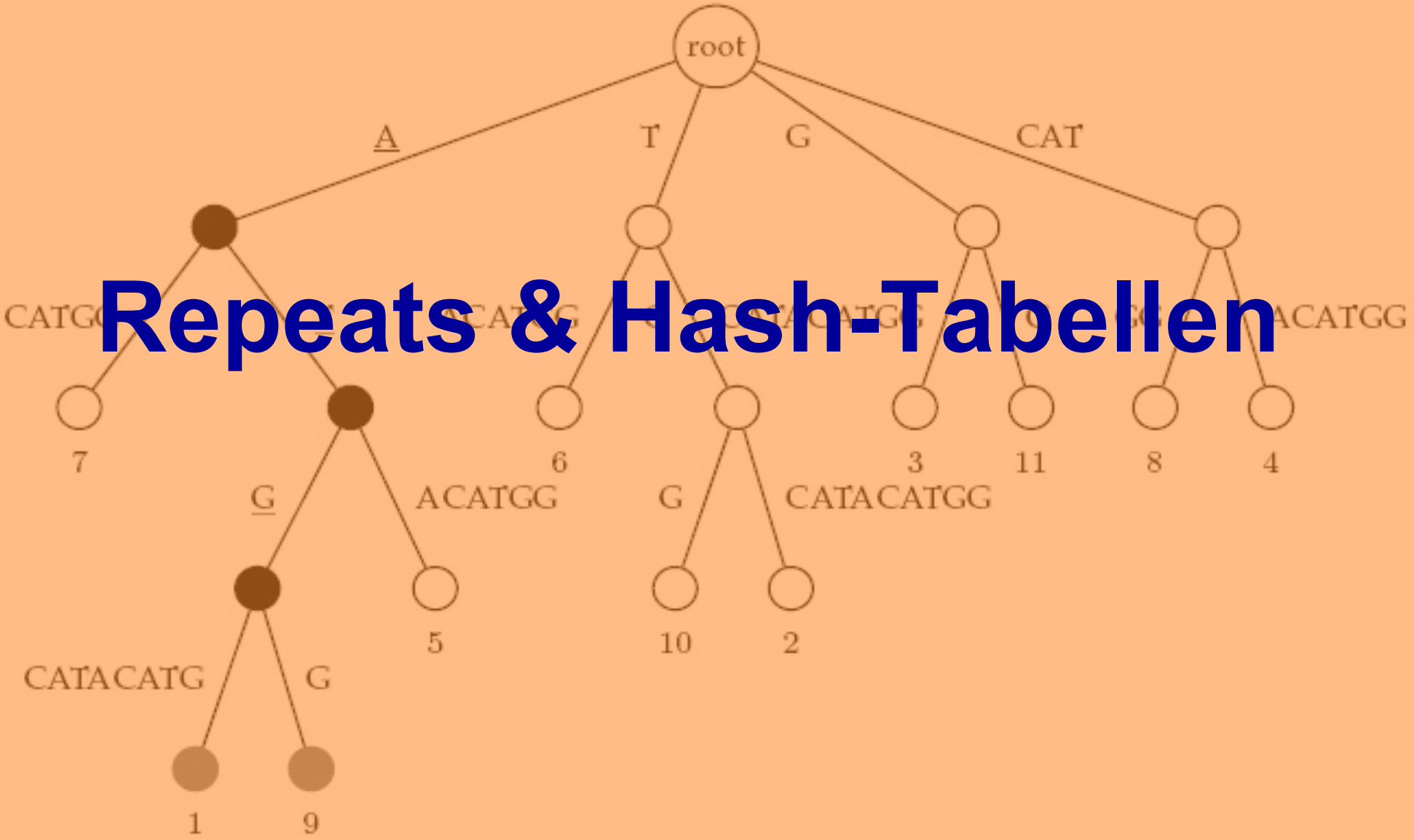
Heinrich Heine Universität Düsseldorf

Algorithmen in der Bioinformatik

7. Kombinatorische Mustersuche

Nach Jones & Pevzner: An Introduction to Bioinformatics Algorithms

Repeats & Hash-Tabellen



Genomische Repeats (Wiederholungen)

ATGGTCTAGGTCCTAGTGGTC

Warum sind die interessant?

- Genomische Rearrangements passieren häufig an Repeats
- Tumoren gehen häufig mit einer “Explosion” von Repeats einher
- Müssen vor einer Motif-Suche herausgefiltert werden
- Vor Assembly auch

Komplikation: Kleine Mutationen sind möglich…

Brachial-Methoden, Branch & Bound

Ziel: Suche nach *lokal maximalen* Repeats, d.h. Repeats, die nicht mehr weiter verlängert werden können

Brachial-Methode: Vergleiche jeden String, der an Position *i* startet, mit jedem gleich langen String, der an Position *j* startet.

Besser: Finde zuerst alle Repeats der Länge *l*, und verlängere die in maximale Repeats.

Warum besser?

- Repeats sind leichter zu finden, wenn wir ihre Länge *l* vorbestimmen
- Suche mit kurzem *l* ist einfach (s. folgende Folien)

Bsp.: GCTTACAGATTCAGTCTTACAGATGG

Hashen

Um alle (maximalen) Repeats zu finden, müssen wir also erst alle Paare von *l*-mer Repeats finden

→ **Hashing**

- Assoziiere eindeutige Schlüssel (keys; hier: *l*-mere aus DNA) mit unterschiedlichen Daten
- Speichere die Anfangsposition(en) jedes *l*-mers der DNA in Tabelle

Definitionen

- Records: Einträge in der Hash-Tabelle
- Keys: Schlüssel zur Identifikation der Records
- Hash-Funktion: Konvertiert den Key in den Index der Hash-Tabelle
- Kollision: Wenn >1 Eintrag in den gleichen Index fällt
  - genau das suchen wir!

Hashen von DNA-Sequenzen

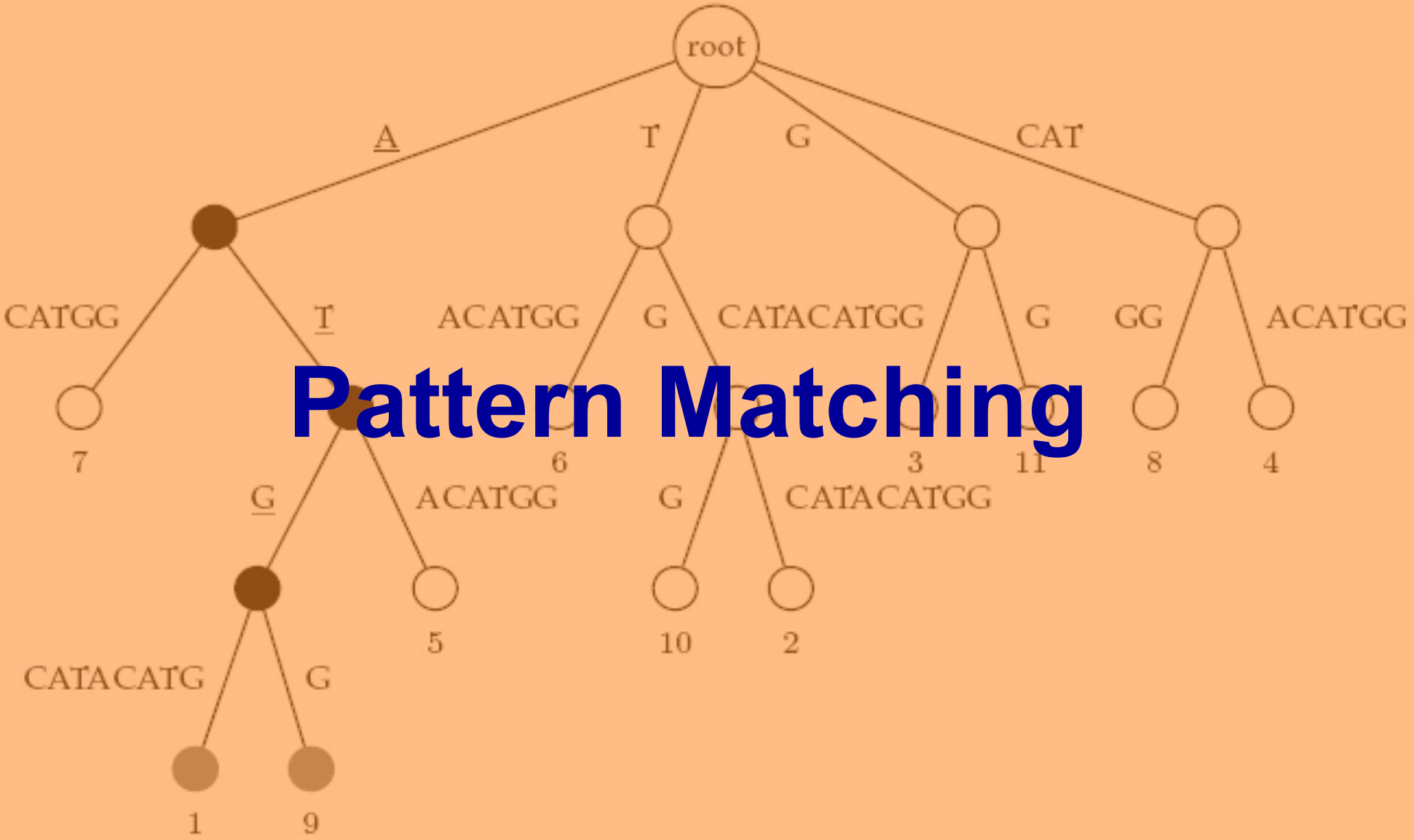
Jedes *l*-mer kann in einen binären String übersetzt werden
(**A→00**, **T→ 01**, **C →10**, **G →11**)

