

Algorithmische Bioinformatik

Ukkonen's Algorithmus:
Konstruktion von Suffixbäumen
in linearer Zeit



Ulf Leser
Wissensmanagement in der
Bioinformatik



Inhalt dieser Vorlesung

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

Überblick

- High-Level: Phasen und Extensionen
- Verbesserungen
 - Suffix-Links
 - Skip/Count Trick
 - Noch zwei Tricks
- Gesamtkomplexität
- Beispiel

Voraussetzungen

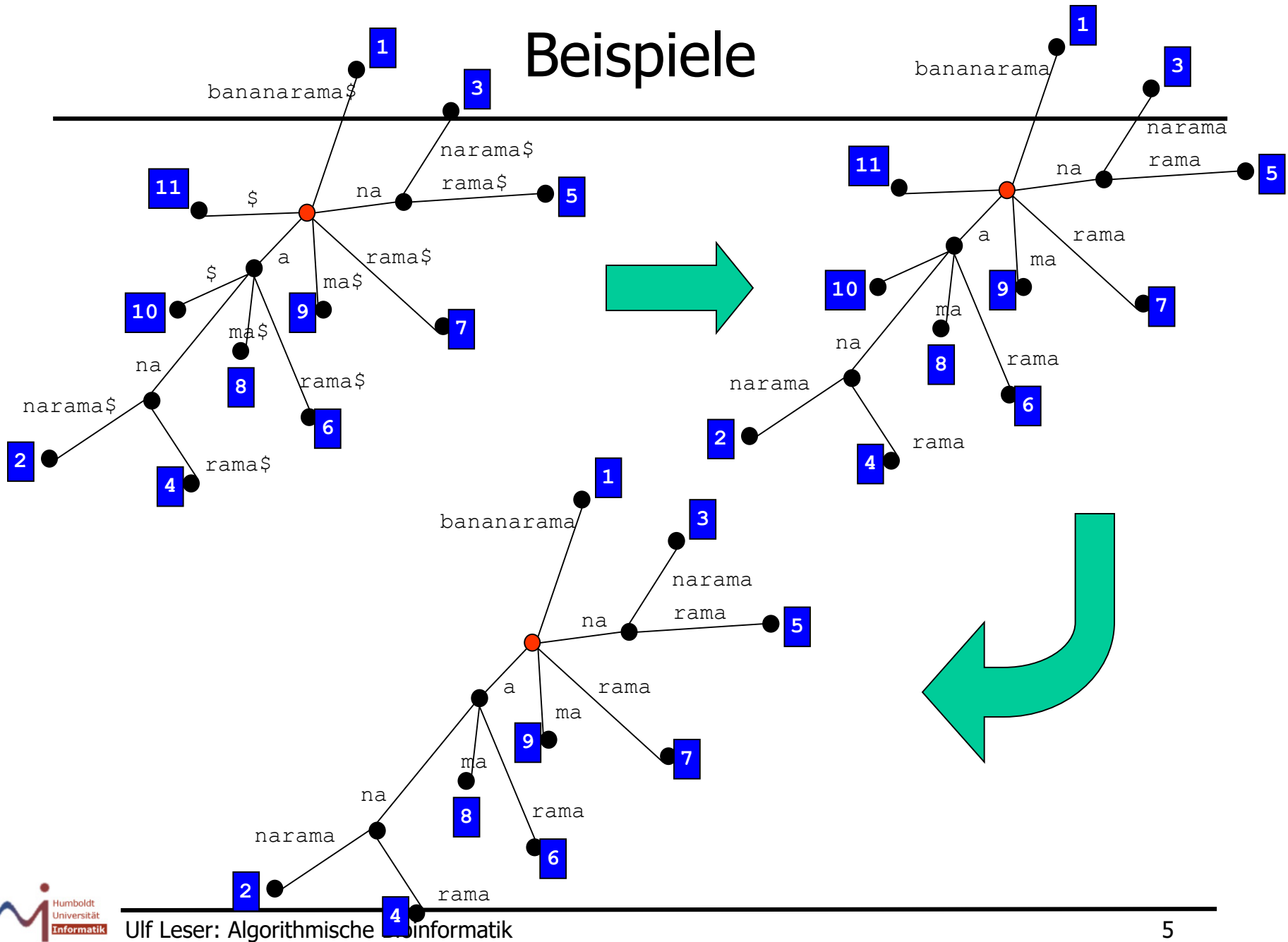
- Definition

*Sei T ein Suffixbaum für S . Der **implizite Suffixbaum T'** entsteht aus T durch*

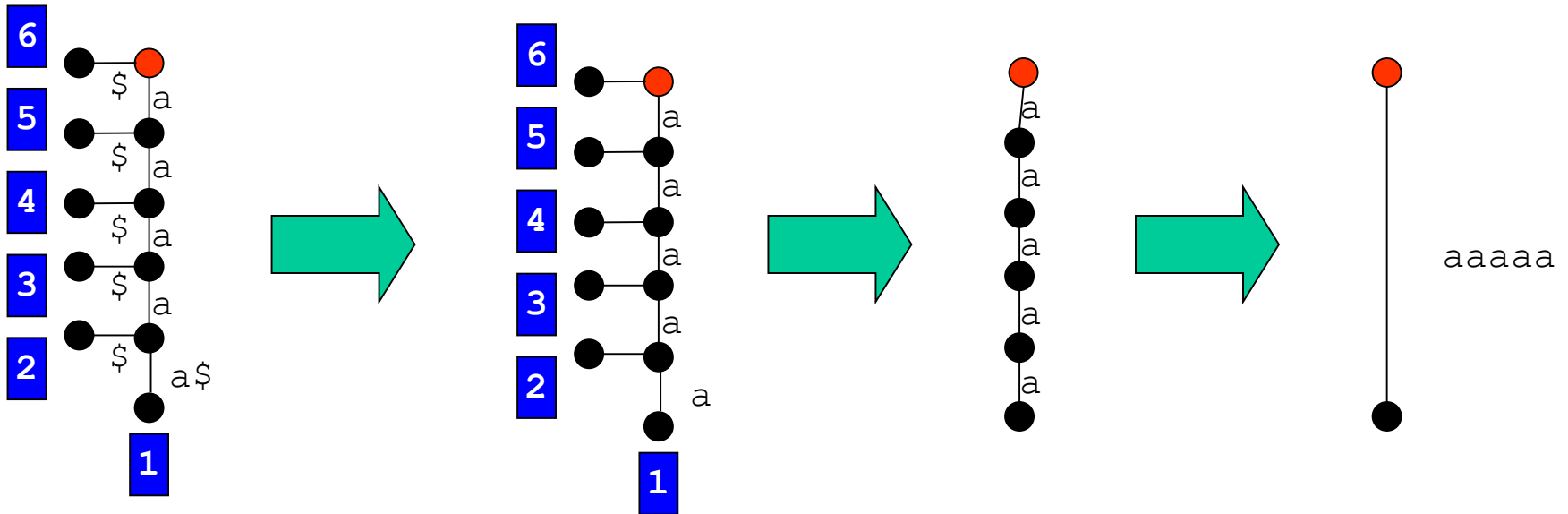
- *Entferne alle Vorkommen von „\$“ aus allen Labels*
- *Entferne alle Kanten ohne Label*
 - *Die Enden von Suffixen, die Präfix eines anderen Suffix sind*
- *Entferne alle inneren Knoten mit weniger als 2 Kindern;
konkatiniere die Label der eingehenden und der ausgehenden
Kante zu dem der neuen (durchgehenden) Kante*

- Das ist die Definition; wir werden anders vorgehen

Beispiele



Beispiel



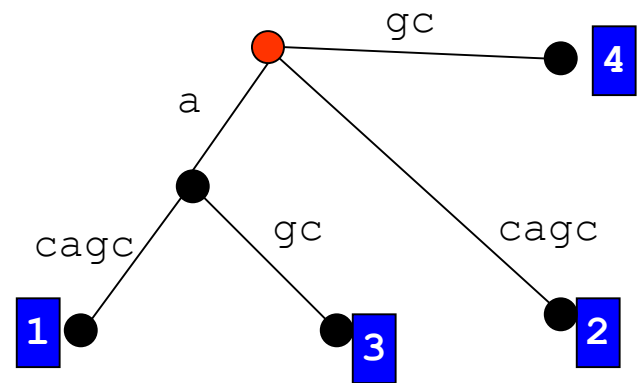
- Implizite Suffixbäume kodieren immer noch alle Suffixe
 - Jedes Suffix matched entlang eines Pfads
 - Aber man kann die Enden der Suffixe nicht mehr erkennen
 - Nicht für alle Suffixe gibt es eine Markierung an einem Blatt

