technische universität
dortmund

Algorithmen auf Sequenzen

Pattern-Matching mit einer Patternmenge

Dominik Kopczynski

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

Einführung

- Häufig von Interesse ist nicht nur die Suche nach einem einzelnen Pattern, sondern
- eine Menge von Pattern aufzufinden.
- Naive Fehlertolleranz ist dadurch möglich, etwa {Maier, Mayer, Meier, Meyer}.
- Im Nachfolgenden wird also eine Menge von Pattern $P = \{P_1, \ldots, P_k\}$ mit
- verschiedenen Längen betrachtet, sei $m_i := |P_i|$ für $1 \leq i \leq k$ und $m = \sum_i m_i$.

Erste Herausforderung: Zählweise

Die Anzahl der Vorkommen von Menge P in einem Text T kann auf (mindestens) drei verschiedene Arten gezählt werden:

- Überlappende Matches; entspricht der Anzahle der Paare (i, j), so dass $T[i : j] \in P$ (Standard-Definition).

Erste Herausforderung: Zählweise

Die Anzahl der Vorkommen von Menge P in einem Text T kann auf (mindestens) drei verschiedene Arten gezählt werden:

- Überlappende Matches; entspricht der Anzahle der Paare (i, j), so dass $T[i : j] \in P$ (Standard-Definition).
- Endpositionen von Matches; entspricht der Anzahl der j, für die es mindestens ein i gibt, so dass $T[i : j] \in P$.

Erste Herausforderung: Zählweise

Die Anzahl der Vorkommen von Menge P in einem Text T kann auf (mindestens) drei verschiedene Arten gezählt werden:

- Überlappende Matches; entspricht der Anzahle der Paare (i, j), so dass $T[i : j] \in P$ (Standard-Definition).
- Endpositionen von Matches; entspricht der Anzahl der j, für die es mindestens ein i gibt, so dass $T[i : j] \in P$.
- Nichtüberlappende Matches; gesucht ist eine maximale Menge von Paaren (i, j), die Matches sind, so dass die Intervalle $[i, j]$ für je zwei verschiedene Paare disjunkt sind (wird in dieser Vorlesung nicht behandelt).

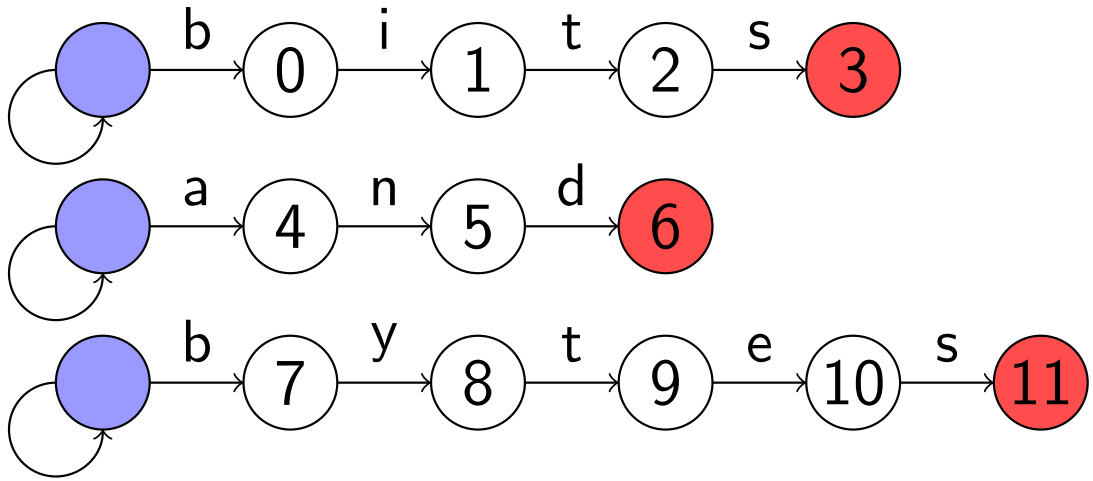
Shift-And-Algorithmus auf Mengen

- Gesucht ist ein NFA, der $\Sigma^* P = \cup_{i=1}^{k} P_k$ akzeptiert.
- Sei hierbei die Zustandsmenge $Q := \{(i,j) \,|\, 1 \leq i \leq k, -1 \leq j \leq |P_i|\}$.
- Zustand (i,j) repräsentiert das Präfix $P_i[:j]$ mit $P_i \in P$.

