technische universität
dortmund

Algorithmen auf Sequenzen

Volltext-Indizes

Dominik Kopczynski

Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik
TU Dortmund

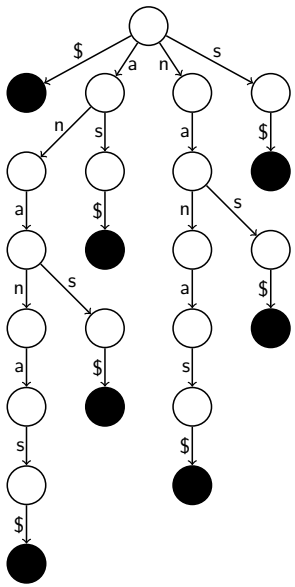
Überblick

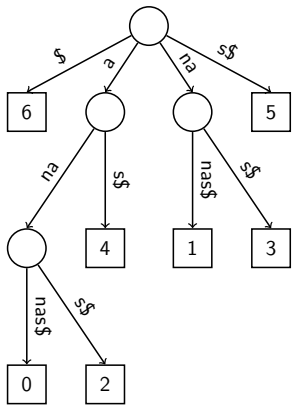
- Bei wiederholten Suchen auf (langen) Texten, ist es sinnvoll den Text vorzuverarbeiten.
- Durch Indizierung des Textes kann die Laufzeit so gesenkt werden, dass sie nur noch von der Länge des Patterns abhängt, also $\mathcal{O}(m)$.
- Bei natürlichsprachlichen Texten können Wort-basierte Indizes eingesetzt werden.
- Indizes, die die Suche nach beliebigen Teilstrings (auch wortübergreifend) erlauben, werden Volltext-Indizes genannt.

Überblick

- Bei wiederholten Suchen auf (langen) Texten, ist es sinnvoll den Text vorzuverarbeiten.
- Durch Indizierung des Textes kann die Laufzeit so gesenkt werden, dass sie nur noch von der Länge des Patterns abhängt, also $\mathcal{O}(m)$.
- Bei natürlichsprachlichen Texten können Wort-basierte Indizes eingesetzt werden.
- Indizes, die die Suche nach beliebigen Teilstrings (auch wortübergreifend) erlauben, werden Volltext-Indizes genannt.
- Im Folgenden wird ein Alphabet konstanter Größe angenommen, also $\mathcal{O}(1)$.

Überblick





6	$
0	ananas$
2	anas$
4	as$
1	nanas$
3	nas$
5	s$

$ananas
ananas$
anas$an
as$anan
nanas$a
nas$ana
s$anana

Suffix-Trie Suffix-Tree Suffix-Array FM-Index

Grundidee der Volltext-Indizes

Indizierung sämtlicher Suffixe eines Textes:

```
mississippi
 ississippi
  ssissippi
   sissippi
    issippi
     ssippi
      sippi
       ippi
        ppi
         pi
          i
```

Grundidee der Volltext-Indizes

Indizierung sämtlicher Suffixe eines Textes:

```
mississippi
 ississippi
  ssissippi
   sissippi
    issippi
     ssippi
      sippi
       ippi
        ppi
         pi
          i
```

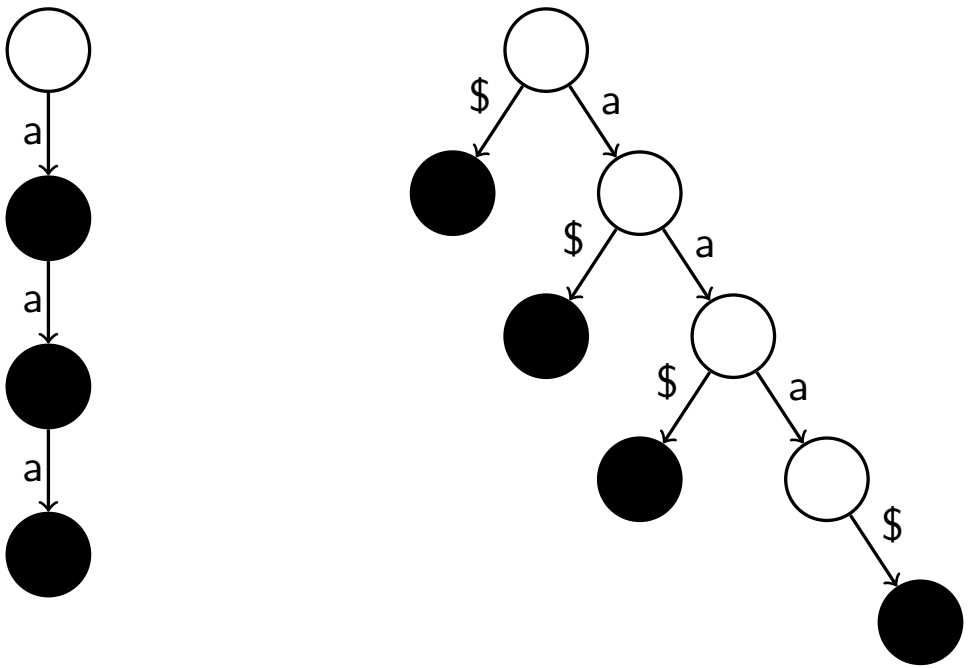
Quadratisch viele
$(n \cdot (n+1))/2$ Zeichen,
wenn alle Suffixe betrachtet werden.

Suffix-Trie

- Alle Suffixe sollen in einen Trie hinzugefügt werden, ähnlich wie beim Aho-Corasick.
- Solch ein Trie sollte folgende Eigenschaften erfüllen: Es gibt eine Bijektion zwischen den den Blättern des Baums und den Suffixen des Textes.
- Um diese Eigenschaft zu gewährleisten, wird im Folgenden ein Wächter (Sentinel) \$ am Ende des Textes eingeführt, für den folgende Eigenschaften
  gelten: $\Sigma \cap \$ = \emptyset$, für alle $\sigma \in \Sigma : \$ < \sigma$.

Suffix-Trie

Suffix-Trie für den Text $T = \texttt{aaa}$ mit und ohne Sentinal:



Alphabet-Abhängigkeit

Ist die Alphabetgröße nicht konstant, hängt die Laufzeit von der Datenstruktur ab, mit der Kinderknoten gesucht werden, sei dabei $c_v \in \mathcal{O}(|\Sigma|)$ die Anzahl der Kinder von Knoten v.

Datenstruktur	*Laufzeit*	*Platz pro Knoten*	*Platz gesamt*
Verkettete Liste	$\mathcal{O}(c_v)$	$\mathcal{O}(c_v)$	$\mathcal{O}(n)$
Balancierter Baum	$\mathcal{O}(\log c_v)$	$\mathcal{O}(c_v)$	$\mathcal{O}(n)$
Array Größe $\|\Sigma\|$	$\mathcal{O}(1)$	$\mathcal{O}(\|\Sigma\|)$	$\mathcal{O}(n\|\Sigma\|)$
Perfektes Hashing[1]	$\mathcal{O}(1)$	$\mathcal{O}(c_v)$	$\mathcal{O}(n)$

[1] theoretisch möglich

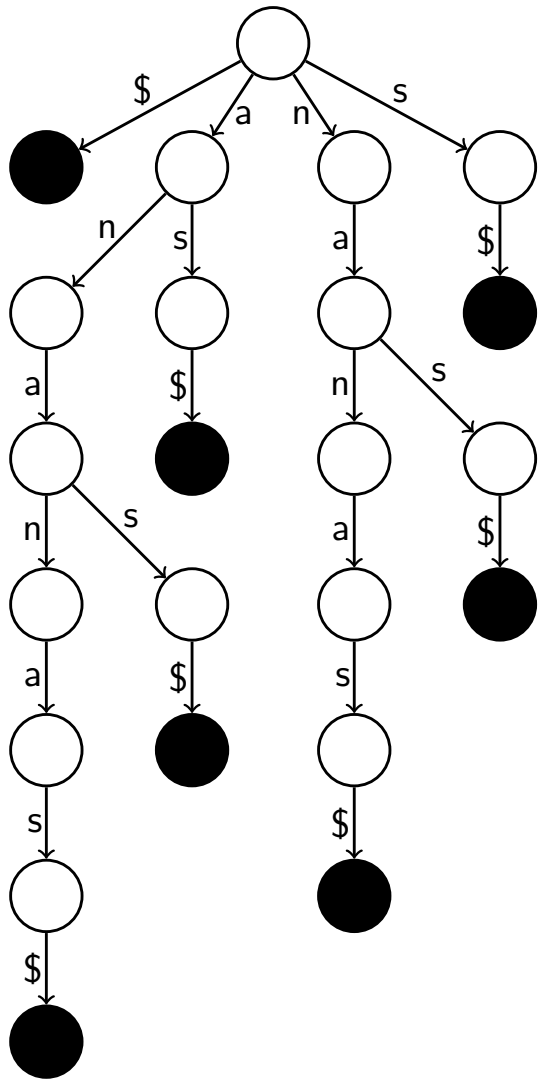
Suffix-Trie

Mustersuche:

- Jeder Teilstring des Textes T ist auch Präfix eines Suffixes von T.
- Die Mustersuche beginnt an der Wurzel.
- Der Trie wird Zeichen für Zeichen des Patterns durchtraversiert, bis entweder das komplette Pattern erkannt wurde (match), oder von einem Knoten aus keine ausgehende Kante $P[j]$ vorhanden ist (mismatch).

Suffix-Trie

Beispiel-Trie
für den Text
$T = \texttt{ananas\$}$:



Implementierung

```
1  def build_trie(T):
2      root, n = dict(), len(T)
3      for t in [T[j:] for j in range(n)]:
4          node = root
5          for c in t:
6              if c not in node:
7                  node[c] = dict()
8              node = node[c]
9      return root
10
11 def has_pattern(node, P):
12     for c in P:
13         if c not in node: return False
14         node = node[c]
15     return True
```

Zusammenfassung

- Mit Hilfe des Suffix-Tries kann eine Patternsuche in $\mathcal{O}(m)$ durchgeführt werden.
- Sowohl die Laufzeit für der Erstellung des Suffix-Tries als auch der Speicherverbrauch beträgt $\mathcal{O}(n^2)$.
- Ein Suffix-Trie ist eine naive Implementierung eines Volltext-Indexes und wird aufgrund seines hohen Speicherverbrauchs nicht genutzt.

Beobachtungen

- Der Trie kann lediglich das Vorkommen von Pattern im Text feststellen, jedoch nicht die Position.
- Im Suffix-Trie kommen oft Ketten von Knoten vor, die nur einen Nachfolger haben.
- Bei der Erstellung des Tries wird der i-te Buchstabe $i + 1$ mal betrachtet. Wünschenswert wäre, wenn dies nur einmal geschähen würde.

Beobachtungen

- Der Trie kann lediglich das Vorkommen von Pattern im Text feststellen, jedoch nicht die Position.
- Im Suffix-Trie kommen oft Ketten von Knoten vor, die nur einen Nachfolger haben.
- Bei der Erstellung des Tries wird der i-te Buchstabe $i + 1$ mal betrachtet. Wünschenswert wäre, wenn dies nur einmal geschähen würde.
- All diese Verbesserungen realisiert der Suffix-Tree.

Σ^+-Baum

Definition (Σ^+-Baum)

Sei Σ ein Alphabet, dann ist ein Σ^+-Baum ein gewurzelter Baum, dessen Kanten jeweils mit einem nichtleeren String über Σ annotiert sind, so dass kein Knoten zwei ausgehende Kanten hat, die mit dem gleichen Buchstaben beginnen.

Σ^+-Baum

Definition (Σ^+-Baum)

Sei Σ ein Alphabet, dann ist ein Σ^+-Baum ein gewurzelter Baum, dessen Kanten jeweils mit einem nichtleeren String über Σ annotiert sind, so dass kein Knoten zwei ausgehende Kanten hat, die mit dem gleichen Buchstaben beginnen.

Definition (Kompakter Σ^+-Baum)

Ein Σ^+-Baum heißt *kompakt*, wenn kein Knoten (außer ggf. der Wurzel) genau ein Kind besitzt.

(String)Tiefe eines Baums

Definition (Tiefe eines Baums)

Die Tiefe $dep(s)$ eines Knotens s ist die Anzahl der Kanten eines eindeutig beschrifteten Pfades von der Wurzel zu s. Die Wurzel r hat per Definition $dep(r) := 0$.

(String)Tiefe eines Baums

Definition (Tiefe eines Baums)

Die Tiefe $dep(s)$ eines Knotens s ist die Anzahl der Kanten eines eindeutig beschrifteten Pfades von der Wurzel zu s. Die Wurzel r hat per Definition $dep(r) := 0$.

Definition (Stringtiefe eines Suffix-Trees)

Für Knoten s sei $str(s)$ die Konkatenation der Kantenbeschriftungen auf einem eindeutigen Pfad von der Wurzel zu s. Dabei sei die Stringtiefe $strdep(s) = |str(s)|$.

(String)Tiefe eines Baums

Definition (Tiefe eines Baums)

Die Tiefe $dep(s)$ eines Knotens s ist die Anzahl der Kanten eines eindeutig beschrifteten Pfades von der Wurzel zu s. Die Wurzel r hat per Definition $dep(r) := 0$.

Definition (Stringtiefe eines Suffix-Trees)

Für Knoten s sei $str(s)$ die Konkatenation der Kantenbeschriftungen auf einem eindeutigen Pfad von der Wurzel zu s. Dabei sei die Stringtiefe $strdep(s) = |str(s)|$.

Die Tiefe eines Knotens s muss nicht der Stringtiefe entsprechen.

Suffix-Tree

Ein Suffix-Tree soll folgende Eigenschaften für einen Text $T\$$ erfüllen:

- Es gibt eine Bijektion zwischen den Blättern der Trees und den Suffixen des Textes.

Suffix-Tree

Ein Suffix-Tree soll folgende Eigenschaften für einen Text $T\$$ erfüllen:

- Es gibt eine Bijektion zwischen den Blättern der Trees und den Suffixen des Textes.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.

Suffix-Tree

Ein Suffix-Tree soll folgende Eigenschaften für einen Text $T\$$ erfüllen:

- Es gibt eine Bijektion zwischen den Blättern der Trees und den Suffixen des Textes.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.

Suffix-Tree

Ein Suffix-Tree soll folgende Eigenschaften für einen Text $T\$$ erfüllen:

- Es gibt eine Bijektion zwischen den Blättern der Trees und den Suffixen des Textes.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.
- Jeder innere Knoten hat ≥ 2 Kinder.

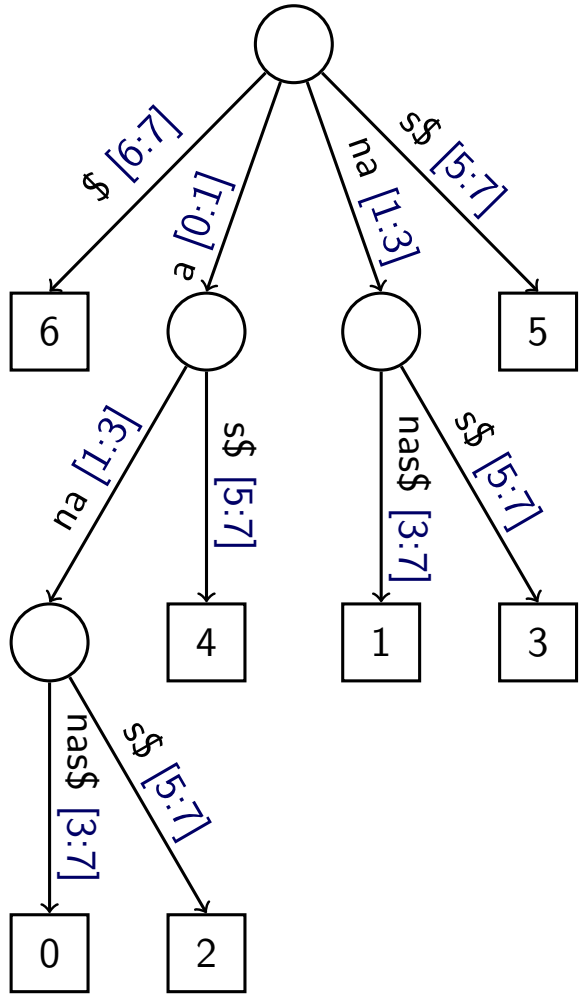
Suffix-Tree

Ein Suffix-Tree soll folgende Eigenschaften für einen Text $T\$$ erfüllen:

- Es gibt eine Bijektion zwischen den Blättern der Trees und den Suffixen des Textes.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.
- Jeder innere Knoten hat ≥ 2 Kinder.
- Jeder Teilsting von $T\$$ kann auf einem Pfad von der Wurzel abgelesen werden.

Suffix-Tree

Beispiel-Tree
für den Text
$T = \texttt{ananas\$}$:



Eigenschaften des Suffix-Tree

- Der Speicherverbrauch pro Knoten sinkt auf $\mathcal{O}(1)$, da nur noch das Indexpaar (b, e) und die Nachkommen des Knotens gespeichert werden.
- Das Paar (b, e) entspricht dem Teilstring $T[b : e]$.
- Da es genau n Blätter gibt und jeder innere Knoten mind. zwei Kinder hat, ist die Anzahl der inneren Knoten auf n beschränkt.

Eigenschaften des Suffix-Tree

- Der Speicherverbrauch pro Knoten sinkt auf $\mathcal{O}(1)$, da nur noch das Indexpaar (b, e) und die Nachkommen des Knotens gespeichert werden.
- Das Paar (b, e) entspricht dem Teilstring $T[b : e]$.
- Da es genau n Blätter gibt und jeder innere Knoten mind. zwei Kinder hat, ist die Anzahl der inneren Knoten auf n beschränkt.
- Der Speicherverbrauch liegt somit bei $\mathcal{O}(n)$.
- Durch die Nummerierung der Blätter mit der Startposition ihres entsprechenden Suffixes lässt sich bei einer Mustersuche die Position im Text bestimmen.

Konstruktion des Suffix-Trees

- Beim naiven Ansatz behilft man sich eines Suffix-Tries, den man zum Suffix-Tree erweitert.
- Durch Zählen der Knoten im Pfad zu den Blättern, kann man diese beschriften und die Kanten indizieren.
- Die Erweiterung lässt sich in $\mathcal{O}(n^2)$ realisieren, die Gesamtlaufzeit beträgt also immer noch $\mathcal{O}(n^2)$.

Konstruktion des Suffix-Trees

- Beim naiven Ansatz behilft man sich eines Suffix-Tries, den man zum Suffix-Tree erweitert.
- Durch Zählen der Knoten im Pfad zu den Blättern, kann man diese beschriften und die Kanten indizieren.
- Die Erweiterung lässt sich in $\mathcal{O}(n^2)$ realisieren, die Gesamtlaufzeit beträgt also immer noch $\mathcal{O}(n^2)$.
- Umso bemerkenswerter ist es, dass der *Ukkonen*-Algorithmus zur Konstruktion eines Suffix-Trees eine Laufzeit von $\mathcal{O}(n)$ hat.

Ukkonen-Algorithmus für Suffix-Trees

- Der Algorithmus konstruiert zu einem Text $T\$$ mit $n := |T\$|$ einen Suffix-Tree in n Iterationen $0, 1, \ldots, n-1$.
- In jeder Iteration i wird das Zeichen $T[i]$ zum Suffix-Tree hinzugefügt.

Ukkonen-Algorithmus für Suffix-Trees

- Der Algorithmus konstruiert zu einem Text $T\$$ mit $n := |T\$|$ einen Suffix-Tree in n Iterationen $0, 1, \ldots, n-1$.
- In jeder Iteration i wird das Zeichen $T[i]$ zum Suffix-Tree hinzugefügt.
- Der Algorithmus ist ein *online*-Algorithmus, das bedeutet, dass nach jeder Iteration i der Baum ein (gültiger) Suffix-Tree für das Präfix $T[:i+1]$ ist.
- Um Linearlaufzeit zu erreichen, darf jede Iteration nur amortisiert $\mathcal{O}(1)$ dauern. Mit verschiedenen Tricks lässt sich diese Laufzeit erreichen.

Tricks des Ukkonen-Algorithmus

1. Es wird das Tupel (s, k) verwaltet, wobei k der Anzahl der Buchstaben an allen bereits beschrittenen Kanten und s dem aktiven Knoten entspricht.

Tricks des Ukkonen-Algorithmus

1. Es wird das Tupel (s, k) verwaltet, wobei k der Anzahl der Buchstaben an allen bereits beschrittenen Kanten und s dem aktiven Knoten entspricht.
2. Einmal eingeführte Blattknoten können nicht mehr erreicht werden. Das Indexpaar zu dem Blattknoten wird mit (b, ∞) beschriftet. So wird das Ende automatisch weitergeführt.

Tricks des Ukkonen-Algorithmus

1. Es wird das Tupel (s, k) verwaltet, wobei k der Anzahl der Buchstaben an allen bereits beschrittenen Kanten und s dem aktiven Knoten entspricht.
2. Einmal eingeführte Blattknoten können nicht mehr erreicht werden. Das Indexpaar zu dem Blattknoten wird mit (b, ∞) beschriftet. So wird das Ende automatisch weitergeführt.
3. Um vom Knoten s mit $str(s) = ax$ und $a \in \Sigma, x \in \Sigma^+$ zum Knoten w mit $str(w) = x$ in konstanter Zeit zu gelangen, werden Verbindungen (Suffixlinks) eingesetzt.

Tricks des Ukkonen-Algorithmus

1. Es wird das Tupel (s, k) verwaltet, wobei k der Anzahl der Buchstaben an allen bereits beschrittenen Kanten und s dem aktiven Knoten entspricht.
2. Einmal eingeführte Blattknoten können nicht mehr erreicht werden. Das Indexpaar zu dem Blattknoten wird mit (b, ∞) beschriftet. So wird das Ende automatisch weitergeführt.
3. Um vom Knoten s mit $str(s) = ax$ und $a \in \Sigma, x \in \Sigma^+$ zum Knoten w mit $str(w) = x$ in konstanter Zeit zu gelangen, werden Verbindungen (Suffixlinks) eingesetzt.
4. Beim Traversieren des Baums können Kanten mit bekanntem Indexpaar (b, e) in konstanter Zeit übersprungen werden.

Implementierung

Die Klasse `STNode` verwaltet die Knoten im Suffix-Tree S mit folgenden Attributen:

- Ein Dictionary $targets : c \rightarrow (s, b, e)$ mit $c := T[b]$, $s \in nodes(S), 0 \leq b < e \leq n$ zum Speichern der Kinderknoten und Indexpaaren (b, e) auf den hinführenden Kanten.
- Eine Referenz *s_link* auf das längste Suffix.
- Startposition *pos* des Suffixes im Text.

```
1 class STNode():
2     def __init__(self, s_link = None, pos= -1):
3         self.targets = dict()
4         self.s_link = s_link
5         self.pos = pos
```

Implementierung

Vorbereitung:

- Der Algorithmus startet mit einem leeren Baum, bestehend aus einer Wurzel *root*.
- Da root selber ein sich ändernder Knoten ist, muss es noch einen Knoten *top* oberhalb der Wurzel geben, der nur für die Konstruktion relevant ist.

Implementierung

Vorbereitung:

- Der Algorithmus startet mit einem leeren Baum, bestehend aus einer Wurzel *root*.
- Da root selber ein sich ändernder Knoten ist, muss es noch einen Knoten *top* oberhalb der Wurzel geben, der nur für die Konstruktion relevant ist.
- Von top aus soll root beim Lesen aller Zeichen aus dem Alphabet erreichbar sein.
- Der Suffixlink von root zeigt auf top.

Implementierung

Vorbereitung:

- Der Algorithmus startet mit einem leeren Baum, bestehend aus einer Wurzel *root*.
- Da root selber ein sich ändernder Knoten ist, muss es noch einen Knoten *top* oberhalb der Wurzel geben, der nur für die Konstruktion relevant ist.
- Von top aus soll root beim Lesen aller Zeichen aus dem Alphabet erreichbar sein.
- Der Suffixlink von root zeigt auf top.
- Es wird von der Wurzel gestartet.
- Mit der Einfüge-Operation *update* sollen in jeder Iteration die Zeichen aus T in den Baum eingefügt werden.

Implementierung

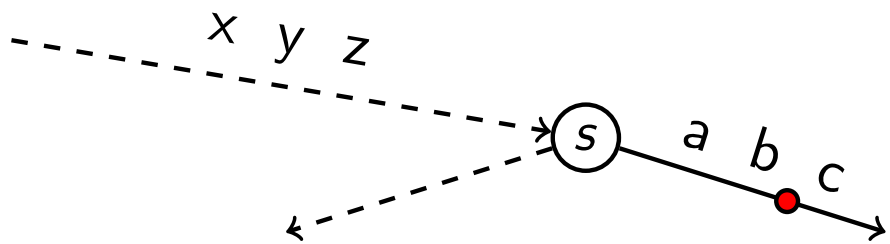
```
1  def ukkonen(T):
2      top = STNode()
3      root = STNode(s_link = top)
4      for c in T:
5          if c not in top.targets:
6              top.targets[c] = (root, -1, 0)
7
8      s, k, pos = root, 0, 0
9      for i in range(len(T)):
10         s, k, pos = update(s, k, i, T, pos)
11     return root
```

Implementierung

- Eine aktuelle Position kann sich entweder direkt an einem inneren Knoten befinden, oder an einer Kante, die von einem Knoten ausgeht.
- Wenn $k = i$, dann befindet sich die aktuelle Position an einem Knoten.
- Wenn $k < i$, dann befindet sich die aktuelle Position an einer Kante. Dabei sei s der ausgehende Knoten.

Implementierung

- Eine aktuelle Position kann sich entweder direkt an einem inneren Knoten befinden, oder an einer Kante, die von einem Knoten ausgeht.
- Wenn $k = i$, dann befindet sich die aktuelle Position an einem Knoten.
- Wenn $k < i$, dann befindet sich die aktuelle Position an einer Kante. Dabei sei s der ausgehende Knoten.
- Beispiel: $T = \overbrace{\underbrace{\texttt{...xyzabc...xyz}}_{k}\texttt{ab}}^{i}\texttt{...}$:



Implementierung

Definition (Referenz)

Sei w ein Teilstring von $T[: i]$ mit $w = uv$, wobei $u = T[j : b]$ und $v = T[b : e]$. Ist $str(s) = u$ dann ist $(s, (b, e))$ eine Referenz auf w.

Implementierung

Definition (Referenz)

Sei w ein Teilstring von $T[:i]$ mit $w = uv$, wobei $u = T[j : b]$ und $v = T[b : e]$. Ist $str(s) = u$ dann ist $(s, (b, e))$ eine Referenz auf w.

Beispiel: $\mathrm{w} = \begin{matrix} abcde \\ 34567 \end{matrix} \widehat{=} (\overline{abc}, (6, 8)) \widehat{=} (\overline{a}, (4, 8)) \widehat{=} (\overline{\epsilon}, (3, 8))$

Implementierung

Definition (Referenz)

Sei w ein Teilstring von $T[: i]$ mit $w = uv$, wobei $u = T[j : b]$ und $v = T[b : e]$. Ist $str(s) = u$ dann ist $(s, (b, e))$ eine Referenz auf w.

Beispiel: $\mathrm{w} = \begin{matrix} abcde \\ 34567 \end{matrix} \widehat{=} (\overline{abc}, (6, 8)) \widehat{=} (\overline{a}, (4, 8)) \widehat{=} (\overline{\epsilon}, (3, 8))$

Definition (Kanonische Referenz)

Eine Referenz $(s, (b, e))$ heißt kanonisch, wenn $e - b$ minimal für alle Referenzen auf w ist.

Implementierung

- Beim Update wird zuerst überprüft, ob der aktuell einzufügende Buchstabe von der aktuellen Position gelesen werden kann.
- Dabei muss unterschieden werden, ob die aktuelle Position sich in einem Knoten befindet oder nicht.

Implementierung

- Beim Update wird zuerst überprüft, ob der aktuell einzufügende Buchstabe von der aktuellen Position gelesen werden kann.
- Dabei muss unterschieden werden, ob die aktuelle Position sich in einem Knoten befindet oder nicht.
- Wenn sich die aktuelle Position an einer Kante befindet, und der nächste Buchstabe nicht gelesen werden kann, wird ein innerer Knoten hinzugefügt.
- Diese Schritte werden in der Funktion *test_and_split* realisiert.

Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

Implementierung

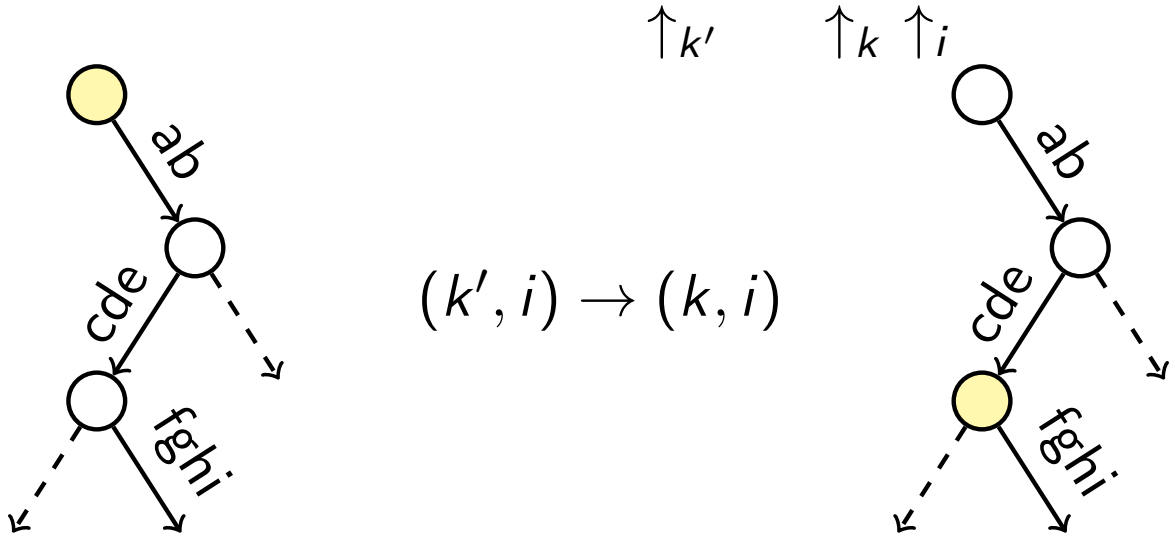
- Während eines Updatevorgangs kommt es vor, dass von der aktuellen Position auf einen inneren Knoten geringerer Tiefe gewechselt wird.
- Um zu wahren, dass die Referenz $(s, (k, i))$ wieder kanonisch wird, wird die Funktion *canonize* aufgerufen.

Beispiel: $T =$ `...abcdefghi...abcdefgh...`

Implementierung

- Während eines Updatevorgangs kommt es vor, dass von der aktuellen Position auf einen inneren Knoten geringerer Tiefe gewechselt wird.
- Um zu wahren, dass die Referenz $(s,(k,i))$ wieder kanonisch wird, wird die Funktion *canonize* aufgerufen.

Beispiel: $T = \texttt{...abcdefghi...abcdefgh...}$



Implementierung

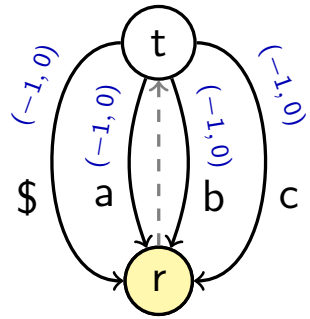
```
1 def canonize(s, k, i, T):
2     if i < k: return s, k
3     s1, b, e = s.targets[T[k]]
4     while e - b <= i + 1 - k:
5         s, k = s1, k + e - b
6         if k <= i:
7             s1, b, e = s.targets[T[k]]
8     return s, k
```

Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

Implementierung

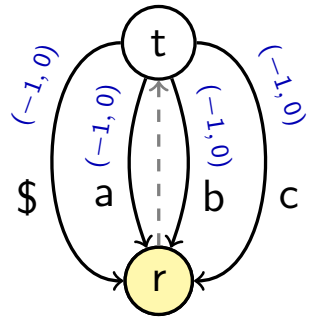
```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```



T	$babacbab\$$	$endPoint$	-
i	0	k	0
s	r	r	—
old_r	—		

Implementierung

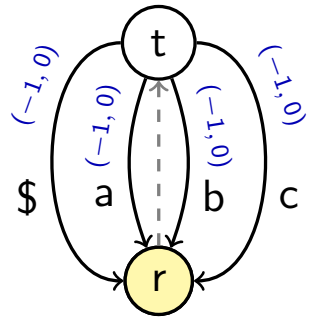
```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```



T	$babacbab\$$	$endPoint$	-
i	0	k	0
s	r	r	—
old_r	—		

Implementierung

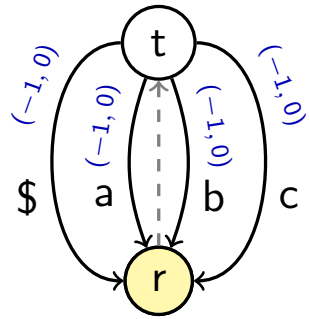
```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```



T	$babacbab\$$	$endPoint$	-
i	0	k	0
s	r	r	—
old_r	—	$s1$	—
b	—	e	—

Implementierung

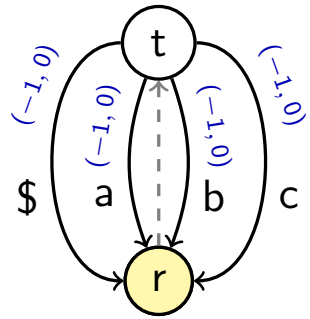
```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```



T	$babacbab\$$	$endPoint$	-
i	0	k	0
s	r	r	—
old_r	—	$s1$	—
b	—	e	—

Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

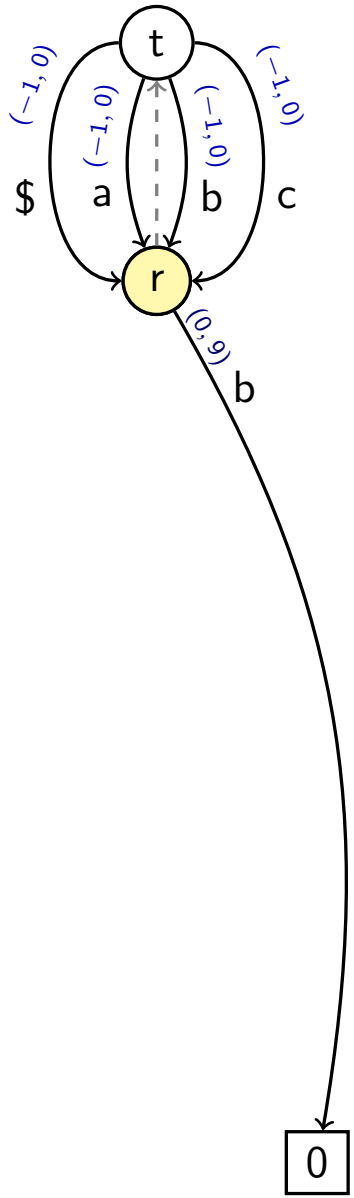


T	$babacbab\$$	$endPoint$	false
i	0	k	0
s	r	r	r
old_r	—		

Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

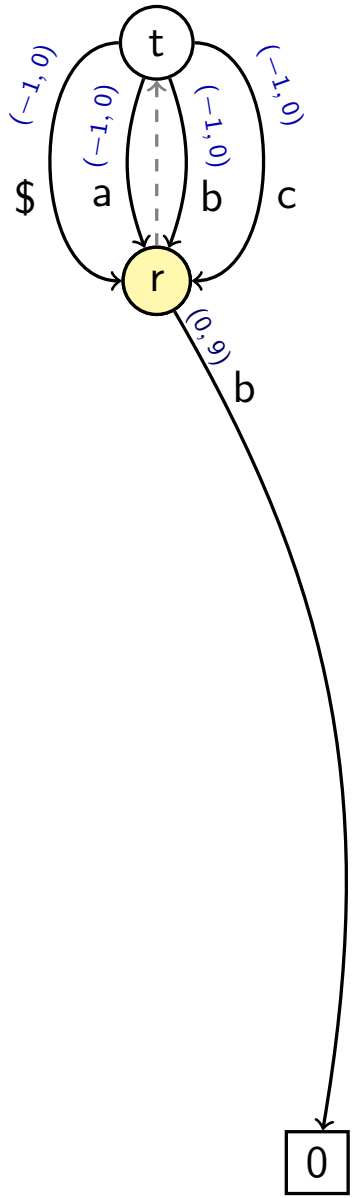
T	*babacbab*\$	*endPoint*	false
i	0	*k*	0
s	r	*r*	r
old_r	r		



Implementierung

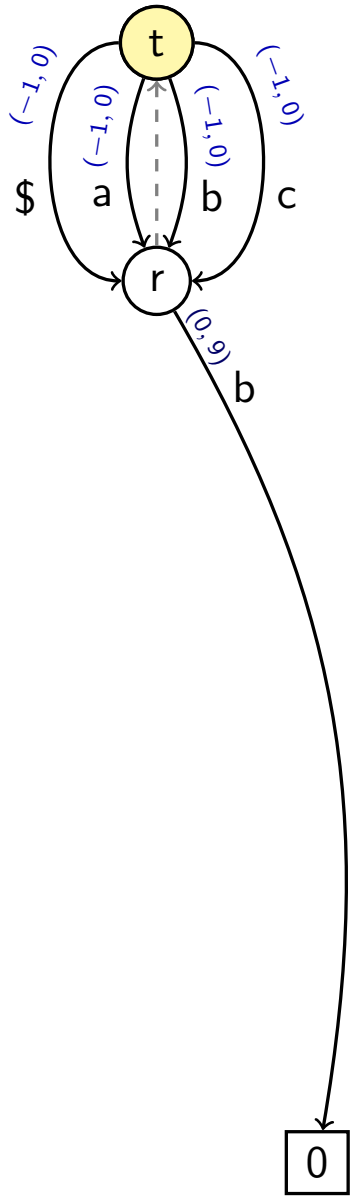
```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

T	*babacbab*\$	*endPoint*	false
i	0	*k*	0
s	r	*r*	r
old_r	r		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

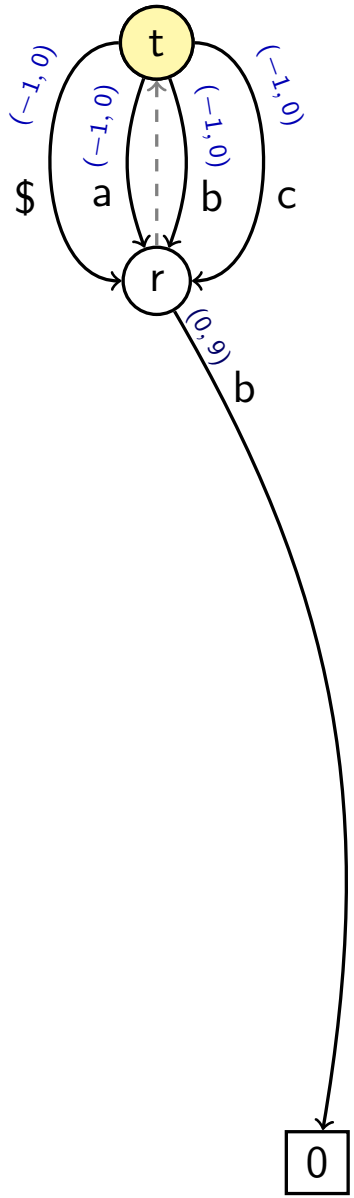


T	$babacbab\$$	$endPoint$	false
i	-1	k	0
s	t	r	r
old_r	r	$s1$	—
b	—	e	—

Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

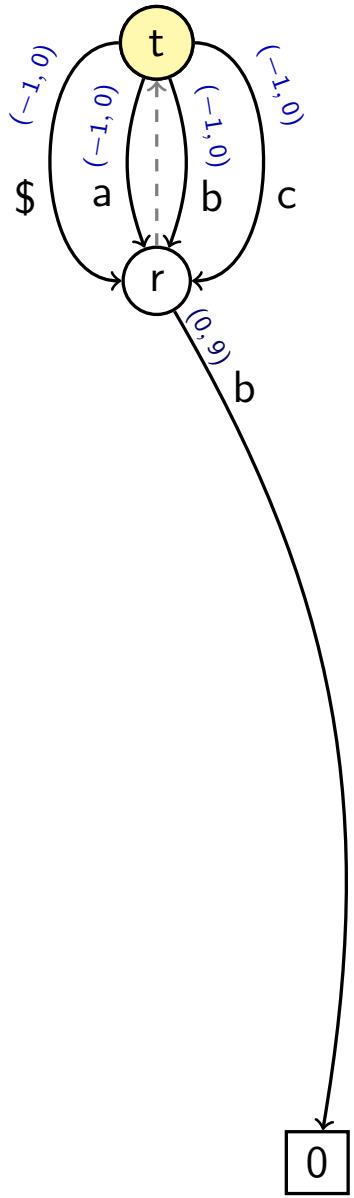
T	$babacbab\$$	$endPoint$	false
i	-1	k	0
s	t	r	r
old_r	r	$s1$	–
b	–	e	–



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

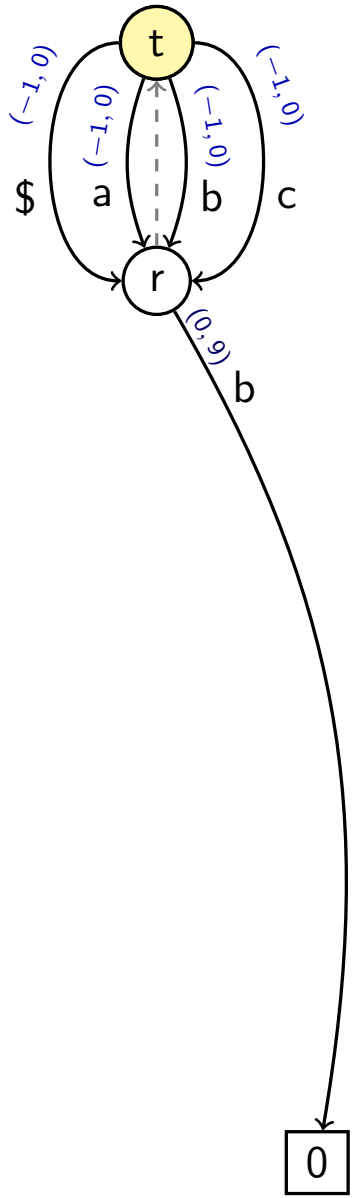
T	$babacbab\$$	$endPoint$	false
i	0	k	0
s	t	r	r
old_r	r		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

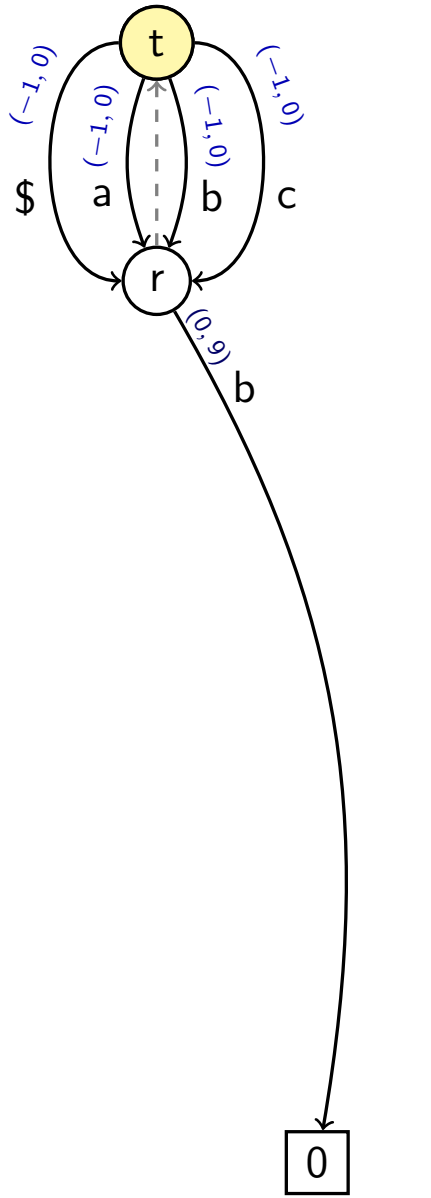
T	$babacbab\$$	$endPoint$	false
i	0	k	0
s	t	r	r
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

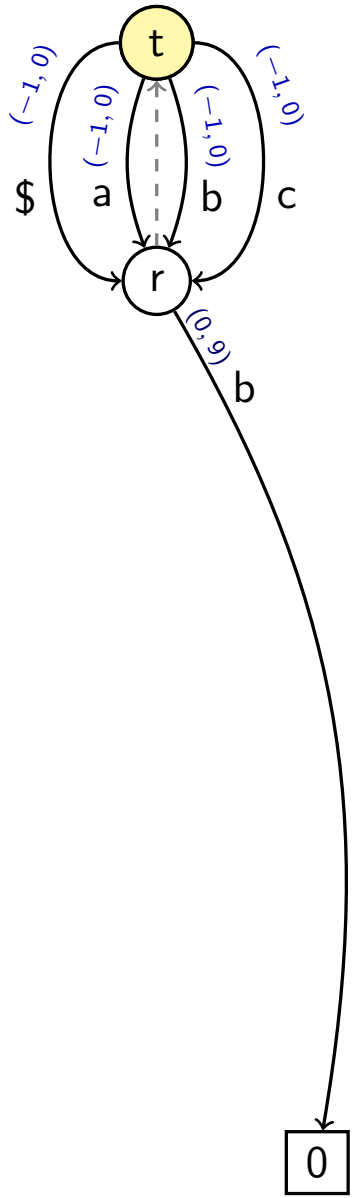
T	$babacbab\$$	$endPoint$	false
i	0	k	0
s	t	r	r
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

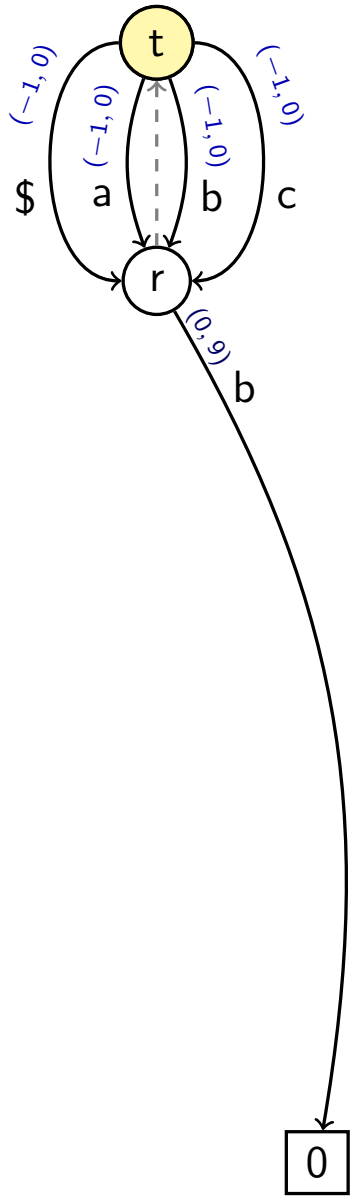
T	$babacbab\$$	$endPoint$	true
i	0	k	0
s	t	r	t
old_r	r		



Implementierung

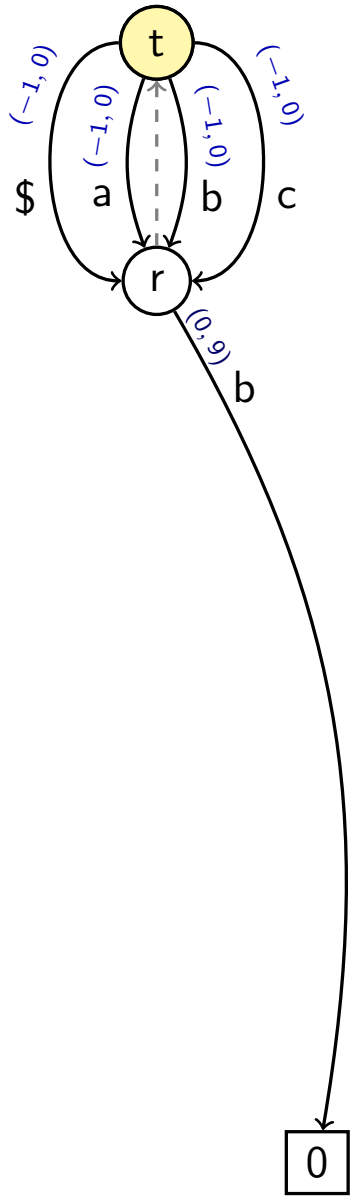
```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

T	$babacbab\$$	$endPoint$	true
i	0	k	0
s	t	r	t
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