Grundaufbau Ukkonen's Algorithmus

- Konstruktion impliziter SBe für wachsende Präfixe von S
 - 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 $S[i+1]$
 - Letzter Schritt verlängert das **leere Suffix** (Einfügen von $S[i+1]$)
 - Reihenfolge der Schritte: von links nach rechts ($S[1..i]$, $S[2..i]$, ...)
- Drei **Extensionsregeln**

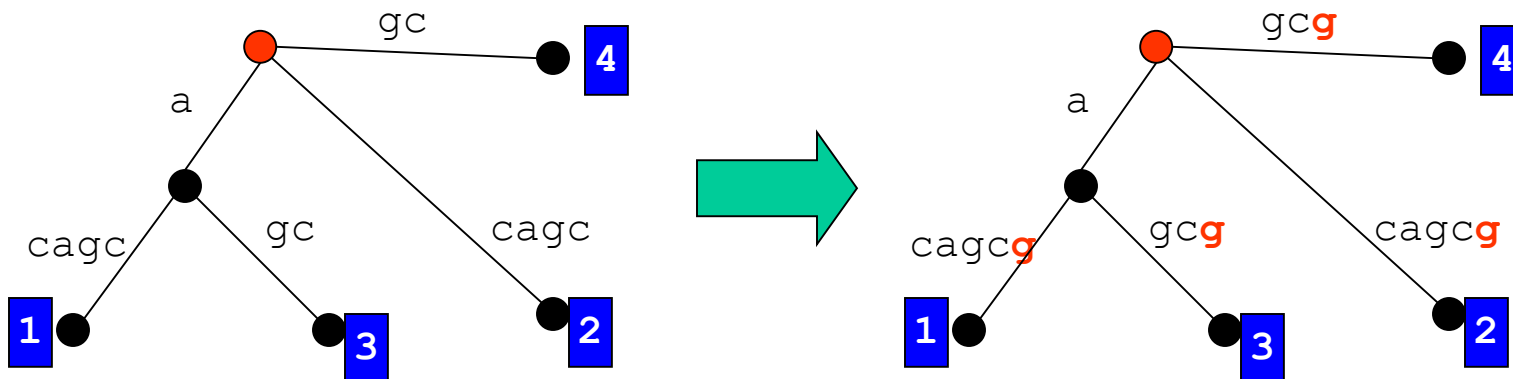
Extensionsregeln

- 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)
 - 3 mögliche Situationen können entstehen
 - Beispiel
 - $S=„acagcg“$
 - Wir haben T_5 und bauen T_6
 - Also: Alle Suffixe von „acagc“ um $S[6]=„g“$ erweitern
-
- ```
graph TD; Root(()) ---|a| NodeA(()); Root ---|gc| NodeGC(()); Root ---|cagc| NodeCAGC(()); NodeA ---|cagc| NodeACAGC(()); NodeA ---|gc| NodeAGC(()); NodeGC ---|cagc| NodeGCGC(()); NodeCAGC ---|g| NodeCAGCG(());
```



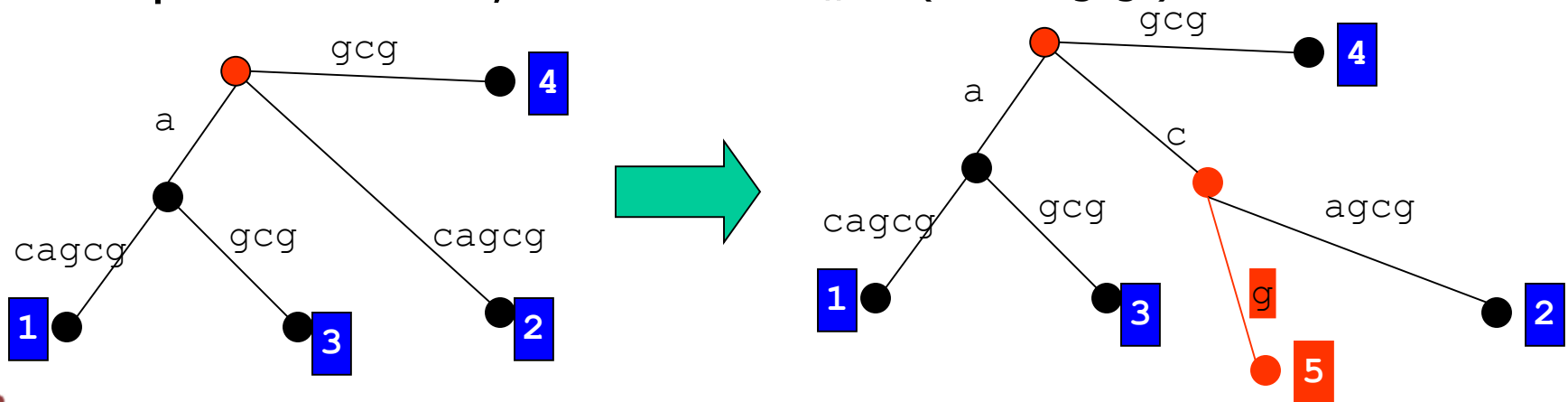
# Extensionsregel 1

- 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“]



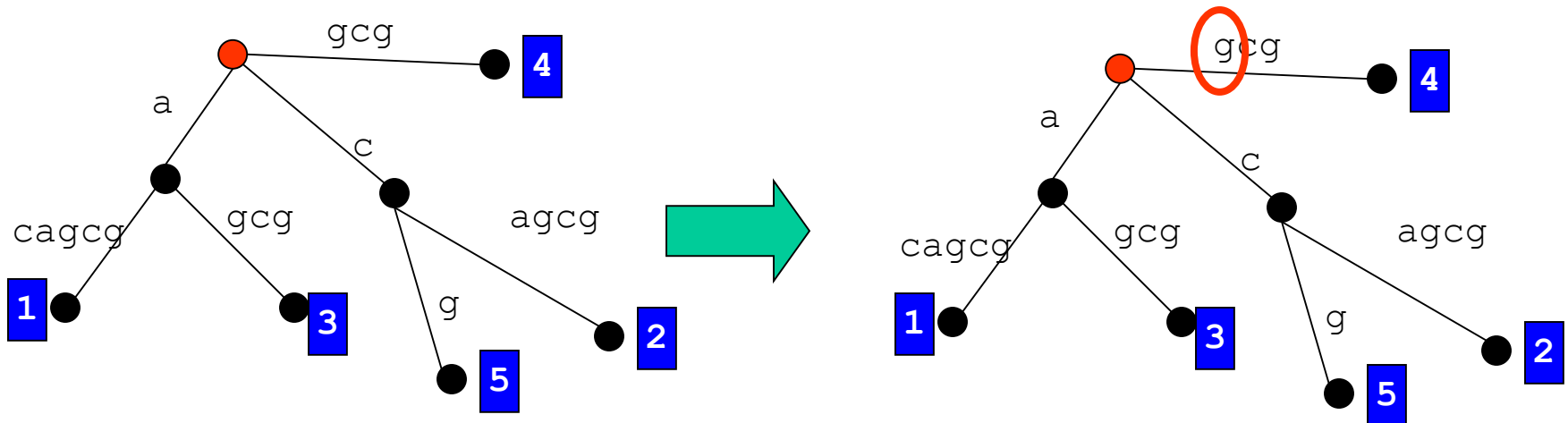
# Extensionsregel 2

- 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, altes Suffix „c“ ( $S = \text{„acagcg“}$ )



# 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, altes Suffix „" ( $S=„acagcg”$ )



# Komplettes Beispiel

---

- Konstruiere implizite Suffixbäume  $T_1 \dots T_6$  für „gtcgtg“
- ...

