Bioinformatik

Ukkonen‘s Algorithmus

Lineare Konstruktion von Suffixbäumen

Ulf Leser

Wissensmanagement in der Bioinformatik

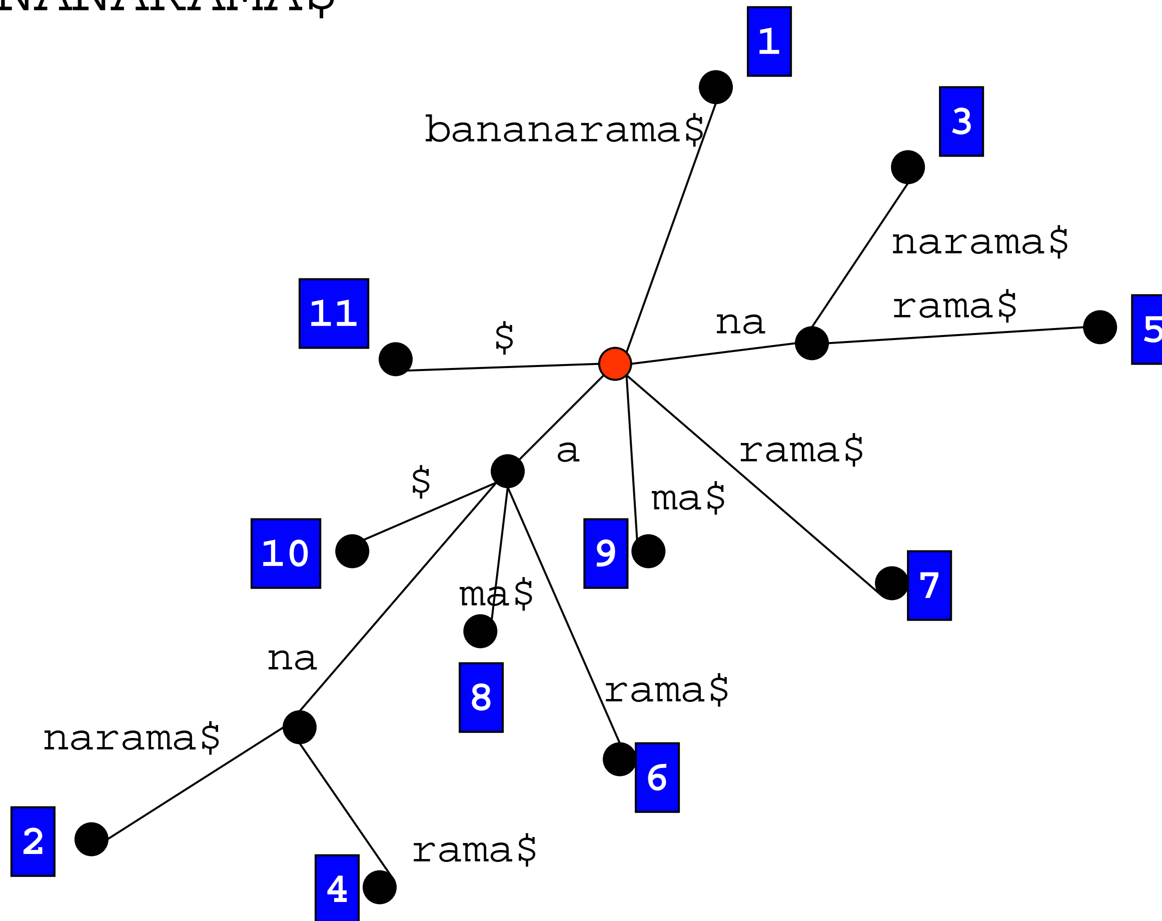
Wo sind wir überhaupt?

- 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

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?

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

Komplexität

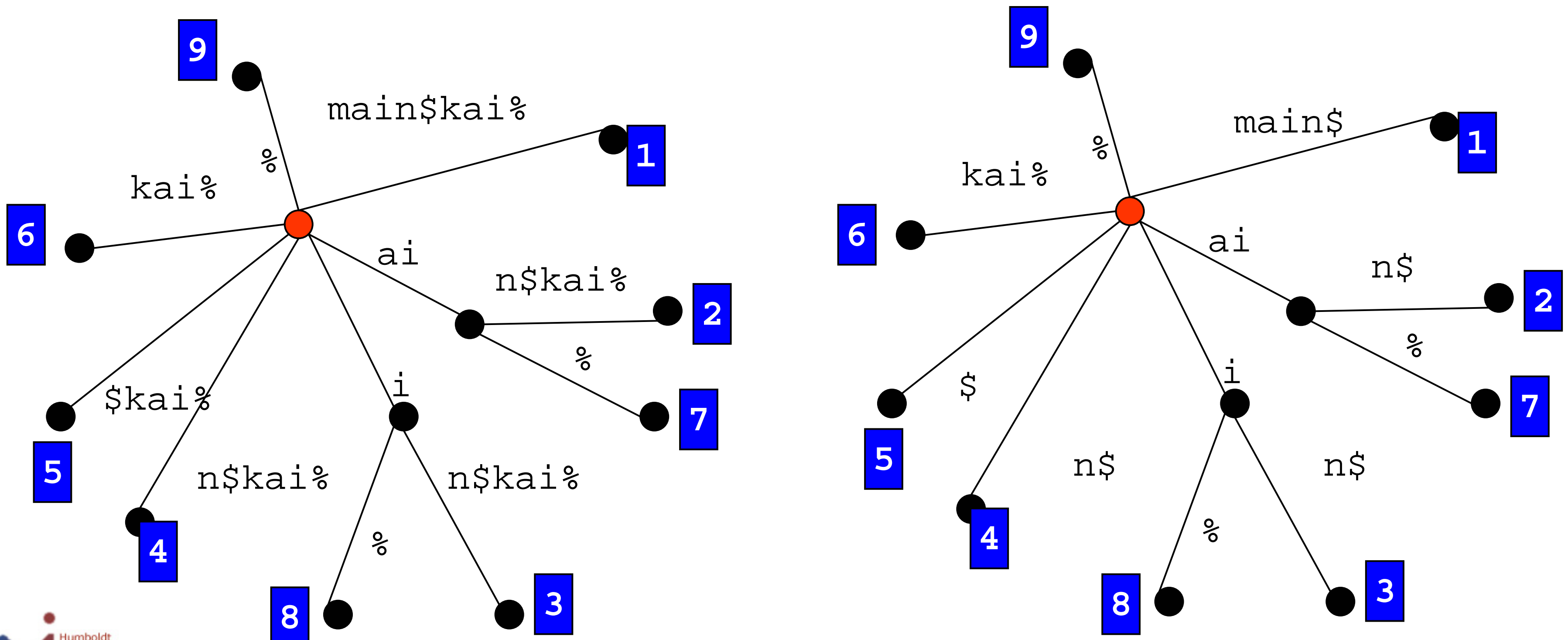
- Theorem
  Sei T der Suffixbaum für S+"$". Die Suche nach allen Vorkommen eines Pattern P, |P|=n, in S ist O(n+k), wenn k die Anzahl Vorkommen von P in S ist.
- Beweisidee
  - P in T matchen kostet O(n)
    - Pfade sind eindeutig – Entscheidung an jedem Knoten ist klar
    - Damit maximal O(n) Zeichenvergleiche
  - 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

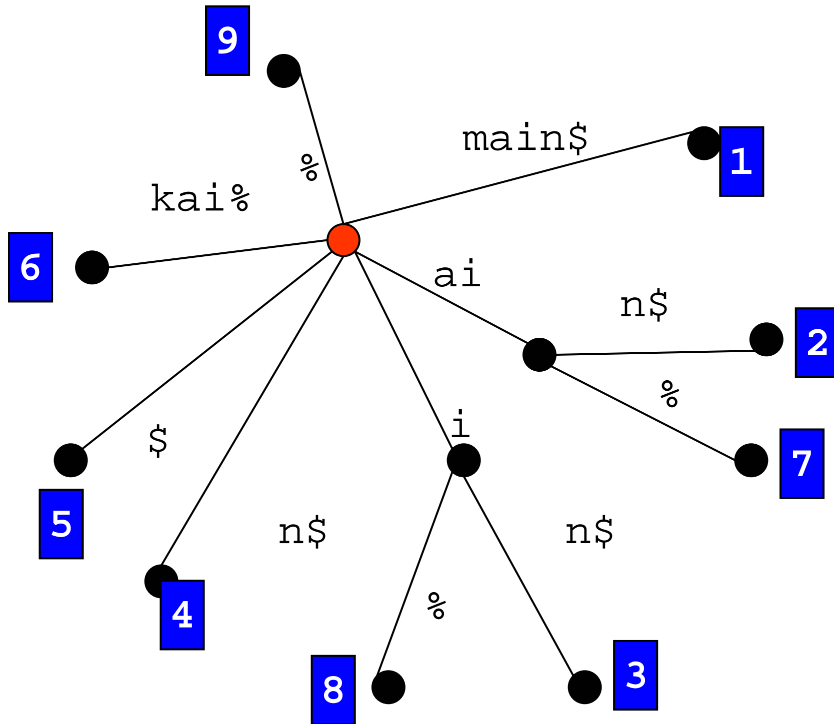
- Gegeben zwei Strings S_1 und S_2
- Gesucht: Längster gemeinsamer Substring s
- Vorschläge ?
- Lösung
  - Konstruiere Suffixbaum T für $S_1\$S_2\%$
  - Streiche aus diesem Baum alle Pfade unterhalb eines „$“
  - Durchlaufe den Baum
    - markiere alle internen Knoten mit 1, wenn im Baum darunter ein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

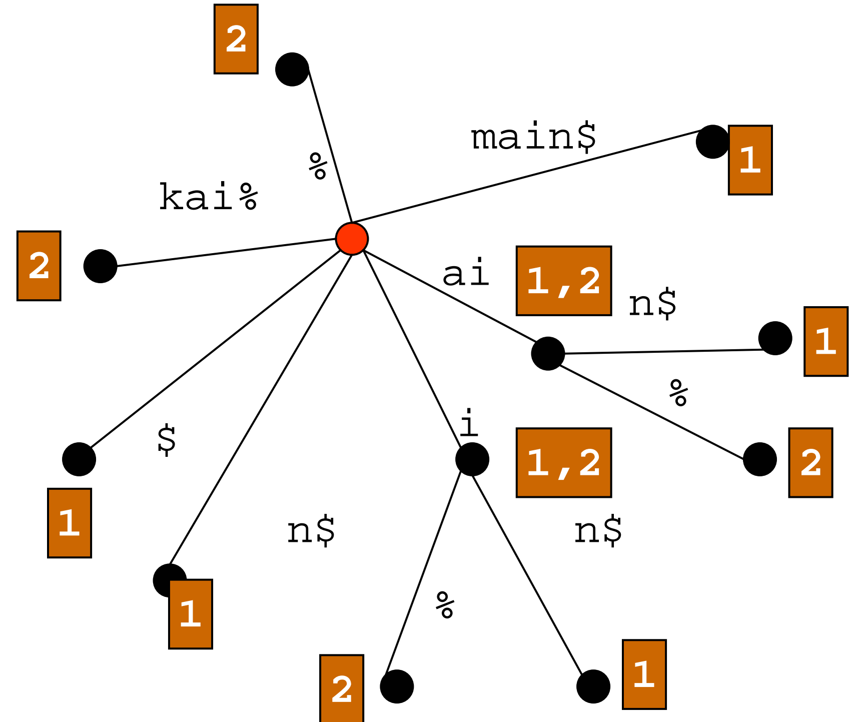
Beispiel

- S_1=main$, S_2=kai%
- ...