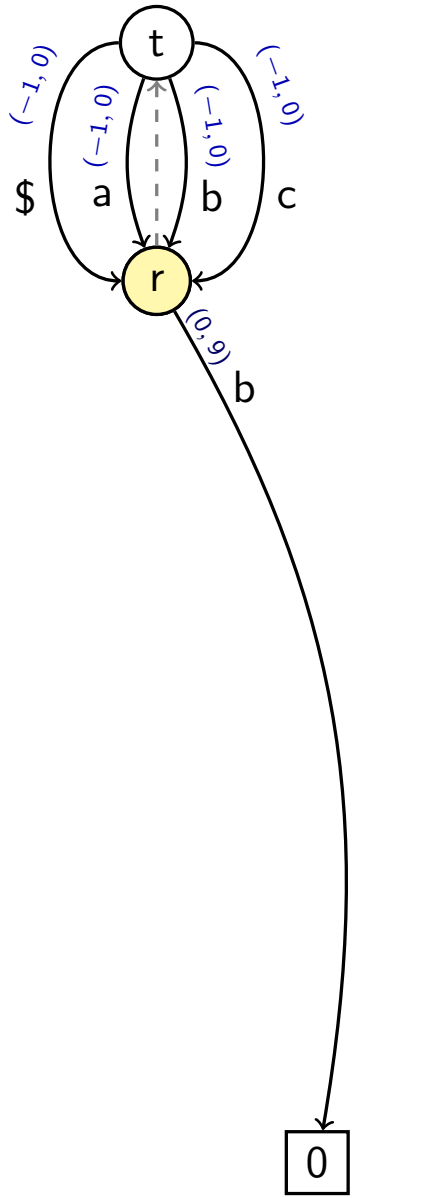


T	$babacbab\$$	$endPoint$	true
i	0	k	0
s	t	r	t
old_r	r	$s1$	r
b	-1	e	0

Implementierung

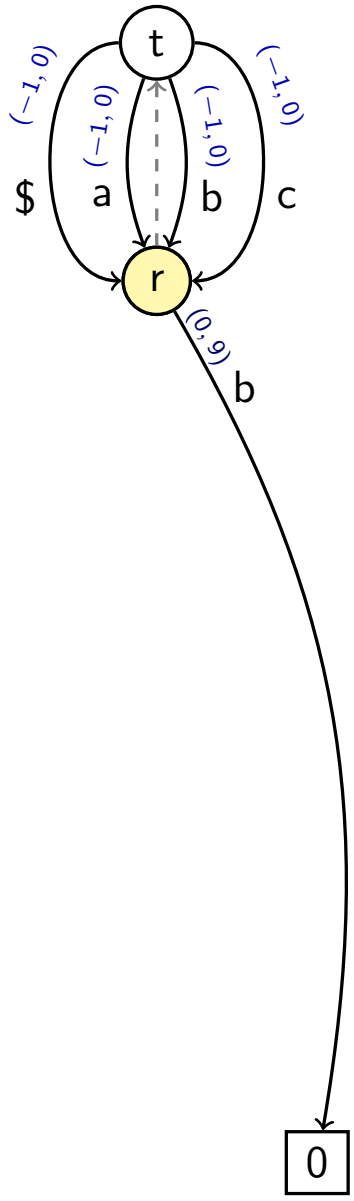
```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

T	$babacbab\$$	$endPoint$	true
i	0	k	1
s	r	r	t
old_r	r	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

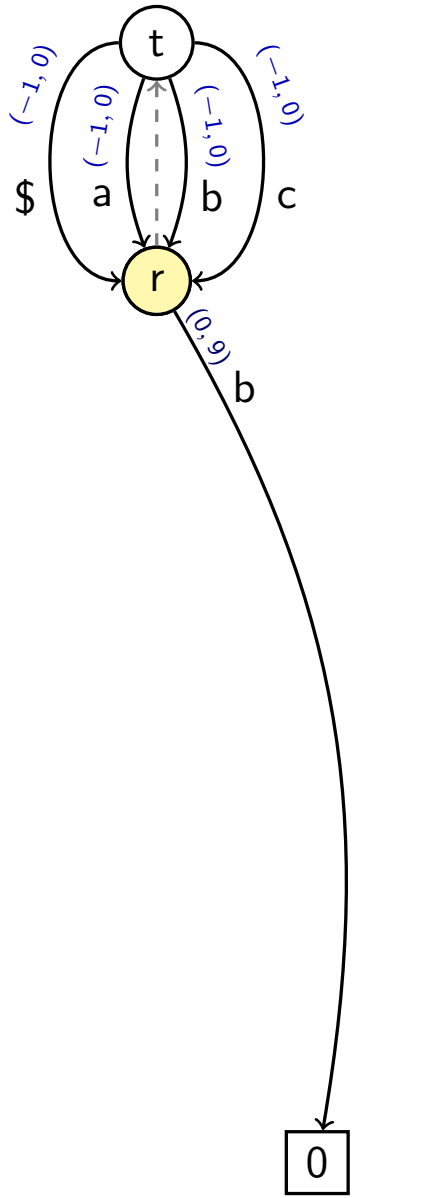


T	$babacbab\$$	$endPoint$	true
i	0	k	1
s	r	r	t
old_r	r	$s1$	r
b	-1	e	0

Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

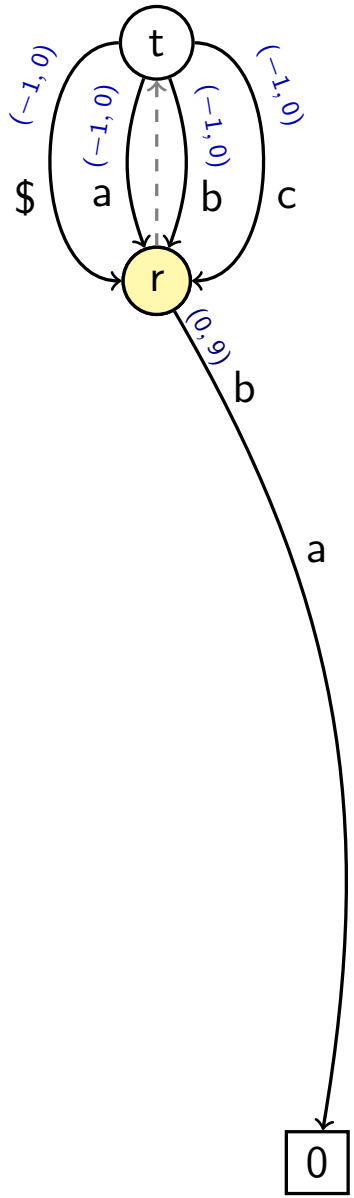
T	$babacbab\$$	$endPoint$	true
i	0	k	1
s	r	r	t
old_r	r		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

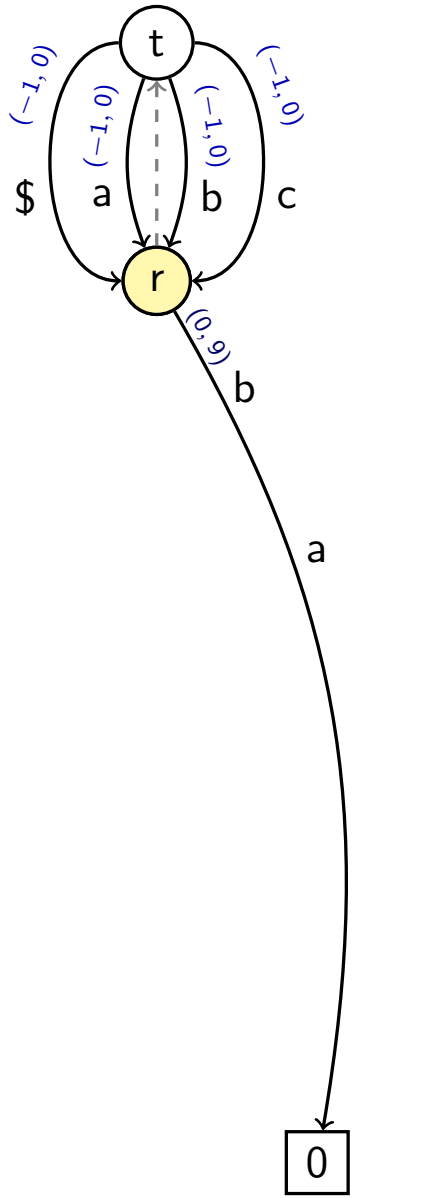
T	$babacbab\$$	$endPoint$	-
i	1	k	1
s	r	r	—
old_r	—		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

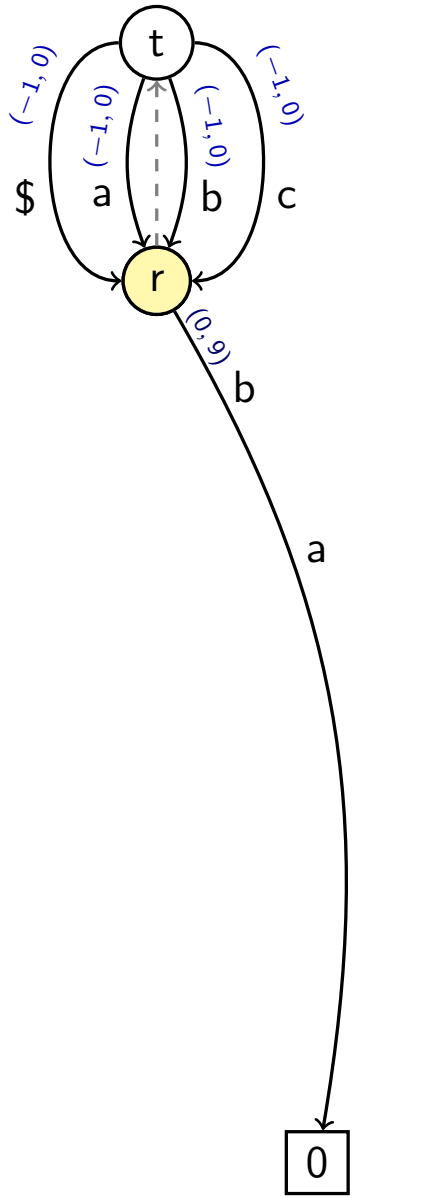
T	$babacbab\$$	$endPoint$	-
i	1	k	1
s	r	r	—
old_r	—		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

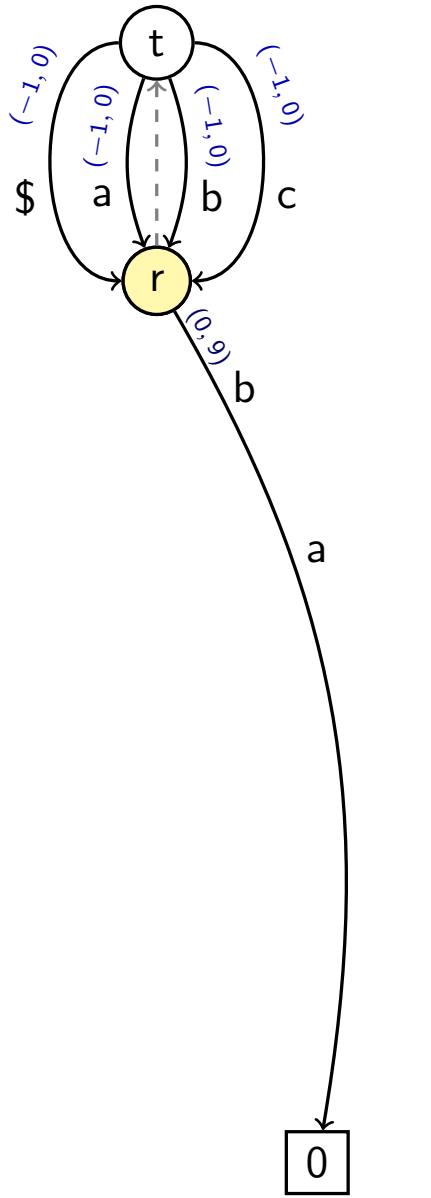
T	$babacbab\$$	$endPoint$	-
i	1	k	1
s	r	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

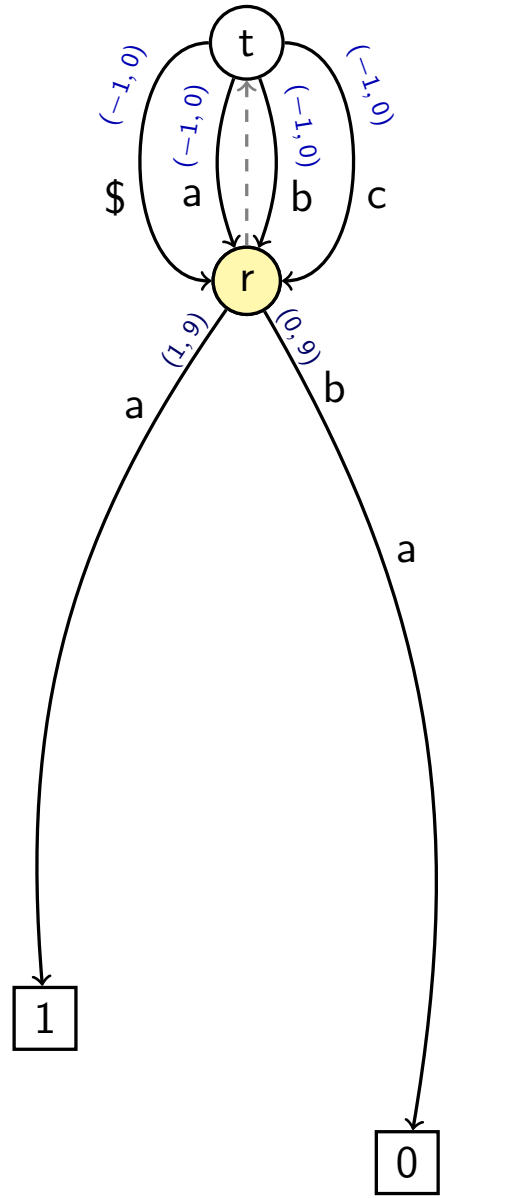
T	$babacbab\$$	$endPoint$	false
i	1	k	1
s	r	r	r
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

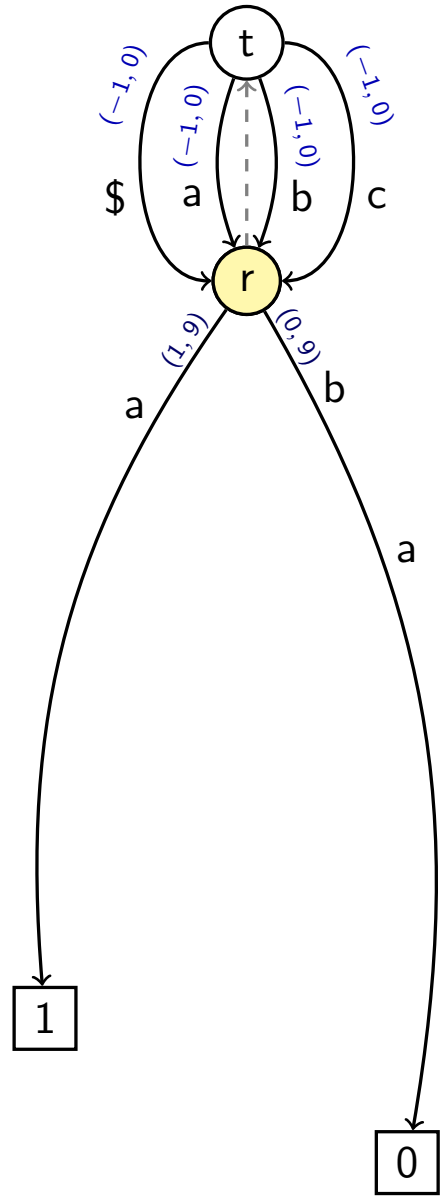
T	$babacbab\$$	$endPoint$	false
i	1	k	1
s	r	r	r
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

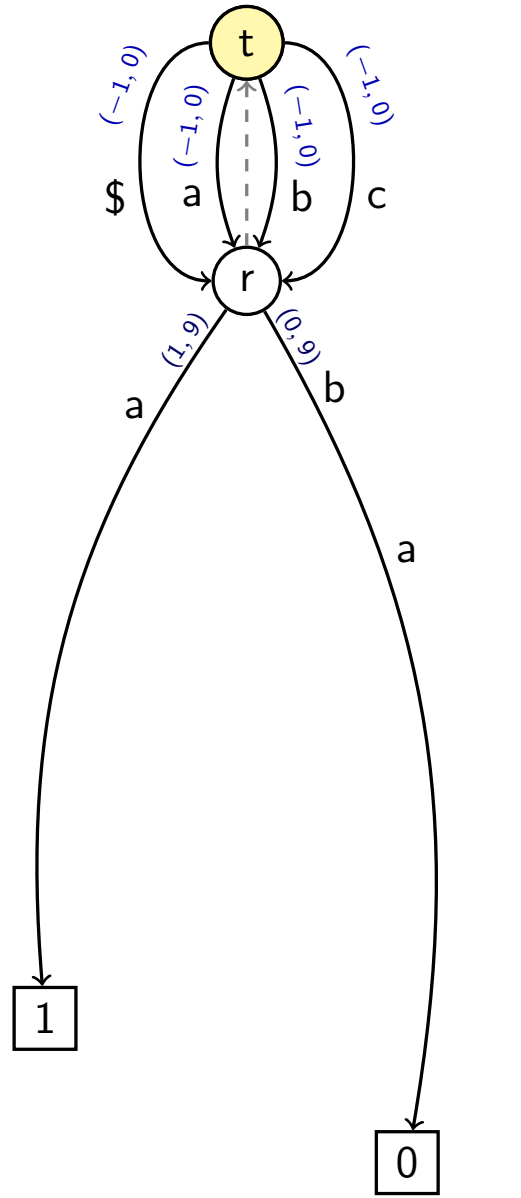
T	$babacbab\$$	*endPoint*	false
i	1	k	1
s	r	r	r
old_r	r		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

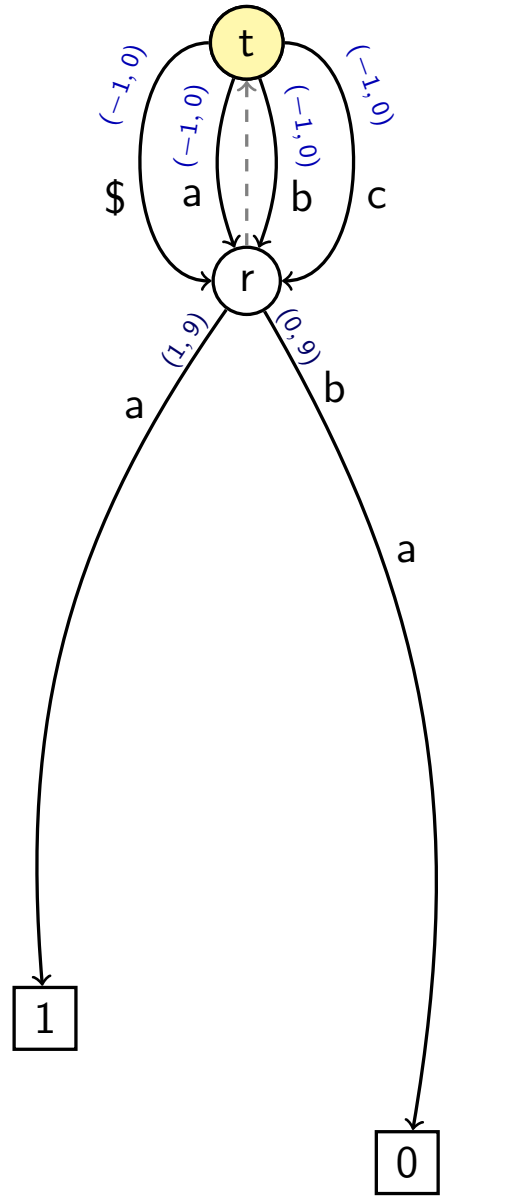
T	$babacbab\$$	$endPoint$	false
i	0	k	1
s	t	r	r
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

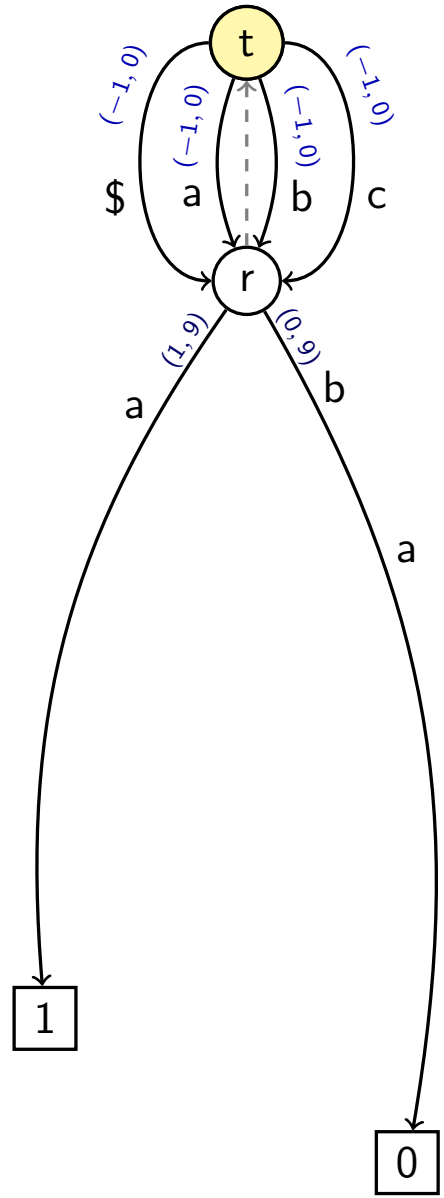
T	$babacbab\$$	$endPoint$	false
i	0	k	1
s	t	r	r
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

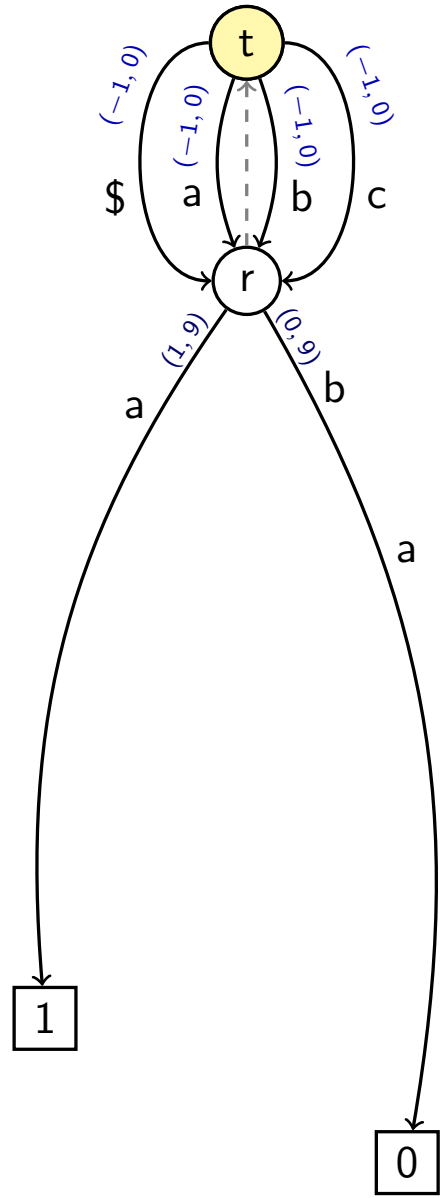
T	$babacbab\$$	$endPoint$	false
i	1	k	1
s	t	r	r
old_r	r		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

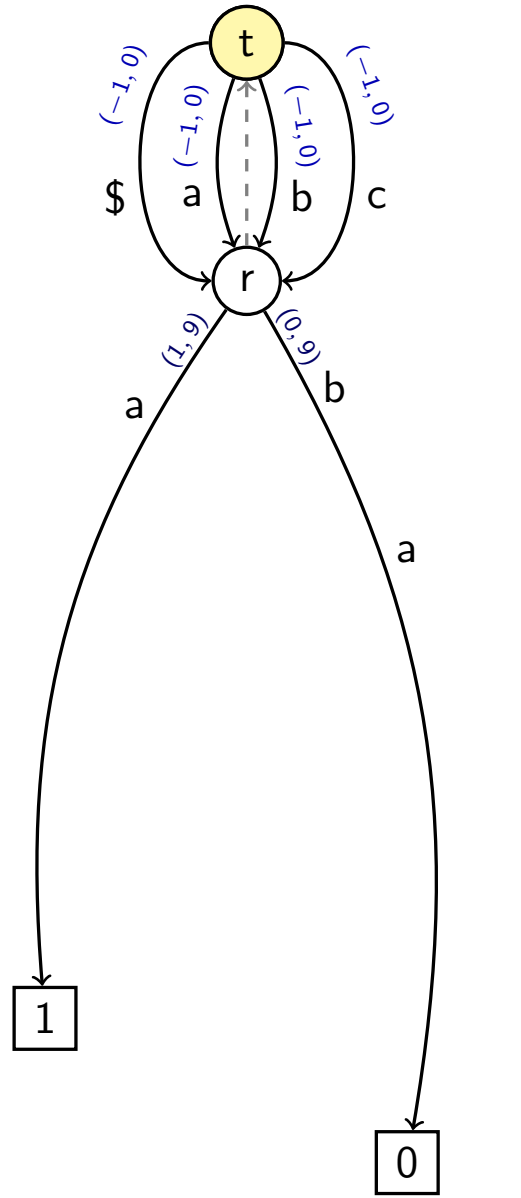
T	$babacbab\$$	$endPoint$	false
i	1	k	1
s	t	r	r
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

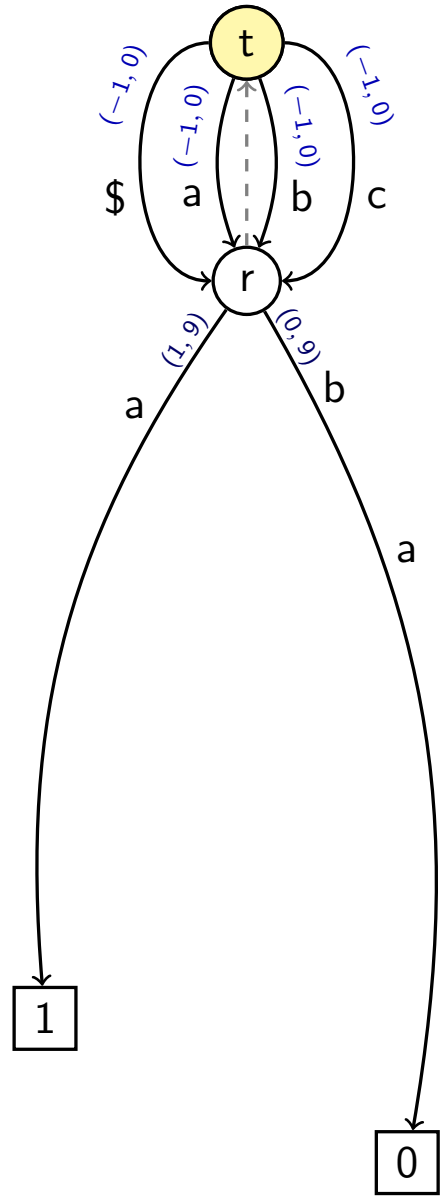
T	$babacbab\$$	$endPoint$	false
i	1	k	1
s	t	r	r
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

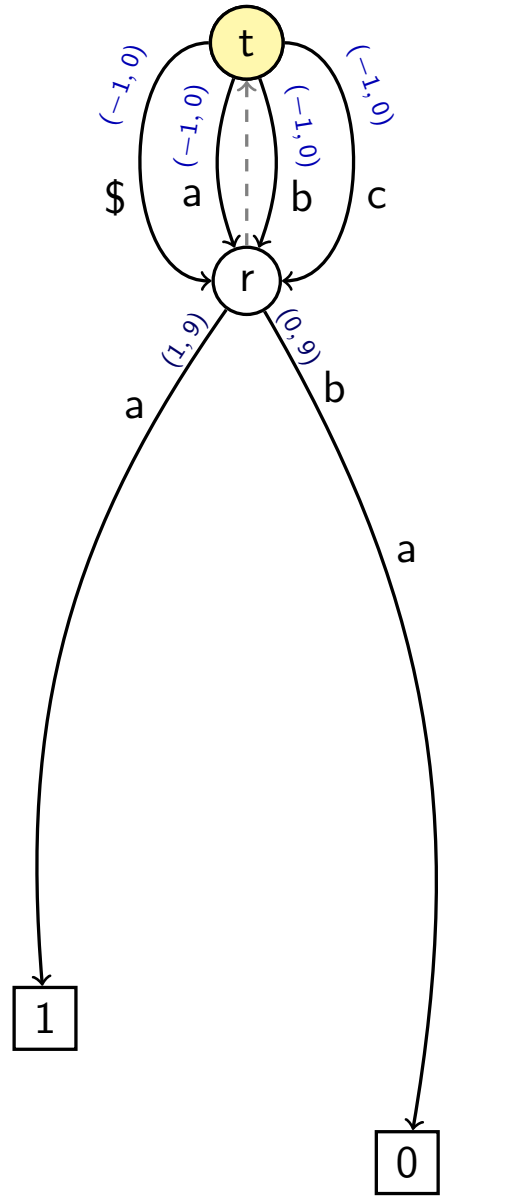
T	$babacbab\$$	$endPoint$	true
i	1	k	1
s	t	r	t
old_r	r		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

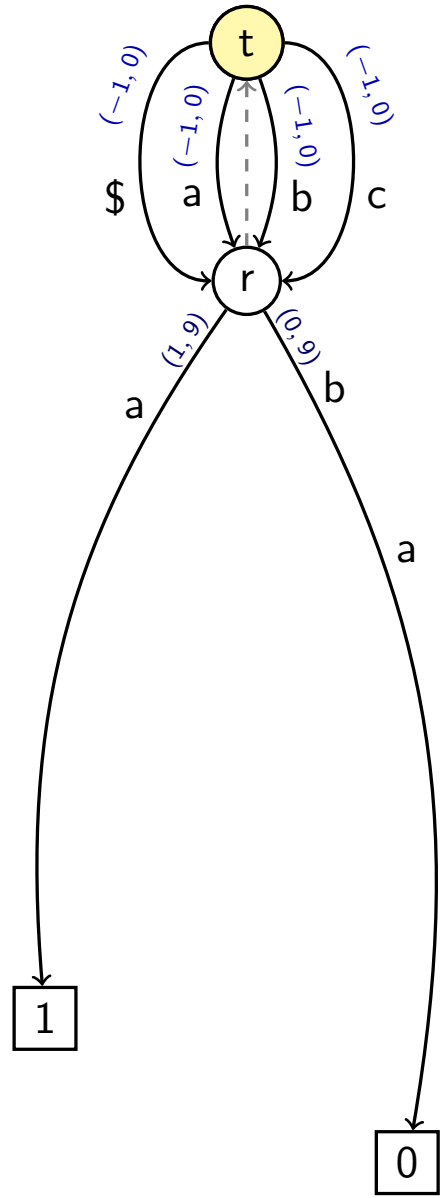
T	$babacbab\$$	$endPoint$	true
i	1	k	1
s	t	r	t
old_r	r	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

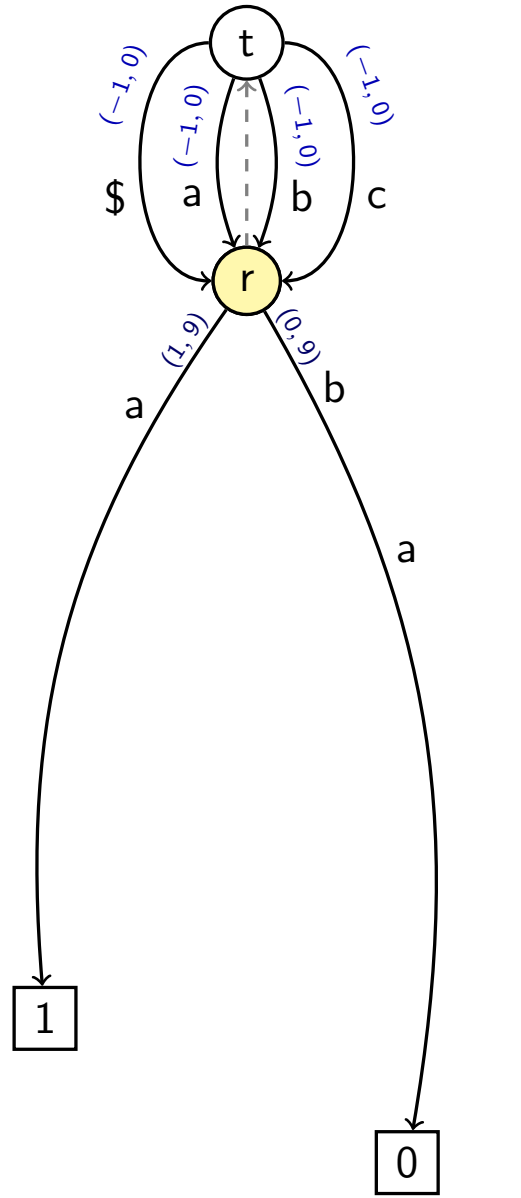
T	$babacbab\$$	$endPoint$	true
i	1	k	1
s	t	r	t
old_r	r	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

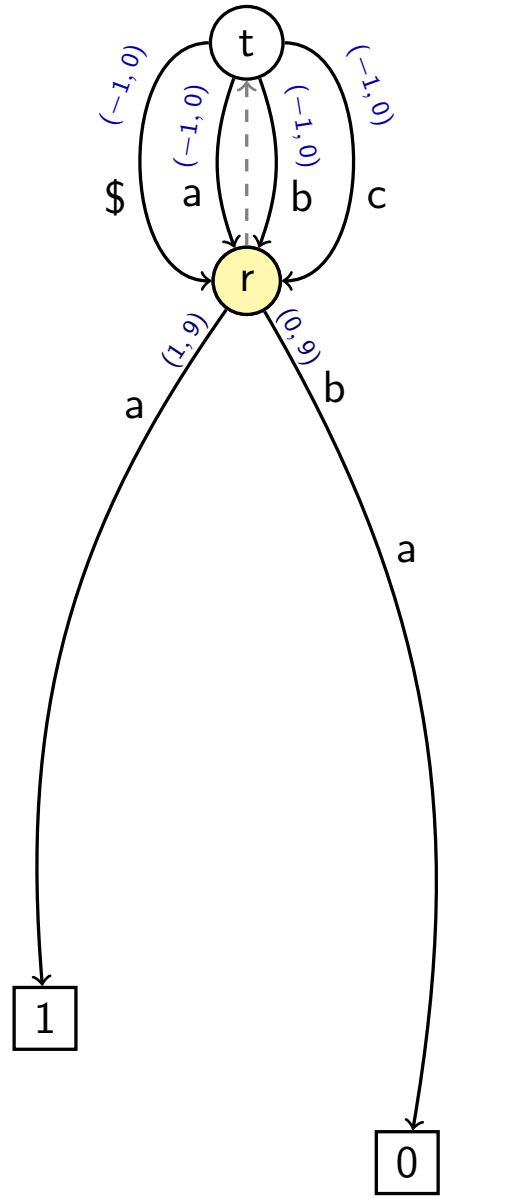
T	$babacbab\$$	$endPoint$	true
i	1	k	2
s	r	r	t
old_r	r	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

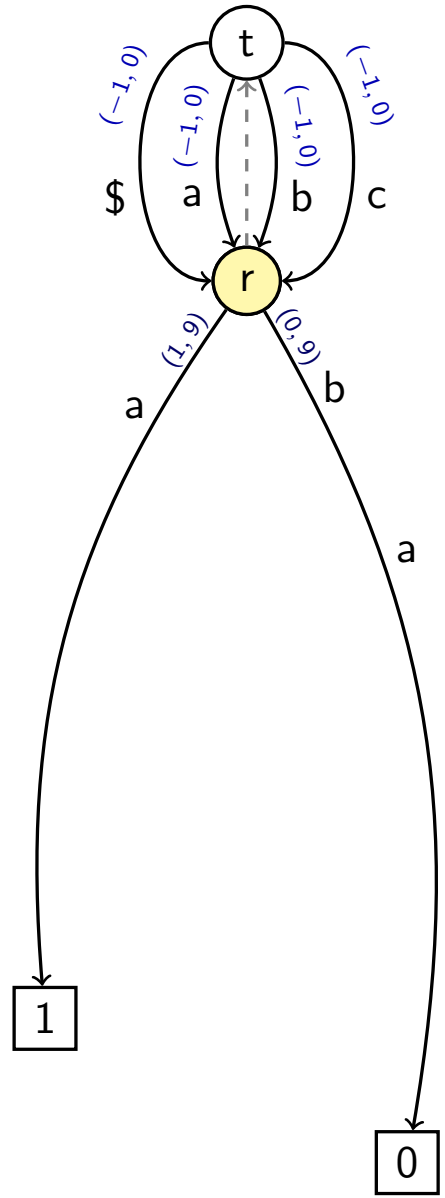
T	$babacbab\$$	$endPoint$	true
i	1	k	2
s	r	r	t
old_r	r	$s1$	r
b	-1	e	0



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

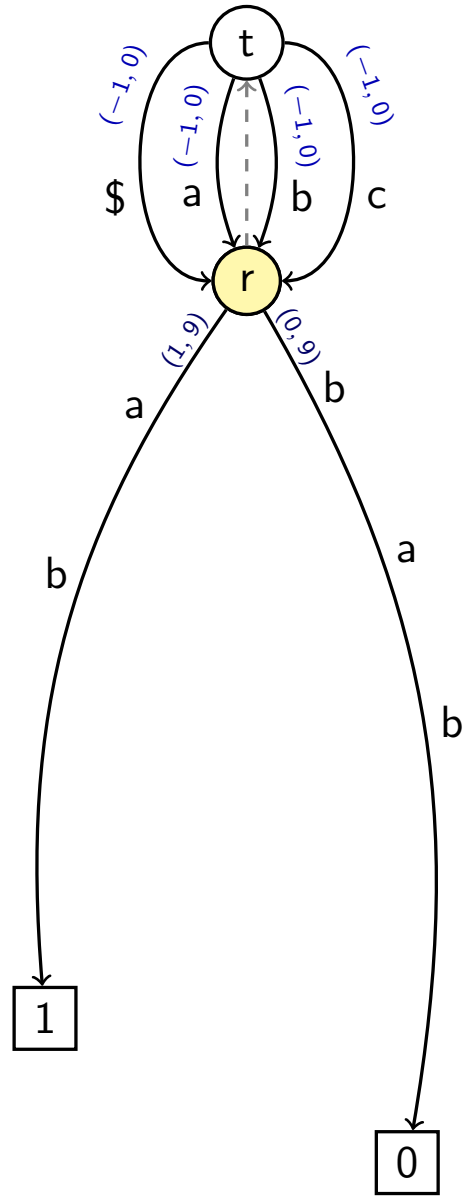
T	$babacbab\$$	*endPoint*	true
i	1	k	2
s	r	r	t
old_r	r		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

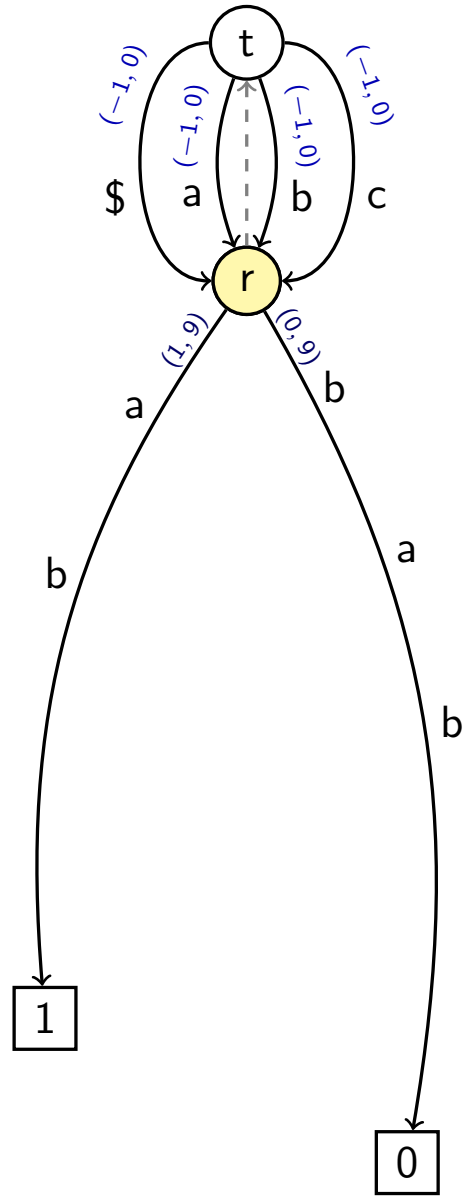
T	$babacbab\$$	$endPoint$	-
i	2	k	2
s	r	r	—
old_r	—		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

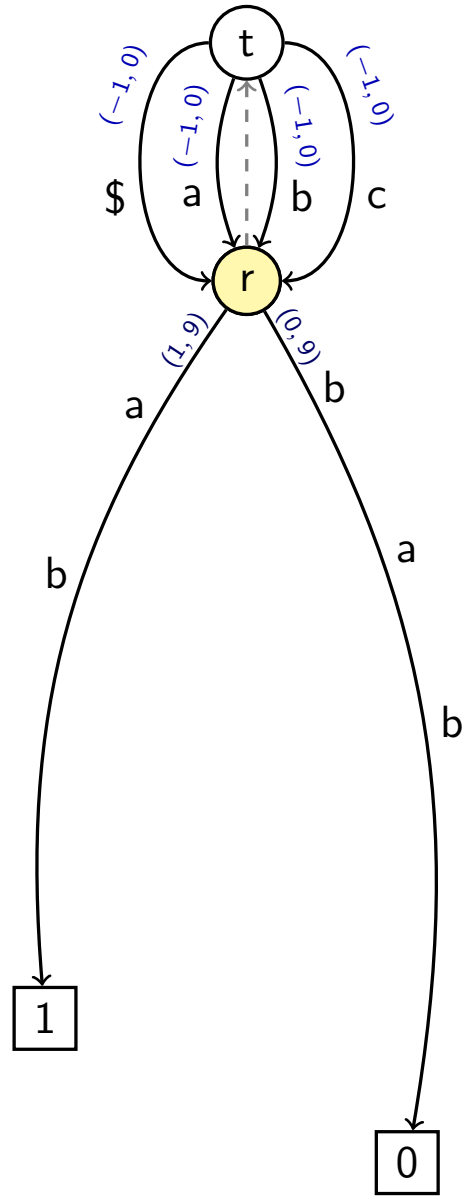
T	$babacbab\$$	$endPoint$	-
i	2	k	2
s	r	r	—
old_r	—	$s1$	—
b	—	e	—



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

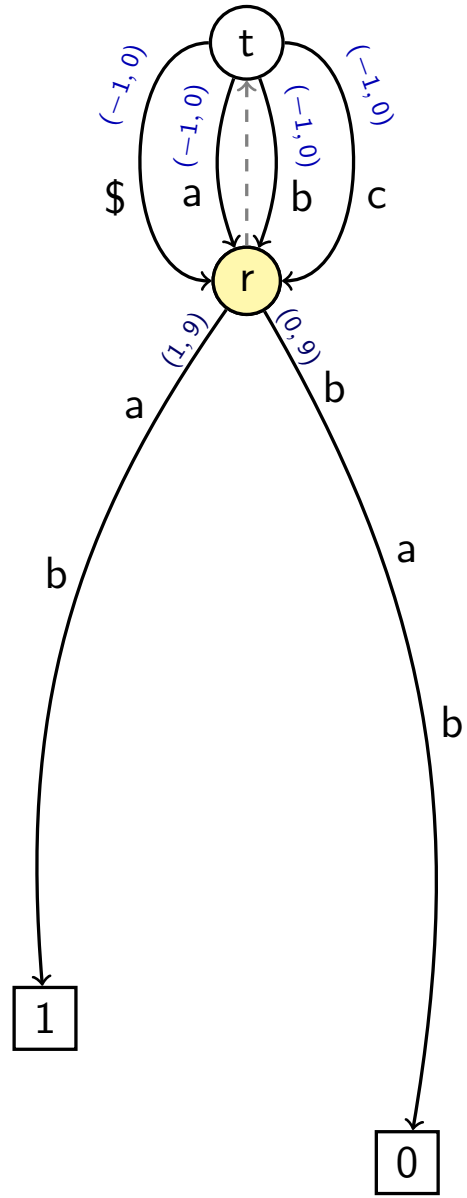
T	$babacbab\$$	$endPoint$	-
i	2	k	2
s	r	r	—
old_r	—	$s1$	—
b	—	e	—



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

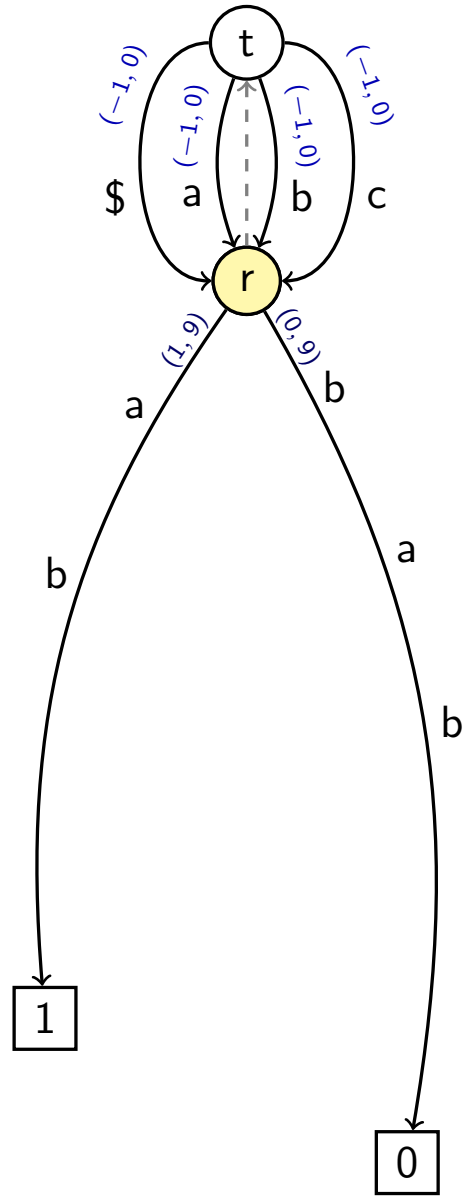
T	$babacbab\$$	$endPoint$	true
i	2	k	2
s	r	r	r
old_r	—		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

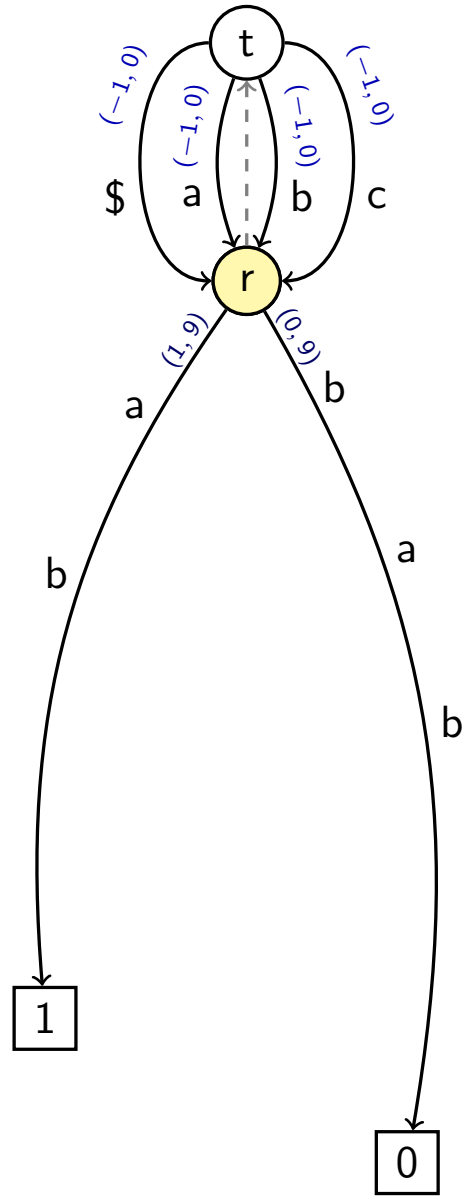
T	$babacbab\$$	$endPoint$	true
i	2	k	2
s	r	r	r
old_r	—	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

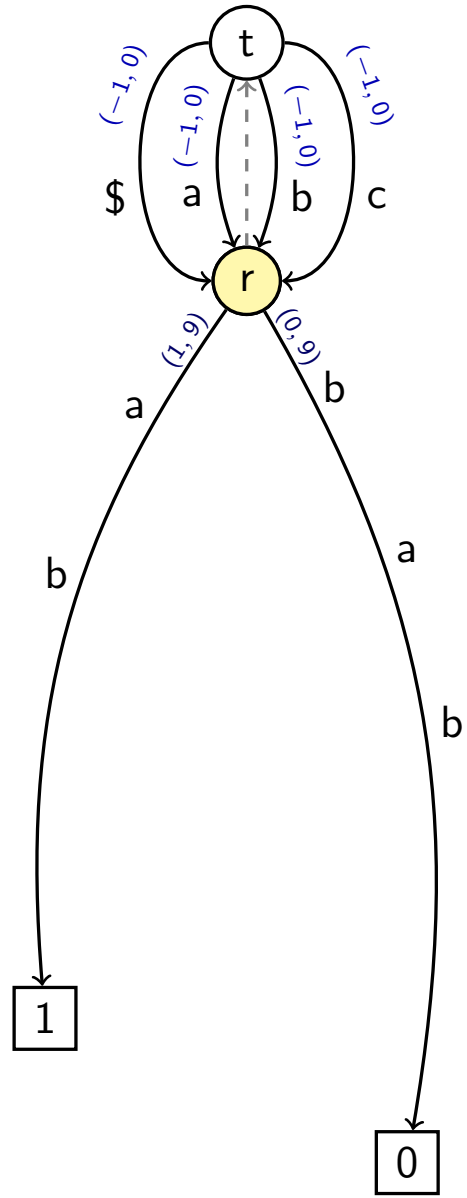
T	$babacbab\$$	$endPoint$	true
i	2	k	2
s	r	r	r
old_r	–	$s1$	0
b	0	e	9



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