# Algorithmus und Komplexität

---

```
construct T_1 ;
for i=1 to m-1 // m-1 phases
 $T_{i+1} := T_i$;
 for j = 1 to i+1 // i+1 extensions, left-right
 b := S[j..i];
 match b in T_{i+1} ;
 extend T_{i+1} with S[i+1]; // Using one of the 3 rules
 end for;
end for;
```

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

# Überblick

---

- High-Level: Phasen und Extensionen
- Verbesserungen
  - Suffix-Links
  - Skip/Count Trick
  - Noch zwei Tricks
- Gesamtkomplexität
- Beispiel

# Suffix-Links

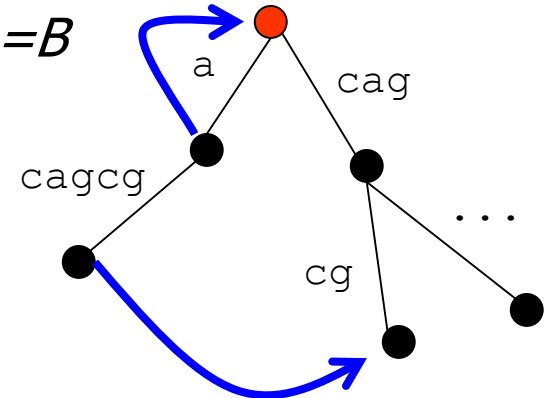
---

- 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 einer Phase auf **zusammen  $O(m)$**
- Intuition
  - Für jedes  $S[j..i]$  gibt es per Konstruktion irgendwo schon  $S[j+1..i]$ , das wir als nächstes verlängern wollen
  - Wir wollen das nicht suchen, sondern **Suffix-Links** merken und in jeder Extension direkt springen statt immer b ab Root zu matchen
  - 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  $\text{label}(k) = xB$ , wobei  $|x| = 1$ ,  $|B|$  beliebig (auch 0)
- Sei  $k'$  ein innerer Knoten von  $T$  mit  $\text{label}(k') = B$ 
  - Wenn  $|B| = 0$ :  $k' = \text{Wurzel}$
- Der Pointer  $(k, k')$  heißt **Suffix-Link**

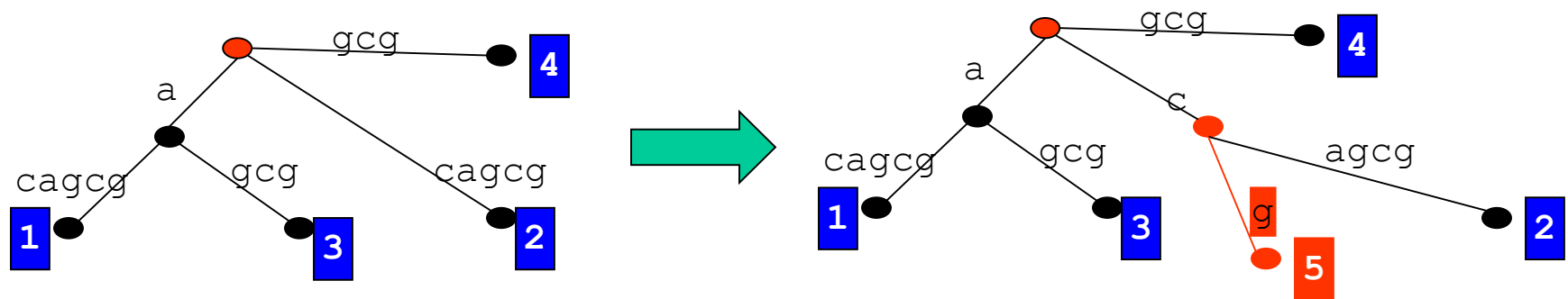


- Wir zeigen, dass **jeder innere Knoten** in  $T$  nach jeder Phase von Ukkonen's Algorithmus ein Suffix-Link-Ziel hat
  - Nicht offensichtlich:  $B$  muss zwar existieren, aber nicht in einem Knoten enden
- Wir müssen die Links dann noch bauen
- Entlang dieser Links springen wir in späteren Phasen

# 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, ... (hier uninteressant)
  - In einer Kante: **Neuer innerer Knoten** teilt diese Kante sowie neues Blatt wie oben



# Suffix-Links in impliziten Suffixbäumen

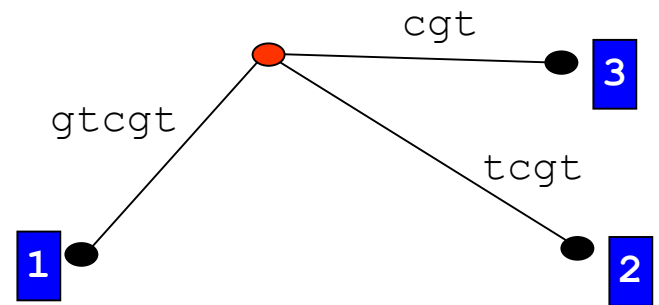
---

- Theorem
  - *In Ukkonen's Algorithmus hat jeder innere Knoten **spätestens nach dem Ende des nächsten Extensionsschritts** ein Suffix-Link-Ziel*
- Lemma
  - *Nach der letzten Extension in einer Phase von Ukkonen's Algorithmus hat **jeder innere Knoten** ein Suffix-Link-Ziel*
  - *Nach jeder Phase von Ukkonen's Algorithmus hat **jeder innere Knoten** ein Suffix-Link-Ziel*
- Das Lemma 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 Beweis)

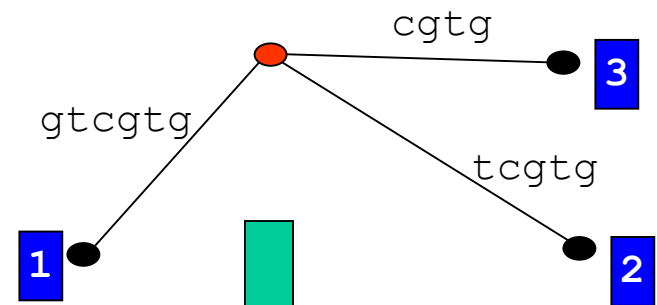
# Beispiel

- „gtcgtg“ in Phase 6

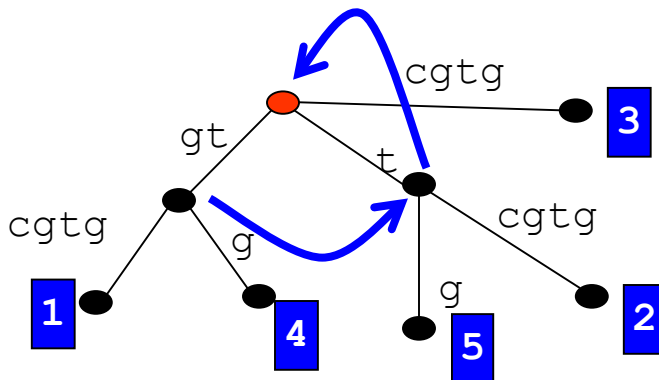
- Beobachtung: „Jeder“ innere Knoten hat nach Phase 5 ein Suffix-Link-Ziel (es gibt hier keinen)
- Wir erweitern in jeder Extension dieser Phase mit  $S[6]=\text{„g“}$



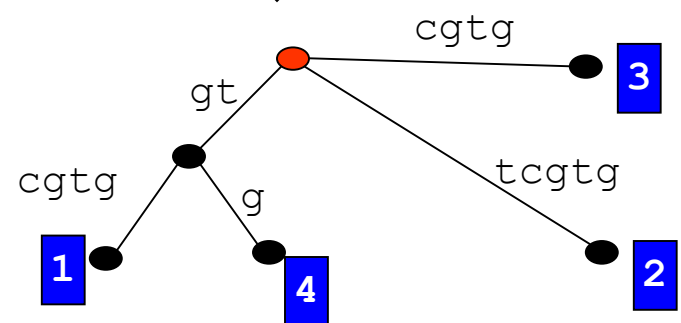
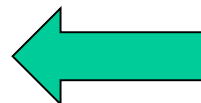
Ext. 1-3