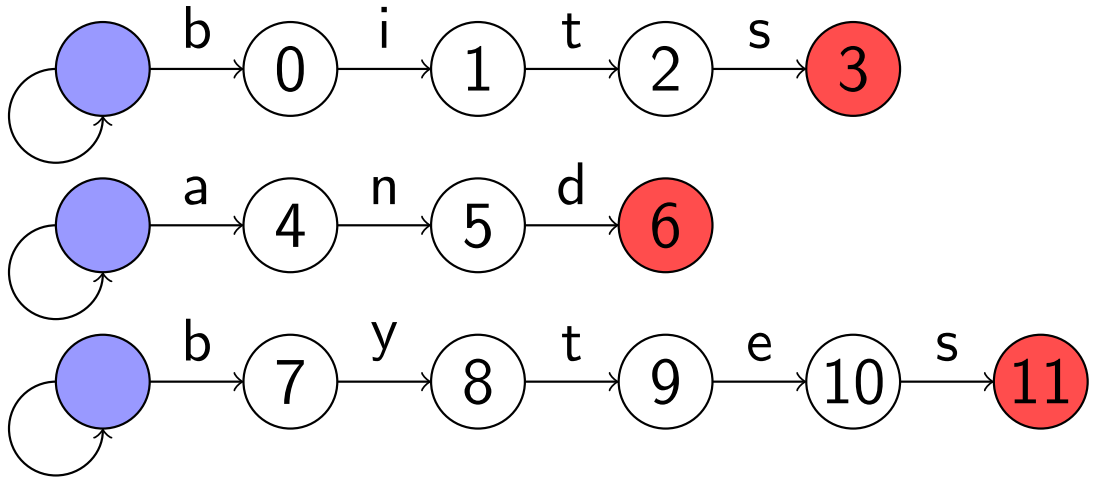
Shift-And-Algorithmus auf Mengen

- Gesucht ist ein NFA, der $\Sigma^* P = \cup_{i=1}^{k} P_k$ akzeptiert.
- Sei hierbei die Zustandsmenge $Q := \{(i,j) \,|\, 1 \leq i \leq k, -1 \leq j \leq |P_i|\}$.
- Zustand (i,j) repräsentiert das Präfix $P_i[:j]$ mit $P_i \in P$.
- Sei $Q_0 = \{(i,-1) \,|\, 1 \leq i \leq k\}$ die Menge der Zustände, die das leere Präfix repräsentieren (Startzustände).
- Sei $T = \{(i,|P_i|-1) \,|\, 1 \leq i \leq k\}$ die Menge der Zustände, die jeweils den ganzen String repräsentieren (Endzustände).
- Übergange von (i,j) nach $(i,j+1)$ durch lesen von Zeichen $P_i[j+1]$ möglich.

Shift-And-Algorithmus auf Mengen



Shift-And-Algorithmus auf Mengen



- NFA zum Finden der Wörter aus der Menge {bits, and, bytes}.
- Die Startzustände sind blau, die akzeptierenden Zustände rot dargestellt.
- Obwohl es mehrere Zusammenhangskomponenten gibt, handelt es sich um einen Automaten.

Der Shift-And-Algorithmus auf Mengen

Implementierung:

- Alle Strings werden zu einem Pattern konkateniert, sei im Beispiel $P = \texttt{bitsandbytes}$.
- Die Startzutände werden in einem Bitfeld gespeichert, sei $A_0[j] = [j \in \{s \mid s = \sum_{i=1}^{g} |P_i|, 0 \leq g < k\}]$ für $0 \leq j < W$. Bsp: $A_0 = \texttt{000010010001}$.
- Die akzeptierenden Zustände werden ebenfalls in einem Bitfeld gespeichert, sei $\mathit{accept}[j] = [j \in \{s \mid s = (\sum_{i=1}^{g} |P_i|) - 1, 1 \leq g \leq k\}]$ für $0 \leq j < W$. Bsp: $\mathit{accept} = \texttt{100001001000}$.

Der Shift-And-Algorithmus auf Mengen

Erstellung der Masken:

```
1  def create_masks(Ps, S):
2      mask = {s: 0 for s in S}
3      A_0, accept, bit = 0, 0, 1
4      for p in Ps:
5          A_0 |= bit
6          for c in p:
7              mask[c] |= bit
8              bit <<= 1
9          accept |= bit
10     return mask, A_0, accept >> 1
```

Der Shift-And-Algorithmus auf Mengen

Mustersuche mit dem Shift-And-Algorithmus:

```
1 def shift_and_multi(Ps, T):
2     S, A = set("".join(Ps) + T), 0
3     mask, A_0, accept = create_masks(Ps, S)
4     for i, c in enumerate(T):
5         A = ((A << 1) | A_0) & mask[c]
6         if A & accept:
7             yield i
```

Der Shift-And-Algorithmus auf Mengen

Mustersuche mit dem Shift-And-Algorithmus:

```
1 def shift_and_multi(Ps, T):
2     S, A = set("".join(Ps) + T), 0
3     mask, A_0, accept = create_masks(Ps, S)
4     for i, c in enumerate(T):
5         A = ((A << 1) | A_0) & mask[c]
6         if A & accept:
7             yield i
```

In diesem Code wird nur die Endposition eines Matches übergeben. Will man zusätzlich noch die Startposition übergeben, so muss erkannt werden, welche akzeptierenden Zustände aktiv sind.

Zusammenfassung

- Der Shift-And-Algorithmus lässt sich mit ein paar Ergänzungen problemlos auf eine Menge von Pattern erweitern.
- Der vorgestellte Algorithmus kann nur verwendet werden, wenn die aufsummierte Länge aller Pattern kleiner als die Registergröße ist.
- Die Laufzeit beträgt $\mathcal{O}(m + n\lceil m/W \rceil)$.
- Es wird lediglich erkannt, ob mind. ein Pattern bei einem aktuellen Zeichen im Text endet. Es findet keine Auflistung der gefundenen Pattern statt.

Ein weiterer Algorithmus auf Mengen

- Offensichtlich lässt sich jedes Pattern einzeln mit dem KMP suchen.
- Bei dem Ansatz beträgt die Laufzeit $\mathcal{O}(kn + m)$.

Ein weiterer Algorithmus auf Mengen

- Offensichtlich lässt sich jedes Pattern einzeln mit dem KMP suchen.
- Bei dem Ansatz beträgt die Laufzeit $\mathcal{O}(kn + m)$.
- Es wird eine Datenstruktur benötigt, bei der
  - nachgehalten werden kann, was das längste Suffix ist, das ein Präfix eines Patterns in der Menge P ist.
  - in konstanter Zeit ein weiteres Zeichen gelesen werden kann, ohne diese Invariante zu verletzen.
- In dem Fall würde die Laufzeit $\mathcal{O}(n + m)$ betragen.