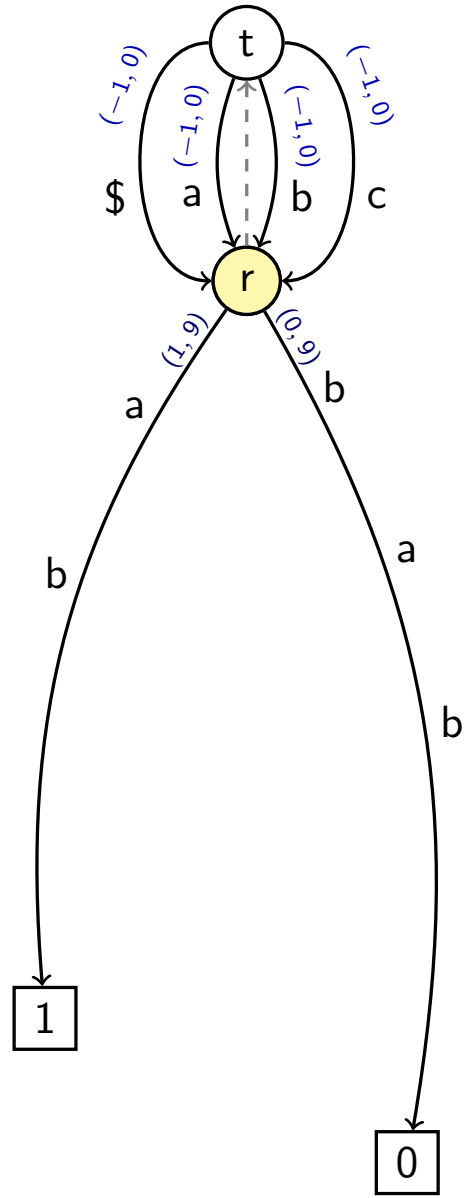
T	$babacbab\$$	$endPoint$	true
i	2	k	2
s	r	r	r
old_r	–	$s1$	0
b	0	e	9



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

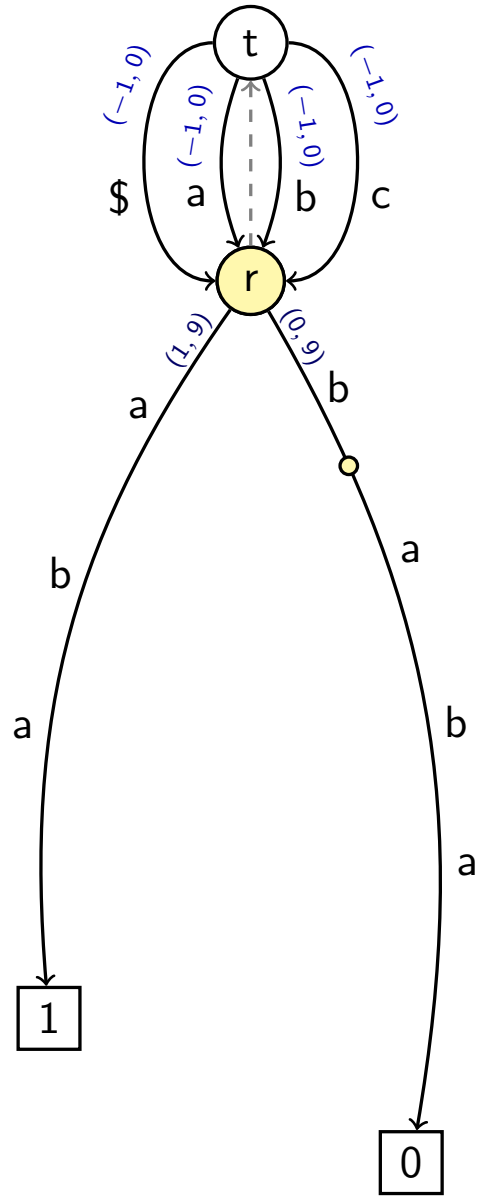
T	$babacbab\$$	$endPoint$	-
i	2	k	2
s	r	r	r
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

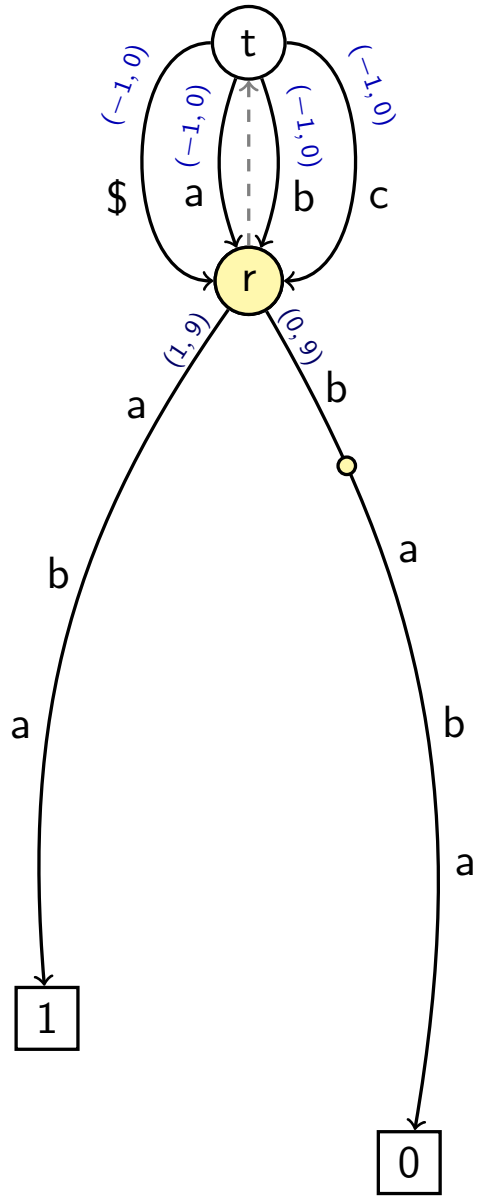
T	*babacbab$*	*endPoint*	-
i	3	*k*	2
s	r	*r*	—
old_r	—		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

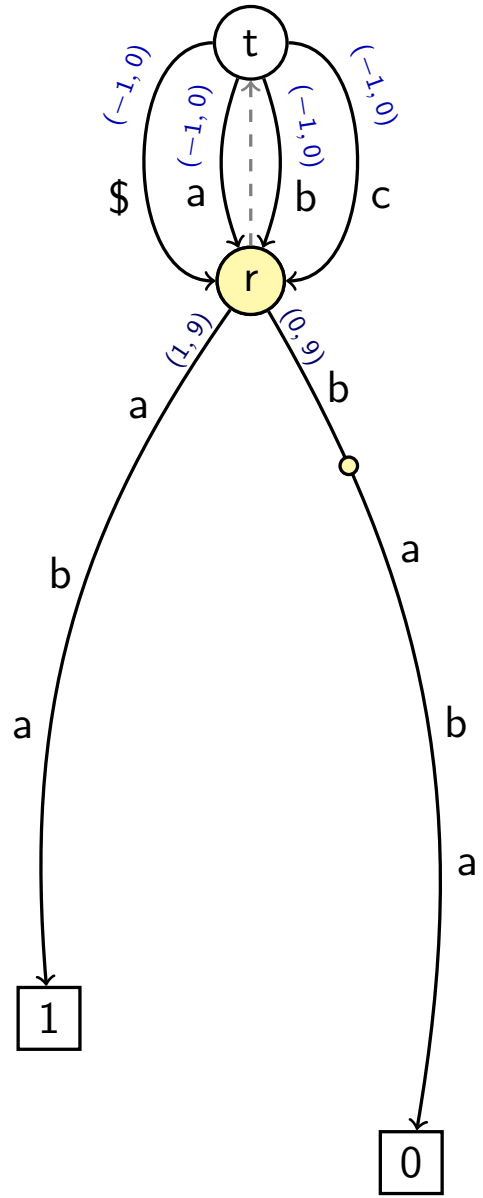
T	$babacbab\$$	$endPoint$	-
i	3	k	2
s	r	r	—
old_r	—	$s1$	—
b	—	e	—



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

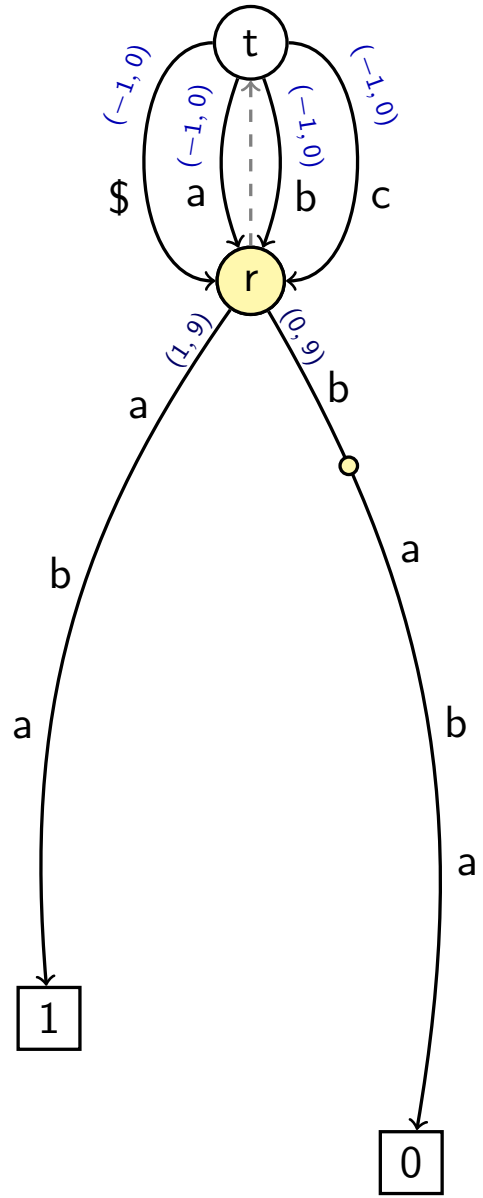
T	$babacbab\$$	$endPoint$	-
i	3	k	2
s	r	r	—
old_r	—	$s1$	0
b	0	e	9



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

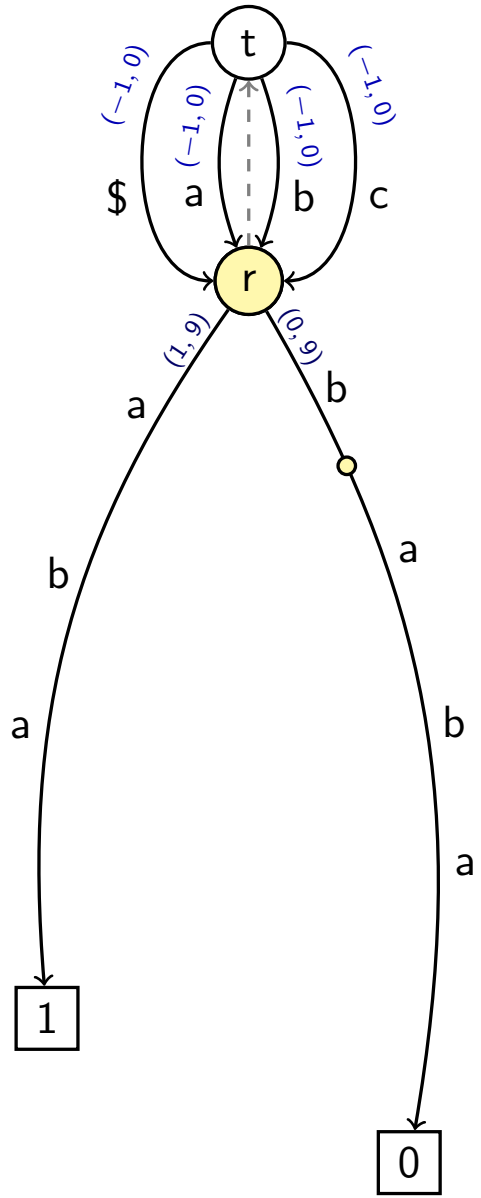
T	$babacbab\$$	*endPoint*	true
i	3	k	2
s	r	r	r
old_r	—		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

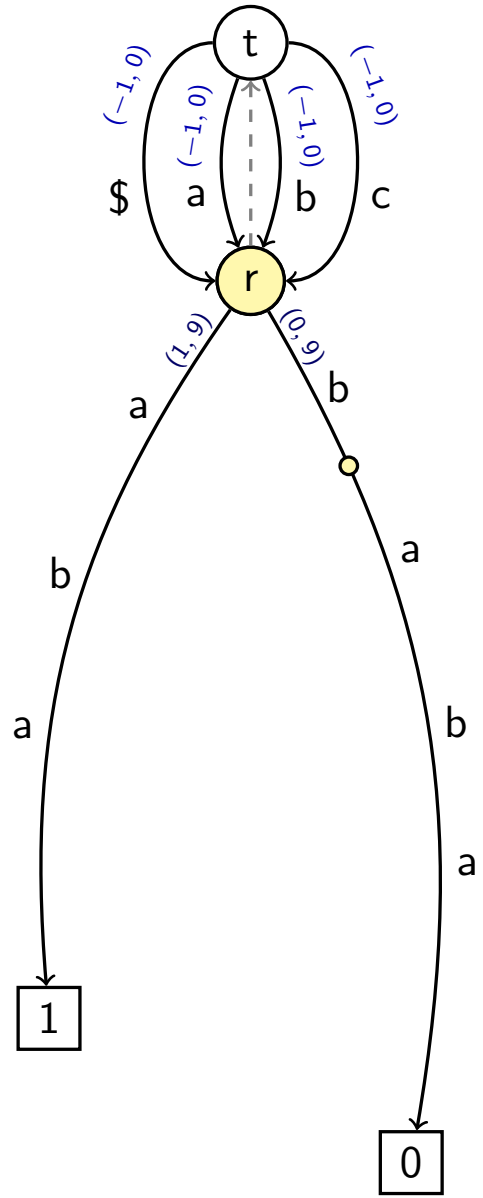
T	$babacbab\$$	$endPoint$	true
i	3	k	2
s	r	r	r
old_r	—	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

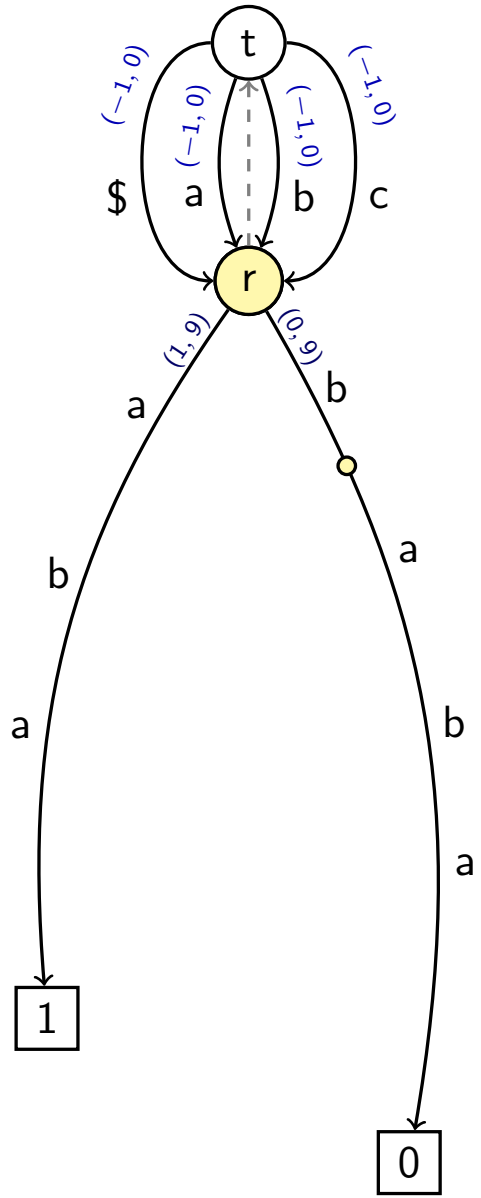
T	$babacbab\$$	$endPoint$	true
i	3	k	2
s	r	r	r
old_r	–	$s1$	0
b	0	e	9



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

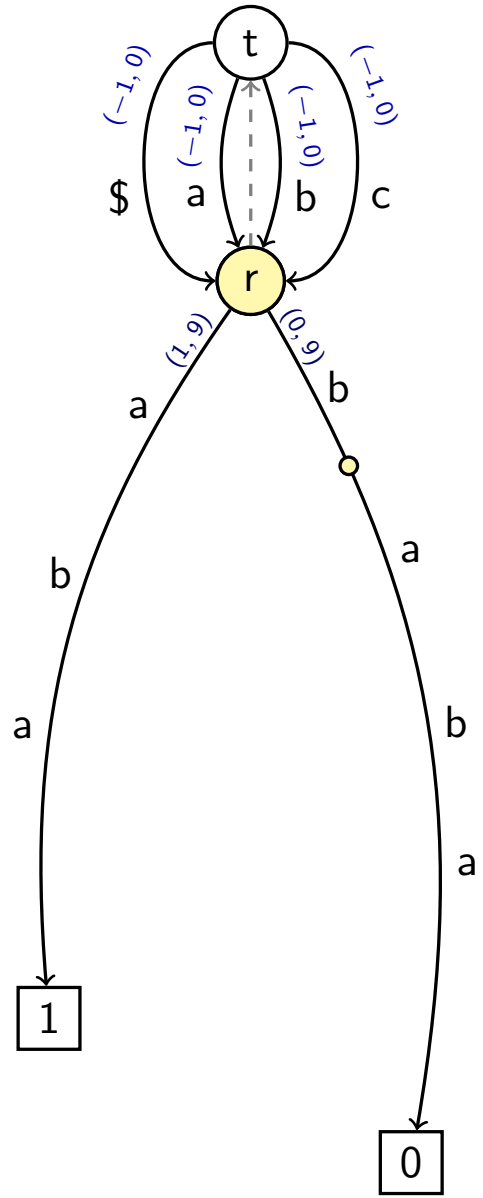
T	$babacbab\$$	$endPoint$	true
i	3	k	2
s	r	r	r
old_r	–	$s1$	0
b	0	e	9



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

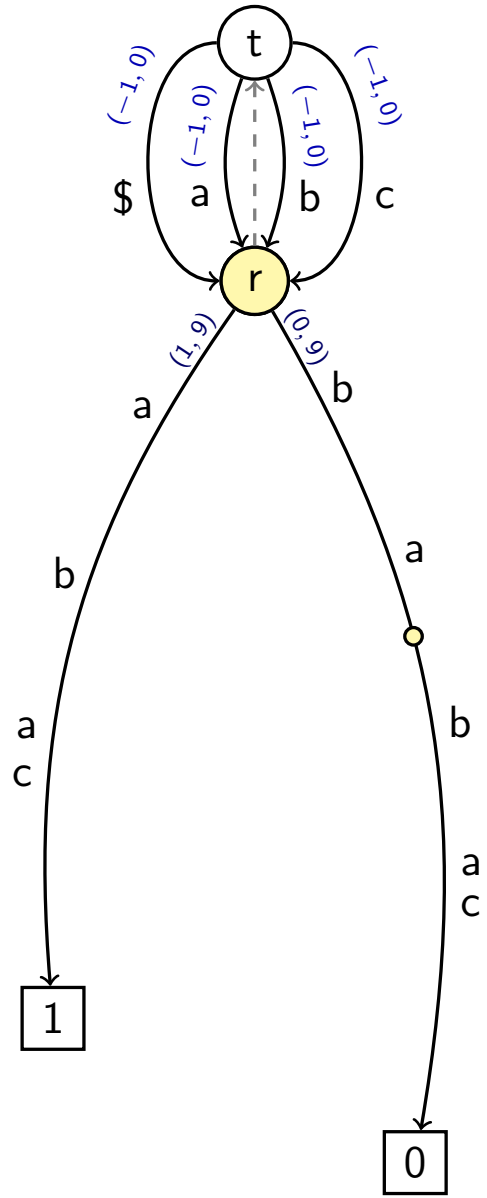
T	*babacbab$*	*endPoint*	true
i	3	*k*	2
s	r	*r*	r
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

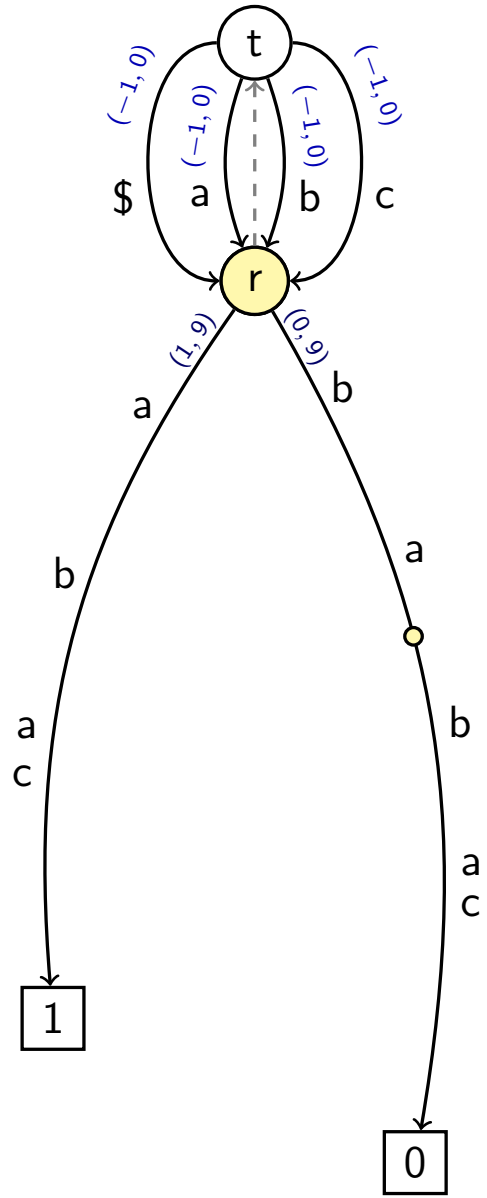
T	$babacbab\$$	$endPoint$	-
i	4	k	2
s	r	r	—
old_r	—		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

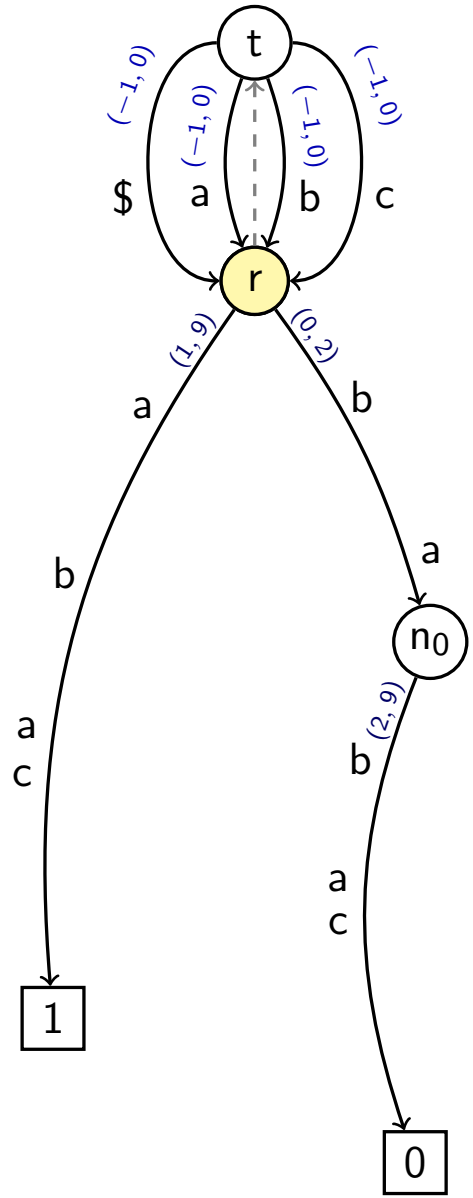
T	$babacbab\$$	$endPoint$	-
i	4	k	2
s	r	r	—
old_r	—	$s1$	0
b	0	e	9



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

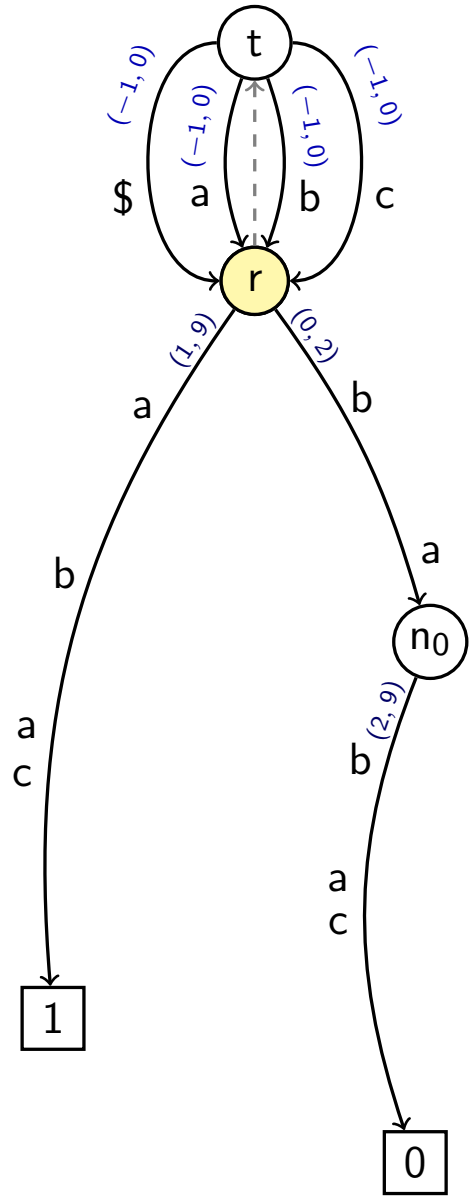
T	$babacbab\$$	$endPoint$	-
i	4	k	2
s	r	r	n_0
old_r	—	$s1$	0
b	0	e	9



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

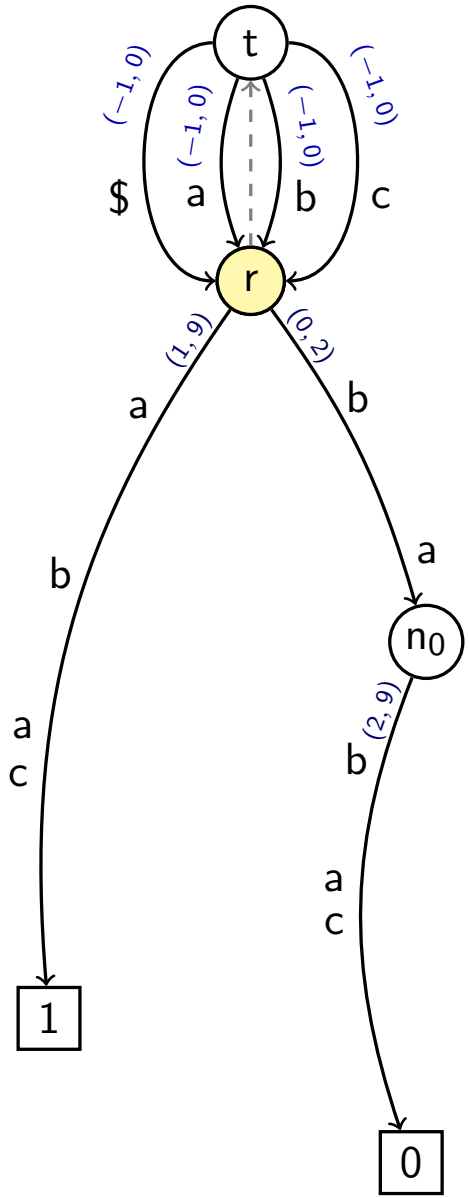
T	$babacbab\$$	$endPoint$	-
i	4	k	2
s	r	r	n_0
old_r	—	$s1$	0
b	0	e	9



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

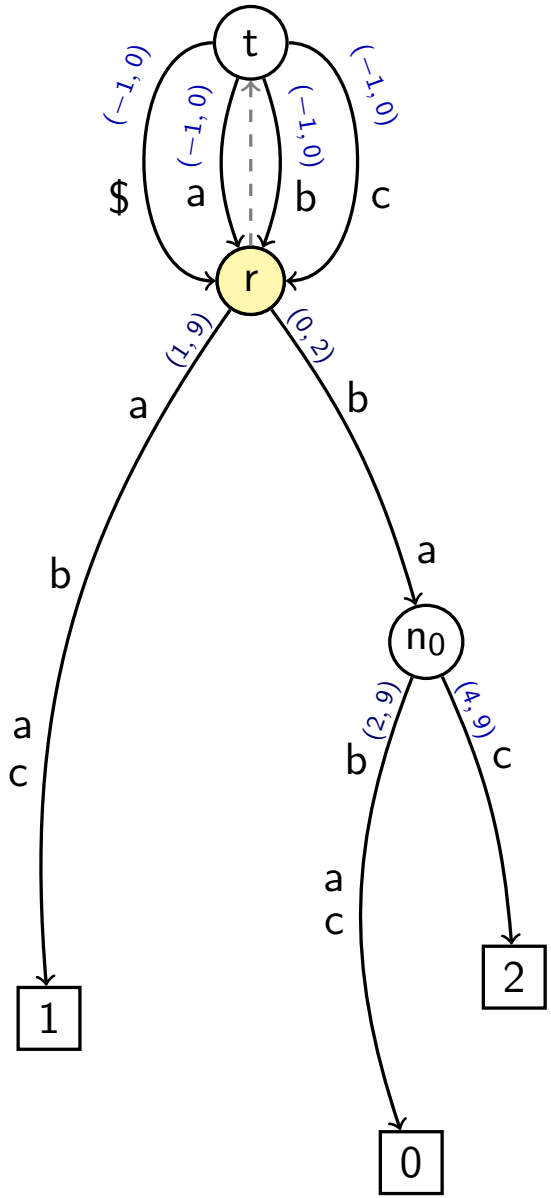
T	$babacbab\$$	$endPoint$	false
i	4	k	2
s	r	r	n_0
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

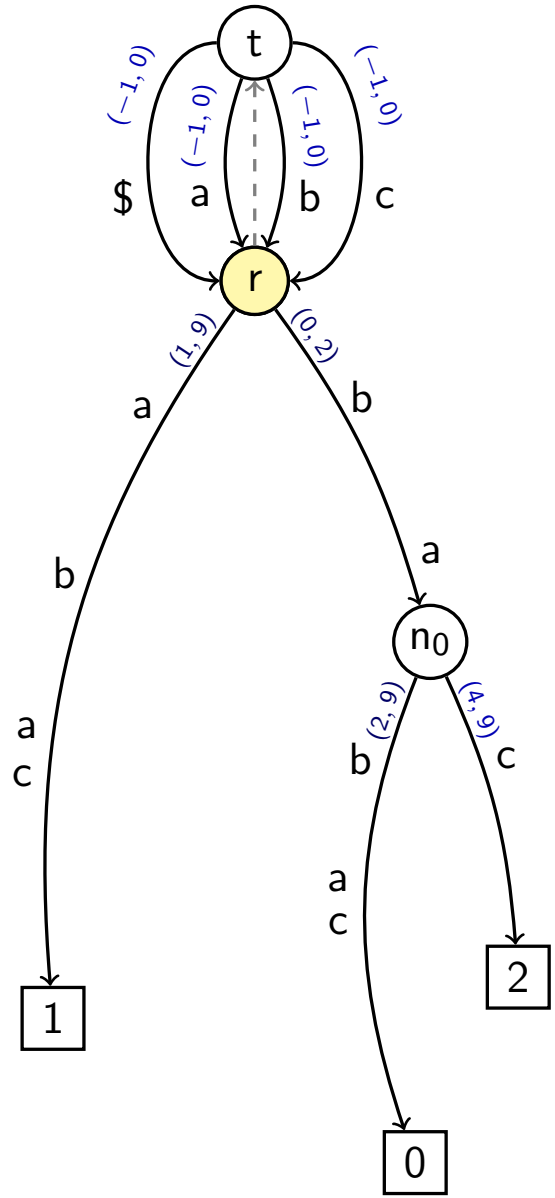
T	$babacbab\$$	$endPoint$	false
i	4	k	2
s	r	r	n_0
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

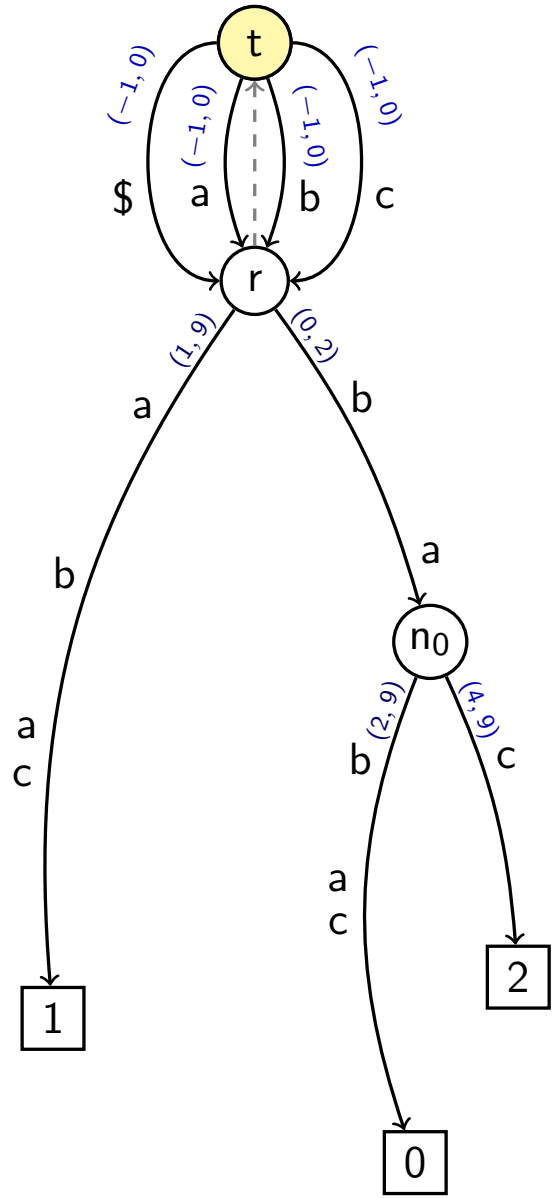
T	$babacbab\$$	$endPoint$	false
i	4	k	2
s	r	r	n_0
old_r	n_0		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

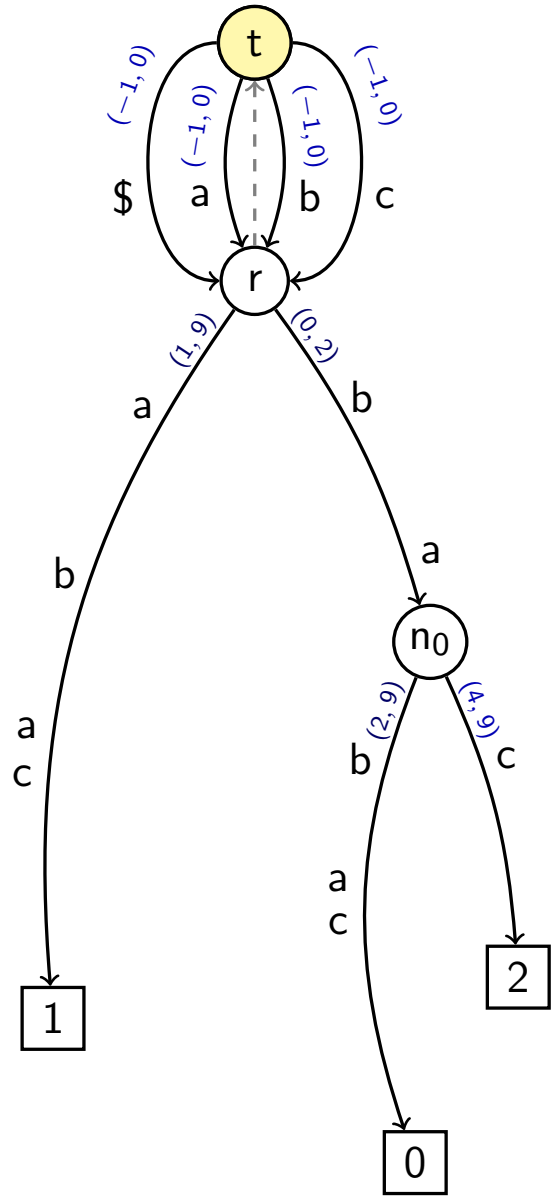
T	$babacbab\$$	$endPoint$	false
i	3	k	2
s	t	r	n_0
old_r	n_0	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

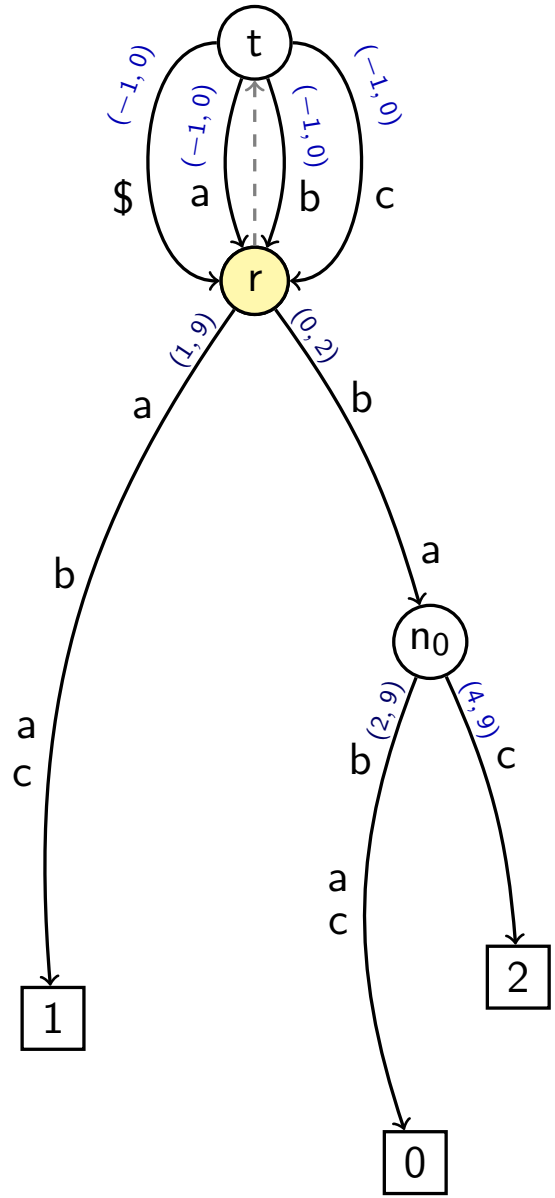
T	$babacbab\$$	*endPoint*	false
i	3	k	2
s	t	r	n_0
old_r	n_0	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

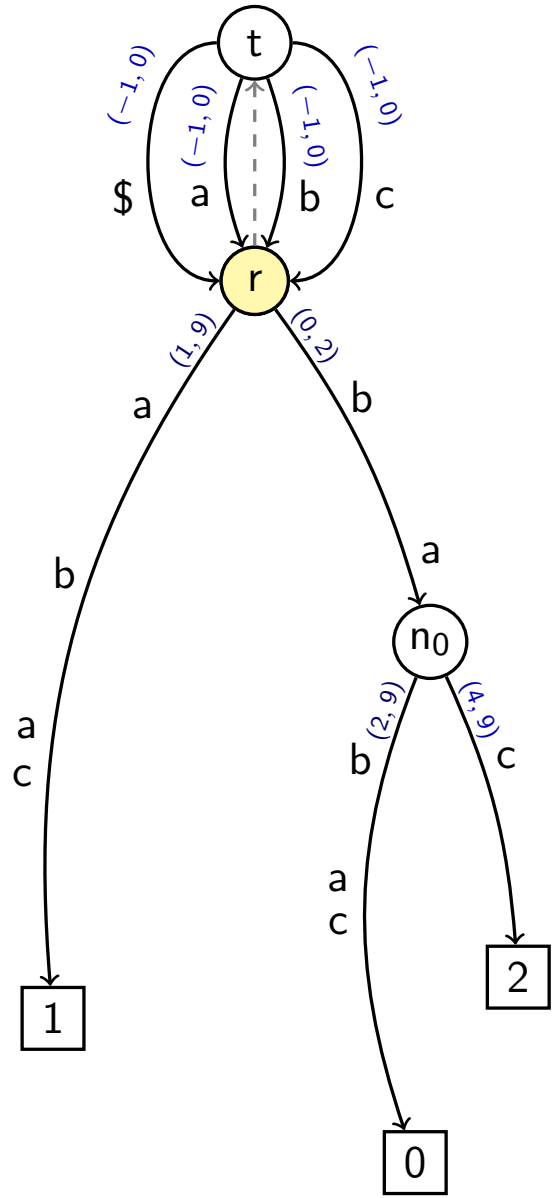
T	$babacbab\$$	$endPoint$	false
i	3	k	3
s	r	r	n_0
old_r	n_0	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

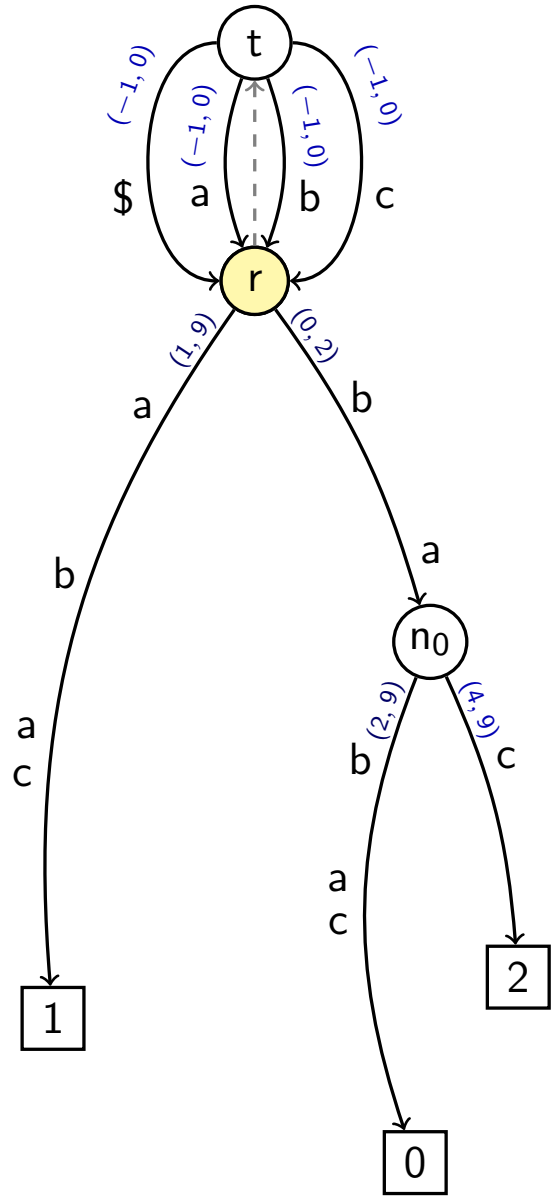
T	$babacbab\$$	$endPoint$	false
i	3	k	3
s	r	r	n_0
old_r	n_0	$s1$	1
b	1	e	9



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

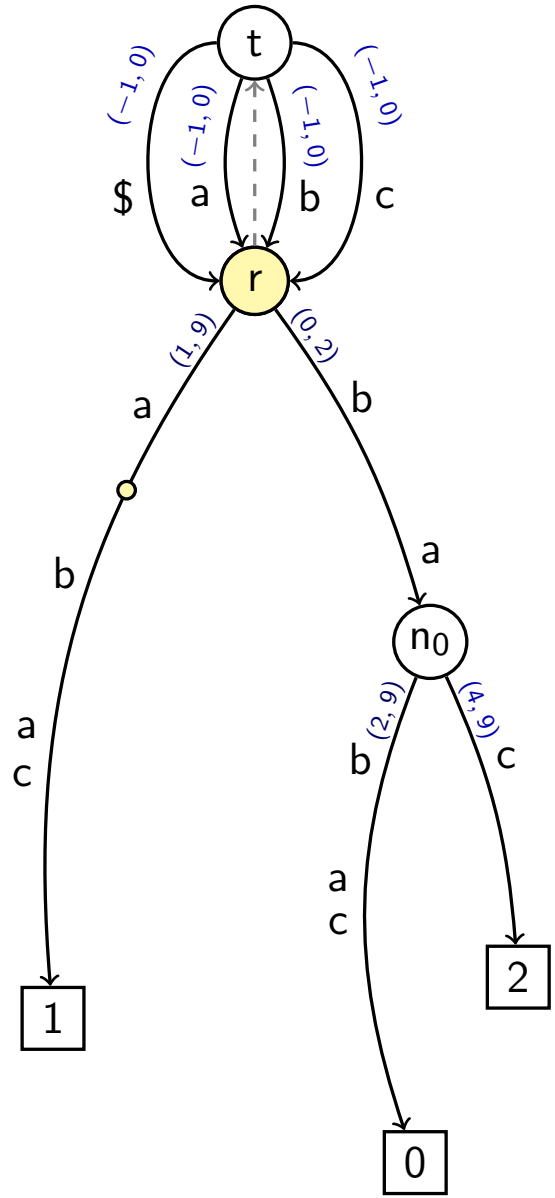
T	$babacbab\$$	$endPoint$	false
i	3	k	3
s	r	r	n_0
old_r	n_0	$s1$	1
b	1	e	9



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

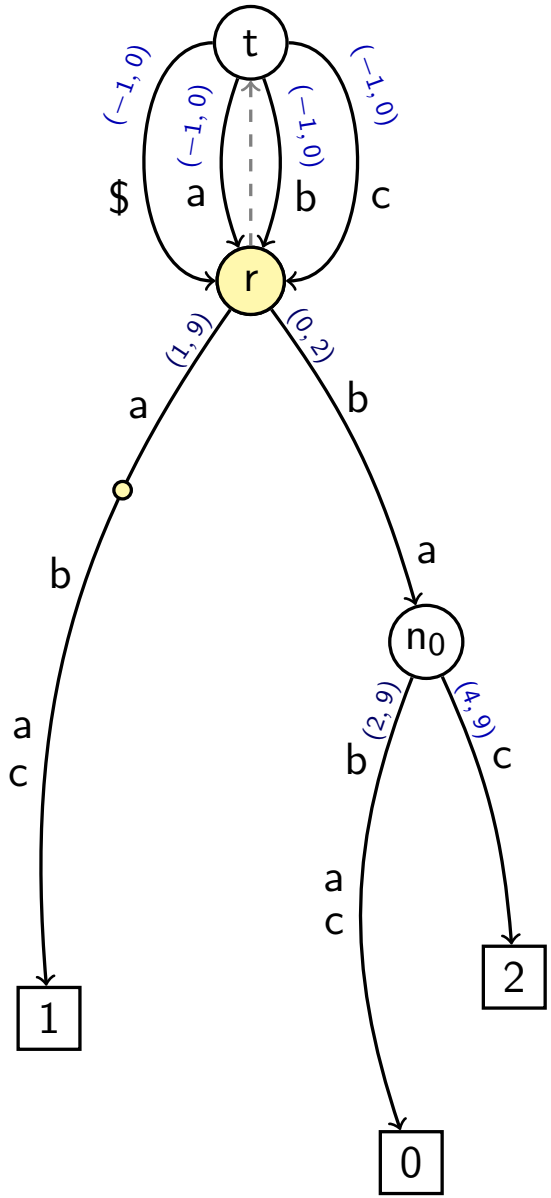
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_0
old_r	n_0		



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

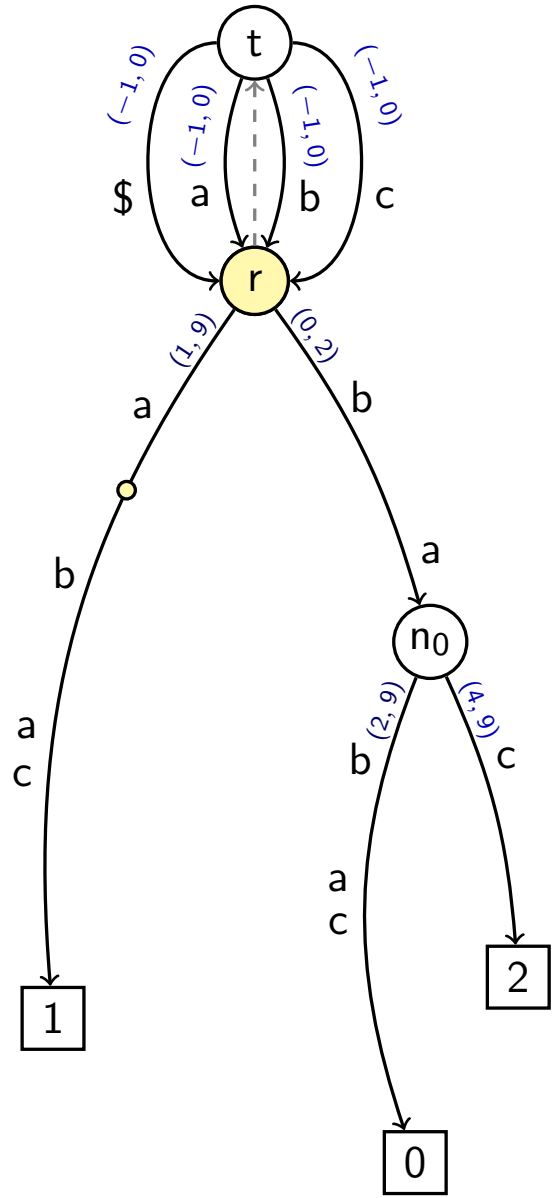
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_0
old_r	n_0	$s1$	—
b	—	e	—



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

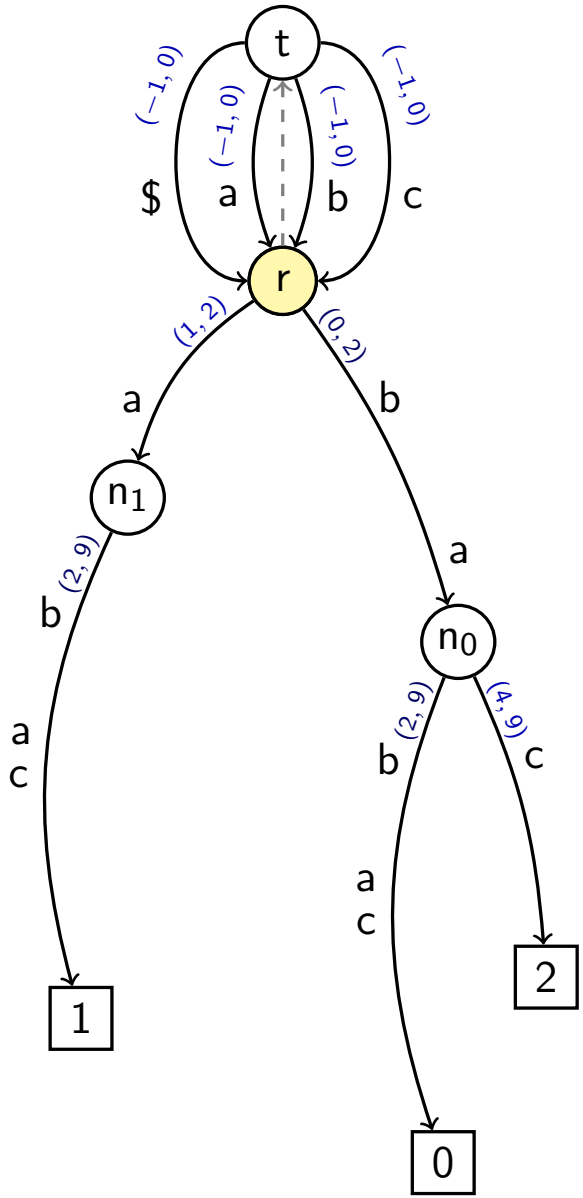
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_0
old_r	n_0	$s1$	1
b	1	e	9



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

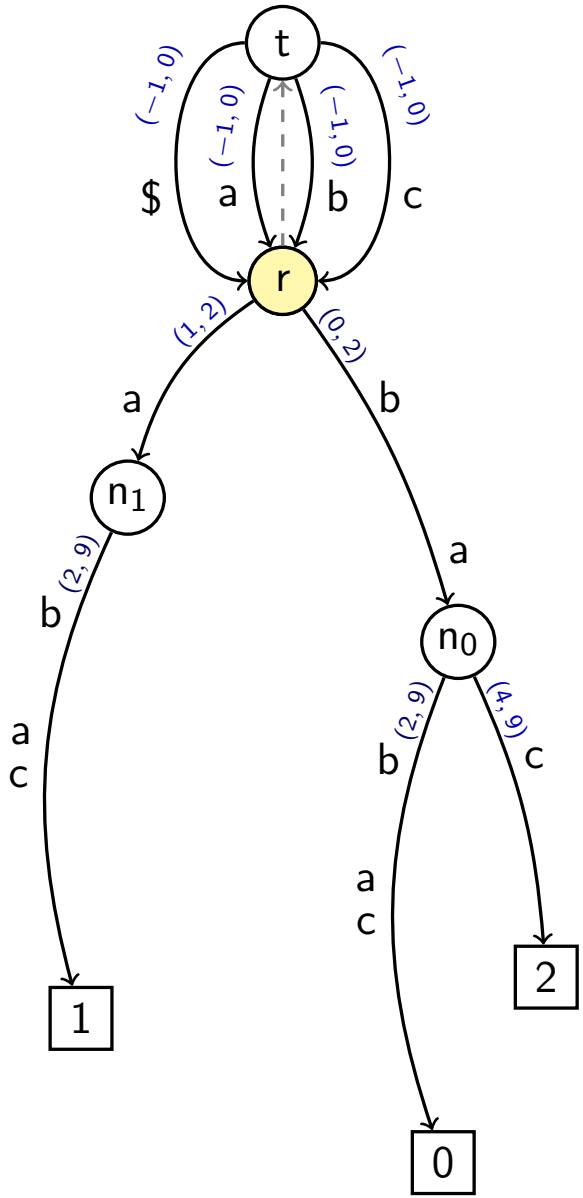
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_1
old_r	n_0	$s1$	1
b	1	e	9



Implementierung