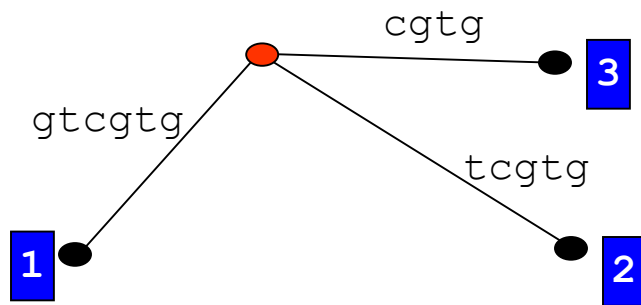


Ext. 4: gt+g

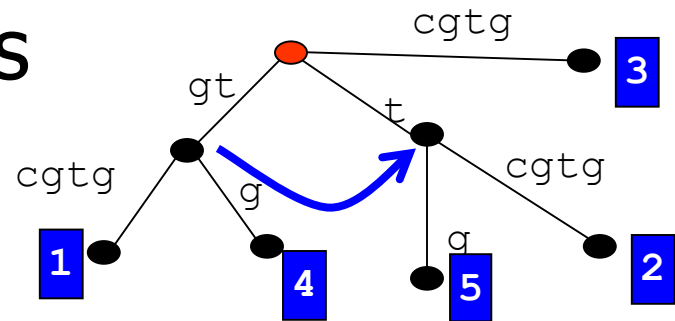


Ext. 5: t+g





## Beweis



- Angenommen, im Extensionsschritt  $j$  der Phase  $i+1$  fügen wir einen neuen inneren Knoten ein (der braucht ein SL-Ziel)
    - Der habe Label  $xB$  mit  $|x|=1$ ,  $|B|$  beliebig
    - Nach  $xB$  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  $z$ 
      - Sonst hätten wir nur das Label eines Blattes verlängert
  - Was passiert dann im nächsten Extensionsschritt  $j+1$ ?
    - Den Pfad  $Bz$  muss es schon aus einer früheren Phase geben
    - Der wird nun spätestens wegen  $S[i+1]$  nach  $B$  geteilt
      - $S[i+1]$  kann es als Fortsetzung nicht geben, sonst hätte es dieselbe Fortsetzung nach  $xB$  gegeben, und in der Extension  $j$  wäre hier kein neuer Knoten notwendig gewesen - Widerspruch
- Also entsteht ein innerer Knoten mit Label  $B$  – unser Ziel
- (oder es gab schon einen)

# Wie bauen wir Suffix-Links?

---

- Wir wissen, dass es für jeden inneren Knoten spätestens nach der nächsten Extension einen Zielknoten gibt
- Aber **wie berechnen wir den Suffix-Link** konkret?
  - Wir haben nicht nur nach dem nächsten Schritt ein Suffix-Link Ziel, 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 Suffix-Link  $(k, k')$ , wobei  $k'$  der im aktuellen Schritt erzeugte oder als letztes besuchte innere Knoten sein muss

# Verwendung der Suffix-Links

---

- In Phase  $i$  sucht man die Enden von  $S[1..i]$ ,  $S[2..i]$ , 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 tiefsten **inneren Knoten  $k$**  über dem Ende von  $S[j..i]$
  - Wenn  $k$  Wurzel ist: Matche einen Ast herunter (Wie? später)
  - 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**
- Anfang jeder Phase
  - Wir merken uns immer einen Zeiger auf Blatt 1
  - Mit dem fängt man **in jeder Phase** an – längstes Suffix

# Nutzen bisher?

---

- Bzgl. Komplexität noch keiner
  - Unterhalb von  $k'$  müssen wir Zeichen für Zeichen matchen
  - Das ist im Worst-Case immer noch  $O(m)$  pro Extensionsschritt
- Noch ein Trick: **Skip/Count**
  - Wir kennen die Länge von  $S[j+1..i]$
  - Wir wissen in  $O(1)$ , welchen ausgehenden Ast wir vom Zielknoten betreten müssen
  - Wir können uns auch die Länge der Kantenlabel merken
    - In konstanter Zeit während des Aufbaus
  - Wir kennen die Länge des Präfix oberhalb von  $k'$  (Tiefe)
  - Wir kennen damit auch die Länge des Suffix unterhalb von  $k'$
  - Damit können wir von **Knoten zu Knoten hüpfen**

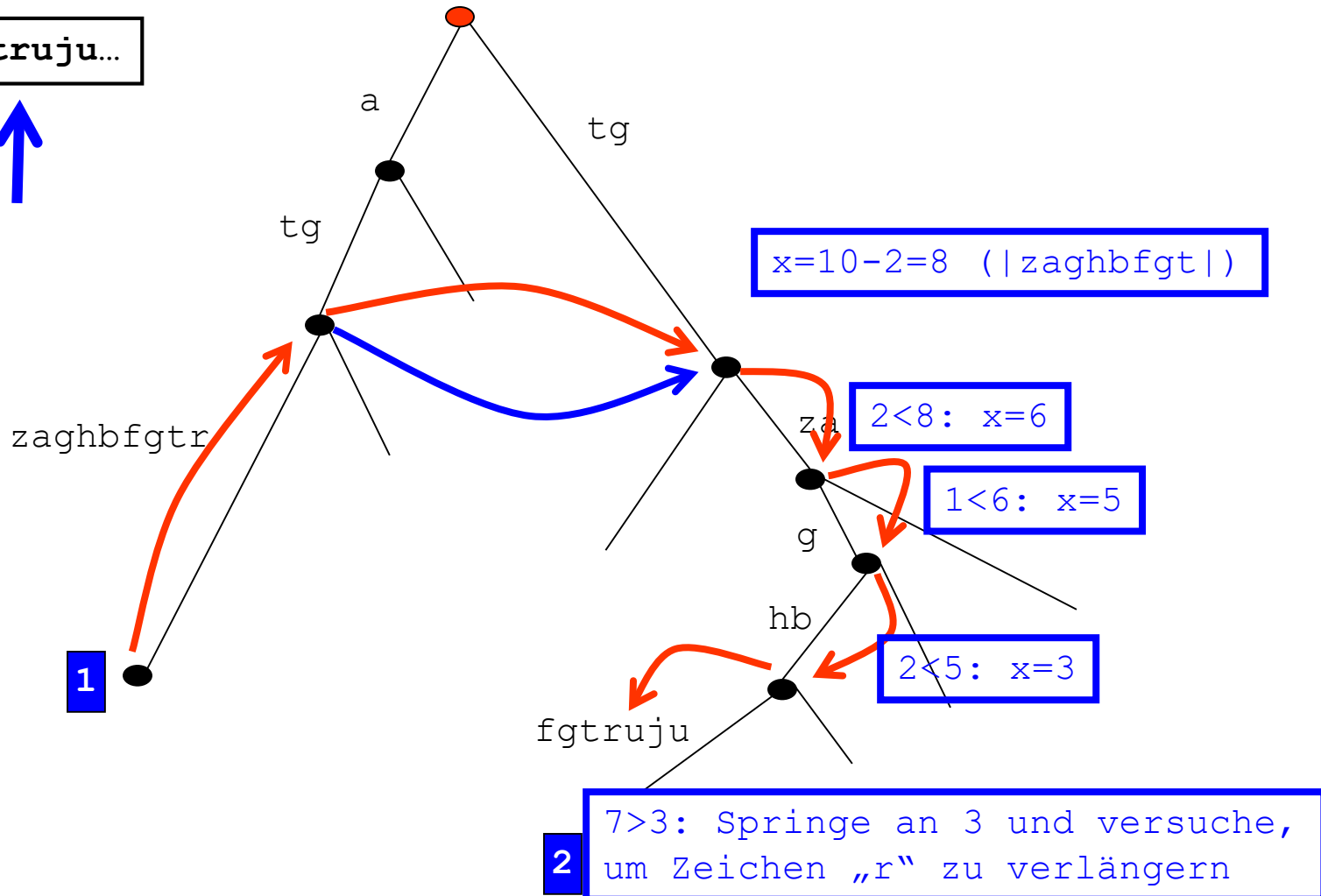
# Details

---

- Für Schritt  $j+1$  seien wir einem Suffix-Link zu  $k'$  gefolgt
- Sei  $x$  die Länge des Pfades von  $k'$  zum Ende von  $S[j+1..i]$ 
  - $S[j+1..i]$  muss kein Blatt sein, deshalb müssen wir vorsichtig sein
- $S[i-x..i]$  matched auf einem Pfad unter  $k'$ 
  - Den genauen Pfad kennen wir nicht, Länge und Label schon
- Wir hüpfen von Knoten zu Knoten
  - Wähle die nächste Kante  $e$  von  $k'$  (ist eindeutig) zu Knoten  $k''$
  - Wenn  $|\text{label}(e)| < x$ :  $x = x - |\text{label}(e)|$ ;  $k' = k''$ ; loop
  - Sonst
    - Springe an Position  $|\text{label}(e)| - x + 1$  im Label von  $e$
    - Bringe dort  $S[i+1]$  unter (Extensionsregel 2 oder 3)

# Beispiel

atgzaghbfgtruju...