Ein weiterer Algorithmus auf Mengen

- Offensichtlich lässt sich jedes Pattern einzeln mit dem KMP suchen.
- Bei dem Ansatz beträgt die Laufzeit $\mathcal{O}(kn + m)$.
- Es wird eine Datenstruktur benötigt, bei der
  - nachgehalten werden kann, was das längste Suffix ist, das ein Präfix eines Patterns in der Menge P ist.
  - in konstanter Zeit ein weiteres Zeichen gelesen werden kann, ohne diese Invariante zu verletzen.
- In dem Fall würde die Laufzeit $\mathcal{O}(n + m)$ betragen.
- Diesen Ansatz realisiert der *Aho-Corasick*-Algorithmus.

Trie

Definition (Trie)

Ein Trie über einer endlichen Menge von Wörtern $S \subset \Sigma^+$ ist ein kantenbeschrifteter Baum über der Knotenmenge Prefixes(S) mit folgender Eigenschaft: Der Knoten s ist genau dann ein Kind von Knoten t, wenn $s = ta$ für ein $a \in \Sigma$; die Kante $t \to s$ ist dann mit a beschriftet.

Trie

Definition (Trie)

Ein Trie über einer endlichen Menge von Wörtern $S \subset \Sigma^+$ ist ein kantenbeschrifteter Baum über der Knotenmenge Prefixes(S) mit folgender Eigenschaft: Der Knoten s ist genau dann ein Kind von Knoten t, wenn $s = ta$ für ein $a \in \Sigma$; die Kante $t \to s$ ist dann mit a beschriftet.

Jeder Knoten v kann somit eindeutig mit dem String identifiziert werden, den man erhält, wenn man die Zeichen entlang des eindeutigen Pfades von der Wurzel zu v geht.

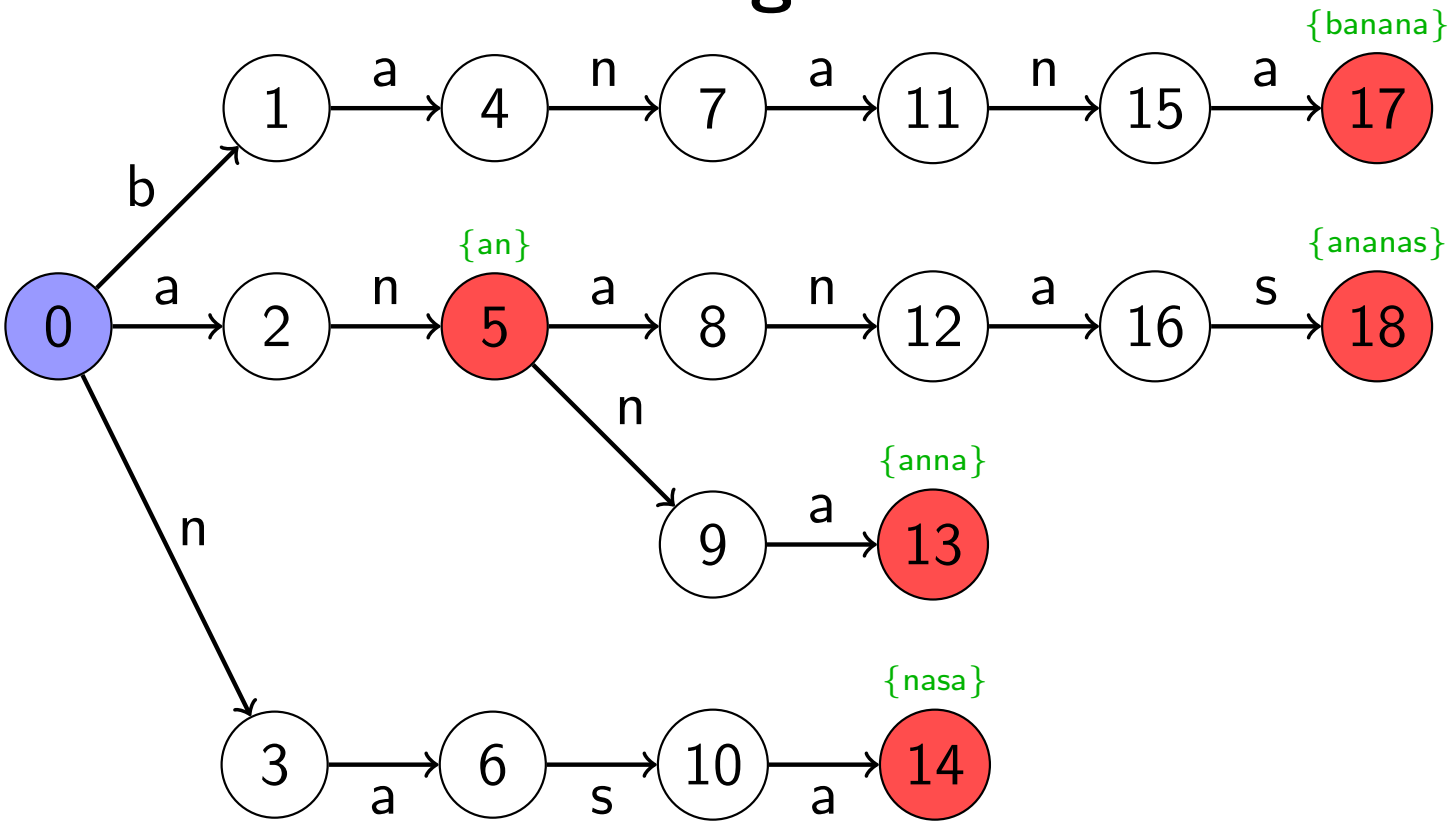
Der Aho-Corasick-Algorithmus

Der AC-Algorithmus nutzt die Ideen des KMP-Algorithmus, um auf einem Trie eine *lps*-Funktion einzuführen. Sei

$$lps(q) := \underset{p \in \text{Prefixes}(P)}{\arg\max} \left\{ |p| \,\middle|\, |p| < |q|, p = q[|q| - |p| :] \right\}.$$