Beispiel





- Verallgemeinerbar zu n Strings $S_1,\ldots,S_n$

Naive Konstruktion

- 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
  - Füge bei Mismatch neue Knoten und Blätter ein
- Komplexität
  - Jeder Schritt von T_i zu T_{i+1} ist O(m)
  - Es gibt m-1 solche Schritte
  - Zusammen: $O(m^2)$

Schöne Wörter

- Rokokokokosnuss

Inhalt dieser Vorlesung

- Ukkonen's Algorithmus zu linearen Konstruktion von Suffixbäumen
  - Einer der komplizierteren der Vorlesung
  - Ziel ist das Verständnis der Schritte und der resultierenden Komplexitätsgrenze
  - Einige Details werden ausgelassen (insb. Hilfsdatenstrukturen)
  - Kein vollständiger Pseudocode

Warum ist Ukkonen's Algorithmus wichtig?

- Gegeben ein String S mit Länge m
- Gesucht: Suffixbaum T für S
- Naives Konstruktionsverfahren braucht $O(m^2)$
  - Damit Suche von P in T: $O(m^2) + O(n)$
  - Das ist (auch für sehr viele) Pattern deutlich teurer als BM etc.
- ➢ Suffixbäume nur sinnvoll, wenn Konstruktion in O(m) gelingt
- Verschiedene Algorithmen
  - Weiner: „Linear pattern matching algorithms", IEEE Symposium on Switching and Automata Theory, 1973
  - McCreight: „A space-economical suffix tree construction algorithm", Journal of the ACM, 1976
  - Ukkonen: „Online construction of suffix trees", Algorithmica, 1995

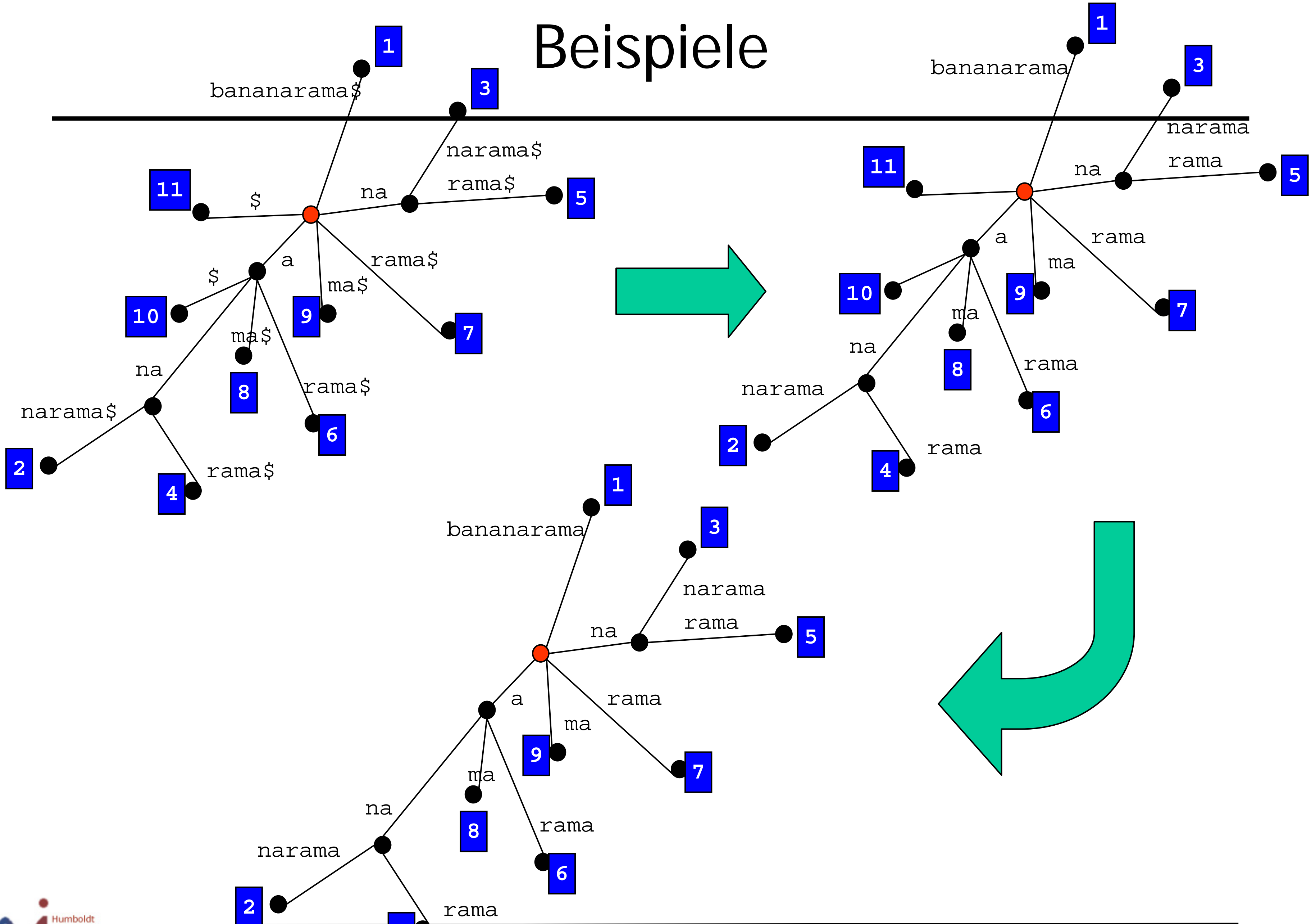
Überblick

- Voraussetzungen
- High-Level: Phasen und Extensionen
  - Führt leider zu $O(m^3)$
- Verbesserungen
  - Suffix-Links
  - Skip/Count Trick
  - Noch zwei Tricks
- Alles zusammen: $O(m)$

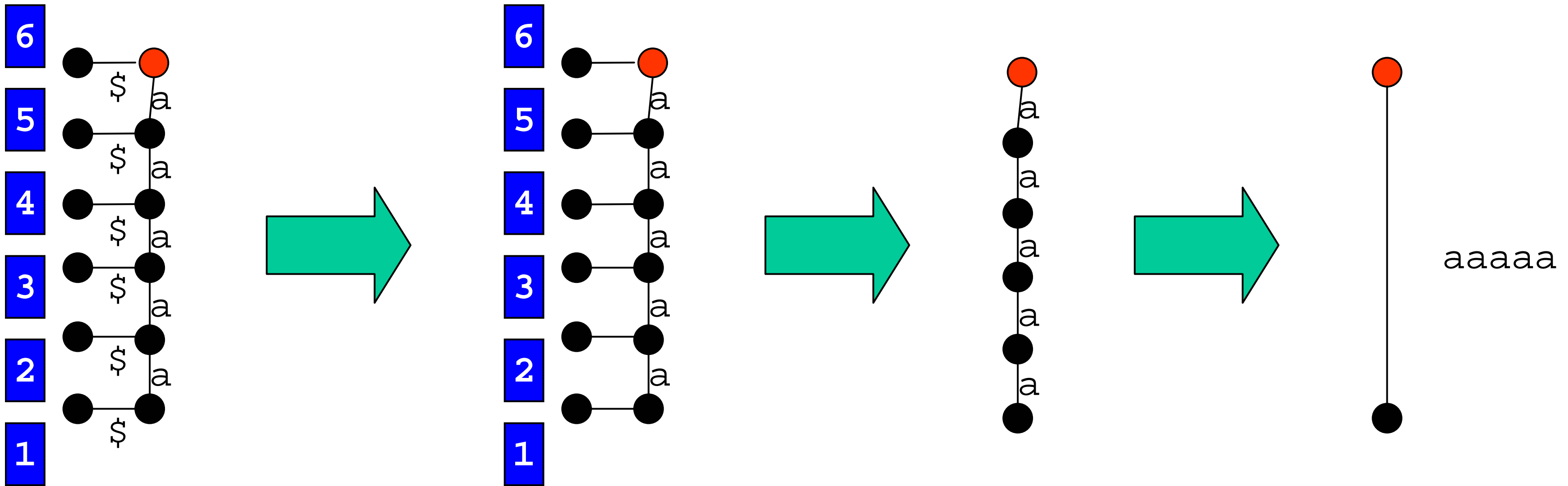
Voraussetzungen

- Definition:
  Seit T ein Suffixbaum für S. Der implizite Suffixbaum T' entsteht aus T durch
  - *Entferne alle Vorkommen von „$" aus allen Labels*
  - *Entferne alle Kanten ohne Label*
  - *Entferne alle inneren Knoten mit weniger als 2 Kindern; konkateniere die Label der eingehenden und der ausgehenden Kante zu dem der neuen (durchgehenden) Kante*
- Das ist die Definition; wir werden ganz anders vorgehen

Beispiele



Beispiel



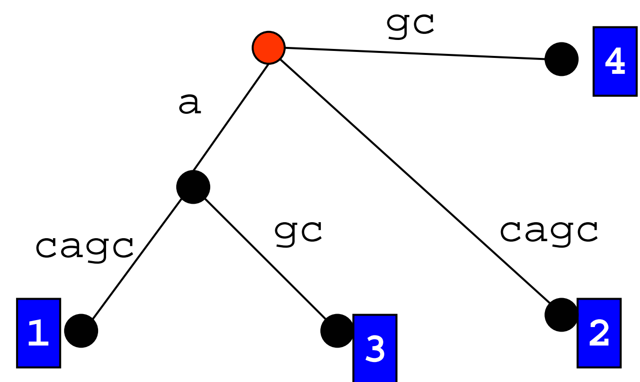
- Implizite Suffixbäume codieren immer noch alle Suffixe
  - Jedes Suffix matched entlang eines Pfads
  - Aber man kann die Enden nicht mehr erkennen
  - Nicht für alle Suffixe gibt es eine Markierung an einem Blatt