# 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 in  $O(1)$ , folgt dem Suffix-Link in  $O(1)$ , 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

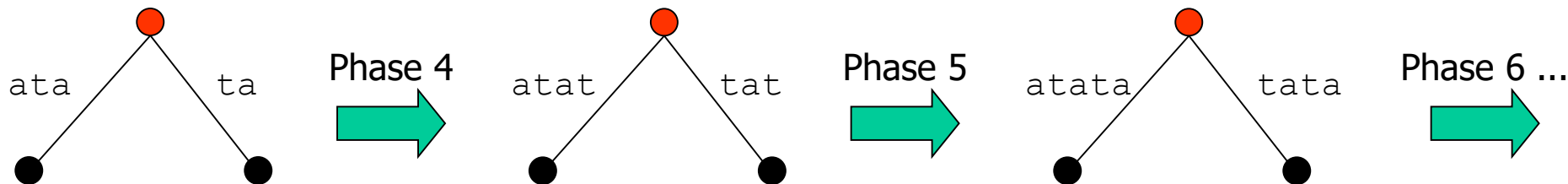
# Überblick

---

- High-Level: Phasen und Extensionen
- Verbesserungen
  - Suffix-Links
  - Skip/Count Trick
  - Noch zwei Tricks
- Gesamtkomplexität
- Beispiel

# Extensionsregeln – auf den zweiten Blick

- Was passiert, wenn Regel 3 **das 1. 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 ...



Stop nach Extension 3

Stop nach Extension 3

# Extensionsregel, Abkürzung 2

---

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

# Extensionsregel, Abkürzung 2

---

- Beobachtung: Keine Regel kann Blätter in irgendwas umwandeln: **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
- In jede Phase greifen also erst Regeln 1 und 2 und dann 3
  - Bei jeder Anwendung von 1 oder 2 wird das Label einer Blattkante verlängert oder Blatt+Kante geschaffen
  - Sei  $j'$  die **letzte Extension** mit Regel 1 oder 2
  - 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**
  - Extensionen bis  $j'$  werden ab Phase  $i+1$  implizit erledigt
  - Wir müssen lediglich Blattkanten anpassen
  - Dazu erhalten Blattkanten statt  $(p,q)$  die Beschriftung  $(p,E)$ 
    - E steht für „Bis zum Ende“ ( $=|S|$ )
    - Am Ende wird jede Blattkante bis zum Ende von S gehen
- 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
    - Kann auch eine Blattkante unterbrechen – p in den Labeln anpassen
  - Neues  $j'$  merken und **nächste Phase** starten

# Algorithmus

---

```
construct T_1 ;
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;
```

# Überblick

---

- High-Level: Phasen und Extensionen
- Verbesserungen
  - Suffix-Links
  - Skip/Count Trick
  - Noch zwei Tricks
- Gesamtkomplexität
- Beispiel

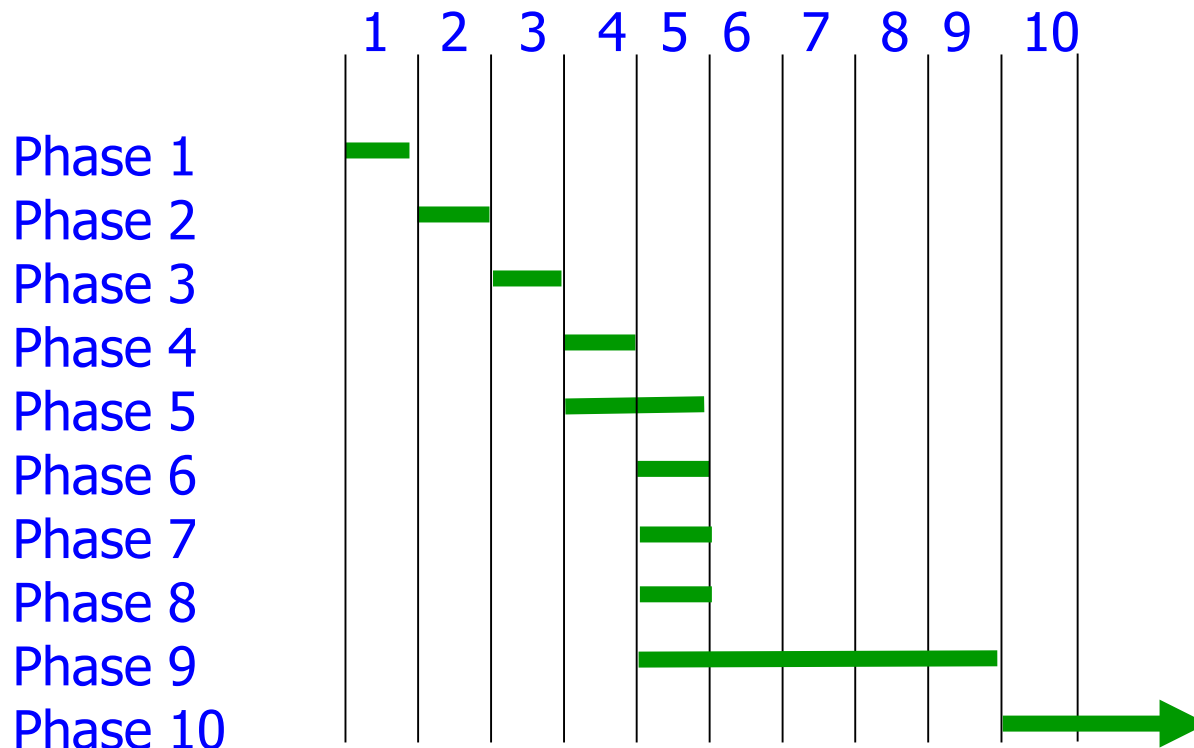
# Komplexität

---

- Theorem
  - *Ukkonen's Algorithmus benötigt  $O(m)$  zur Konstruktion von  $T_m$*
- Beweisidee
  - Jede Phase führt **explizit höchstens einen Extensionsschritt** aus, der schon in einer früheren Phase ausgeführt wurde
    - Das war der erste Schritt mit Regel 3 aus der letzten Phase
  - Danach kommt in der Phase entweder
    - ein weiterer expliziter Schritt
      - Der ist keine Wiederholung und wird auch nicht wieder wiederholt
    - oder es greift Extensionsregel 3 und die Phase ist beendet
  - Es gibt  $m$  Phasen
  - Alle Phasen zusammen führen also **höchstens  $2m$  explizite** Extensionsschritte aus
  - Außerdem werden insgesamt nur  $O(m)$  Knotensprünge ausgeführt
    - Was wir hier nicht zeigen

# Komplexität

- Welche Schritte haben wir ausgeführt?



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

# 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

# Ein komplettes Beispiel

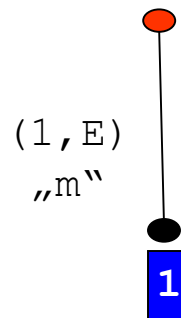
---

12345678901

mississippi

Phase 1

$j_1' = 1$



Blatt „1“ wird nie wieder angefasst  
Blattkantenlabel werden im Algorithmus ignoriert  
(hier nur zur Verdeutlichung weitergeführt)

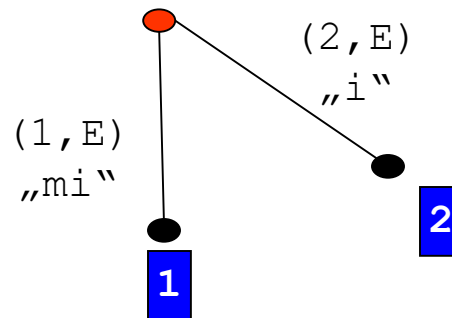
# Beispiel 2

12345678901  
mississippi

Phase 2 ( $=i+1$ )