- Key: *l*-mer
- Records: Startpositionen des *l*-mers
- Hash-Funktion: Übersetzung des *l*-mers in binären String der Länge 2*l*
- Tabelle hat 2^{2l} Einträge (Anzahl der mit 2*l* bits darstellbaren Zahlen)

Um maximale Repeats im Genom zu finden:

- Für alle *l*-mere im Genom: speichere Start-Position in Hash-Tabelle
- Für alle Paare von *l*-meren an jeder Position der Tabelle: erweitere in maximalen Repeat

Pattern Matching



Pattern Matching Problem

Motivation: Suche bekanntes Muster in Datenbank

Ziel: Finde alle Positionen eines Musters im Text

Eingabe: Muster $\boldsymbol{p} = p_1 \ldots p_n$ und Text $\boldsymbol{t} = t_1 \ldots t_m$

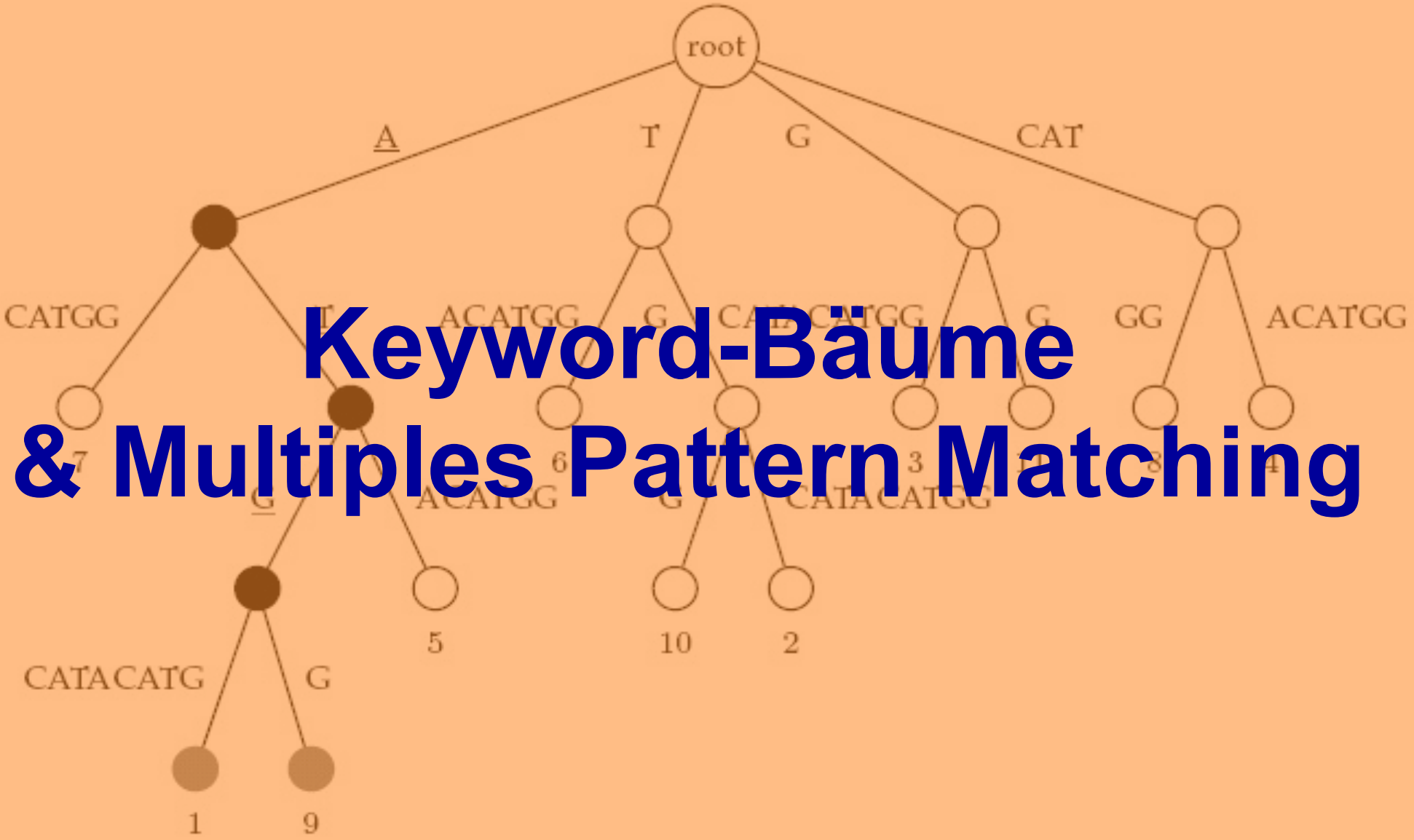
Ausgabe: Alle Positionen i ($1 \leq i \leq m\text{-}n\text{+}1$), für die $(t_i, \ldots, t_{i+n-1}) = \boldsymbol{p}$

Pattern Matching: Brachial-Methode

PatternMatching(**p**,**t**)

1. $n \leftarrow$ länge(**p**)
2. $m \leftarrow$ länge(**t**)
3. for $i \leftarrow 1$ to $(m - n + 1)$
4. if $(\boldsymbol{t}_i \ldots \boldsymbol{t}_{i+n-1} = \mathbf{p})$
5. output i

- Laufzeit: O(nm)
- Für die meisten Eingaben ist es eher O(m)
  (wenn wir in Zeile 4 nur bis zum 1. Mismatch gehen)
- Bessere Lösung: Suffix-Bäume
  - Lösung immer in O(m)
  - Verwandt zu Keyword-Bäumen



Keyword-Bäume & Multiples Pattern Matching

Multiples Pattern Matching Problem

Motivation:

- Durchsuche eine Datenbank nach **mehreren** bekannten Mustern (oder Musterstücken)
- Read Mapping

Ziel: Für k Muster, finde alle Positionen irgendeines Musters in einem Text

Eingabe: k Muster $\mathbf{p}^1,\ldots,\mathbf{p}^k$, und Text $\mathbf{t} = t_1 \ldots t_m$

Ausgabe: Positionen i ($1 \leq i \leq m$), für die ein Substring von **t**, beginnend an Position i, identisch zu einem der $\mathbf{p}^j$ ist.