Grundaufbau Ukkonen's Algorithmus

- Induktive Konstruktion impliziter Suffixbäume für jedes Präfix von T
  - Wir konstruieren alle T_i, d.h., implizite Suffixbäume für S[1..i]
  - Startpunkt T_1: Wurzel und ein Knoten mit Kantenlabel S[1]
  - Phasen: Konstruktion von T_{i+1} aus T_i
  - Abschluss: Transformation von T_m in den „echten" Suffixbaum T
- Jede der m-1 Phasen besteht aus Extensionsschritten
  - Phase i hat i Extensionsschritte
  - Jeder Schritt verlängert ein Suffix von S[1..i] um das Zeichen S[i+1]
  - Der letzte Schritt jeder Phase verlängert das leere Suffix – einfügen von S[i+1]
  - Reihenfolge der Schritte: von links nach rechts (S[1..i], S[2..i], ...)
- Wie wird verlängert?
  - Drei Extensionsregeln

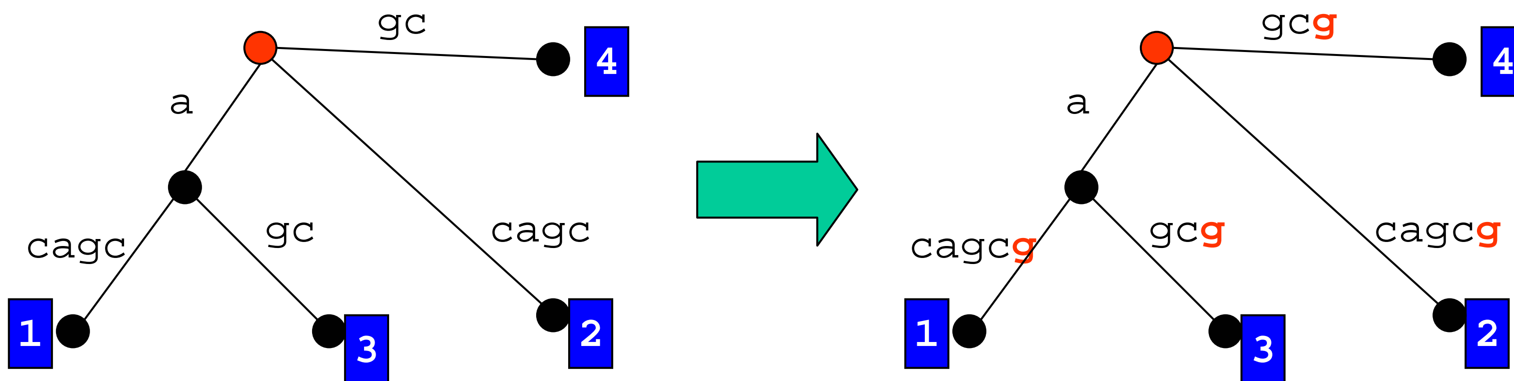
Extensionsregeln

- In Extensionsschritt j in Phase i+1 verlängern wir S[j..i] um das Zeichen S[i+1]
  - Sei b=S[j..i]
- Matche b in T_{i+1} (im Entstehen). Das geht bis …
  - 3 mögliche Situationen können entstehen
- Beispiel
  - S=„acagcg “
  - Wir haben T_5 und bauen T_6
  - Also: Alle Suffixe von „acagc“ erweitern
  - und zwar um S[6]=„g“



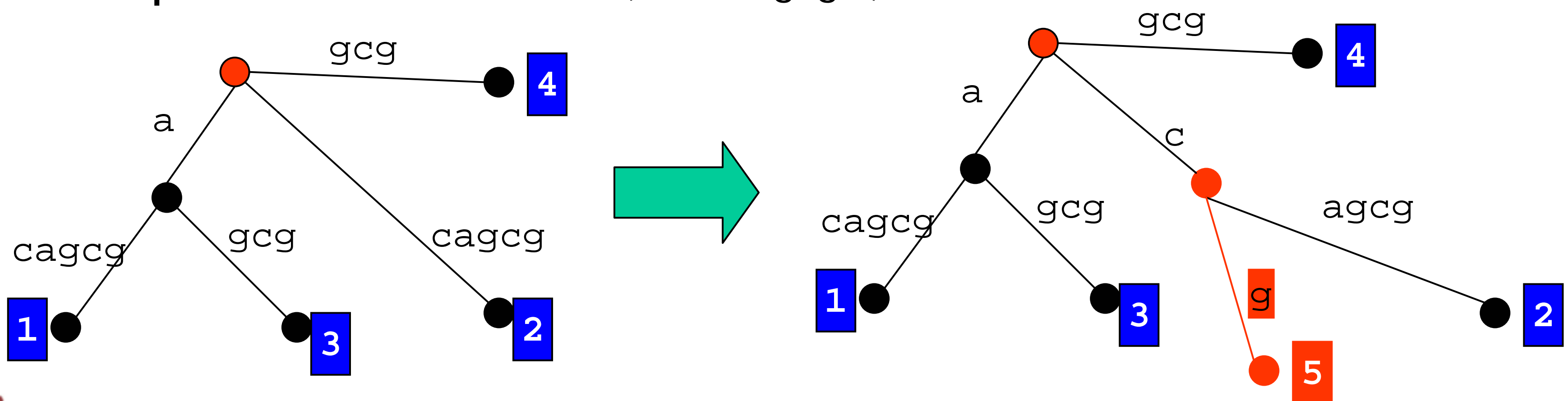
Extensionsregel 1

- Matche b in T_{i+1}. Das geht bis ...
  - Regel 1: b endet in einem Blatt
  - ➢ Erweitere das Label der letzten Kante um S[i+1]
- Beispiel (wir hängen „g“ an Suffixe von „acagc“)
  - Erweiterung von „acagc“, „cagc“, „agc“, „gc“
  - [es bleiben Schritt 6 „“ und Schritt 5 „c“]



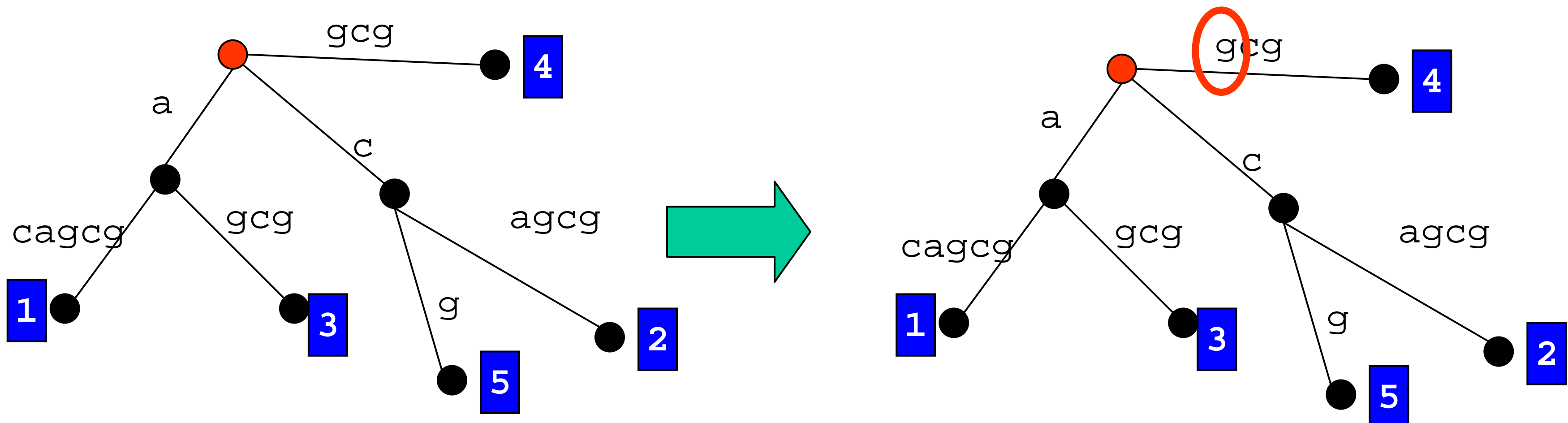
Extensionsregel 2

- Matche b in T_{i+1}. Das geht bis …
  - Regel 2: b endet an einem inneren Knoten oder in einer Kante, und kein weiterer Pfad beginnt mit S[i+1]
    - Innerer Knoten: Neues Blatt unterhalb dieses Knotens mit Kantenlabel S[i+1]; markiere Blatt mit „j“
    - In einer Kante: Neuer innerer Knoten, der diese Kante teilt; neues Blatt wie oben
- Beispiel: Schritt 5, „c“ (S=„acagcg “)



Extensionsregel 3

- Matche b in T_{i+1}. Das geht bis ...
  - Regel 3: b endet an einem inneren Knoten oder in einer Kante, und einer der weiteren Pfade beginnt mit S[i+1]
  - ➢ Tue gar nichts
- Beispiel: Schritt 6, „“ (S=„acagcg “)



Komplettes Beispiel

- Konstruiere implizite Suffixbäume $T_1 \ldots T_6$ für „gtcgtg“
- …

Algorithmus und Komplexität

```
construct T1;
for i=1 to m-1                              // m-1 phases
    Ti+1 := Ti;
    for j = 1 to i+1                        // i+1 extensions, left-right
        match S[j..i] in Ti+1;
        extend Ti+1 with S[i+1];            // Using one of the 3 rules
    end for;
end for;
```

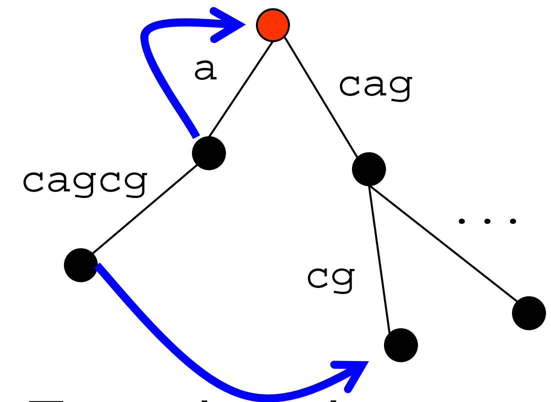
- Die zwei Schleifen sind $O(m^2)$
- Finden der Suffixe b in T_{i+1} ist $O(m)$
- Extension ist konstant
- Zusammen: $O(m^3)$