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

$j_2' = 2$



Blatt „2“ wird nie wieder angefasst

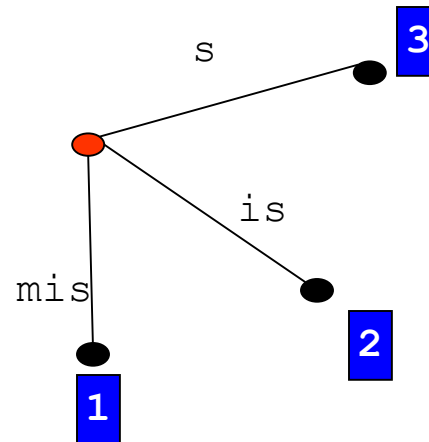
# Beispiel 3

12345678901  
mississippi

Phase 3

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

$j_3' = 3$



Blatt „3“ wird nie wieder angefasst

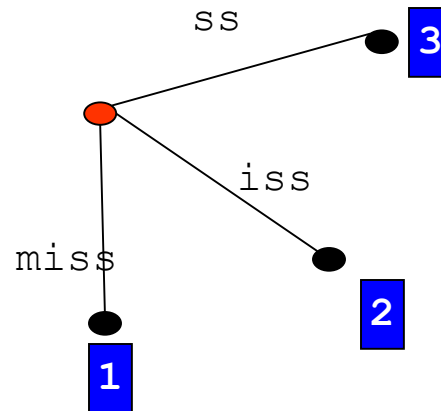
# Beispiel 4

12345678901  
mississippi

Phase 4

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

$j_4' = 3$



Phasenabbruch nach  $j=4$

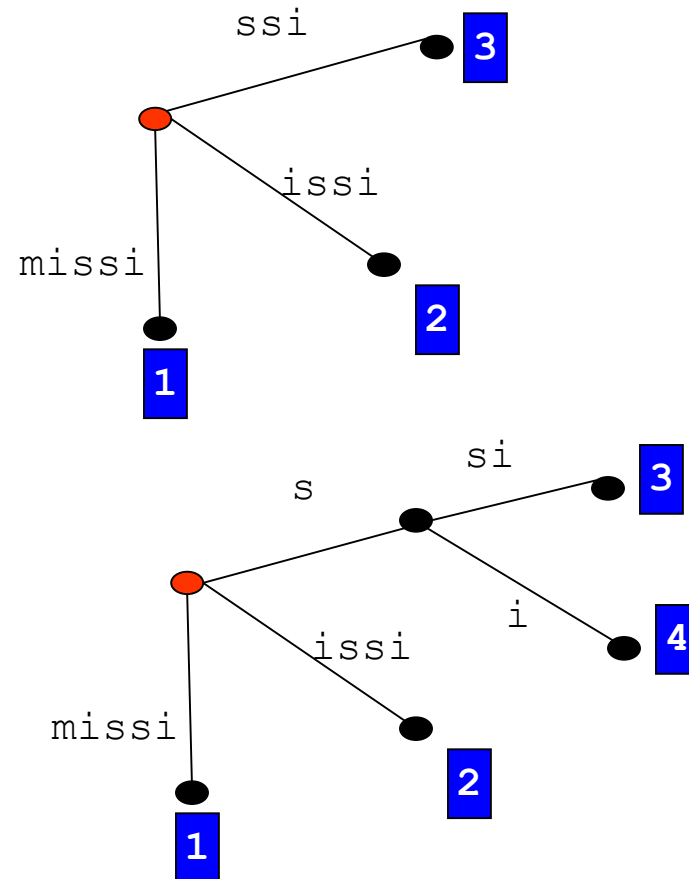
# Beispiel 5

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  $j=5$

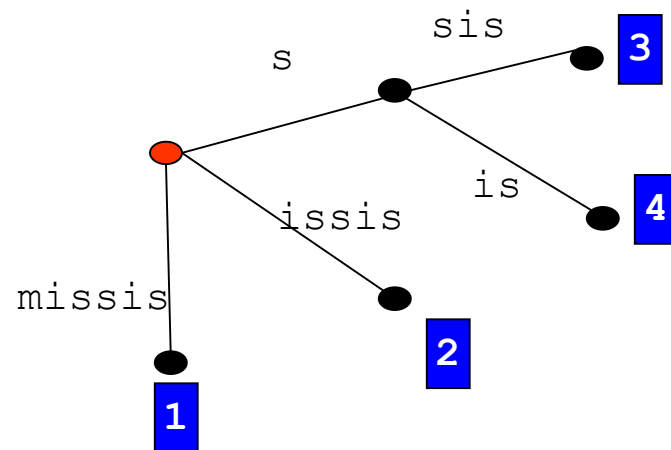
# Beispiel 6

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  $j=5$