Multiples Pattern Matching

Einfachste Lösung: *k* unabhängige Pattern-Matching-Probleme

Laufzeit: **O*(kmn)*** mit *k* Läufen des *PatternMatching* Algorithmus
(*k* = # Muster, *m* = Länge(text), *n* = Länge(Muster))

Alternative: Keyword-Bäume (s. nächste Folie)

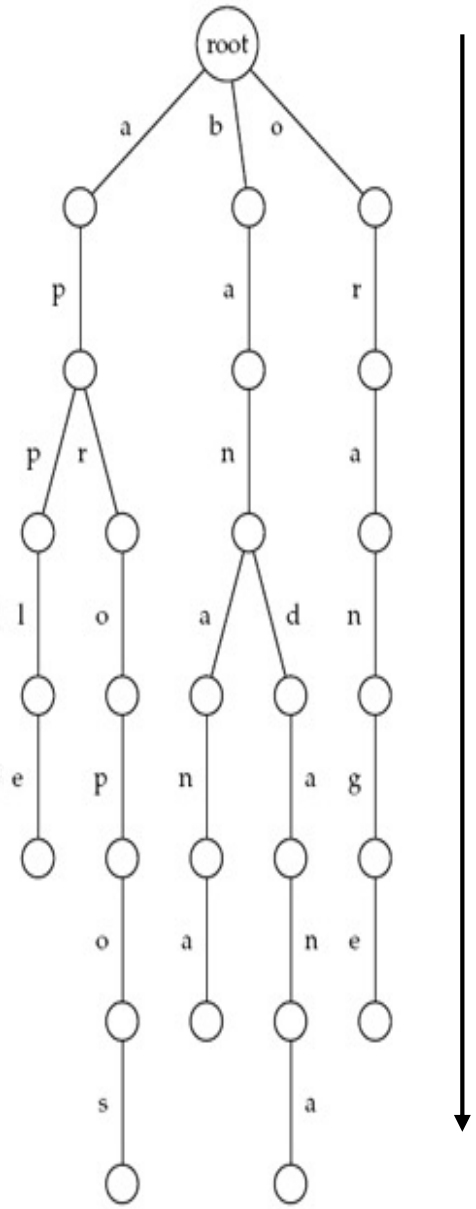
- engl.: *keyword "tries"*
- Bilde Keyword-Baum in O(*N*), mit *N*=Länge($\mathbf{p}^1$,…,$\mathbf{p}^k$)
- Insgesamt (mit naivem Threading): O(*N* + *nm*)
  - Mit Aho-Corasick Algorithmus (nicht diese VL): O(*N* + *m*)

Keyword-Bäume

Eigenschaften

- Speichert *k* Keywörter in einem Baum mit Wurzel
- Jede Kante ist mit 1 Buchstaben gekennzeichnet
- Kanten, die einen Knoten verlassen, haben unterschiedliche Kennzeichen
- Jedes Keywort kann auf einem Pfad von der Wurzel gelesen werden

Multiples Pattern Matching:
Threading (Einfädeln) des Textes an jeder Position *i*



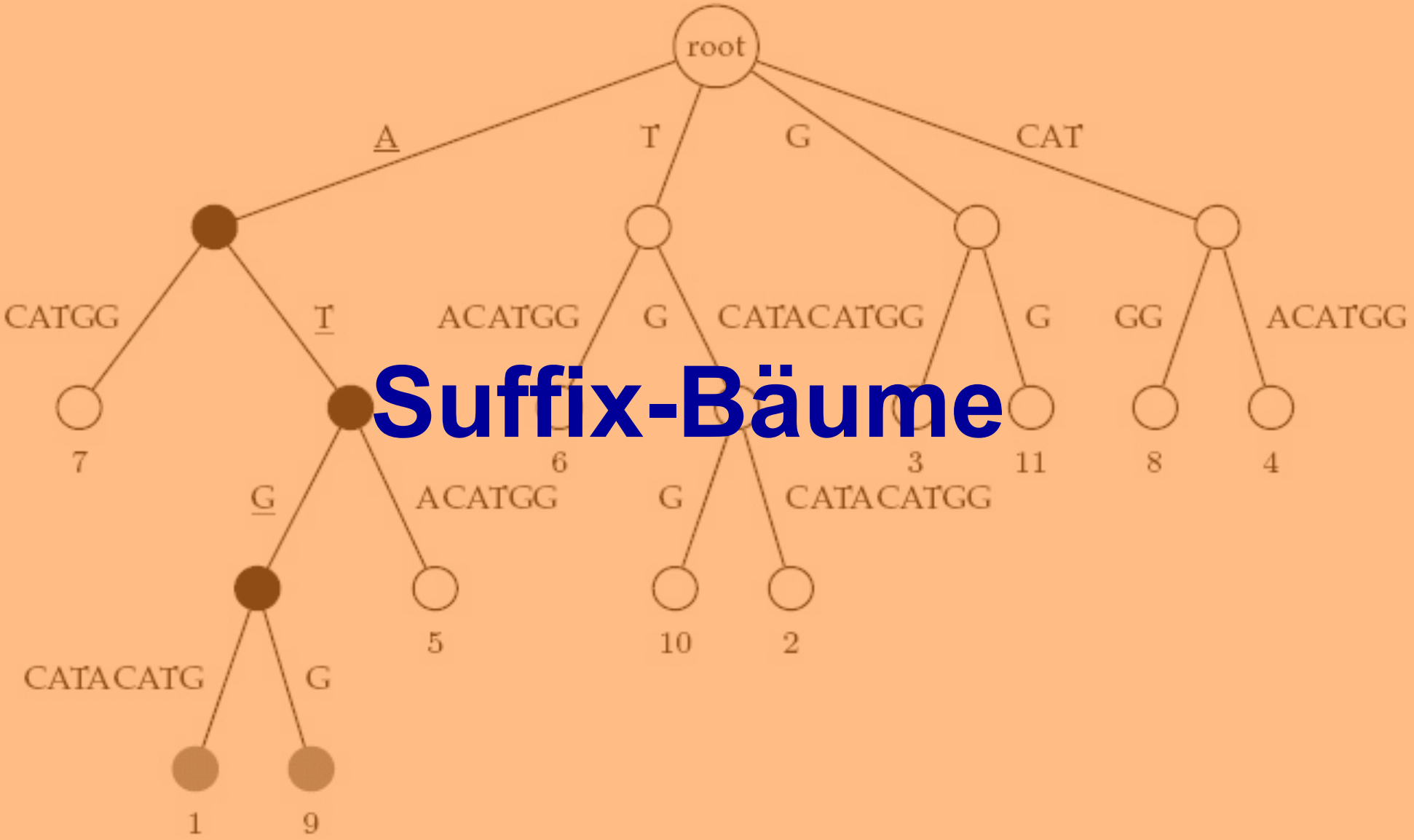
Beispiel: Keywörter

- apple
- apropos
- banana
- orange
- bandana

Beispiel: Threading (Einfädeln)