```
1  def test_and_split(s, k, i, T):
2      if k == i:
3          return T[i] in s.targets, s
4
5      s1, b, e = s.targets[T[k]]
6      active = i - k + b
7      if T[i] == T[active]:
8          return True, s
9
10     r = STNode()
11     s.targets[T[b]] = (r, b, active)
12     r.targets[T[active]] = (s1, active, e)
13     return False, r
```

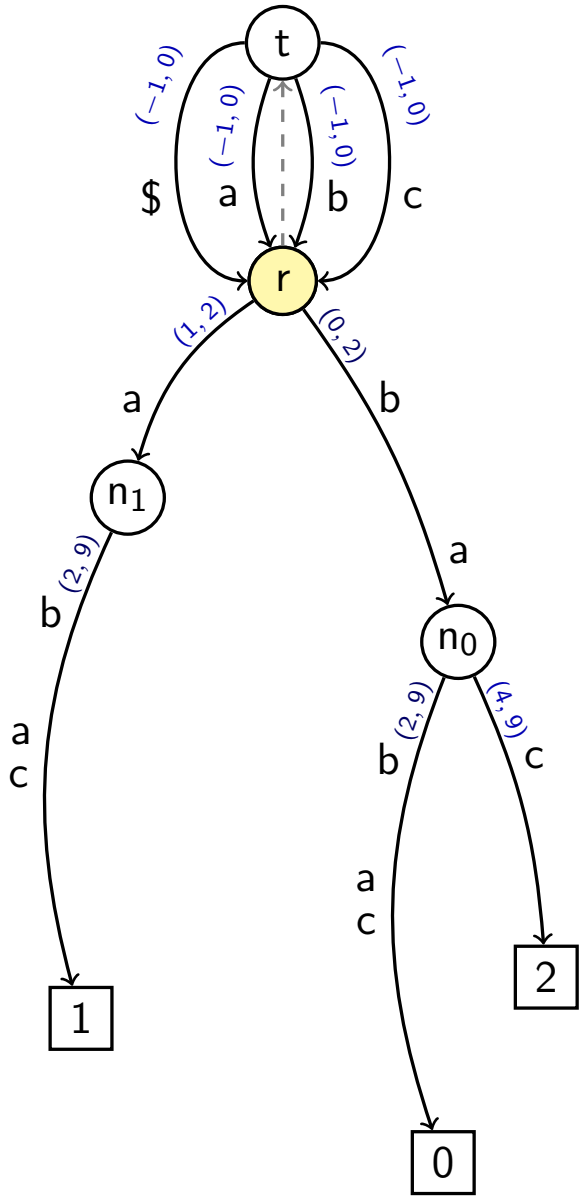
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_1
old_r	n_0	$s1$	1
b	1	e	9



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

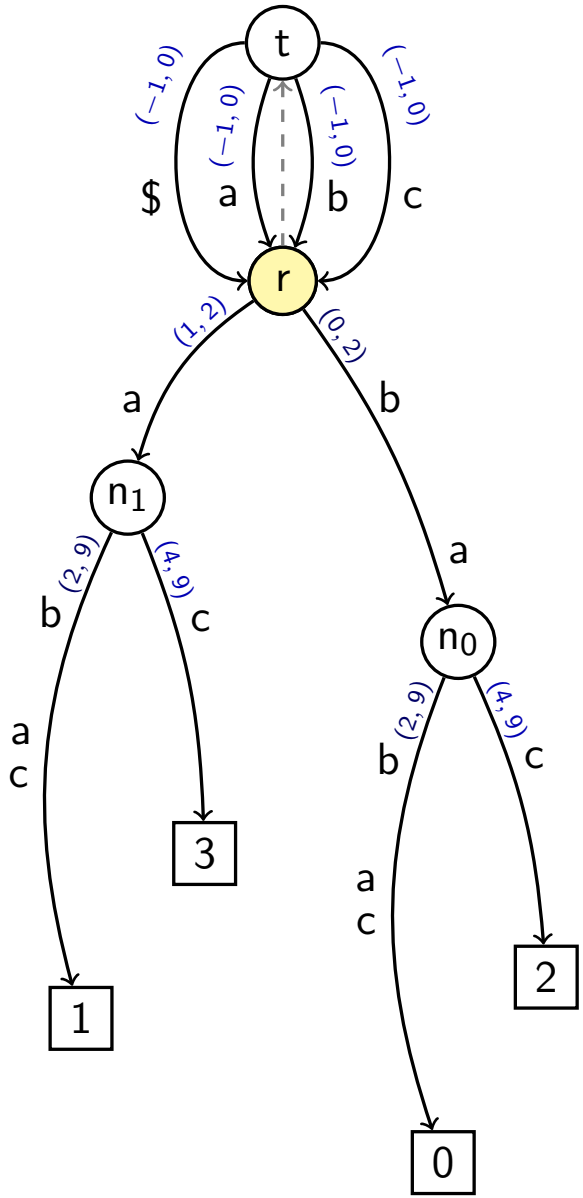
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_1
old_r	n_0		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

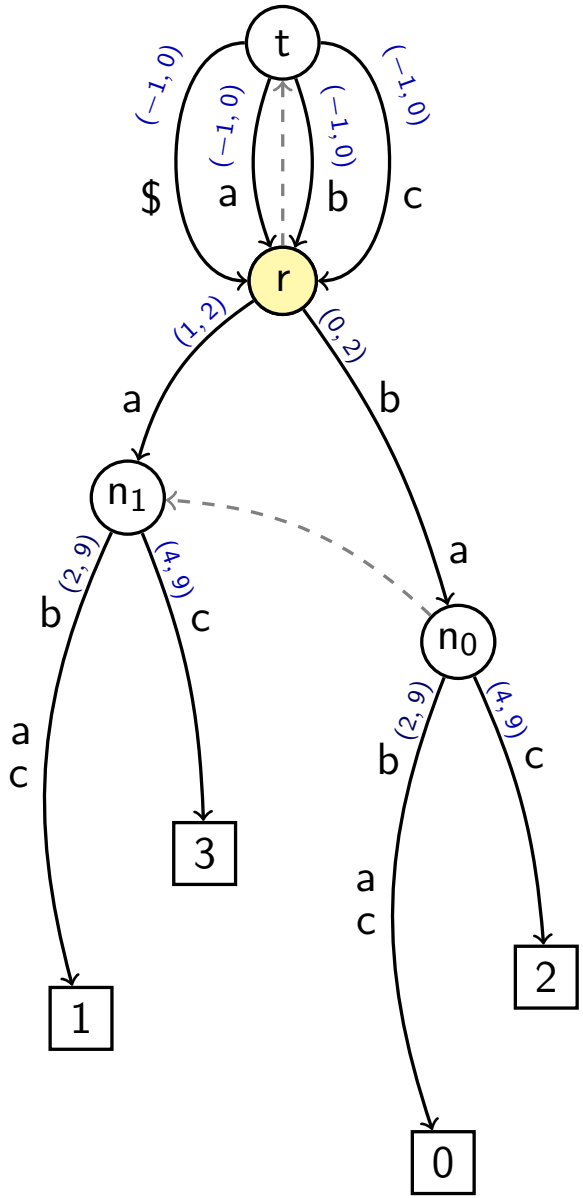
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_1
old_r	n_0		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

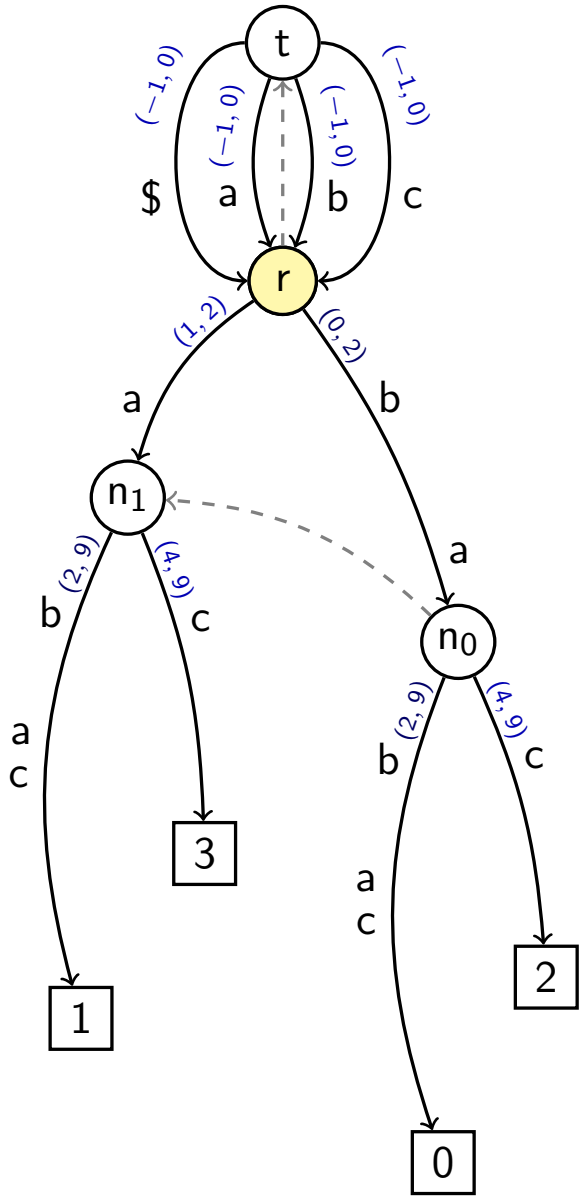
T	$babacbab\$$	$endPoint$	false
i	4	k	3
s	r	r	n_1
old_r	n_0		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

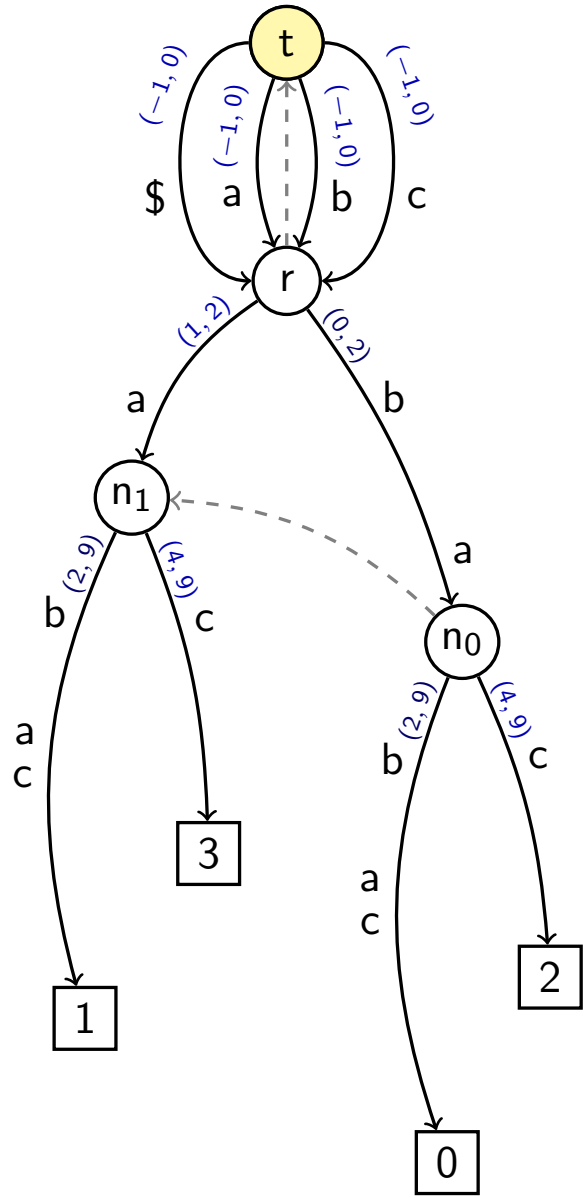
T	$babacbab\$$	*endPoint*	false
i	4	k	3
s	r	r	n_1
old_r	n_1		



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

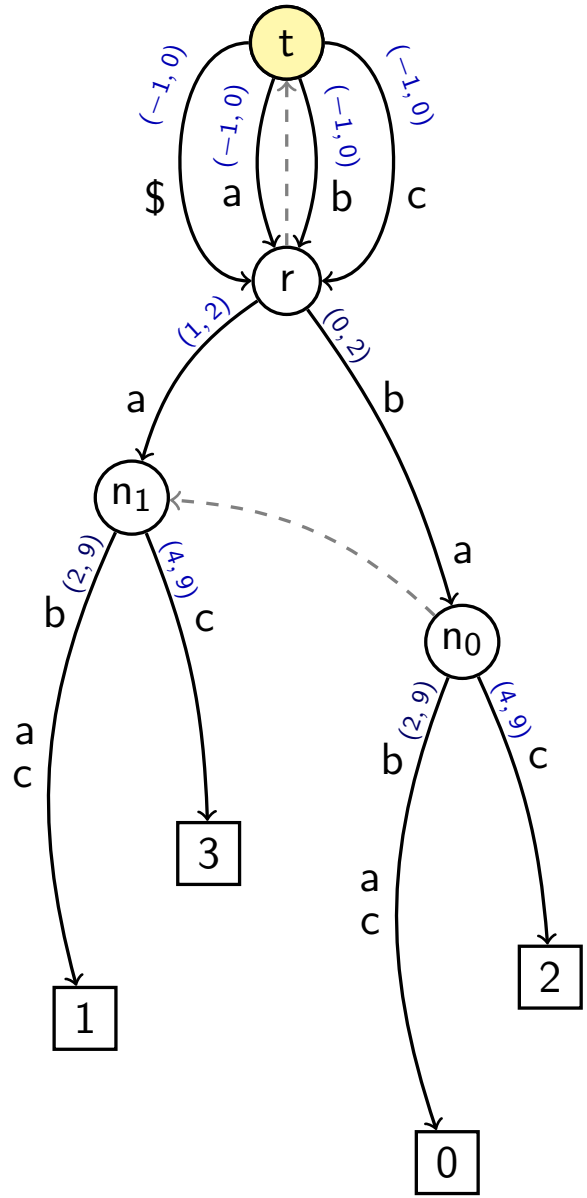
T	$babacbab\$$	$endPoint$	false
i	3	k	3
s	t	r	n_1
old_r	n_1	$s1$	—
b	—	e	—



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

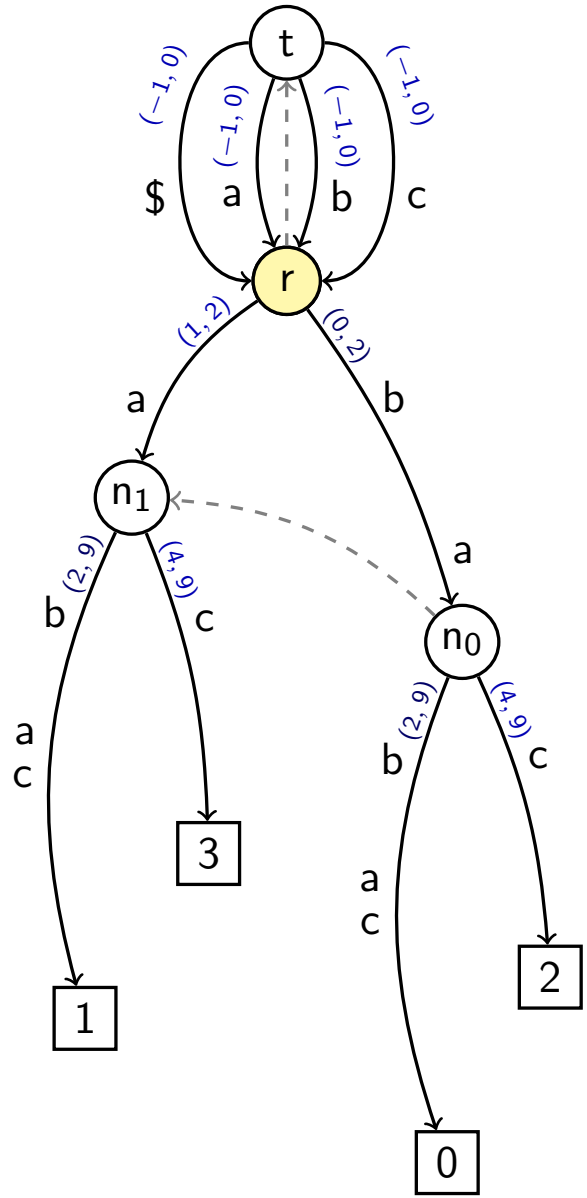
T	$babacbab\$$	*endPoint*	false
i	3	k	3
s	t	r	n_1
old_r	n_1	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

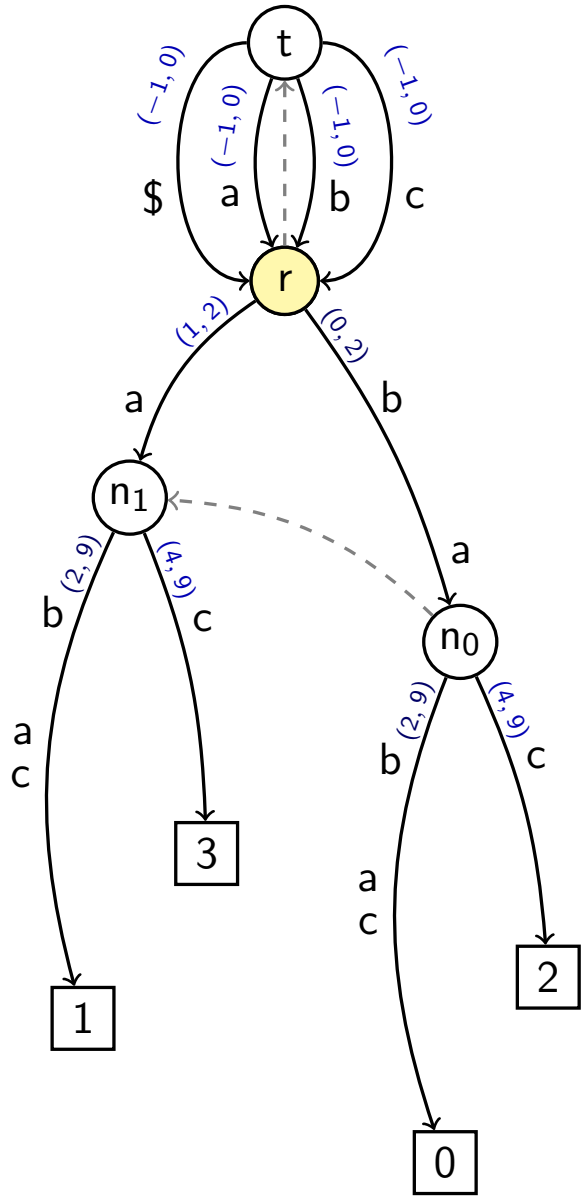
T	$babacbab\$$	*endPoint*	false
i	3	k	4
s	r	r	n_1
old_r	n_1	$s1$	r
b	-1	e	0



Implementierung