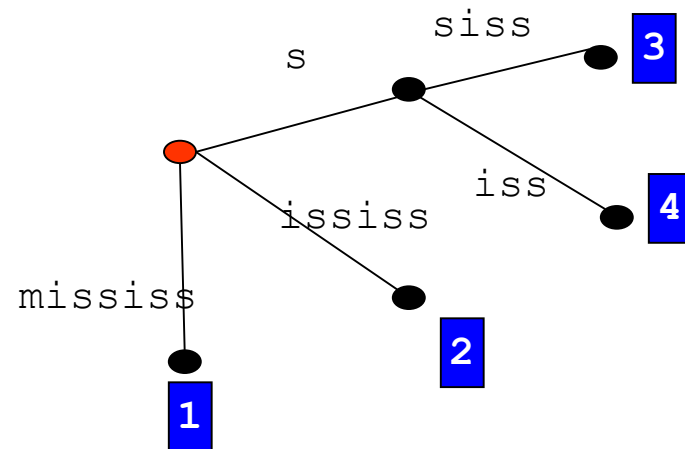
# Beispiel 7

12345678901  
mississ**s**ippi

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

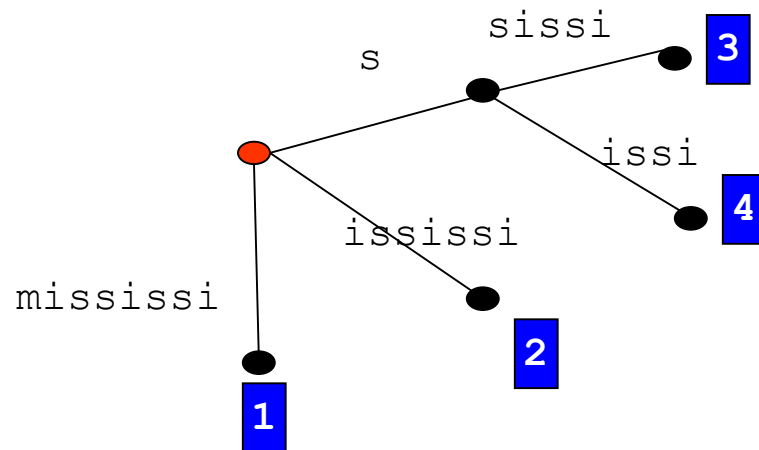
# Beispiel 8

12345678901  
mississ**i**ppi

## 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  $j=5$

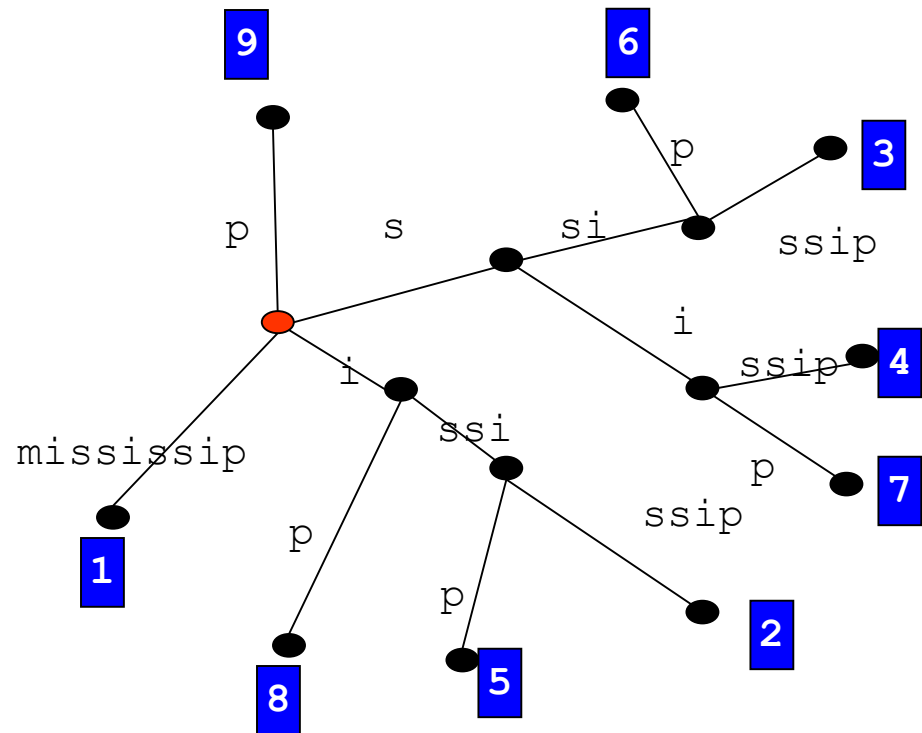
12345678901  
mississippip

## 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!

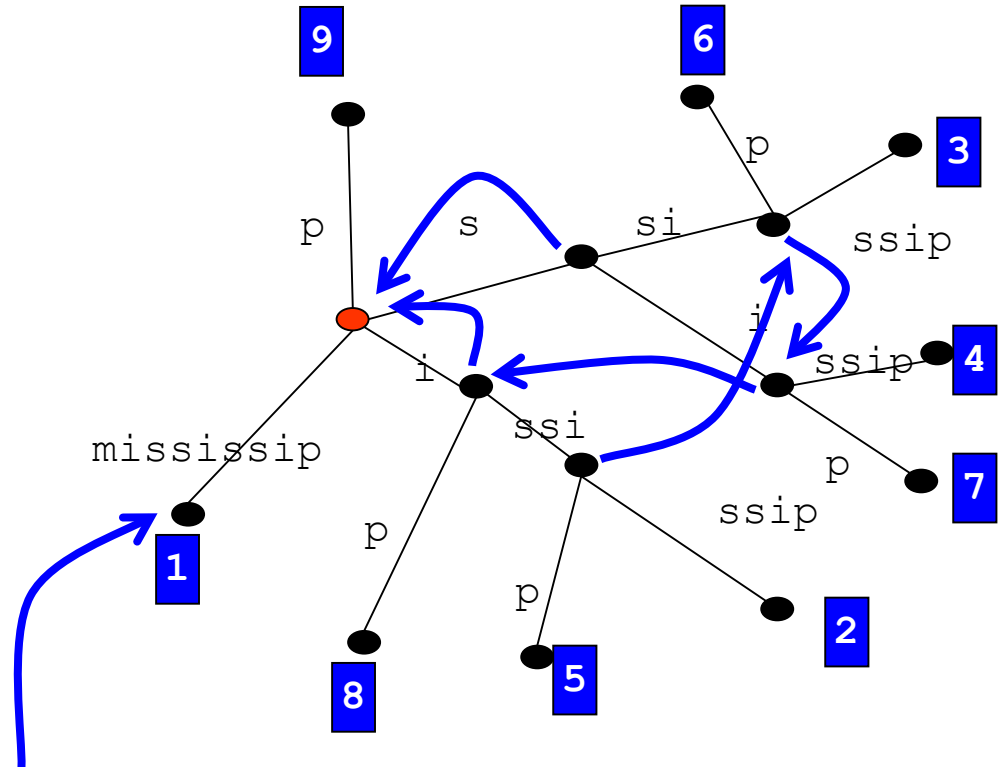


# Beispiel 9: Die Suffix-Links

12345678901  
mississipp*i*

## 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

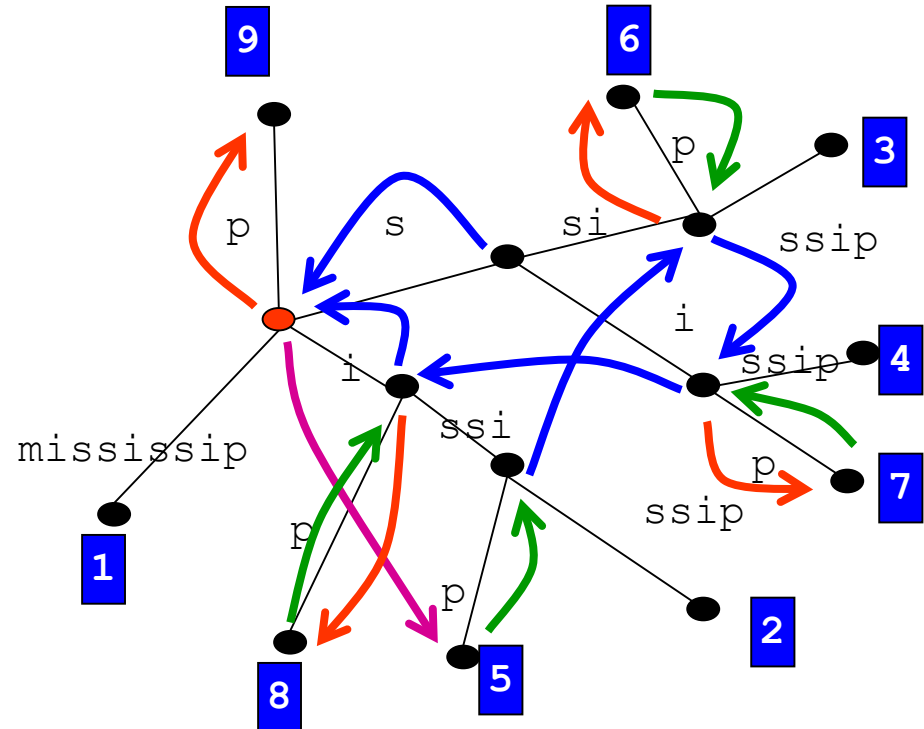
Sonst: Erstes Suffix matchen, ausgehend von Root