Suffix-Links

- Wir wenden uns dem „innersten“ Problem zu – immer wieder das Ende von Suffixen zu finden
  - Wir reduzieren die Zeit für alle „match S[j..i] …“ Operationen in einer Phase auf zusammen O(m)
- Intuition
  - Für jedes gefundene S[j…i] gibt es per Konstruktion irgendwo schon S[j+1..i]
  - Wo? Wir wollen nicht suchen, sondern Suffix-Links merken und direkt anspringen
  - Außerdem: Da S[j..i] und S[j+1..i] bis auf S[j] identisch sind, werden wir von Knoten zu Knoten springen – Skip/Count Trick – statt Zeichen einzeln im Baum zu matchen
  - Zusammen: Komplexität in einer Phase abhängig von Anzahl der Knoten, nicht der Zeichen

Suffix-Links formal

- Definition
  - *Sei k ein innerer Knoten des impliziten Suffixbaums T' für S*
  - *Sei label(k)=xa, wobei |x|=1, |a| beliebig (auch 0)*
  - *Sei k' ein innerer Knoten von T' mit label(k')=a*
  - *Der Pointer (k,k') heißt Suffix-Link*



- Spezialfall für |a|=0
  - Der Suffix-Link geht dann zur Wurzel
- Wir zeigen, dass jeder innere Knoten in T' nach jeder Phase von Ukkonen's Algorithmus einen Suffix-Link hat
- Diese Links werden wir dann in späteren Phasen entlang springen

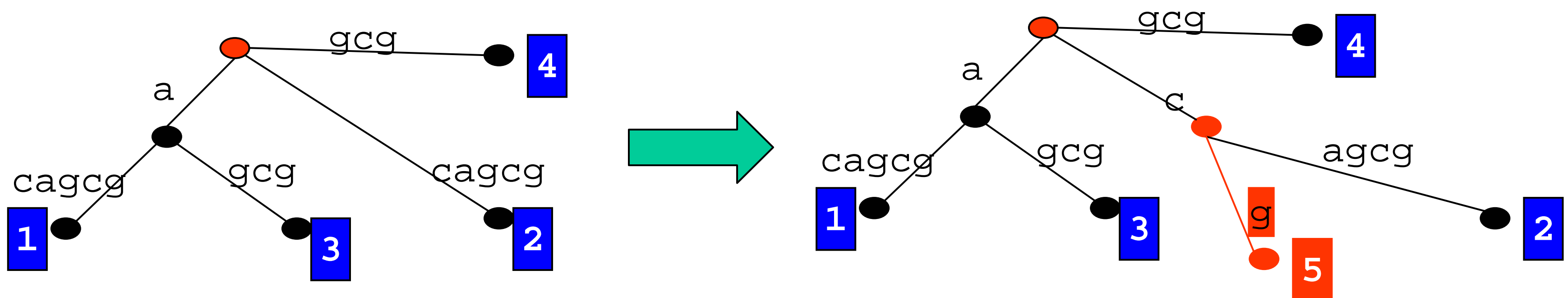
Suffix-Links in impliziten Suffixbäumen

- Neue innere Knoten entstehen nur in Regel 2

– Regel 2: b endet an einem inneren Knoten oder in einer Kante, und kein weiterer Pfad beginnt mit S[i+1]

➢ Innerer Knoten: Neues Blatt, … (uninteressant)

➢ In einer Kante: Neuer innerer Knoten teilt diese Kante sowie neues Blatt wie oben

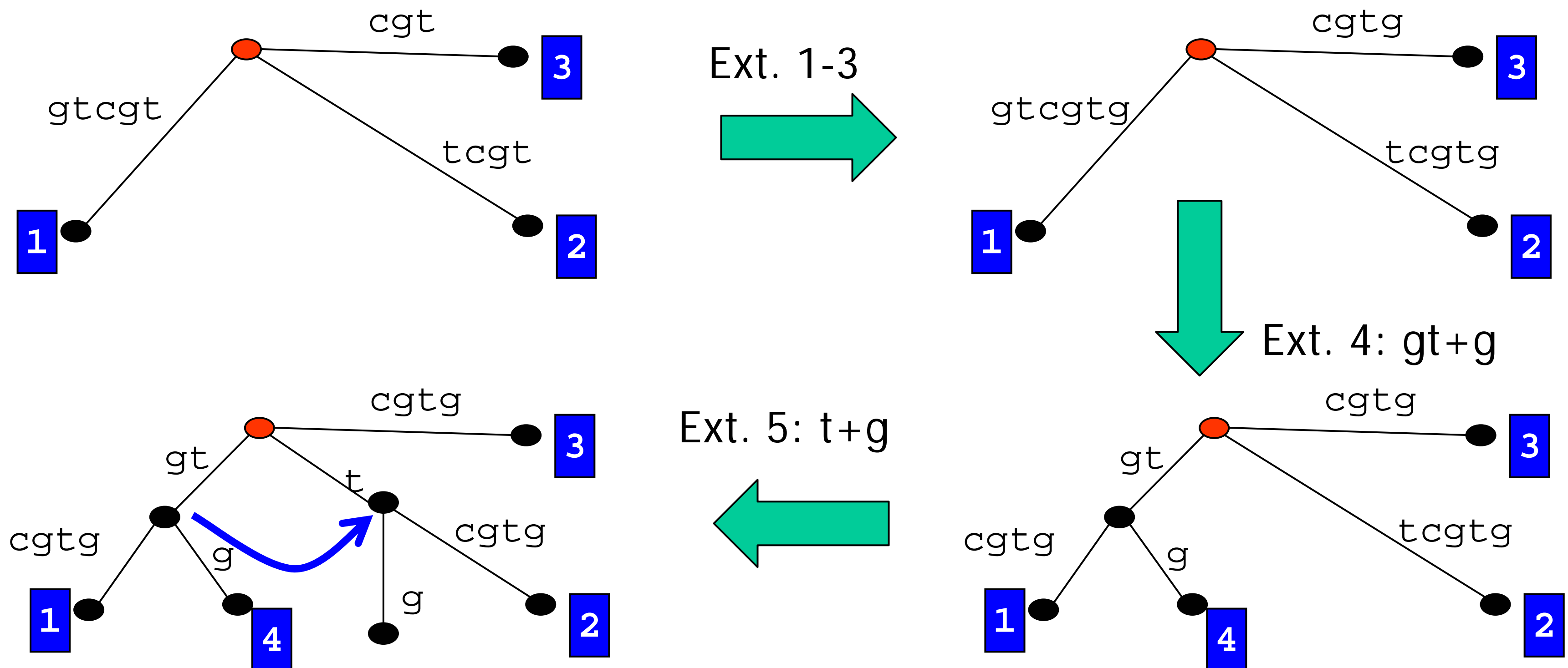


Suffix-Links in impliziten Suffixbäumen

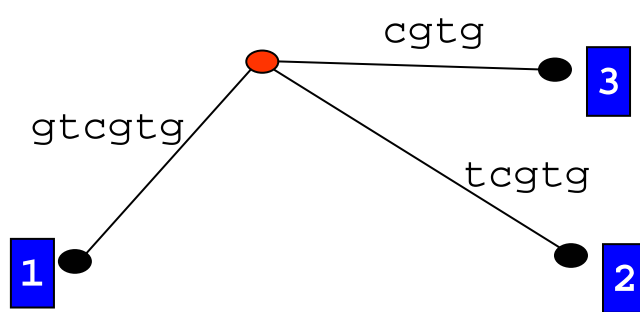
- Theorem
  - *In Ukkonen's Algorithmus hat jeder innere Knoten spätestens nach dem Ende des nächsten Extensionsschritts einen Suffix-Link*
- Folgerung
  - *Nach der letzten Extension in einer Phase von Ukkonen's Algorithmus hat jeder innere Knoten einen Suffix-Link*
  - *Nach jeder Phase von Ukkonen's Algorithmus hat jeder innere Knoten einen Suffix-Link*
- Die Folgerung ist einfach …
  - Denn: In der letzten Extension einer Phase wird S[i+1] eingefügt – das kann keine inneren Knoten mehr erzeugen
- … das Theorem nicht
  - Erst ein Beispiel … dann der Beweis

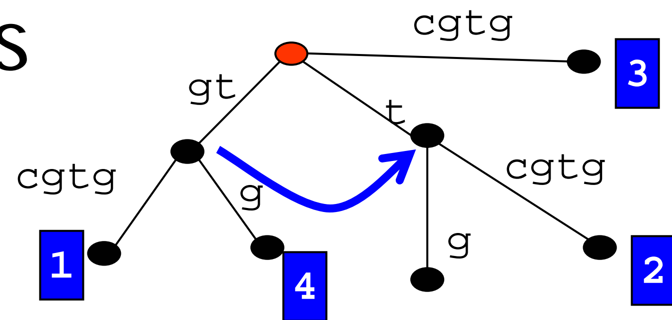
Beispiel

- „gtcgtg“ in Phase 6
  - Beobachtung: Jeder innere Knoten hat nach Phase 5 einen Suffix-Link (es gibt keine)
  - Wir erweitern in jeder Extension dieser Phase mit S[6]=„g“



Beweis





- Angenommen, im Extensionsschritt j der Phase i+1 fügen wir einen neuen inneren Knoten ein
  - Der habe Label xa mit |x|=1
  - Nach xa gab es nach dem Knoten keine Fortsetzung mit S[i+1]
    - Sonst hätten wir keinen neuen Knoten gebraucht
  - Aber es gab mindestens eine Fortsetzung, z.B. mit c
    - Sonst hätten wir nur das Label eines Blattes verlängert
- Was passiert dann im nächsten Extensionsschritt j+1?
  - Den Pfad ac muss es schon geben
    - Wurde in einer früheren Phase konstruiert
  - Der wird nun wegen S[i+1] nach a geteilt
    - S[i+1] kann es als Fortsetzung nicht geben, sonst hätte es dieselbe Fortsetzung nach xa gegeben, und in der Extension j wäre hier kein neuer Knoten notwendig gewesen - Widerspruch

➢ Also entsteht ein innerer Knoten mit Label a
  ➢ (oder es gab schon einen)