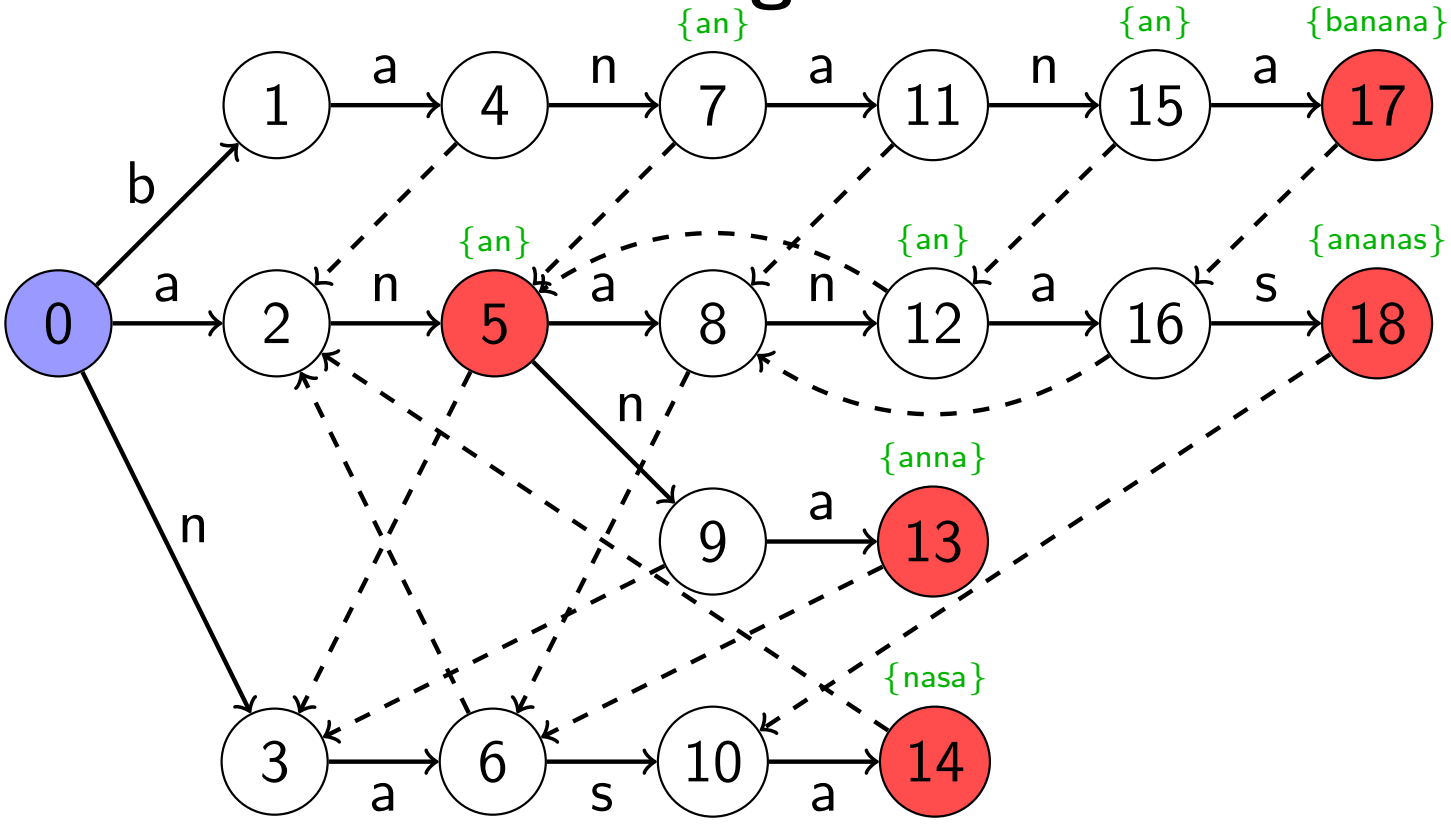
- Hierbei sind q und $lps(q)$ keine Zahlen, sondern Strings, bzw. Knoten im Trie, die jeweils auf einen String abbilden.
- Die Funktion $lps(q)$ verweist auf den Knoten, der zum längsten Präfix p eines Patterns in P gehört, welches gleichzeitig ein echtes Suffix von q ist.

Der Aho-Corasick-Algorithmus



Beispiel: Trie über Patternmenge $P = \{$an, ananas, anna, banana, nasa$\}$. Startzustand: blau; Wörter aus P: rot; auszugebende Wörter: grün;

Der Aho-Corasick-Algorithmus



Beispiel: Trie über Patternmenge $P = \{$an, ananas, anna, banana, nasa$\}$. Startzustand: blau; Wörter aus P: rot; auszugebende Wörter: grün; Gestrichelte Pfeile: *lps*-Funktion.

Der Aho-Corasick-Algorithmus

Erzeugen der *lps*-Funktion:

1. Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten ϵ, $lps[\epsilon]$ ist nicht definiert.

Der Aho-Corasick-Algorithmus

Erzeugen der *lps*-Funktion:

1. Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten ϵ, $lps[\epsilon]$ ist nicht definiert.
2. Wenn der Vorgängerknoten ϵ war, so gilt $lps[q] = \epsilon$ für alle Knoten q.