```
1  def canonize(s, k, i, T):
2      if i < k: return s, k
3
4      s1, b, e = s.targets[T[k]]
5      while e - b <= i + 1 - k:
6          k += e - b
7          s = s1
8          if k <= i:
9              s1, b, e = s.targets[T[k]]
10     return s, k
```

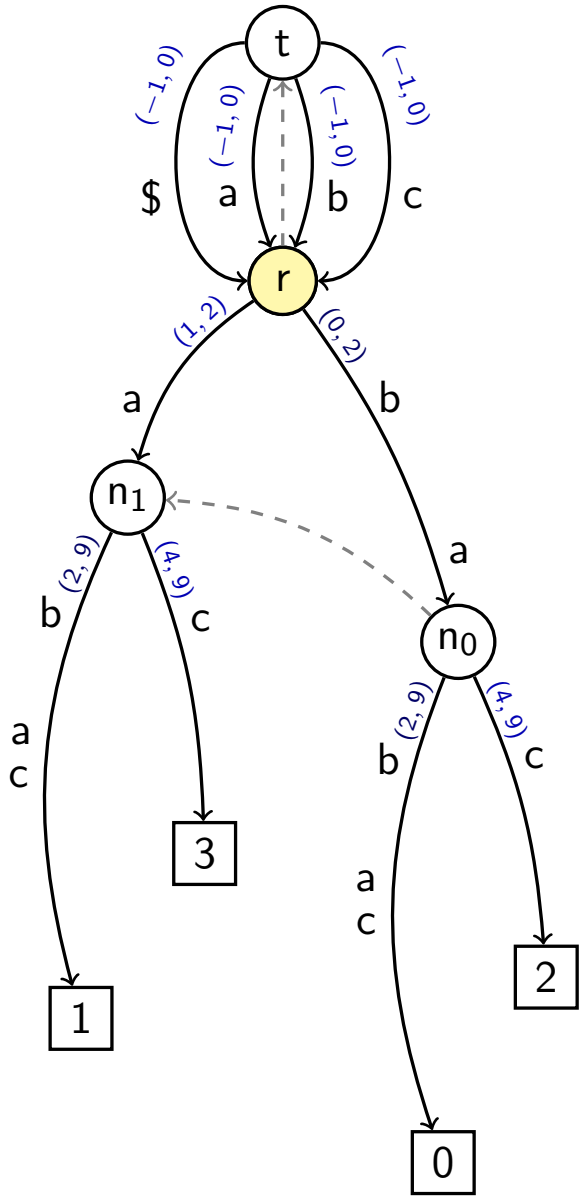
T	$babacbab\$$	$endPoint$	false
i	3	k	4
s	r	r	n_1
old_r	n_1	$s1$	r
b	-1	e	0



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

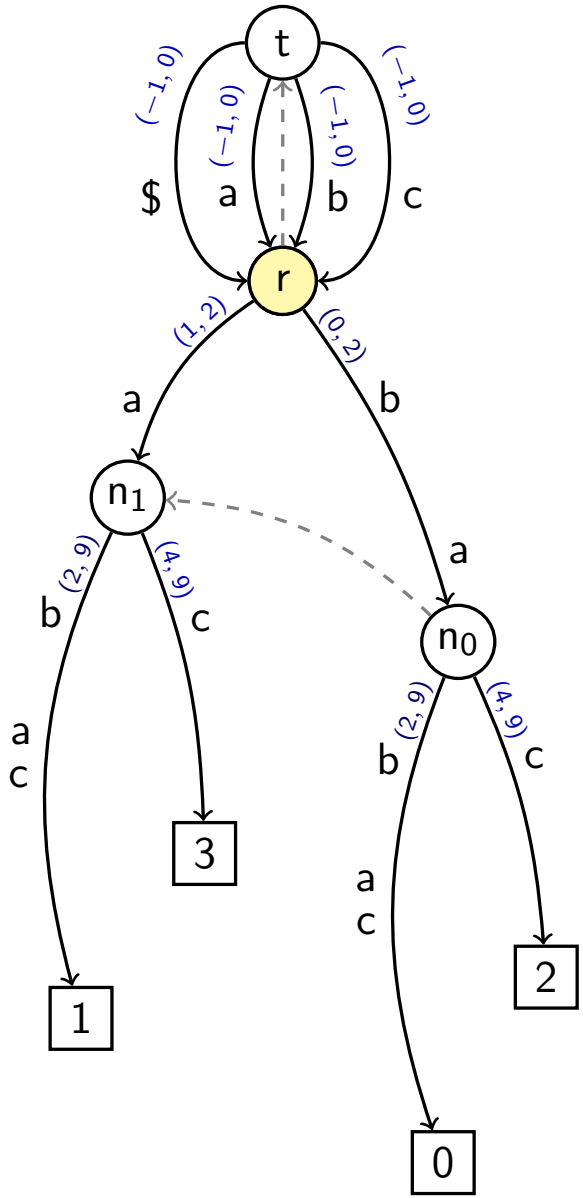
T	$babacbab\$$	$endPoint$	false
i	4	k	4
s	r	r	n_1
old_r	n_1		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

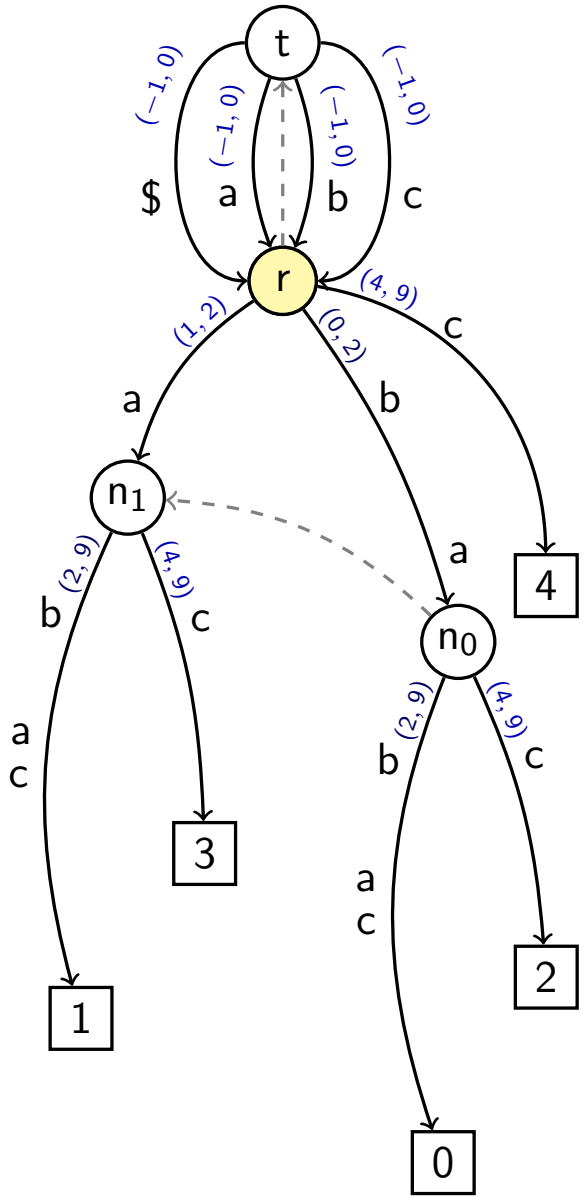
T	$babacbab\$$	$endPoint$	false
i	4	k	4
s	r	r	r
old_r	n_2		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

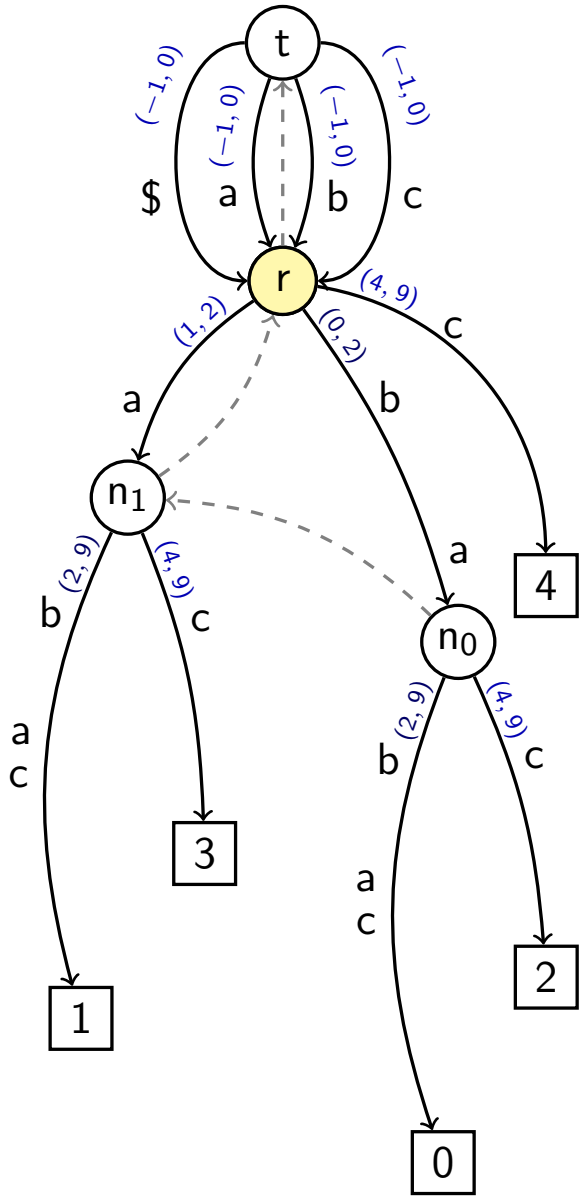
T	$babacbab\$$	$endPoint$	false
i	4	k	4
s	r	r	r
old_r	n_1		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

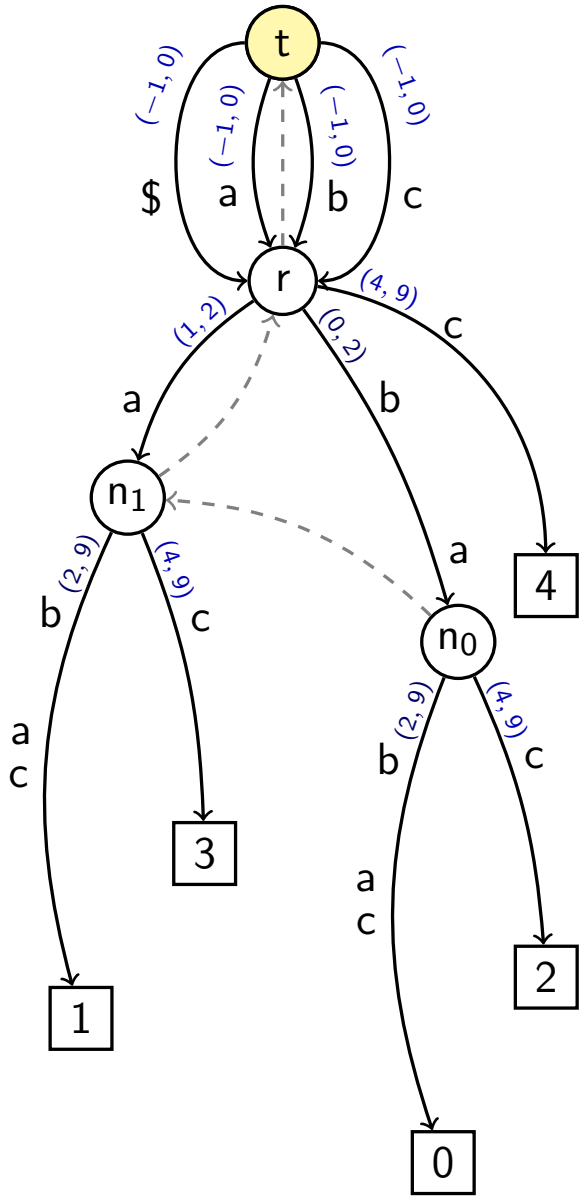
T	$babacbab\$$	$endPoint$	false
i	4	k	4
s	r	r	r
old_r	n_1		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

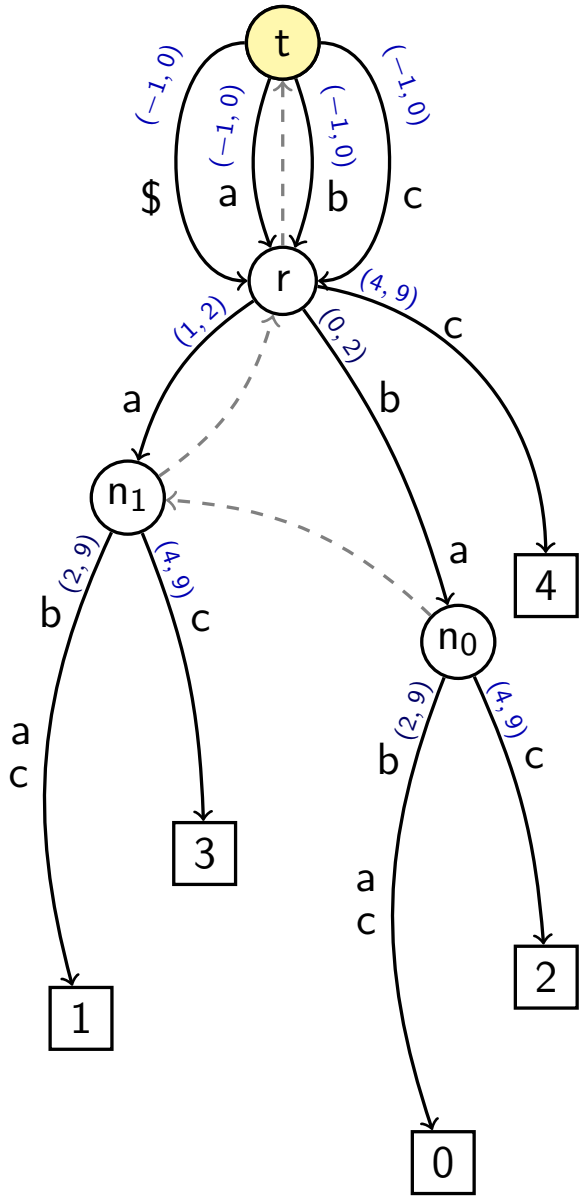
T	*babacbab$*	*endPoint*	false
i	4	*k*	4
s	t	*r*	r
old_r	r		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

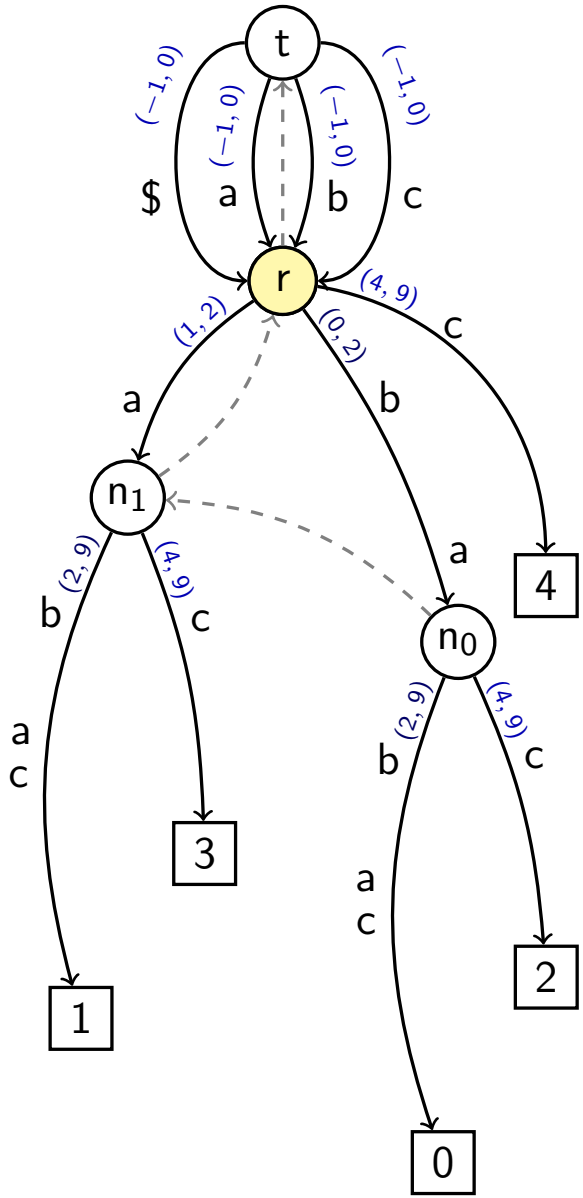
T	*babacbab$*	*endPoint*	true
i	4	*k*	4
s	t	*r*	t
old_r	r		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

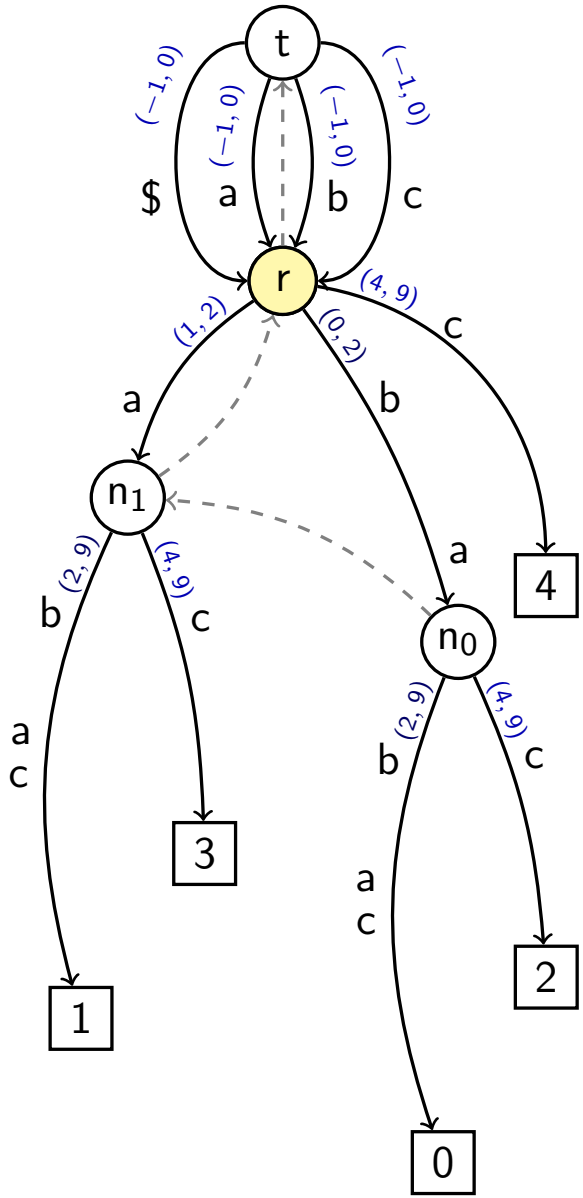
T	*babacbab$*	*endPoint*	true
i	4	*k*	5
s	r	*r*	t
old_r	r		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

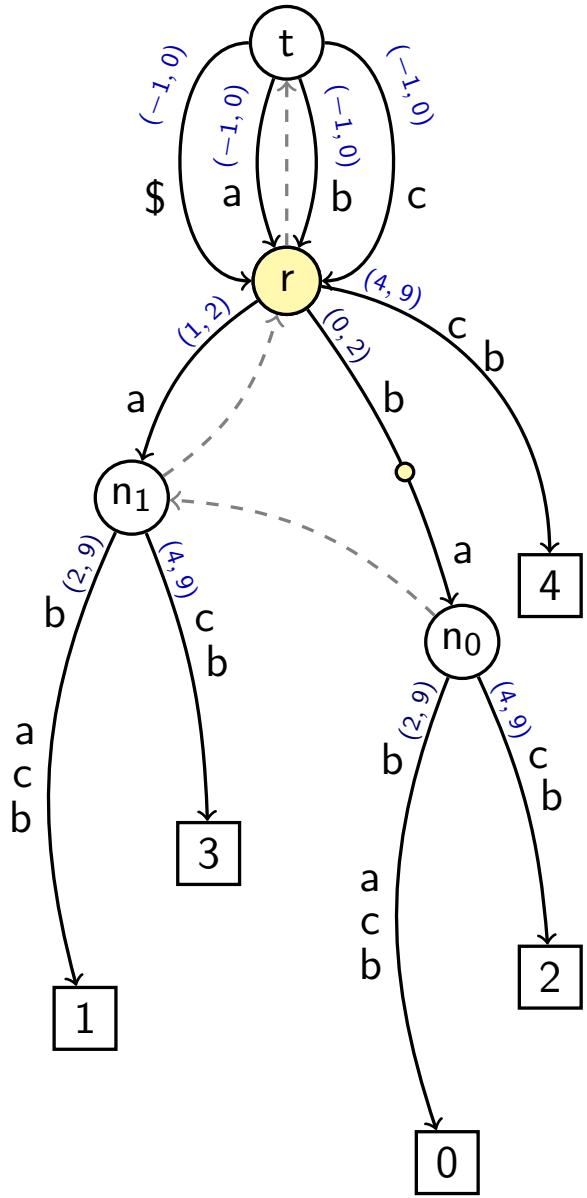
T	*babacbab$*	*endPoint*	true
i	4	*k*	5
s	r	*r*	t
old_r	r		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

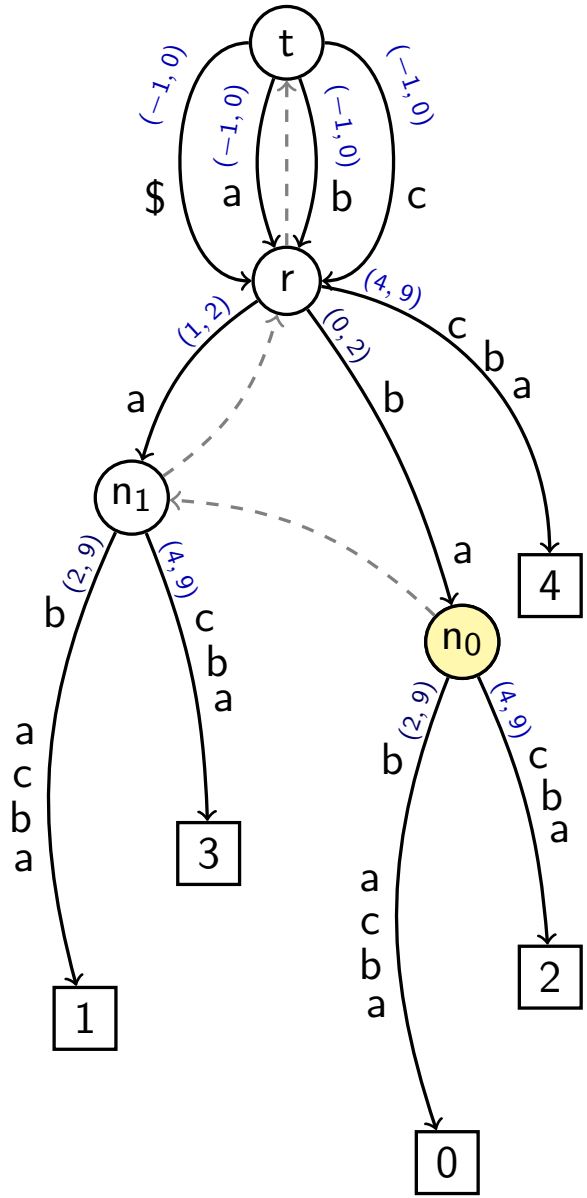
T	$babacbab\$$	$endPoint$	-
i	5	k	5
s	r	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

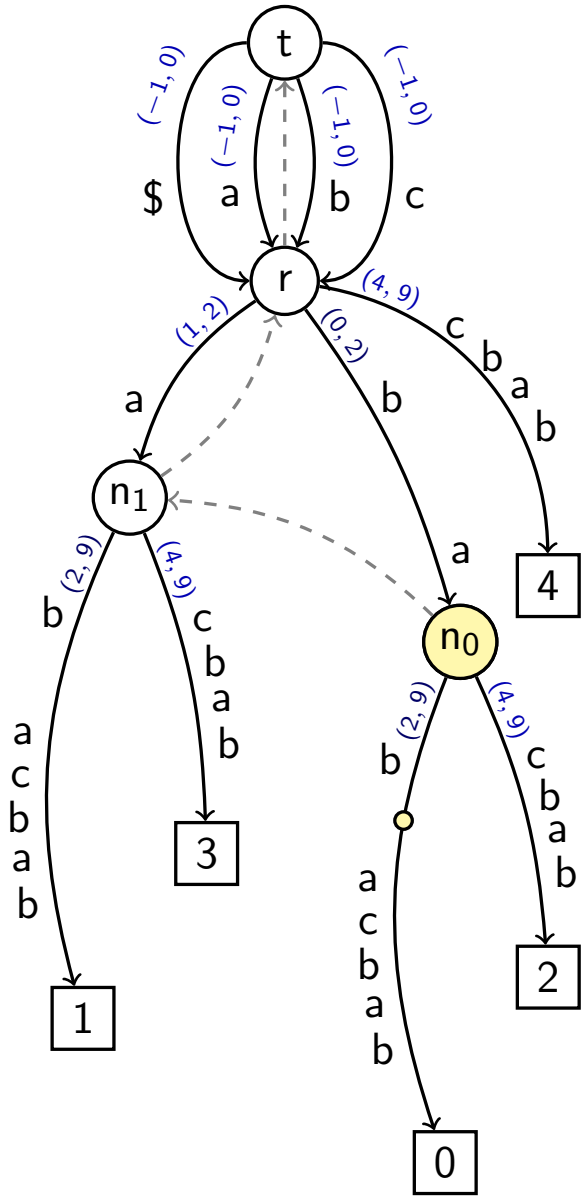
T	$babacbab\$$	$endPoint$	-
i	6	k	7
s	n_0	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

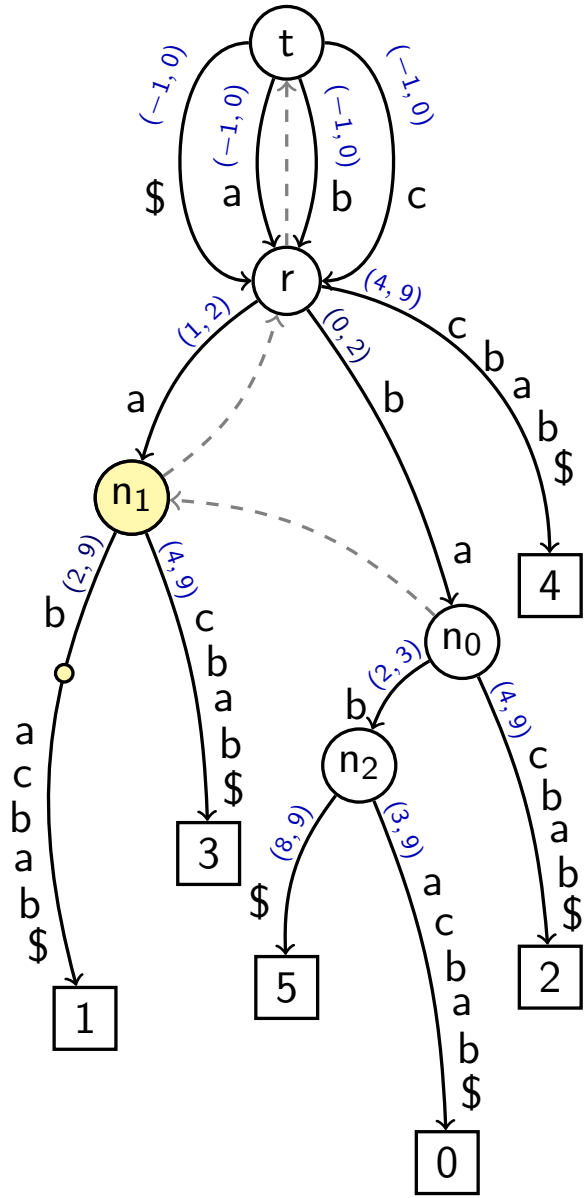
T	$babacbab\$$	$endPoint$	-
i	7	k	7
s	n_0	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

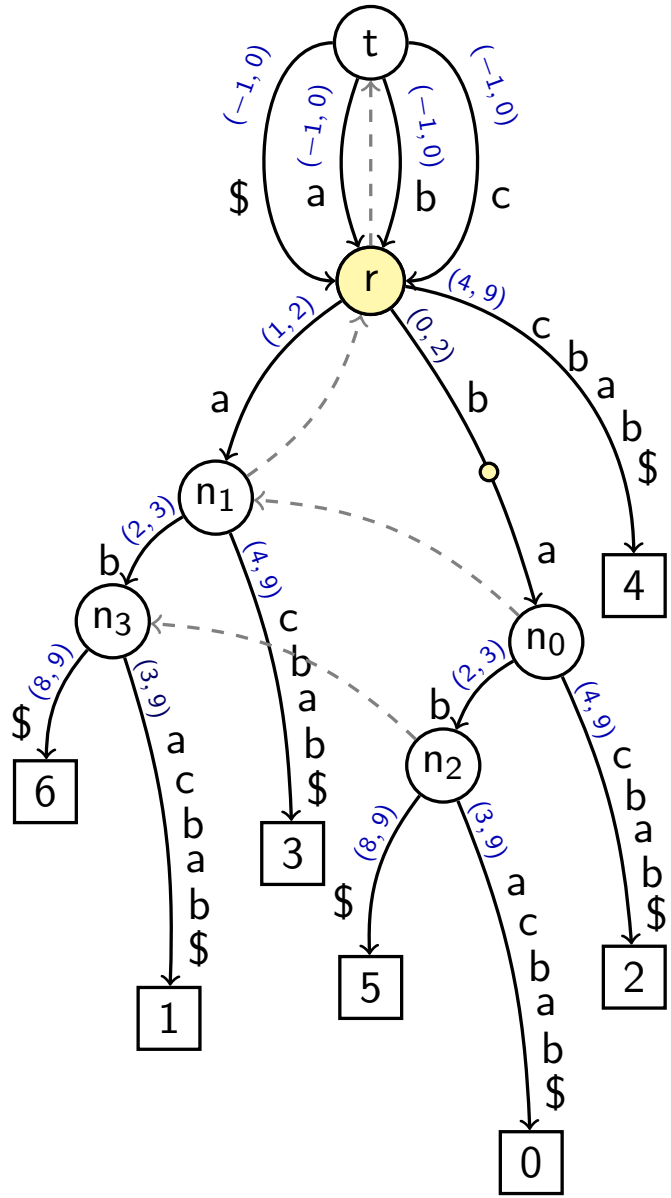
T	$babacbab\$$	$endPoint$	-
i	8	k	7
s	n_1	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

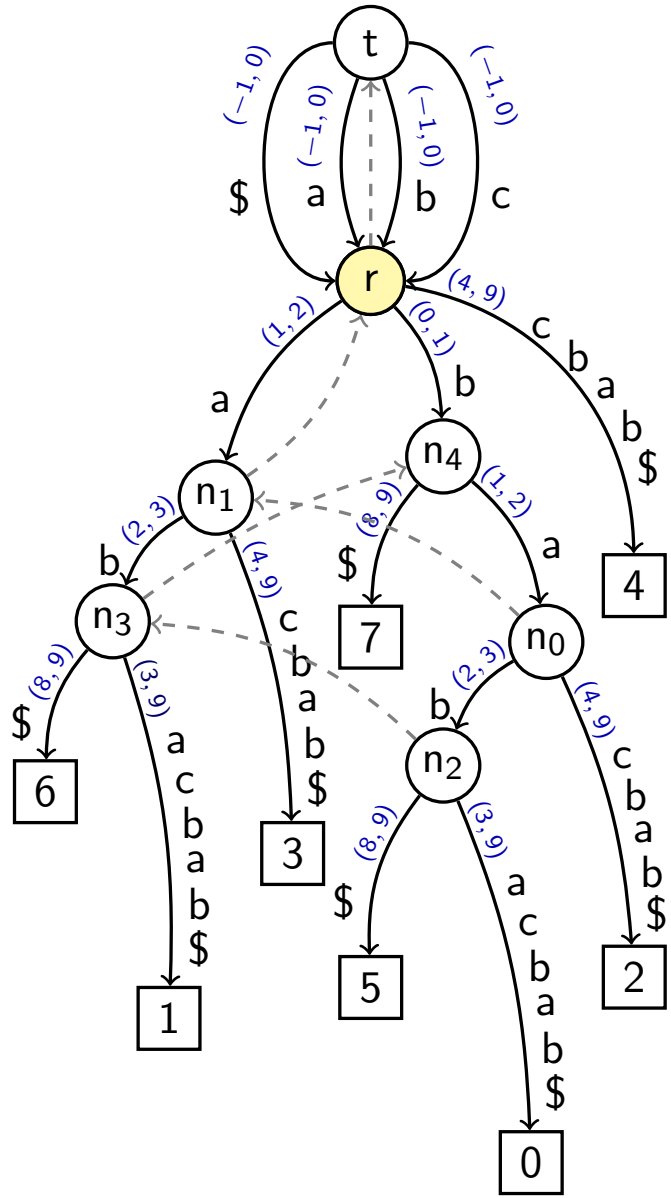
T	*babacbab$*	*endPoint*	-
i	8	*k*	7
s	r	*r*	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

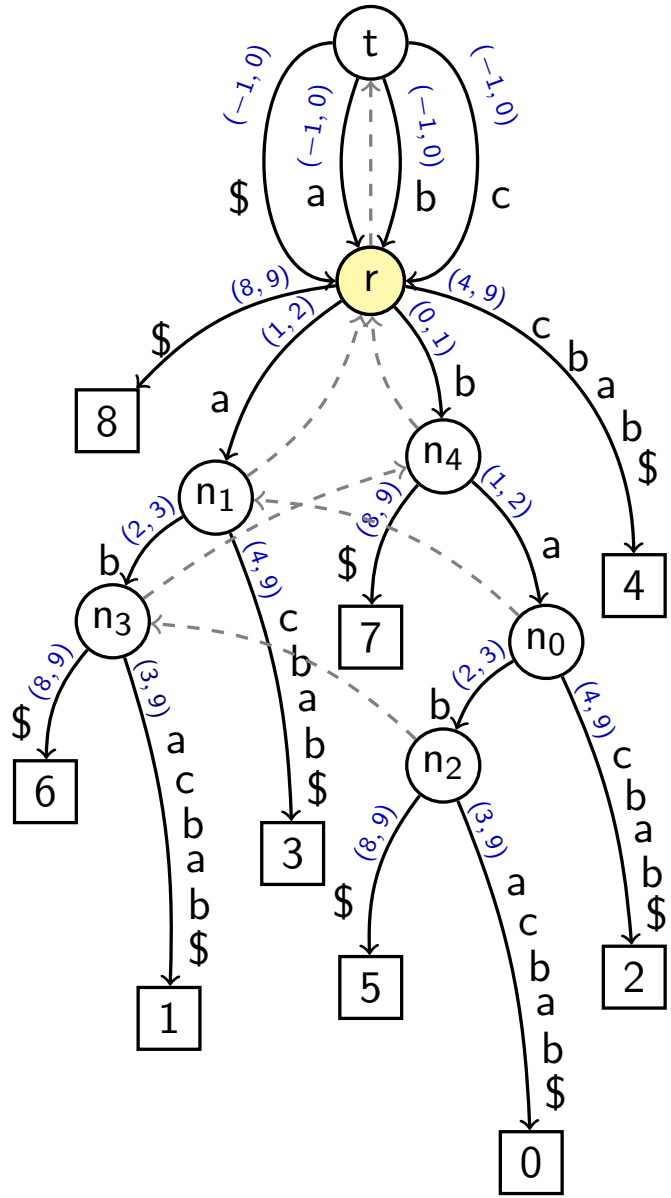
T	$babacbab\$$	$endPoint$	-
i	8	k	8
s	r	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

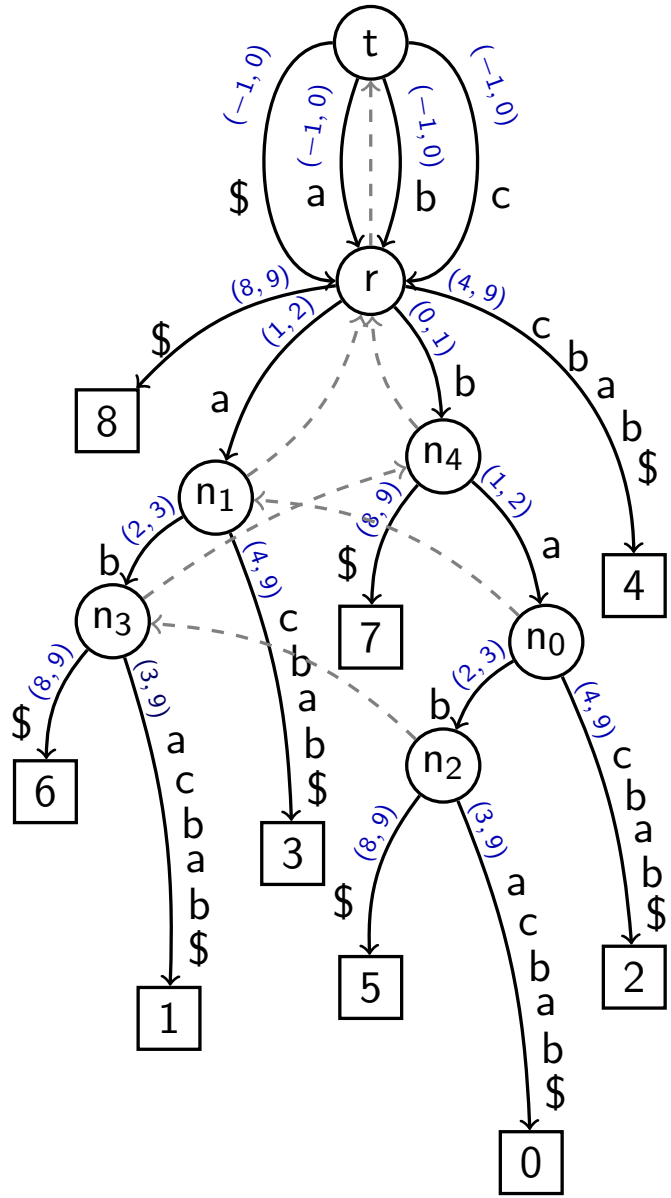
T	$babacbab\$$	$endPoint$	-
i	8	k	9
s	r	r	—
old_r	—		



Implementierung