Genauer

- Wir wissen, dass es für jeden inneren Knoten spätestens nach der nächsten Extension einen Zielknoten gibt
  - Dahin hätten wir gerne einen Suffixlink
- Aber wie setzen wir denn den Suffixlink konkret?
  - Wir haben nicht nur spätestens nach dem nächsten Schritt einen Suffixlinkpartner, sondern wir kommen im nächsten Schritt auch daran vorbei
  - Also merken wir uns bei Auftreten von Regel 2 mit neuem inneren Knoten k genau diesen Knoten k
  - Nach jedem Schritt sehen wir nach, ob im letzten Schritt ein neuer Knoten erzeugt wurde (also k) und setzen den Suffixlink (k,k‘), wobei k‘ der im aktuellen Schritt erzeugte innere Knoten sein muss (und den muss es geben – siehe Beweis)

Bisher

- Wir haben nach jeder Phase Suffix-Links für alle inneren Knoten
- Wie verwenden wir die?
- Beachte
  - Jede Extension endet
    - Regel 1 und 2: Auf einem Blatt
    - Regel 3: Auf / am Ende einer Kante

Verwendung der Suffix-Links

- Während einer Phase sucht man nacheinander die Enden von S[1..i], S[2..i], S[3..i] etc.
  - Sprich: xyz..., yz..., z... – genau das Suffix-Link Szenario
- Wenn wir in Schritt j das Ende von S[j..i] gefunden haben und zu Schritt j+1 übergehen
  - Suche den inneren Knoten k über dem Ende von S[j..i]
  - Wenn k die Wurzel ist
    - Matche wie bei naivem Algorithmus (siehe Skip/Count)
  - Sonst ist k ein innerer Knoten
    - Folge dem Suffix-Link von k zu Knoten k'
    - Das Ende von S[j+1..i] muss unter k' liegen (aber wo?)
    - Das Präfix von S[j+1..i] oberhalb von k' müssen wir nicht mehr beachten, sondern nur das Suffix unterhalb von k'
- Anfang in jeder Phase
  - Wir merken uns immer einen Pointer auf Blatt 1
  - Mit dem fängt man immer an – längstes Suffix

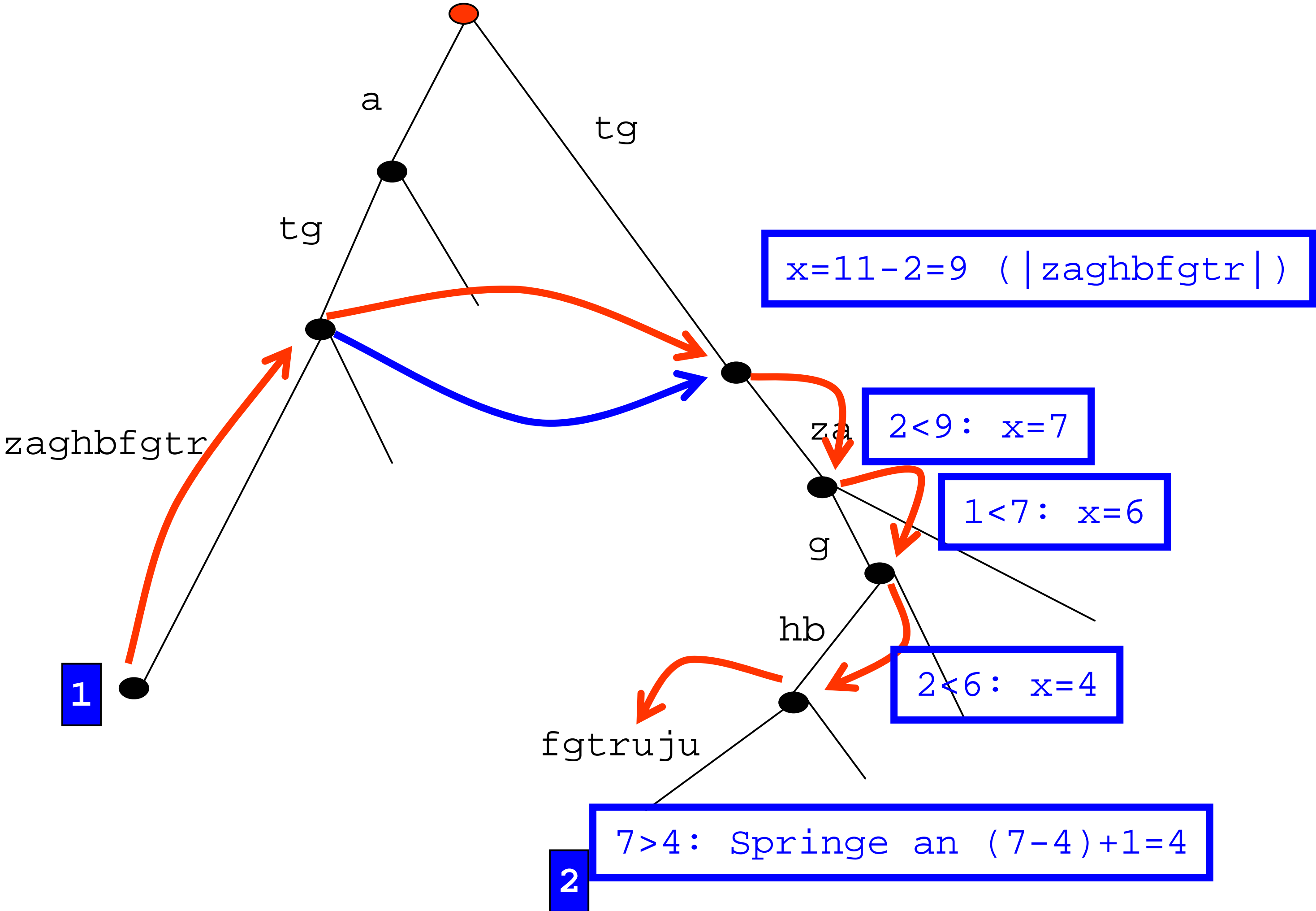
Nutzen bisher?

- Bzgl. Komplexität noch keiner ...
  - Unterhalb von k' müssen wir wieder Zeichen für Zeichen matchen
  - Das ist O(m) pro Extensionsschritt
- Noch ein Trick: Skip/Count
  - Wir kennen die Länge von S[j+1..i]
  - Wir können uns auch die Länge der Kantenlabel merken
    - In konstanter Zeit während des Aufbaus
  - Wir kennen damit die Länge des Präfix oberhalb von k'
  - Wir können damit auch die Länge des Suffix unterhalb von k' berechnen
  - Damit können wir von Knoten zu Knoten hüpfen ...

Details

- Ausgangspunkt: Für Schritt j+1 sind wir einem Suffix-Link zu k' gefolgt
- Sei x die Länge des Pfades von k' zum Ende von S[j+1..i]
  - In konstanter Zeit berechenbar, wenn man sich an jedem inneren Knoten k auch |label(k)| merkt
  - S[j+1..i] muss kein Blatt sein, deshalb müssen wir vorsichtig sein
- S[i-x..i] matched auf dem Pfad unter k (aber bis wohin)
  - Den genauen Pfad kennen wir nicht – seine Länge und sein Label schon
  - Wir müssen nicht mehr Zeichen für Zeichen vergleichen
  - Nur an den Kreuzungen kurz nachsehen
- Wir hüpfen von Knoten zu Knoten
  - Wähle die nächste Kante e von k' (ist eindeutig) zu Knoten k''
  - Wenn |label(e)|<x
    - x = x - |label(e)|; k'=k''
    - Loop
  - Sonst
    - Springe an Position |label(e)|-x+1 im Label von e
    - Bringe dort S[i+1] unter (Extensionsregel 2 oder 3)

Beispiel



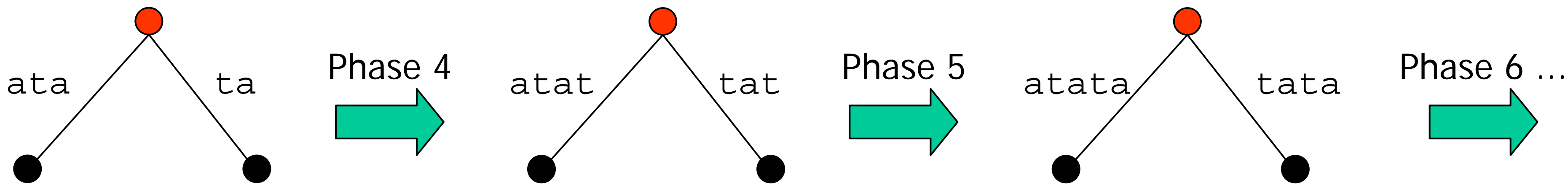
Erste Komplexitätsreduktion

- Theorem
  - *Ukkonen's Algorithmus mit Suffix-Links und Skip/Count braucht pro Phase nur O(m) Laufzeit*
- Beweis
  - Formal: Siehe Gusfield, p. 101-103
  - Idee: Über die Tiefe der Knoten k, k'. Für eine Extension geht man einen Knoten hoch, folgt dem Suffix-Link, und dann 0-n Knoten runter. Man zeigt, dass man dadurch pro Phase jeden Knoten höchstens 3 mal besucht
- Damit sind wir insgesamt bei $O(m^2)$
- Also sind noch ein paar Anstrengungen notwendig

Extensionsregeln – auf den zweiten Blick

- Beobachtung: Was passiert, wenn Regel 3 das erste Mal in einer Phase greift?
  - Wir verlängern ein Suffix S[j..i] um S[i+1]
  - Regel 3: Tue nichts, denn es gibt das Suffix S[j..i+1] schon im Baum (kam schon in einer früheren Phase vor)
  - Dann gibt es auch S[j+1..i+1], S[j+2..i+1], …
  - ➢ Wir können die Phase beenden – in dieser Phase wird nur noch Regel 3 greifen, und die ändert nichts am Baum

- Beispiel: „atatatatatc“, Phase …



Extensionsregel, Abkürzung 2

- Wir unterscheiden
  - Explizite Extension: Schritt mit Anwendung einer Regel
  - Implizite Extension: Schritt nur „gedacht“
    - Alle Schritte nach einer erste Anwendung von Regel 3 sind implizit
- Gesucht
  - Welche Schritte können wir in einer Phase noch sparen?
- Wichtiger und einfacher Trick für das folgende
  - An eine Kante k zu einem Blatt schreiben wir nicht das Label label(k), sondern Indizes p,q mit label(k)=S[p..q]

Extensionsregel, Abkürzung 2

- Beobachtung
  - Es gibt keine Regel zur Umwandlung von Blättern
  - ➢ Blätter bleiben immer Blätter
- Was passiert in den Schritten
  - Regel 1: Verlängerung des Labels einer Blattkante
  - Regel 2: Neues Blatt oder: neuer Knoten und neues Blatt
  - Regel 3: Nichts, Abbruch der Phase
- Jede Phase läuft also ab ala 1,2,2,2,1,2,1,3
  - Erst einige Anwendungen von 1 oder 2, dann einmal 3
  - Sei j' die letzte explizite Extension in Phase i
- Bei jeder Anwendung von 1 oder 2 wird das Label einer Blattkante verlängert oder ein Blatt+Kante geschaffen
  - Alle Schritte bis j' sind nach Phase i durch Blätter repräsentiert
  - ➢ In Phase i+1 verlängern die Schritte 1…j' nur Blätter