Der Aho-Corasick-Algorithmus

Erzeugen der *lps*-Funktion:

1. Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten ϵ, $lps[\epsilon]$ ist nicht definiert.
2. Wenn der Vorgängerknoten ϵ war, so gilt $lps[q] = \epsilon$ für alle Knoten q.
3. Für alle restlichen Knoten gilt: In Knoten xa setze $v := x$. Prüfe, ob Knoten $lps[v]$ eine ausgehende Kante a hat. Wenn ja, setze $lps[xa] := lps[v]a$. Sonst gehe solange über zu $v := lps[v]$, bis entweder $lps[v]$ nicht mehr existiert oder es eine ausgehende Kante a von v gibt.

Der Aho-Corasick-Algorithmus

Erzeugen der *lps*-Funktion:

1. Traversiere den Trie mittels Breitensuche. Beginne beim Startknoten ϵ, $lps[\epsilon]$ ist nicht definiert.
2. Wenn der Vorgängerknoten ϵ war, so gilt $lps[q] = \epsilon$ für alle Knoten q.
3. Für alle restlichen Knoten gilt: In Knoten xa setze $v := x$. Prüfe, ob Knoten $lps[v]$ eine ausgehende Kante a hat. Wenn ja, setze $lps[xa] := lps[v]a$. Sonst gehe solange über zu $v := lps[v]$, bis entweder $lps[v]$ nicht mehr existiert oder es eine ausgehende Kante a von v gibt.

Punkt 3 entspricht der *delta*-Funktion des KMP-Algorithmus.

Der Aho-Corasick-Algorithmus

Ausgabe der gefundenen Pattern:

- Es findet eine Ausgabe statt, wenn der aktuelle Knoten
  - einem akzeptierenden Zustand entspricht.
  - über (mehrfache) *lps*-Transitionen einen akzeptierenden Zustand erreicht.

Der Aho-Corasick-Algorithmus

Ausgabe der gefundenen Pattern:

- Es findet eine Ausgabe statt, wenn der aktuelle Knoten
  - einem akzeptierenden Zustand entspricht.
  - über (mehrfache) *lps*-Transitionen einen akzeptierenden Zustand erreicht.
- Die Indizes der auszugebenden Pattern können
  - direkt im Knoten als Liste durch Konkatenation mit ihren *lps*-Vorgängern abgespeichert werden.
  - im Knoten als Liste abgespeichert werden. Bei der Ausgabe wird über die *lps*-Transitionen. traversiert.

Der Aho-Corasick-Algorithmus

Implementierung:
Es wird eine Klasse `ACNode` mit folgenden Attributen angelegt:

- Ein Dictionary *targets* : $c \rightarrow q$ mit $c \in \Sigma, q \in \text{Prefixes}(P)$ zum speichern der Kinderknoten.
- Eine Referenz *lps* auf den *lps*-Vorgängerknoten.
- Eine Liste *out* mit den Indizes der auszugebenden Pattern.

```
1 class ACNode():
2     def __init__(self):
3         self.targets = dict()
4         self.lps = None
5         self.out = []
```

Der Aho-Corasick-Algorithmus

Implementierung:
Die *delta*-Funktion ist ähnlich aufgebaut, wie die herkömmliche *delta*-Funktion des KMP, nur dass die Zustände hierbei durch Knoten abgebildet werden.

```
1 def delta(q, c):
2     while q.lps != None and c not in q.targets:
3         q = q.lps
4     if c in q.targets: q = q.targets[c]
5     return q
```

Der Aho-Corasick-Algorithmus

Implementierung:
Das Erstellen des Aho-Corasick-Automates ist aufgeteilt in zwei Phasen:

1. Erstellen des Tries.
2. Hinzufügen der lps-Kanten.

```
1 def build_AC(P):
2     root = build_trie(P)
3     root = build_lps(root)
4     return root
```

Der Aho-Corasick-Algorithmus

Implementierung:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

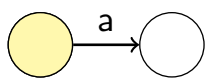
$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