- appeal
- pineapple

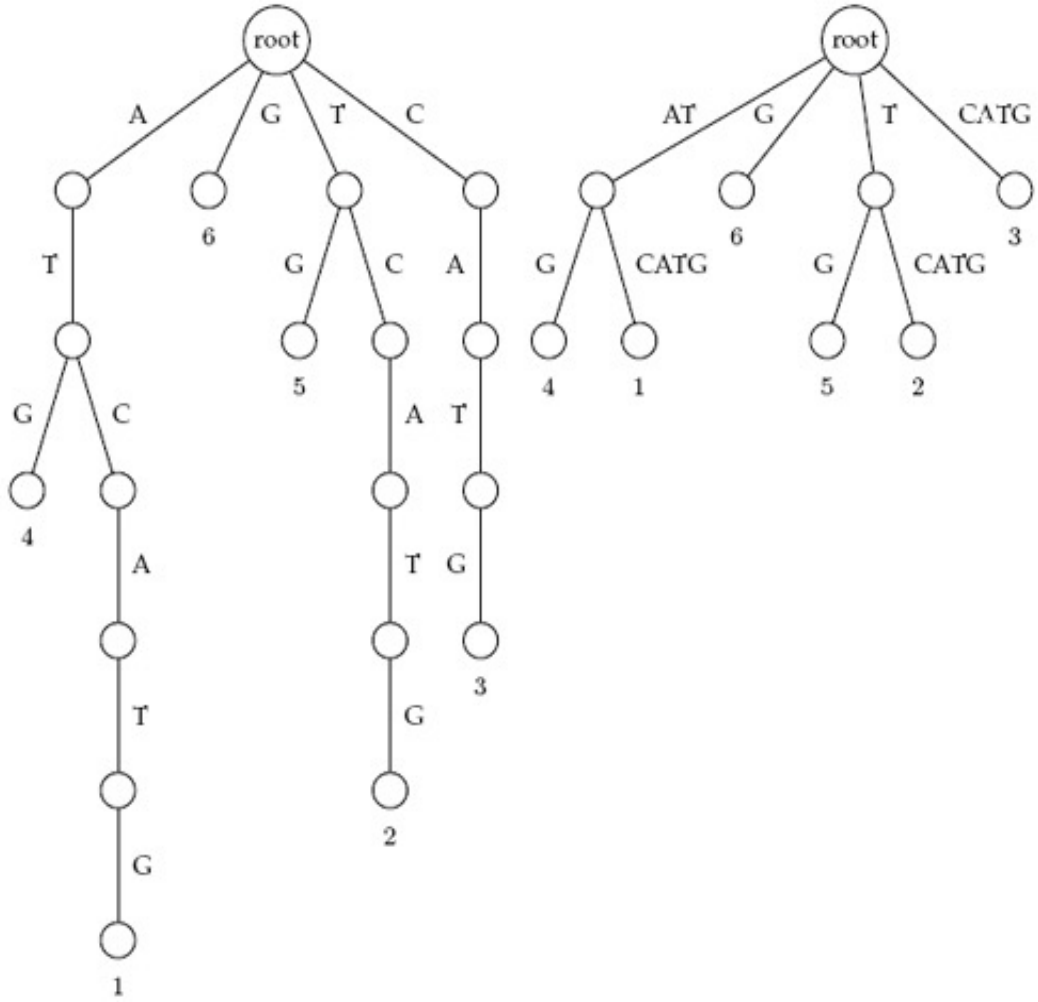
Suffix-Bäume



Suffix-Bäume = kollabierte Keywort-Bäume

Wie Keywort-Bäume aller **Suffixes** eines Textes, aber Knoten ohne Verzweigungen werden entfernt.

- Jede Kante ist mit einem *Substring* gekennzeichnet
- Alle internen Knoten haben ≥ 2 ausgehende Kanten
- Blätter tragen den Startpunkt des endenden Strings (Musters); s.u.

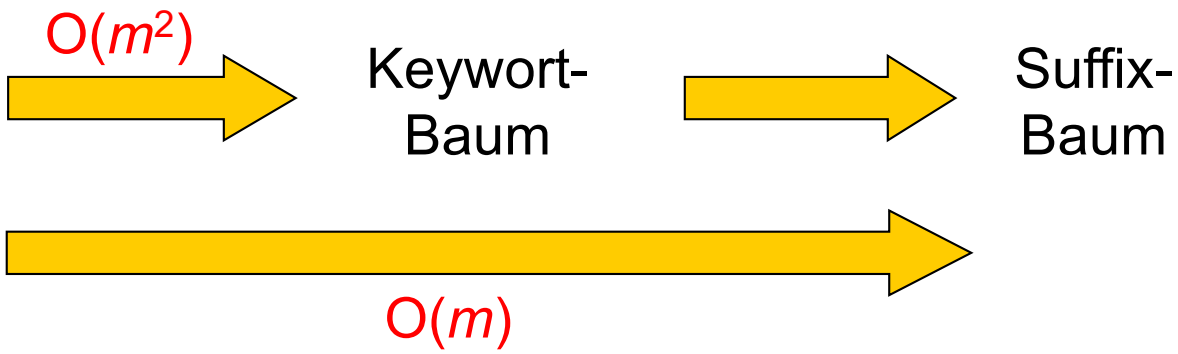


(a) Keyword tree (b) Suffix tree

Suffix-Baum eines Textes

Bilde Suffix-Baum aus allen Suffixen des Textes

ATCATG
TCATG
CATG
ATG
TG
G



Laufzeit?

Linear in der totalen Größe aller Suffixe → O(m^2)

Ausnutzen der Redundanz der Suffixe:
Ukkonen (oder Weiner) Algorithmus: O(m)
(das ist auch die untere Schranke dafür)

Pattern Matching mit Suffix-Bäumen

SuffixTreePatternMatching(**p**,**t**)

1 Bilde **Suffix-Baum** für Text **t**

2 Threading: “Fädele” Muster **p** durch den **Suffix-Baum**

3 **if** (Threading vollständig)

4 **output** (Positionen aller Blätter,
die (Kindes-)Kinder des threads sind)

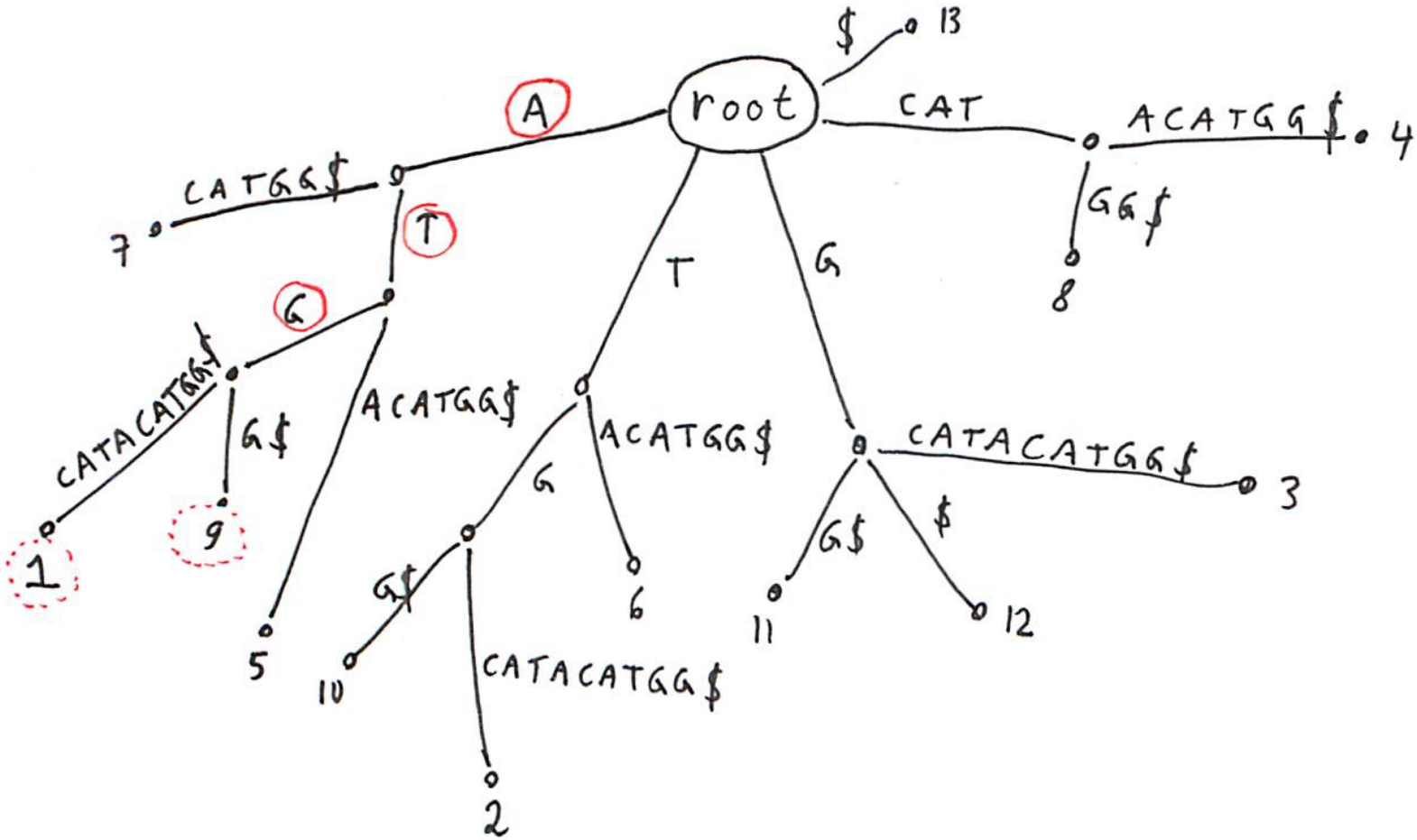
Laufzeit für *k* Muster der Länge *n* in Text der Länge *m*:

- Bilde Suffix-Baum für Text - O(*m*)
- Fädele die Muster durch den Suffix-Baum – *O(k(n* + #Matches)) [1]
- Insgesamt O(*m* + *k(n* + #Matches))

[1]Im VL-Video steht hier nur *O(kn).* Wenn wir viele Matches haben (z. B. ‘a’ in ‘aaaaaaa...aaaa’ suchen, kommt noch die Anzahl der Matches dazu.