Extensionsregel, Abkürzung 2

- Diese Schritte können wir uns schenken
  - Blattkanten erhalten statt (p,q) die Beschriftung (p,E)
  - E steht für „Bis zum Ende“
    - Am Ende wird jede Blattkante bis zum Ende von S gehen (denn Blättern repräsentieren schließlich Suffixe von S)
- Zusammen: Eine Phase im einzelnen
  - Überspringe alle Schritte bis j‘ der letzten Phase
  - Führe Extensionen aus, entweder bis i+1 oder bis zur ersten Anwendung von Regel 3
    - Hier werden neue Blätter / innere Knoten geschaffen
    - Das kann auch eine Blattkante unterbrechen – p anpassen
  - Neues j‘ merken und nächste Phase starten

Algorithmus

```
construct T1;
j` := 0;                        // Points to next ext. in next phase
for i=1 to m-1 do               // m-1 phases
    for j=j`+1 to i+1 do
        find end of S[j..i];    // Using Suffix-Links
        apply rules;
        if (rule 3 applied)     // The show stopper rule
            j := j-1;
            break;              // End phase
        end if;
    end for;
    j` := j;                    // Next start point
end for;
```

Komplexität

- Theorem
  - *Ukkonen's Algorithmus benötigt O(m) zur Konstruktion von T_m*
- Beweisidee
  - Jede Phase führt explizit höchstens einen Extensionsschritt j aus, der schon in einer früheren Phase ausgeführt wurde (das ist j')
  - Alle Phasen zusammen führen also höchstens 2m explizite Extensionsschritte aus
  - Außerdem werden insgesamt höchstens m Knotensprünge ausgeführt

Was bleibt

- Wie gewinnen wir T aus T_m?
  - Führe eine weitere Phase aus mit S[m+1]=$
  - Das markiert alle Suffixe von S
    - Kein Suffix kann mehr Präfix eines anderen Suffix sein
  - Außerdem die Label von Blattkanten ([p,E]) mit echtem Label ersetzen
    - Alle Blattkanten in O(m) finden

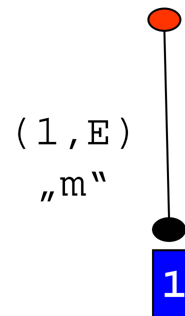
Humboldt Universität Informatik

Ein komplettes Beispiel

```
12345678901
mississippi
```

Phase 1

$j_1' = 1$



Blatt „1" wird nie wieder angefasst

Blattkantenlabel werden im Algorithmus ignoriert

(hier nur zur Verdeutlichung weitergeführt)

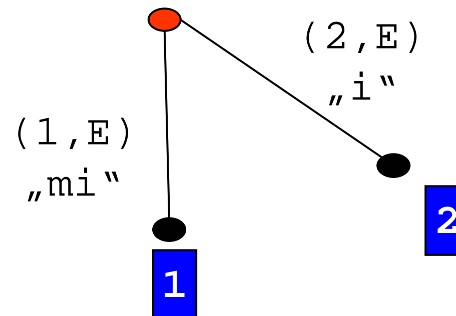
Beispiel 2

```
12345678901
mississippi
```

Phase 2 (=i+1)

- $j_1'=1$
- 2: Rule 2

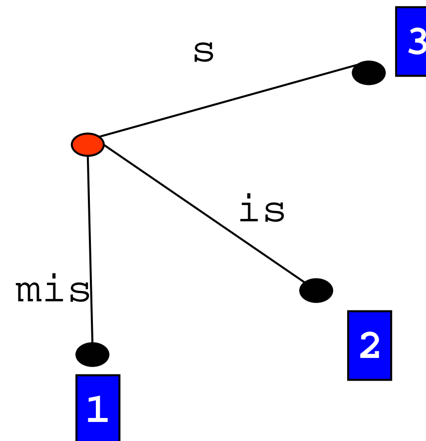
$j_2'=2$



Blatt „2" wird nie wieder angefasst

Beispiel 3

12345678901
mi**s**sissippi



Phase 3

- $j_2'=2$
- 3: Rule 2

$j_3'=3$

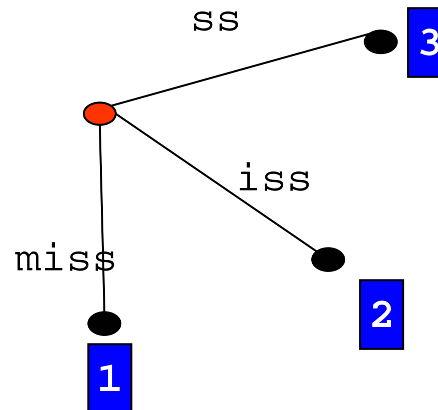
Blatt „3“ wird nie wieder angefasst

Beispiel 4

```
12345678901
mississippi
```

Phase 4
- $j_3' = 3$
- 4: Rule 3

$j_4' = 3$



Phasenabbruch nach $j = 4$

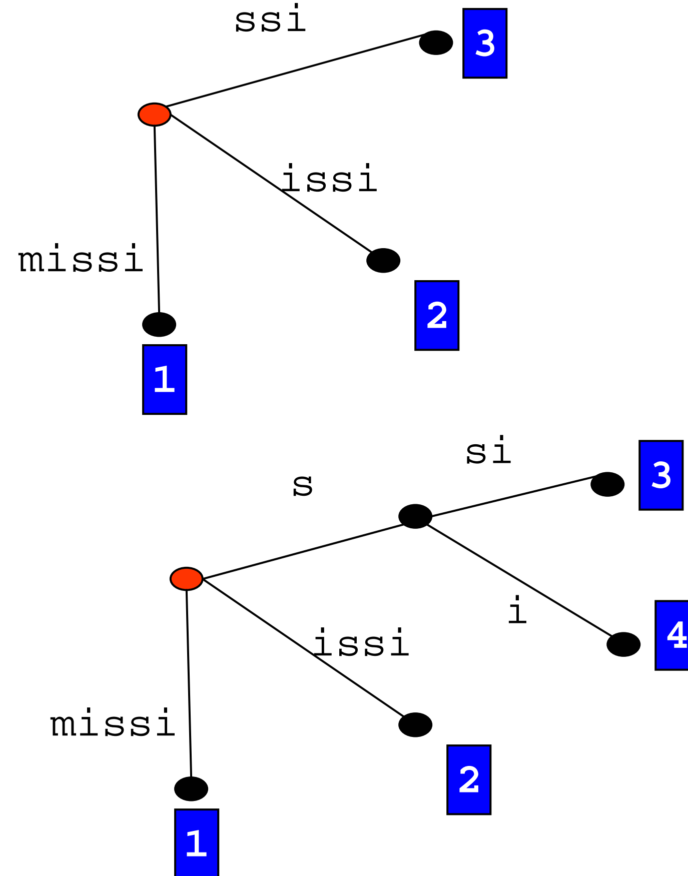
Beispiel 5

```
12345678901
mississippi
```

Phase 5

- $j_4'=3$
- 4 (s+i): Rule 2
- 5 („“+i): Rule 3

$j_5'=4$



j=4 wurde zweimal betrachtet (Phasen 4 und 5)
Phasenabbruch nach j=5

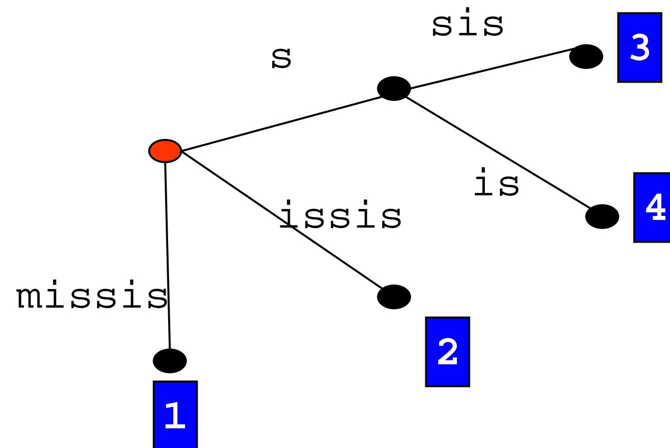
Beispiel 6

```
12345678901
mississippi
```

Phase 6

- $j_5'=4$
- 5 (i+s) : Rule 3

$j_6'=4$



j=5 muss zweimal betrachtet werden (5-6)
Phasenabbruch nach j=5

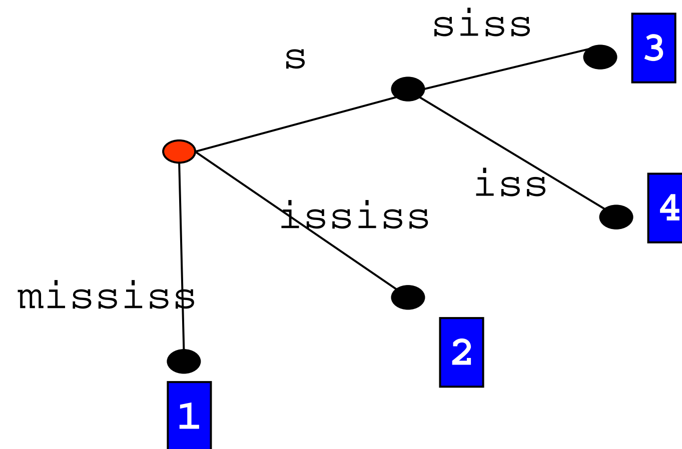
Beispiel 7

```
12345678901
mississippi
```

Phase 7

- $j_6' = 4$
- 5 (is+s) : Rule 3

$j_7' = 4$



J=5 muss dreimal betrachtet werden (5-7)
Phasenabbruch nach j=5

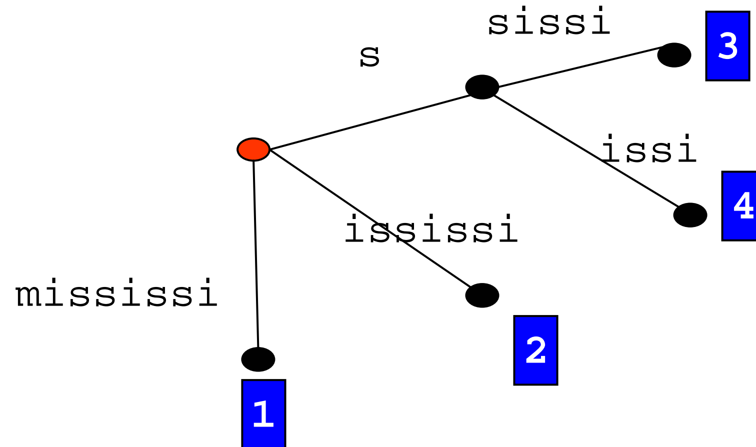
Beispiel 8

```
12345678901
mississippi
```