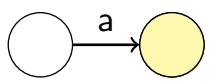


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

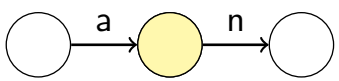


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

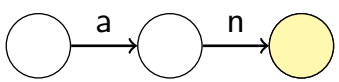


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

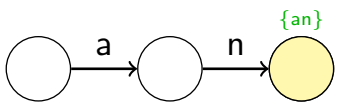


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

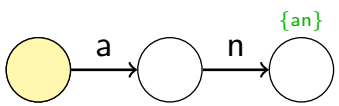


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

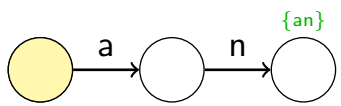


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

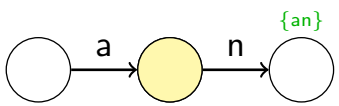


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

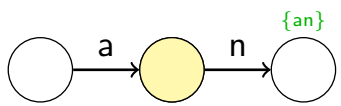


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

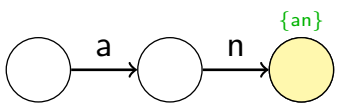


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

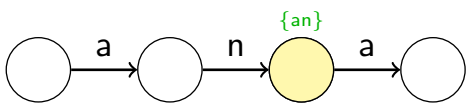


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

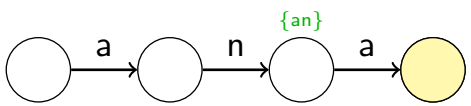


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

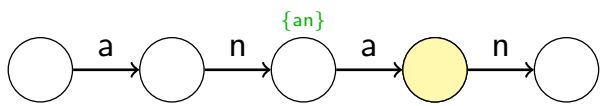


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

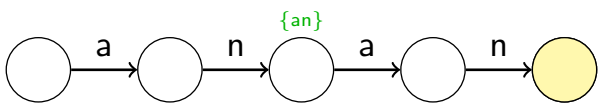