# Beispiel 9: Die Suffix-Links

12345678901  
mississippipi

## 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

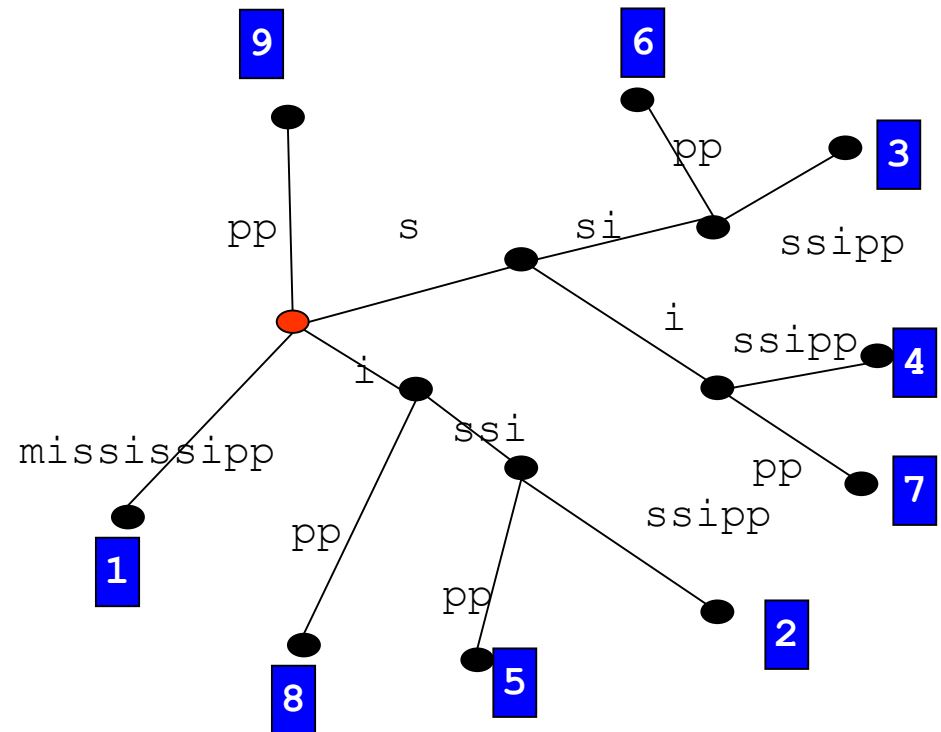
# Beispiel 10

12345678901  
mississippi

Phase 10

- $j_9' = 9$
- 10 („" + p): Rule 3

$j_{10}' = 9$



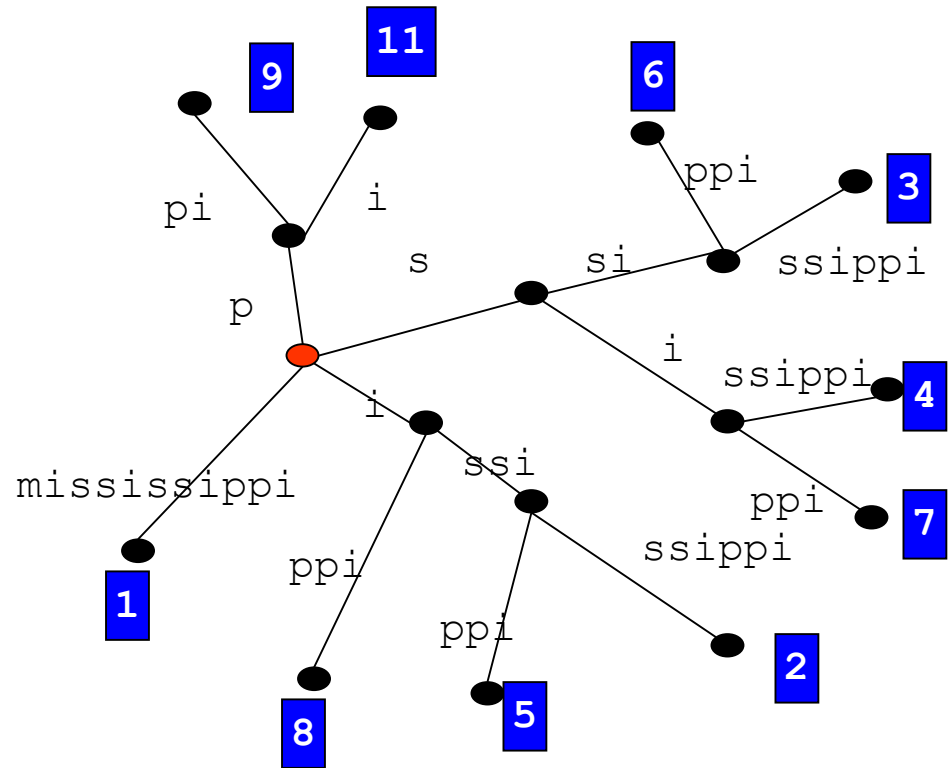
Nichts passiert  
(außer implizite Verlängerung der Blattkanten)

# 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

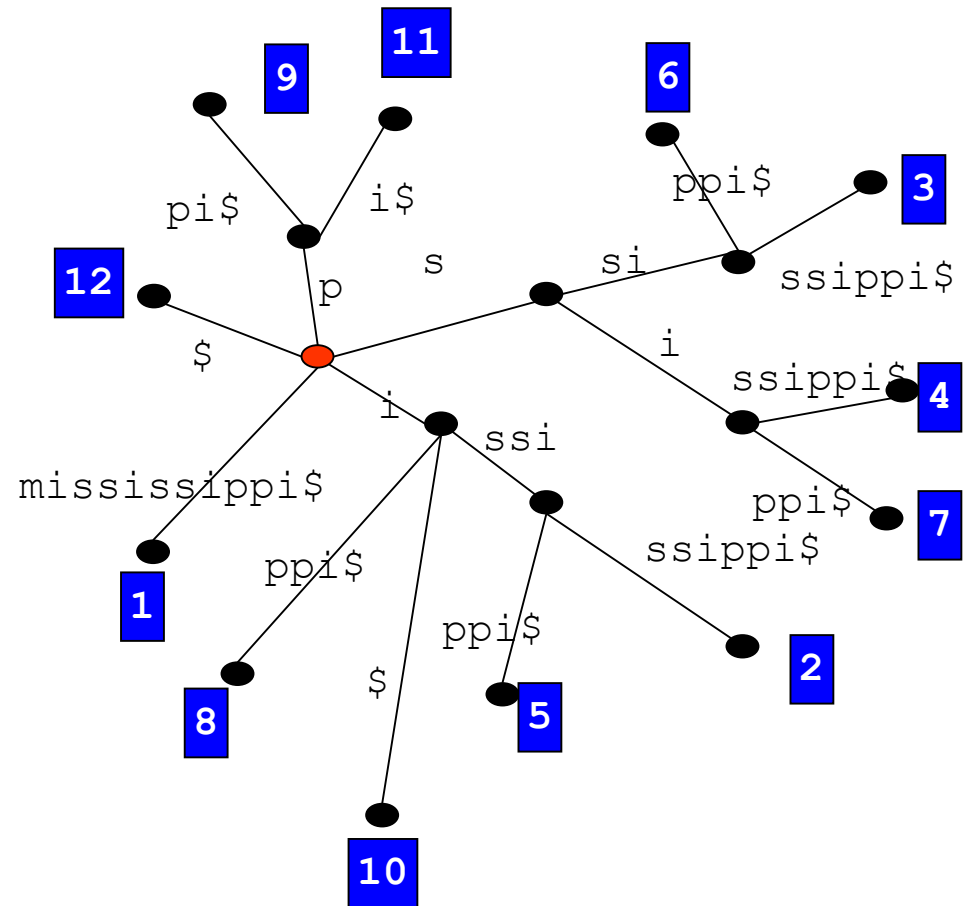
# Transformation $T_{11} \rightarrow T$

12345678901  
mississippi\$

Neuer Durchlauf

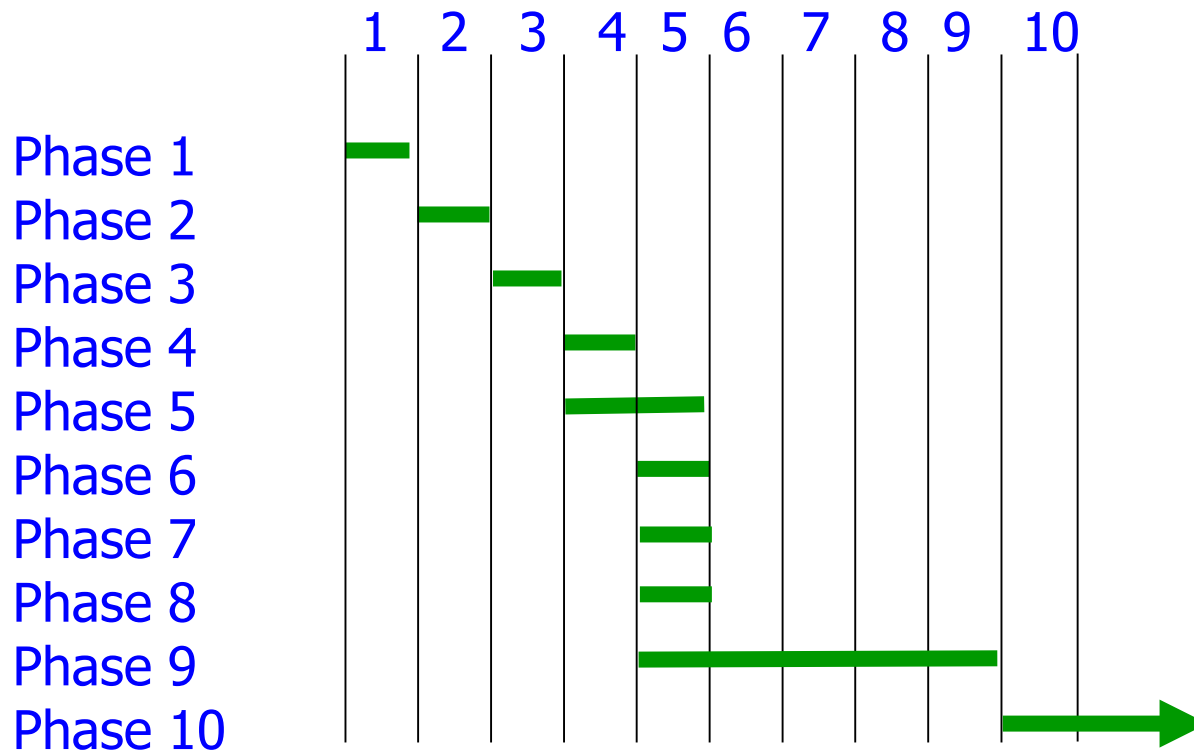
- 1-10: Regel 1
- 11: Regel 2

Fertig



# Komplexität

- Welche Schritte haben wir ausgeführt?



- Schritte markieren einen Pfad von links oben nach rechts unten
- Das können zusammen **höchstens  $2 \cdot m$  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 ...

---

- Suffixbäume sind sehr **speicherintensiv**
  - Viele Pointer, Kantenrepräsentation
  - Beste Implementierungen mit  $O(1)$  Branching brauchen  $\sim 15$  Byte/Zeichen
- Konstruktion schlecht auf **Sekundärspeicher**
  - Viel Random Access (durch Suffix-Links)
  - Aber: Hauptspeicher sind heutzutage riesig
- Abhilfe
  - **(Enhanced) Suffixarrays** – deutlich geringerer Speicherverbrauch
  - Methoden, die lineare Laufzeit zugunsten weniger IO aufgeben
  - Mehr Hauptspeicher
- Aber: Auch schlechte Ausnutzung von Cache-Hierarchien