Phase 8

- $j_7' = 4$
- 5 (iss+i) : Rule 3

$j_8' = 4$



j=5: Phasen 5-8 (Aber bisher alle nur implizit)
Phasenabbruch nach j=5

Beispiel 9

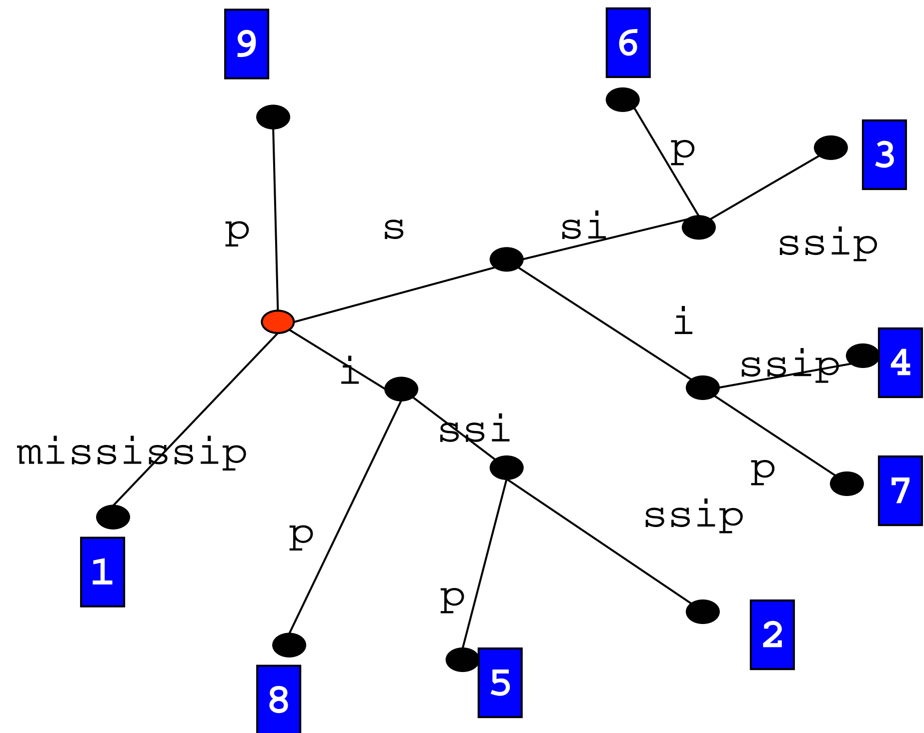
```
12345678901
mississippi
```

Phase 9

- $j_8'=4$
- 5 (issi+p) : Rule 2
- 6 (ssi+p): Rule 2
- 7 (si+p): Rule 2
- 8 (i+p) : Rule 2
- 9 („“+p) : Rule 2

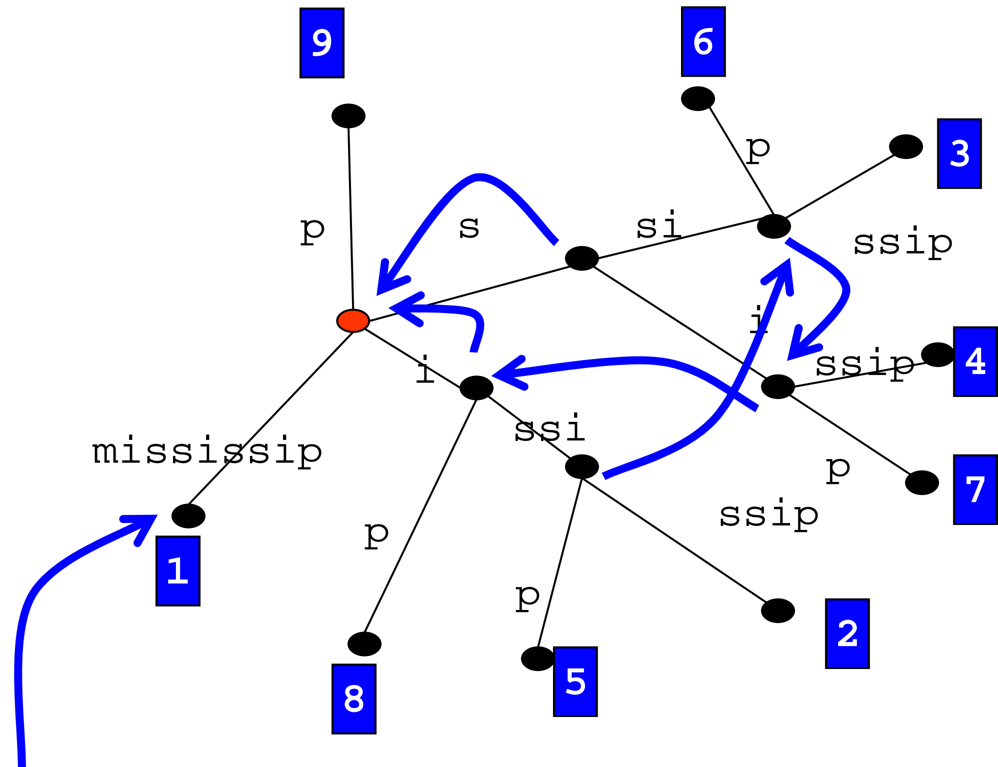
$j_9'=9$

Großer Sprung!



Beispiel 9: Die Suffix-Links

```
12345678901
mississippi
```



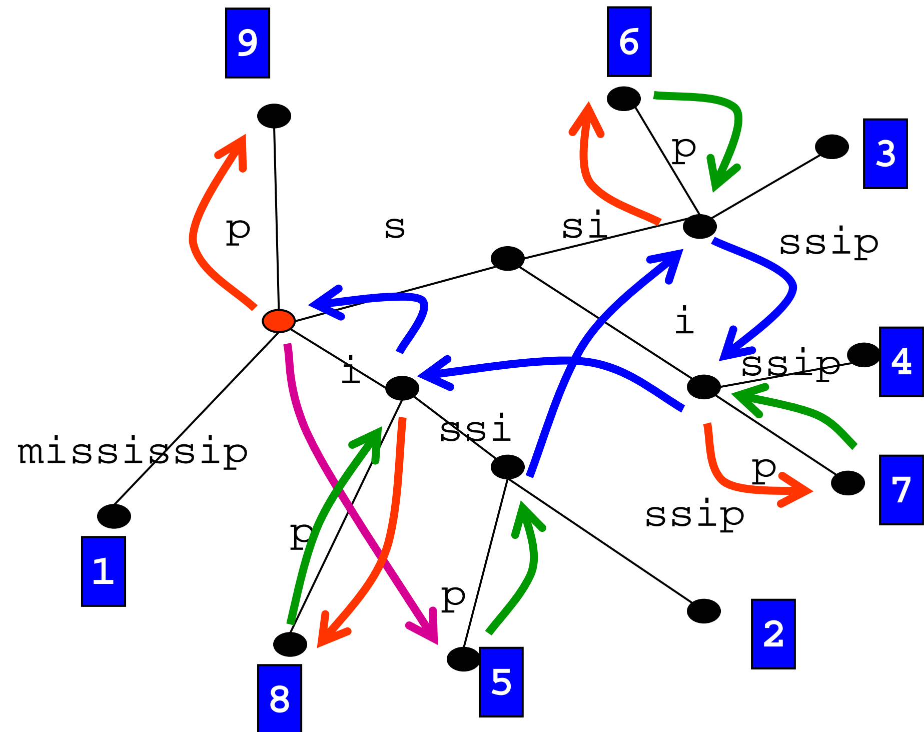
Phase 9

- j_8'=4
- 5 (issi+p) : Rule 2
- 6 (ssi+p): Rule 2
- 7 (si+p): Rule 2
- 8 (i+p) : Rule 2
- 9 („“+p) : Rule 2

Pointer auf Blatt 1 bleibt immer konstant
Dort geht Traversierung in Phase los, die mit Schritt 1 starten
Sonst: Erstes Suffix matchen, ausgehend von Root

Beispiel 9: Die Suffix-Links

12345678901
mississippi



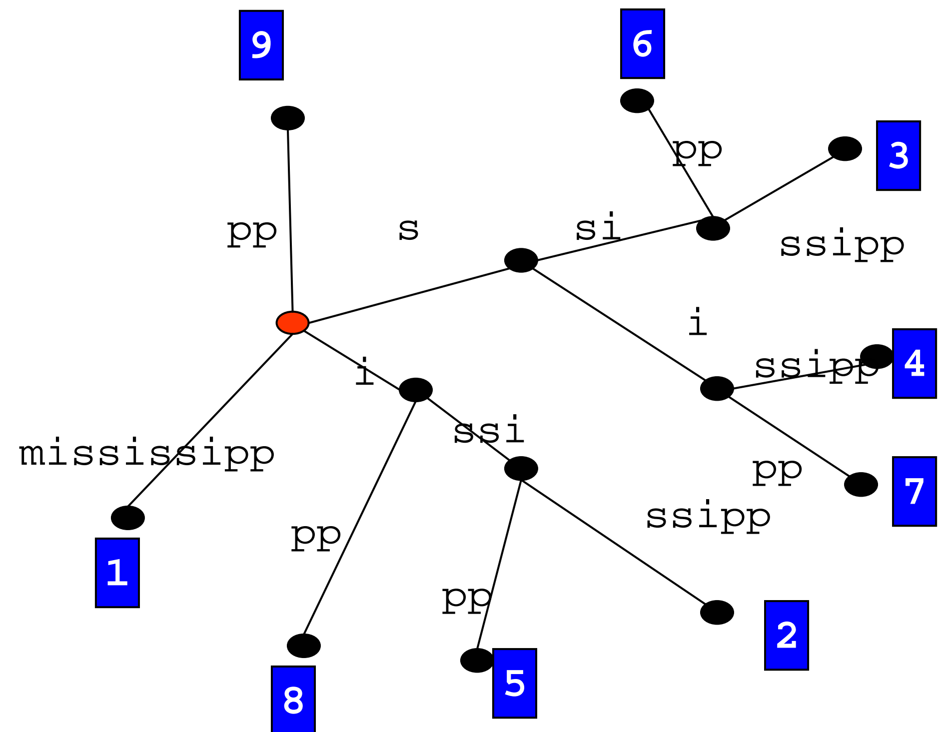
Phase 9

- j_8'=4
- 5 (issi+p) : Rule 2
- 6 (ssi+p): Rule 2
- 7 (si+p): Rule 2
- 8 (i+p) : Rule 2
- 9 („"+p) : Rule 2