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

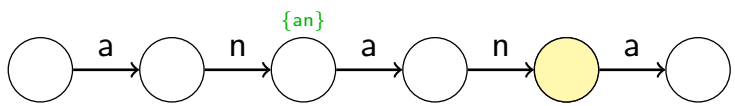


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

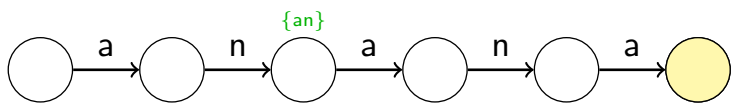


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

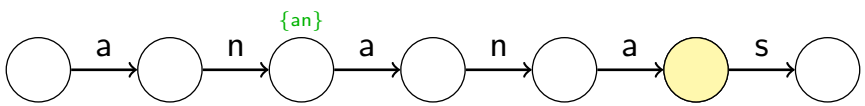


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

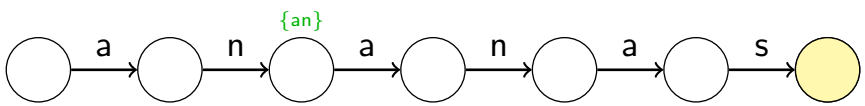


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

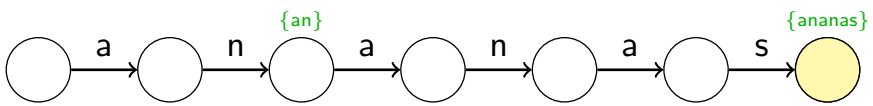


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

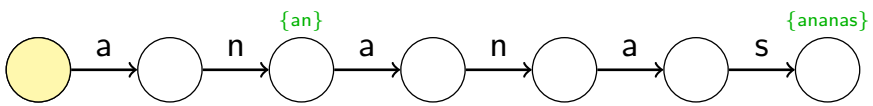


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

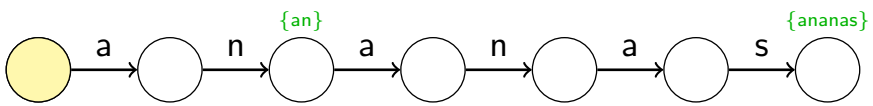


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

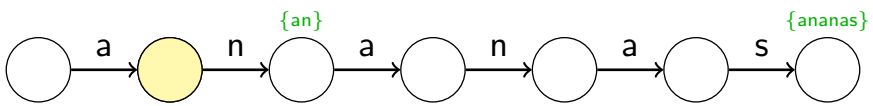


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

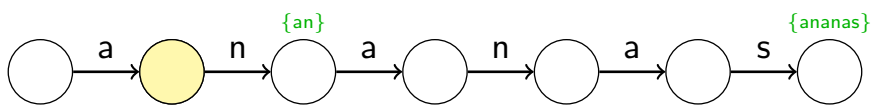


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

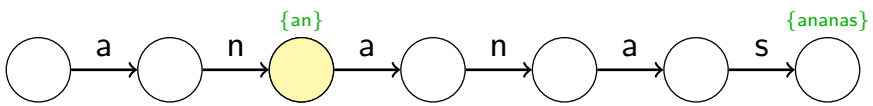


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

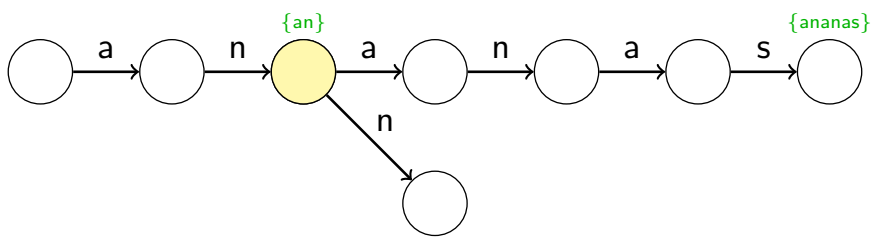


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

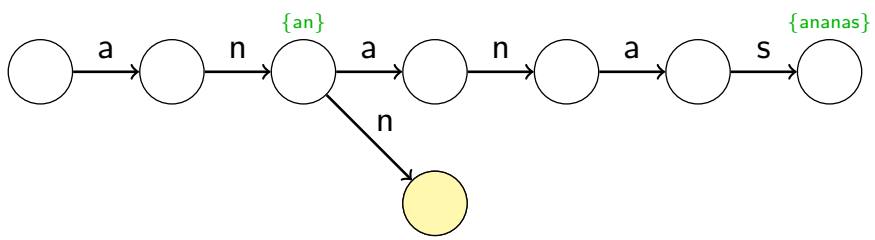


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

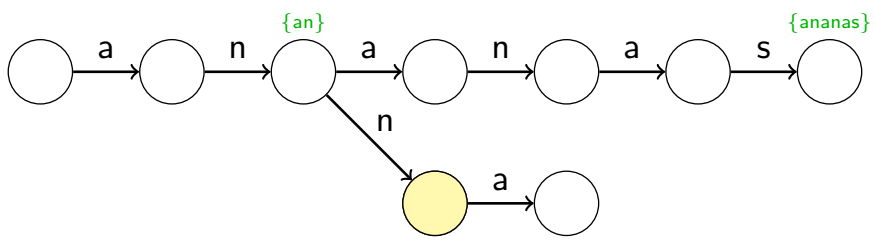