Suffix-Baum: Beispiel

Fädele `ATG` durch den Suffix-Baum für `ATGCATACATGG$`



Die Präfixe von `ATGCATACATGG` sowie `ATGG` passen

→ Positionen 1 und 9 als Ausgabe

Keywort-Bäume *vs.* Suffix-Bäume

Keywort-Bäume und Suffix-Bäume helfen, Muster aufzufinden

Keywort-Baum:

Baum aus *k* Mustern

Threading: Text

Suffix-Baum:

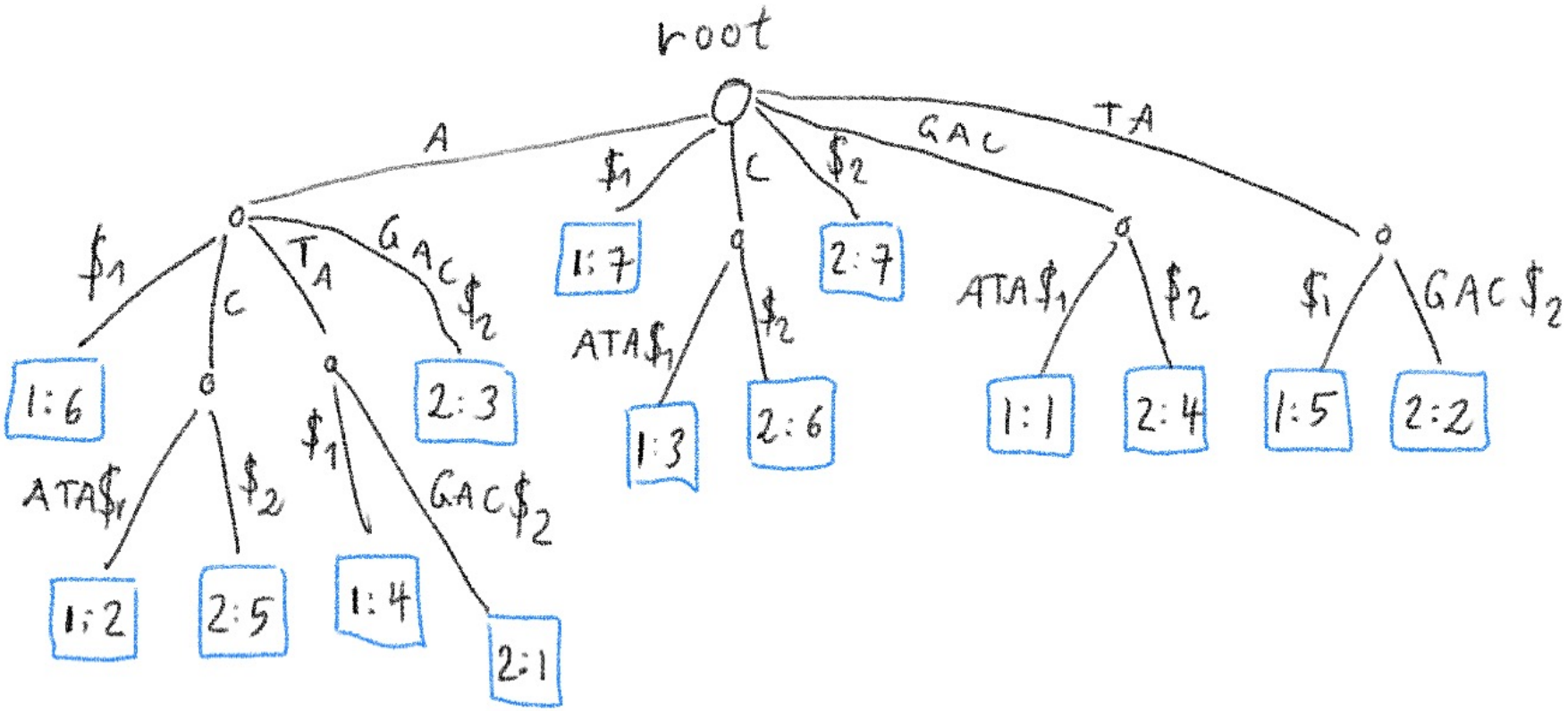
Baum aus Text

Threading: Muster

Generalisierte Suffixbäume für mehrere Strings

- Füge individuelle Spezialsymbole an die Strings s_1, s_2, ... an: $\$_1$, $\$_2$, ...
- Beschrifte Blätter mit *i:j*
  → Offset *j* in s_i

- **Bsp: {“GACATA”, “ATAGAC”}**
- `1234567  1234567`
  `GACATA$1, ATAGAC$2`



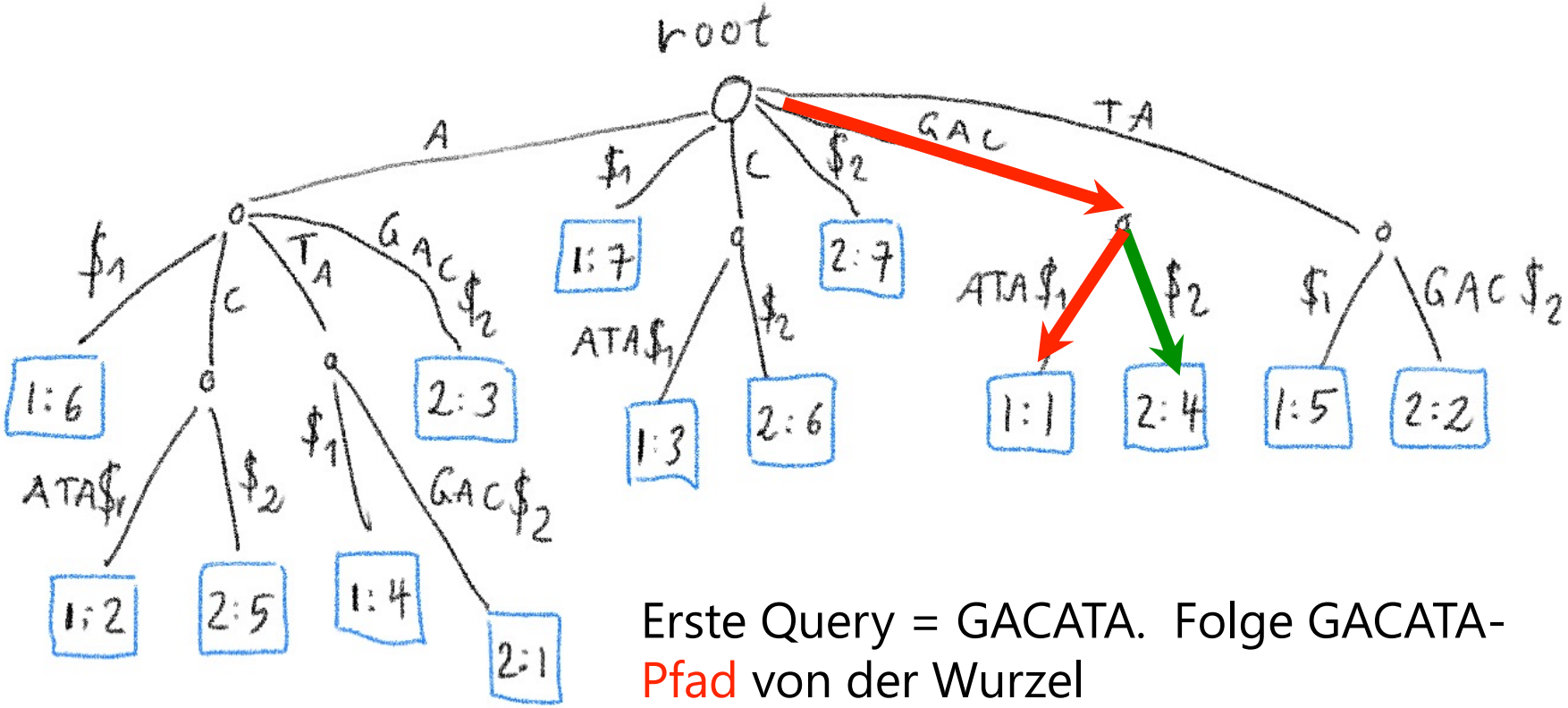
Anwendung: Overlaps finden

Problem: Gegeben eine Menge von Strings *S*, finde für jeden String *x* in *S* alle Präfix-Suffix-Overlaps mit anderen Strings *y* in *S*