- Suffix-Link springen / von Root matchen
- Von Knoten zu Knoten zum Blatt hüpfen
- Zum untersten inneren Knoten zurück
- Nächster Extensionsschritt

Beispiel 10

```
12345678901
mississippi
```



Phase 10

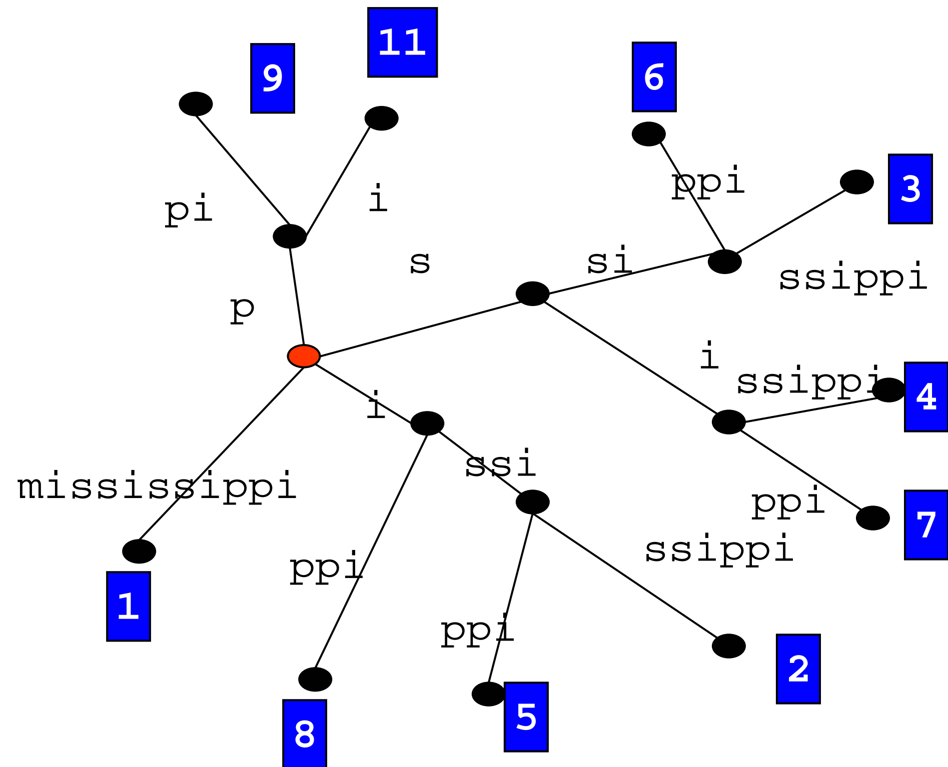
- $j_9'=9$
- 10 („“+p): Rule 3

$j_{10}'=9$

Nichts passiert
(außer implizite Verlängerung der Blattkanten)

Beispiel 11

12345678901
mississippi



Phase 11

- j_9'=9
- 10 (p+i): Rule 2
- 11 („“+i): Rule 3

T_{11} ist fertig
Nicht alle Suffixe sind als Blätter vertreten

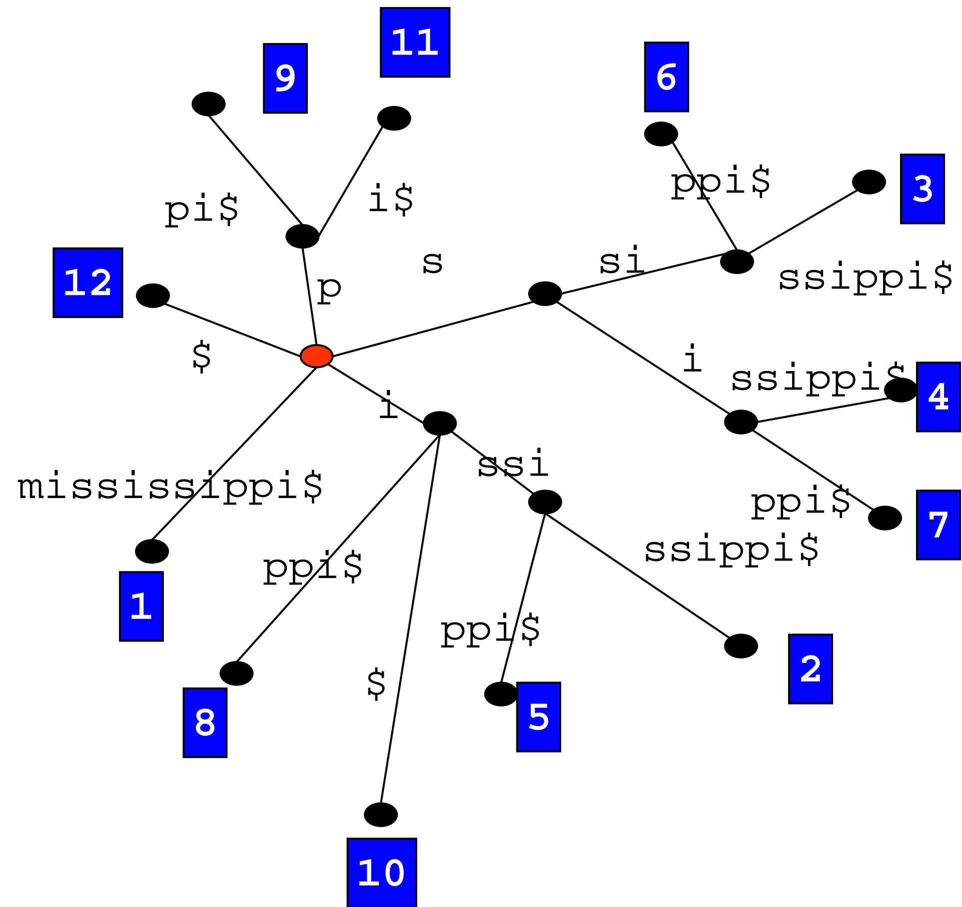
Transformation $T_{11} \rightarrow T$

12345678901
mississippi$

Neuer Durchlauf

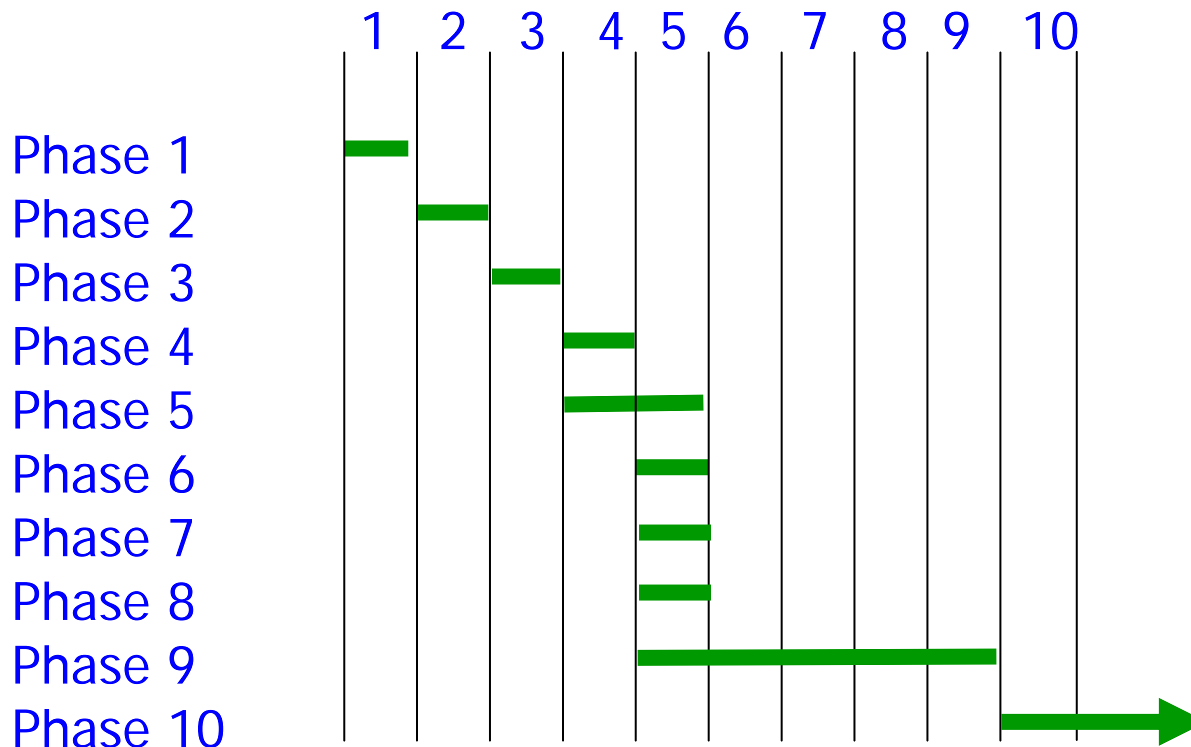
- 1-10: Regel 1
- 11: Regel 2

Fertig



Komplexität

- Welche Schritte haben wir ausgeführt?



- Schritte markieren einen Pfad von links oben nach rechts unten
- Das können zusammen höchstens 2*m Schritte sein

Zusammenfassung

- Algorithmus verläuft konzeptionell in Phasen und Extensionen
- Aber praktisch alle Extensionsschritte können implizit ausgeführt werden
- Nur linear viele Extensionsschritte führen tatsächlich zu Änderungen in der Baumstruktur
- Navigation wird durch Suffix-Links beschleunigt

Aber ...

- Konstruktion sehr schlecht auf Sekundärspeichern
  - Man braucht den ganzen Baum im Hauptspeicher (durch Suffix-Links)
  - Eingrenzung der Verwendbarkeit (keine Genom–Genom Vergleiche)
- Gleichzeitig sehr speicherintensiv
  - Viele Pointer, Kantenrepräsentation
  - Beste Implementierungen brauchen ca. 15 Byte/Zeichen
- Abhilfe
  - Suffixarrays – deutlich geringerer Speicherverbrauch
  - Konstruktion ohne Suffix-Links – bessere Lokalität