```
1  def update(s, k, i, T, pos):
2      old_r, n = None, len(T)
3      endPoint, r = test_and_split(s, k, i, T)
4
5      while not endPoint:
6          r.targets[T[i]] = (STNode(pos=pos), i, n)
7          if old_r != None: old_r.s_link = r
8          old_r, pos = r, pos + 1
9          s, k = canonize(s.s_link, k, i - 1, T)
10         endPoint, r = test_and_split(s, k, i, T)
11
12     if old_r != None: old_r.s_link = s
13     s, k = canonize(s, k, i, T)
14     return s, k, pos
```

T	$babacbab\$$	$endPoint$	-
i	8	k	9
s	—	r	—
old_r	—		



Laufzeitanalyse

While-Schleife in *canonize*:

- Zu beginn ist $k = 0$.
- Die While-Bedingung lautet $e - b \leq i + 1 - k$.

Laufzeitanalyse

While-Schleife in *canonize*:

- Zu beginn ist $k = 0$.
- Die While-Bedingung lautet $e - b \leq i + 1 - k$.
- Für alle Intervalle gilt: $e - b \geq 1$.
- Pro Betreten der While-Schleife wird k um mind. 1 erhöht.

Laufzeitanalyse

While-Schleife in *canonize*:

- Zu beginn ist $k = 0$.
- Die While-Bedingung lautet $e - b \leq i + 1 - k$.
- Für alle Intervalle gilt: $e - b \geq 1$.
- Pro Betreten der While-Schleife wird k um mind. 1 erhöht.
- Sobald $i < k$ ist, bricht die While-Schleife ab.
- Insgesamt kann k max. n mal um 1 erhöht werden. Somit kann die Schleife auch nur $\mathcal{O}(n)$ mal betreten werden.

Laufzeitanalyse

While-schleife in *update*:

- Pro *update*-Aufruf wird die Stringtiefe der aktuellen Position um 1 erhöht.
- Pro While-Durchlauf wird die Stringtiefe durch das Beschreiten der Suffixlinks um 1 verringert.

Laufzeitanalyse

While-schleife in *update*:

- Pro *update*-Aufruf wird die Stringtiefe der aktuellen Position um 1 erhöht.
- Pro While-Durchlauf wird die Stringtiefe durch das Beschreiten der Suffixlinks um 1 verringert.
- Die Stringtiefe kann nicht unter 0 sinken.
- Die Stringtiefe wird genau n mal erhöht, also kann sie auch nur n mal um 1 verringert werden.

Laufzeitanalyse

While-schleife in *update*:

- Pro *update*-Aufruf wird die Stringtiefe der aktuellen Position um 1 erhöht.
- Pro While-Durchlauf wird die Stringtiefe durch das Beschreiten der Suffixlinks um 1 verringert.
- Die Stringtiefe kann nicht unter 0 sinken.
- Die Stringtiefe wird genau n mal erhöht, also kann sie auch nur n mal um 1 verringert werden.
- Der Ukkonen-Algorithmus konstruiert einen Suffix-Tree also in $\mathcal{O}(n)$.

Suffix-Array aus Suffix-Tree erstellen

- Ein Suffix-Array *pos* gibt die Startpositionen der Suffixe in lexikographischer Reihenfolge an.
- Für $T = \texttt{babacbab\$}$: $pos = [8, 6, 1, 3, 7, 5, 0, 2, 4]$.
- Das Suffix-Array enthält dieselben Informationen wie die Blätter eines Suffix-Trees.

Suffix-Array aus Suffix-Tree erstellen

- Ein Suffix-Array *pos* gibt die Startpositionen der Suffixe in lexikographischer Reihenfolge an.
- Für $T = \texttt{babacbab\$}$: $pos = [8, 6, 1, 3, 7, 5, 0, 2, 4]$.
- Das Suffix-Array enthält dieselben Informationen wie die Blätter eines Suffix-Trees.