→ Baue einen generalisierten Suffixbaum der Strings in *S*

Overlaps finden mit generalisiertem Suffixbaum

Generalisierter Suffixbaum für{ “GACATA”, “ATAGAC” }



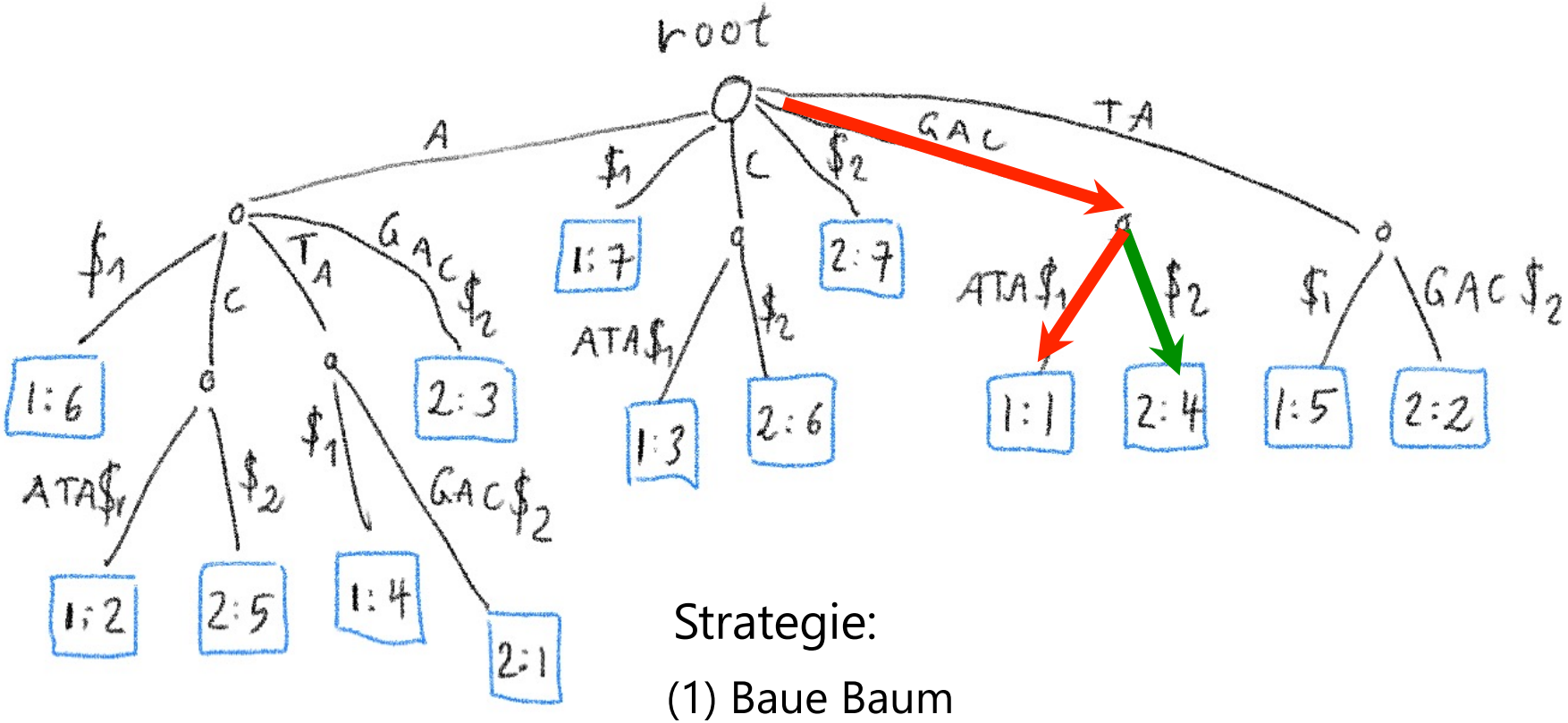
Erste Query = GACATA. Folge GACATA-Pfad von der Wurzel

Grüne Kante: impliziert Suffix des zweiten Strings der Länge 3 mit Präfix der Query

ATAGAC
|||
GACATA

Overlaps finden mit generalisiertem Suffixbaum

Generalisierter Suffixbaum für{ “GACATA”, “ATAGAC” }



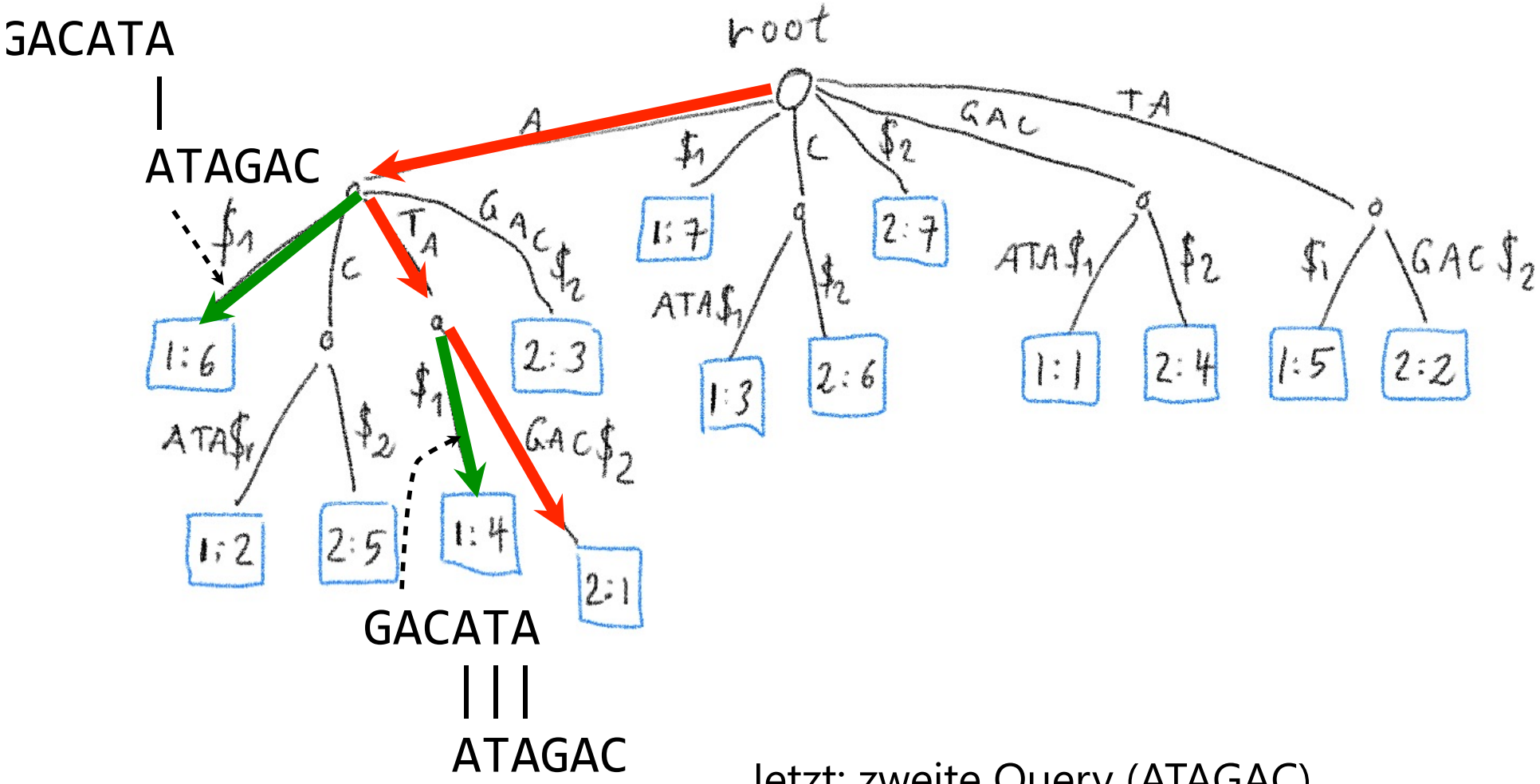
Strategie:

(1) Baue Baum

(2) Für jeden String: Fädele von der Wurzel durch den Baum. Gebe jede ausgehende Kante aus, die mit einem Spezialzeichen $\$_j$ gelabelt ist. Diese führen zu Matches von Präfixen der Query und Suffixen von s_j

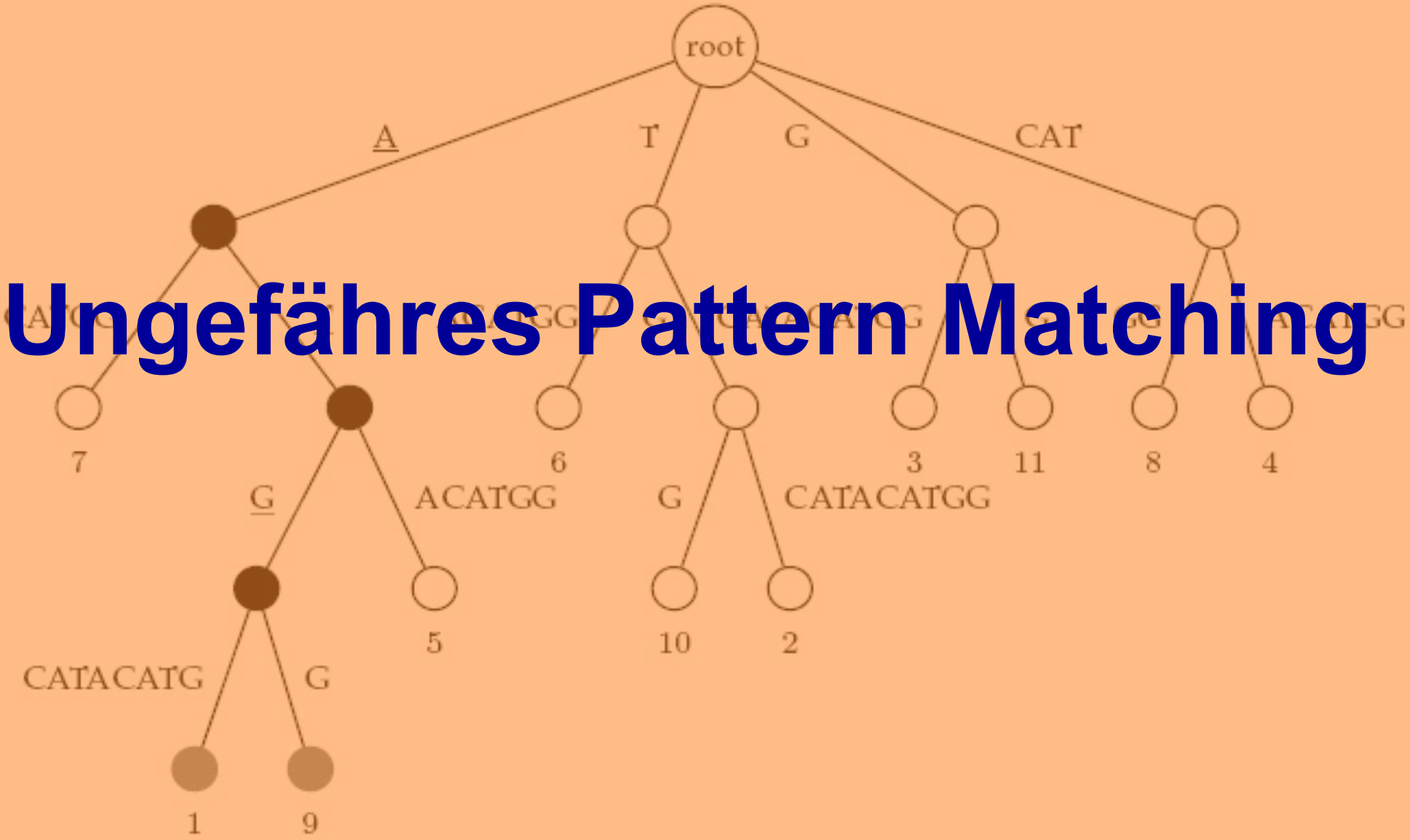
Overlaps finden mit generalisiertem Suffixbaum

Generalisierter Suffixbaum für{ “GACATA”, “ATAGAC” }



Jetzt: zweite Query (ATAGAC)

Ungefähres Pattern Matching



Ungefähres Pattern Matching

Mutationen! Sequenzierfehler!

Biologisch macht es mehr Sinn, *ungefähre* Matches zu suchen

Ziel: Finde alle ungefähren Vorkommnisse eines Musters in einem Text.

Eingabe: Muster $\mathbf{p}=(p_1 \ldots p_n)$, Text $\mathbf{t}=(t_1 \ldots t_m)$, sowie die maximale Anzahl Mismatches k

Ausgabe: Alle Positionen $1 \le i \le m\text{-}n+1$, für die $d_H(t_i \ldots t_{i+n-1}, \mathbf{p}) \le k$ (Hamming-Abstand)

- Local Alignment ist O(mn) - zu langsam für großes m
- Praxis: daher häufig heuristische Such-Algorithmen
- Alignments haben häufig kurze *identische* Regionen.
  Viele heuristische Methoden *filtern* zunächst
  Finde kurze exakte Matches, und nutze diese als *Seeds* für die Erweiterung (mit Mismatches).

Ungefähres Pattern Matching: Brachial

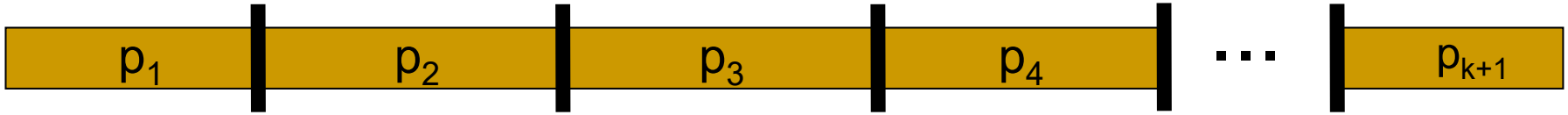
```
ApproximatePatternMatching(p, t, k)
1    n ← Länge(p)
2    m ← Länge(t)
3    for i ← 1 to m−n+1
4        dist ← 0
5        for j ← 1 to n
6            if t_{i+j−1} ≠ p_j
7                dist ← dist + 1
8        if dist ≤ k
9            output i
```

- Laufzeit: O(*nm*).
- (Landau-Vishkin Algorithmus: O(*km*) – beste worst-case Laufzeit, aber praktisch schlechter als filterbasierte Verfahren)

Filtern und Erweitern

Idee: Anstatt direkt nach ungefähren Matches zu suchen (schwer), suchen wir erst nach exakt matchenden Substrings (leicht)

- Teile das Pattern p in k+1 gleichgroße Teile auf (sogenannte Seeds)