```
1 def get_suffixarray(node, pos = None):
2     if pos == None: pos = []
3     if len(node.targets):
4         for c in sorted(node.targets):
5             get_suffixarray(node.targets[c][0], pos)
6     else: pos += [node.pos]
7     return pos
```

Mustersuche im Suffix-Tree

Fragestellung: kommt P in T vor?

- Für die Mustersuche ist wieder das Tupel (s, k, i) erforderlich:
- Die Mustersuche startet mit $(root, 0, 0)$:

Mustersuche im Suffix-Tree

Fragestellung: kommt P in T vor?

- Für die Mustersuche ist wieder das Tupel (s, k, i) erforderlich:
- Die Mustersuche startet mit $(root, 0, 0)$:
- Drei Möglichkeiten:
  - Wenn $i = |P|$: `True` zurückgeben.
  - Wenn $i < |P|$ und $k == i$: Prüfen ob s eine ausgehende Kante mit $P[i]$ hat.
  - Wenn $i < |P|$ und $k < i$: sei $s1, b, e = s.targets[P[k]]$, prüfen ob $P[i] = T[i - k + b]$.

Mustersuche im Suffix-Tree

Fragestellung: kommt P in T vor?

- Für die Mustersuche ist wieder das Tupel (s, k, i) erforderlich:
- Die Mustersuche startet mit $(root, 0, 0)$:
- Drei Möglichkeiten:
  - Wenn $i = |P|$: `True` zurückgeben.
  - Wenn $i < |P|$ und $k == i$: Prüfen ob s eine ausgehende Kante mit $P[i]$ hat.
  - Wenn $i < |P|$ und $k < i$: sei $s1, b, e = s.targets[P[k]]$, prüfen ob $P[i] = T[i - k + b]$.
- Wenn Fall 2 oder 3 nicht zutrifft: `False` zurückgeben.

Mustersuche im Suffix-Tree

Fragestellung: kommt P in T vor?

- Für die Mustersuche ist wieder das Tupel (s, k, i) erforderlich:
- Die Mustersuche startet mit $(root, 0, 0)$:
- Drei Möglichkeiten:
  - Wenn $i = |P|$: `True` zurückgeben.
  - Wenn $i < |P|$ und $k == i$: Prüfen ob s eine ausgehende Kante mit $P[i]$ hat.
  - Wenn $i < |P|$ und $k < i$: sei $s1, b, e = s.targets[P[k]]$, prüfen ob $P[i] = T[i - k + b]$.
- Wenn Fall 2 oder 3 nicht zutrifft: `False` zurückgeben.
- Überprüfen ob nach gelesenem Zeichen im Tree ein Knoten kommt und ggf. betreten:
  wenn $e - b \leq i + 1 - k : s = s1, k = k + e - b$.

Mustersuche im Suffix-Tree

Fragestellung: kommt P in T vor?

- Für die Mustersuche ist wieder das Tupel (s, k, i) erforderlich:
- Die Mustersuche startet mit $(root, 0, 0)$:
- Drei Möglichkeiten:
  - Wenn $i = |P|$: `True` zurückgeben.
  - Wenn $i < |P|$ und $k == i$: Prüfen ob s eine ausgehende Kante mit $P[i]$ hat.
  - Wenn $i < |P|$ und $k < i$: sei $s1, b, e = s.targets[P[k]]$, prüfen ob $P[i] = T[i - k + b]$.
- Wenn Fall 2 oder 3 nicht zutrifft: `False` zurückgeben.
- Überprüfen ob nach gelesenem Zeichen im Tree ein Knoten kommt und ggf. betreten:
  wenn $e - b \leq i + 1 - k : s = s1, k = k + e - b$.
- Suche mit $i + 1$ wiederholen. Laufzeit insgesamt: $\mathcal{O}(|P|)$.

Mustersuche im Suffix-Tree

Fragestellung: wo kommt P in T vor?

- Patternsuche leicht modifizieren.
- Wenn $i = |P|$: überprüfen, ob die aktuelle Position an einem Knoten ist, wenn $k < i : s = s.targets[P[k]][0]$.

Mustersuche im Suffix-Tree

Fragestellung: wo kommt P in T vor?

- Patternsuche leicht modifizieren.
- Wenn $i = |P|$: überprüfen, ob die aktuelle Position an einem Knoten ist, wenn $k < i : s = s.targets[P[k]][0]$.
- Vom Knoten s alle Blätter besuchen und deren *pos*-Wert in einer Liste zurückgeben.
- Bei Mismatch leere Liste zurück geben.

Mustersuche im Suffix-Tree

Fragestellung: wo kommt P in T vor?

- Patternsuche leicht modifizieren.
- Wenn $i = |P|$: überprüfen, ob die aktuelle Position an einem Knoten ist, wenn $k < i : s = s.targets[P[k]][0]$.
- Vom Knoten s alle Blätter besuchen und deren *pos*-Wert in einer Liste zurückgeben.
- Bei Mismatch leere Liste zurück geben.
- Sei z die Anzahl der Blätter, dann ist die Gesamtlaufzeit $\mathcal{O}(|P| + z)$.

Mustersuche im Suffix-Tree

```
1  def search_pattern(s, T, P, k = 0):
2      for i, c in enumerate(P):
3          if k == i and c not in s.targets: return []
4
5          s1, b, e = s.targets[P[k]]
6          if k < i and c != T[i - k + b]: return []
7
8          if e - b <= i + 1 - k:
9              s, k = s1, k + e - b
10
11     if k < len(P): s = s.targets[P[k]][0]
12     return get_suffixarray(s)
```

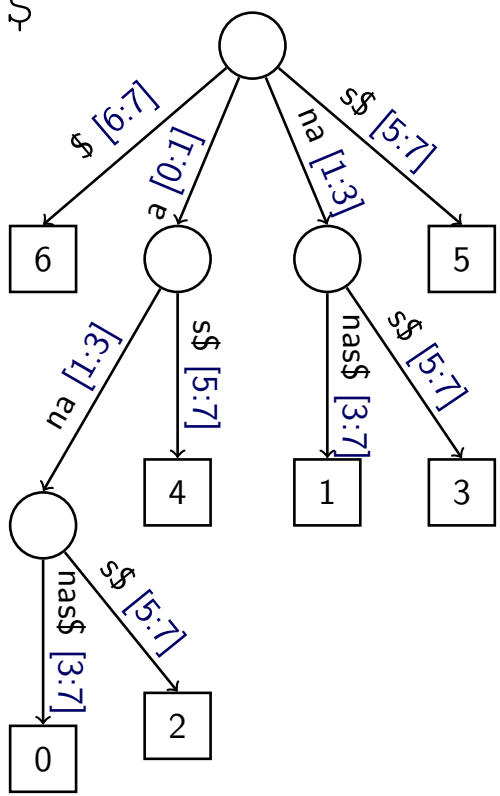
Längster wiederholter Teilstring

Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ ist innerer Knoten}\}$, dann ist der längste wiederholte Teilstring (LRS) $t = str(s^*)$.

Längster wiederholter Teilstring

Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ ist innerer Knoten}\}$, dann ist der längste wiederholte Teilstring (LRS) $t = str(s^*)$.

Beispiel: $T = \texttt{ananas\$}$



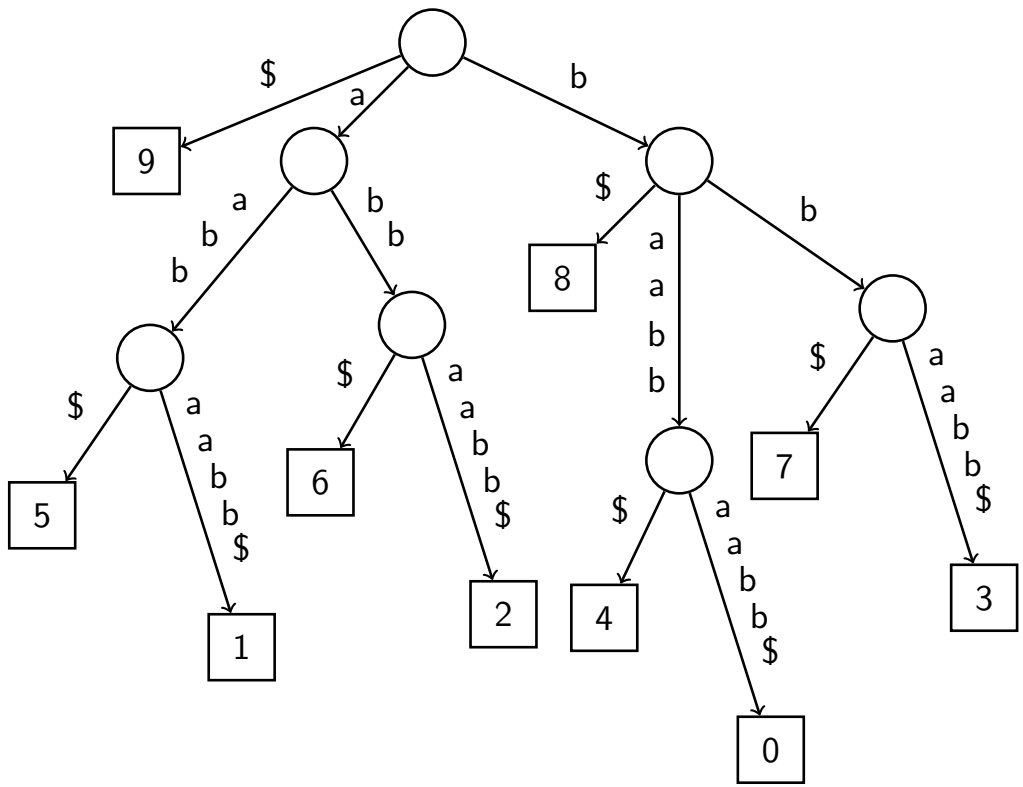
LRS hier: ana

Kürzester eindeutiger Teilstring

Sei $s^* := \text{argmin}_{s \in nodes(S)}\{strdep(s) \mid s \text{ hat Blattkante } e, |e| > 1\}$, dann ist der kürzeste eindeutige Teilstring (SUS) $t = str(s^*) \circ e[0]$.

Kürzester eindeutiger Teilstring

Sei $s^* := \text{argmin}_{s \in nodes(S)}\{strdep(s) \mid s \text{ hat Blattkante } e, |e| > 1\}$, dann ist der kürzeste eindeutige Teilstring (SUS) $t = str(s^*) \circ e[0]$.
Beispiel: $T =$ `baabbaabb$`



SUS hier: `bba`

Längster gemeinsamer Teilstring

- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.

Längster gemeinsamer Teilstring

- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.
- Sei dementsprechend der verallgemeinerte Text $T = T_1\$_1 T_2\$_2$ mit $\$_1 < \$_2$.
- Suffix-Tree auf T aufbauen

Längster gemeinsamer Teilstring

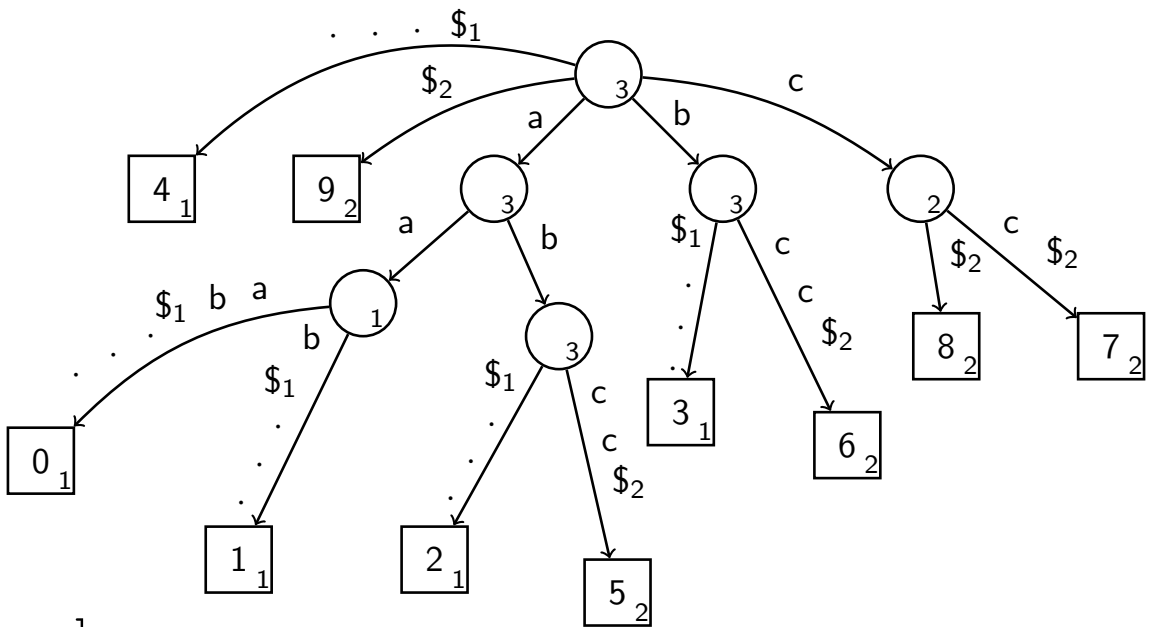
- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.
- Sei dementsprechend der verallgemeinerte Text $T = T_1\$_1 T_2\$_2$ mit $\$_1 < \$_2$.
- Suffix-Tree auf T aufbauen
- Alle Blätter mit *pos*-Wert $< |T_1\$_1|$ mit 1 beschriften.
- Alle restlichen Blätter mit 2 beschriften.

Längster gemeinsamer Teilstring

Innere Knoten bitweise mit den Labels seiner Kinder verodern. Sei $s^* := \arg\max_{s \in nodes(S)}\{strdep(s) \mid s \text{ ist inn. Knoten}, label(s) = 3\}$, dann ist der längste gemeinsame Teilstring (LCS) $t = str(s^*)$.

Längster gemeinsamer Teilstring

Innere Knoten bitweise mit den Labels seiner Kinder verodern. Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ ist inn. Knoten}, label(s) = 3\}$, dann ist der längste gemeinsame Teilstring (LCS) $t = str(s^*)$.
Beispiel: $T = \texttt{aaab\$}_1\texttt{abcc\$}_2$



LCS hier: ab

Zusammenfassung

- Der Ukkonen-Algorithmus erstellt online einen Suffix-Tree in $\mathcal{O}(n)$.
- Verschiedene Fragestellungen können mit dem Suffix-Tree beantwortet werden:
  - Kommt P in T vor? Wenn ja, wo?
  - Welcher ist der längste wiederholte Teilstring im Text T?
  - Welcher ist der kürzeste eindeutige Teilstring im Text T?
  - Welcher ist der längste gemeinsame Teilstring der beiden Texte T_1 und T_2?
- Leider hohe Konstante beim Speicherverbrauch, bei guten Implementierungen ca. 20 Bytes pro Zeichen.