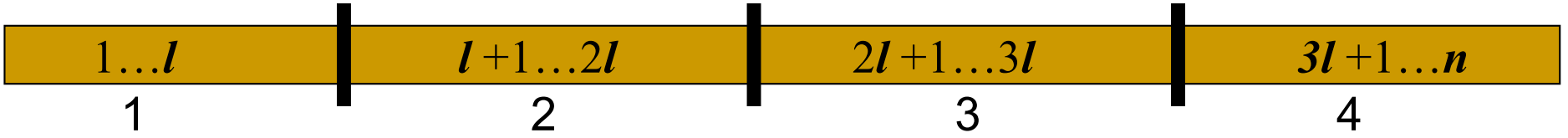


- *k* Mismatches können bis zu *k* Teile betreffen
  $\Rightarrow$ mindestens 1 Teil ist Mismatch-frei
- Suche für jeden Seed alle exakten Matches im Text T
- Erweitere alle gefundenen Matches links und rechts
  - Beispiel: an den Stellen im Text, wo p_2 exakt matcht, müssen wir links um $|p_1|$ Zeichen erweitern und rechts um $|p_3| + ... + |p_{k+1}|$
- Prüfen, ob die Hamming-Distanz ≤ k ist

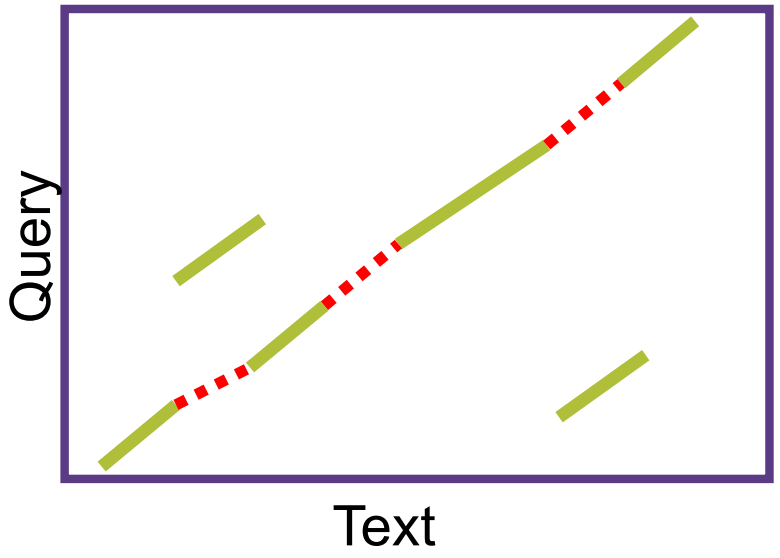
Filtern: Match-Identifikation & -Erweiterung

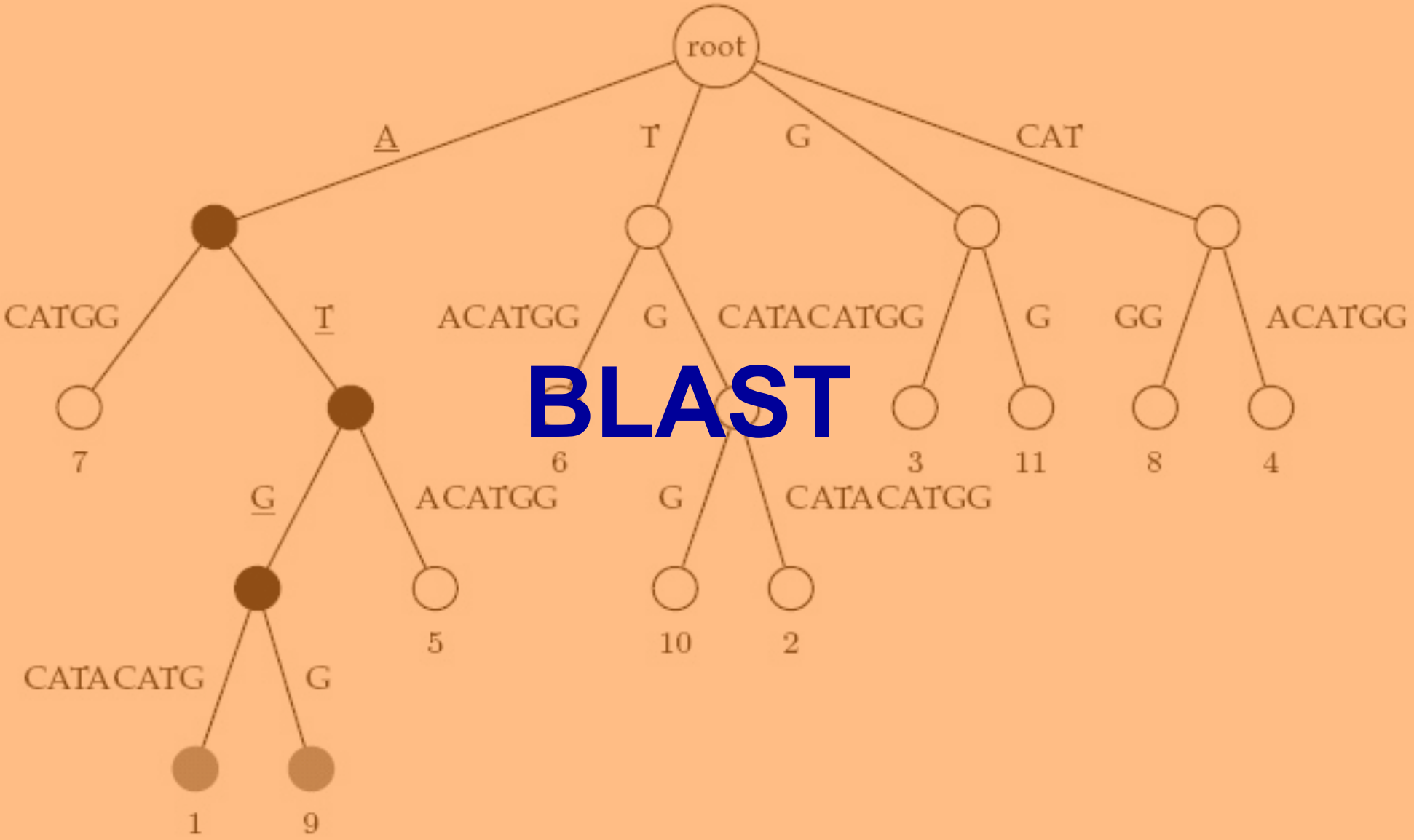
$d_H(x_1 \ldots x_n, y_1 \ldots y_n) \leq k$
$\Rightarrow$ Es gibt ein i mit $d_H(x_i \ldots x_{i+l-1}, y_i \ldots y_{i+l-1}) = 0$, und $l = \left\lfloor \frac{n}{k+1} \right\rfloor$

Beweis: Breche **x** und **y** in k+1 Teile mit Länge l. k Mismatches können höchstens k Teile betreffen $\Rightarrow$ 1 Teil ist Mismatch-frei.$_{qed}$



Match-Erweiterung
Vergrößere exakte Matches der Länge l zu ungefähren Matches der Länge n mit $d_H < k$




root
A
T
G
CAT
CATGG
T
ACATGG
G
CATACATGG
G
GG
ACATGG
7
6
3
11
8
4
G
ACATGG
G
CATACATGG
5
10
2
CATACATG
G
1
9

BLAST

BLAST

Basic **L**ocal **A**lignment **S**earch **T**ool (Altschul *et al.* 1990)

- Viel schneller als dynamische Programmierung (Smith-Waterman), mit nur geringem Verlust an Sensitivität (fast gleich viele Matches werden erkannt)
- Minimiert zuerst den Suchraum für ungefähre Matches: Findet kurze exakte Matches (Seeds) und sucht dort lokal

BLASTen ist ein Verb (so wie Googlen):

- BLASTen eines neuen Gens gegen eine Datenbank
  - Evolutionäre Verwandtschaft
  - Funktions-Vorhersage
- BLASTen eines Genoms gegen eine Datenbank
  - Potentielle Gene
- BLASTen eines Genoms gegen ein anderes Genom
  - Orthologe Gene
- BLASTen von Transkripten gegen ein Genom
  - Identifikation von transkribierten Genen

BLAST Algorithmus

- Keywort-Suche für alle Wörter der Länge *w* in der Query
  mit Score > Cutoff
  (nicht nur exakte Matches: Transitionsmatrizen wie bei Alignment)
  - *w*=11 für DNA, *w*=3 für Proteine
- Erweiterung zu lokalem Alignment für jedes gefundene Keywort,
  bis Score unter Cutoff fällt
- Laufzeit O(*wm*)

BLAST Algorithmus