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

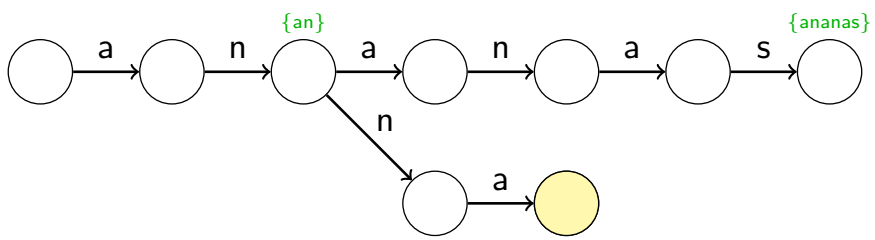


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

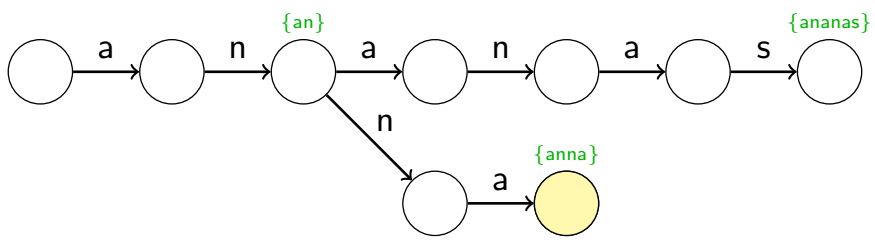


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

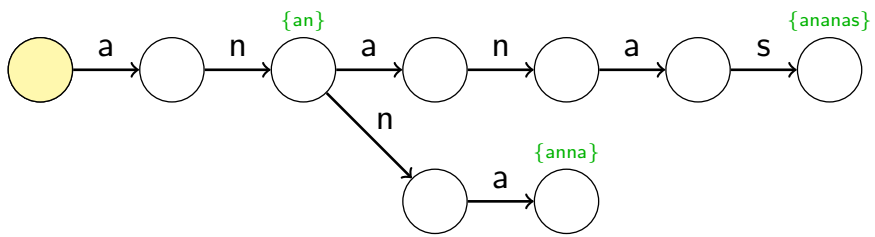


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

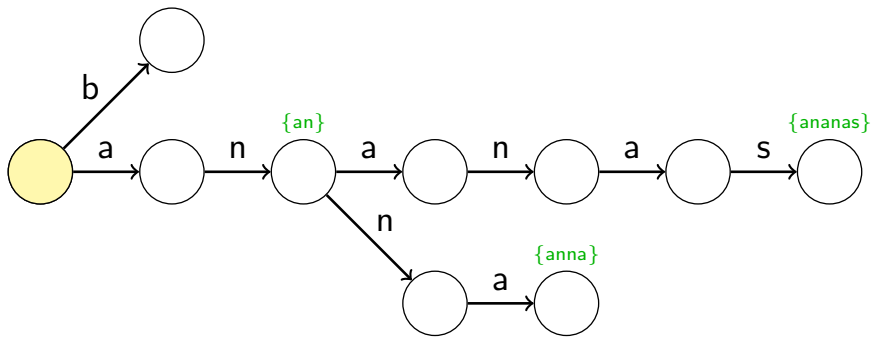


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

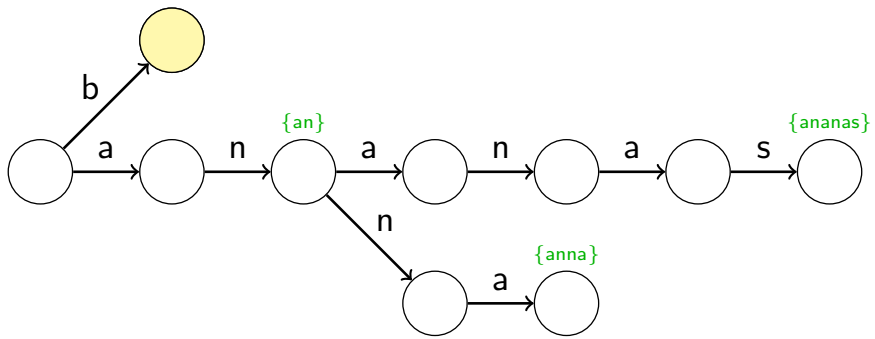


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

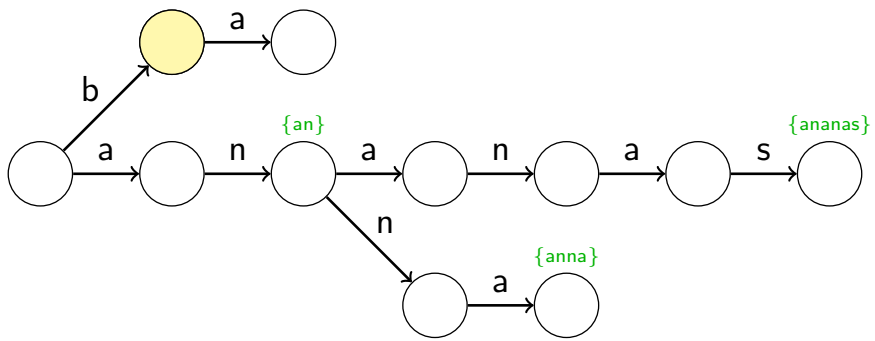


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

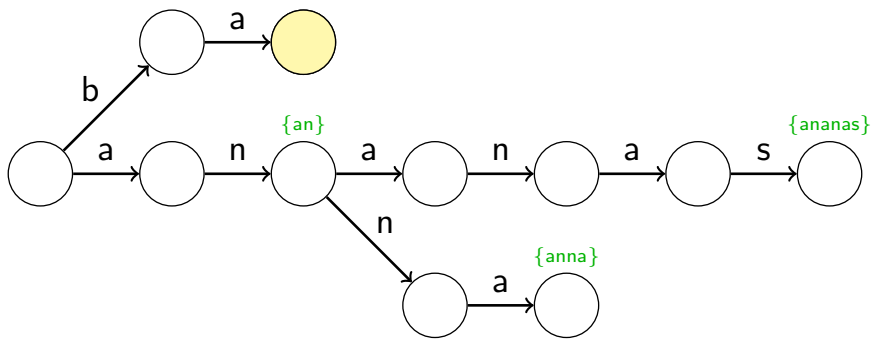


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

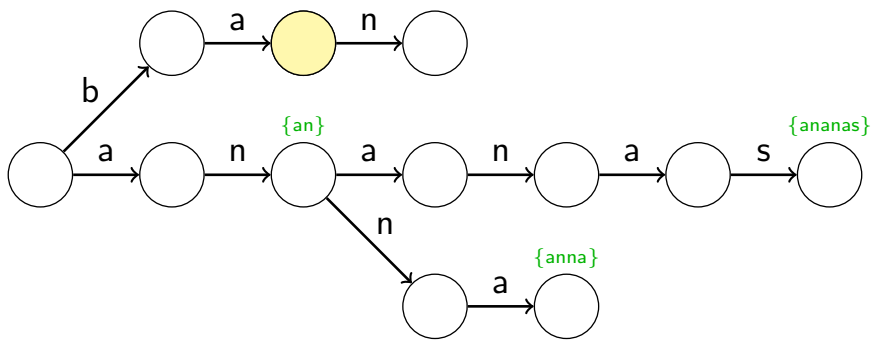


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

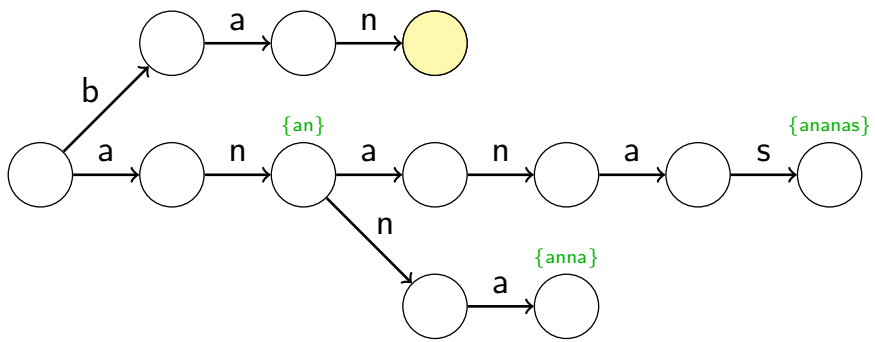


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

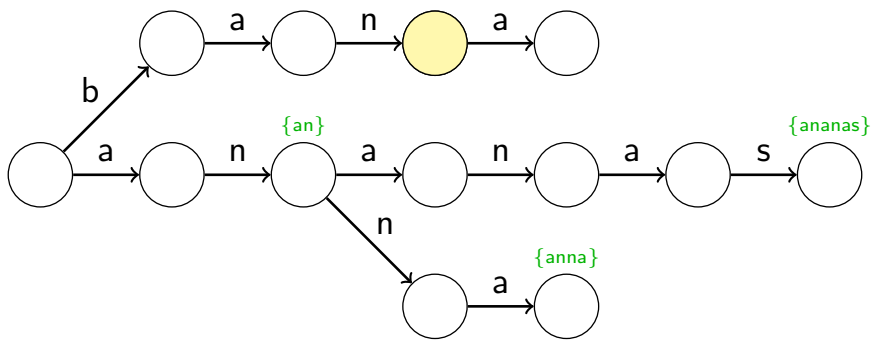


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

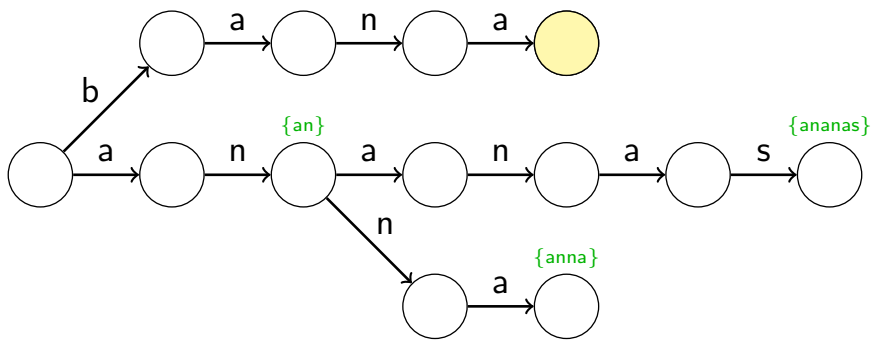


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

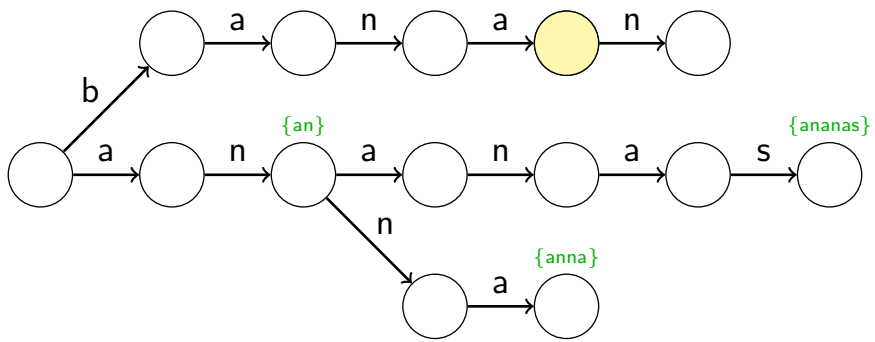


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

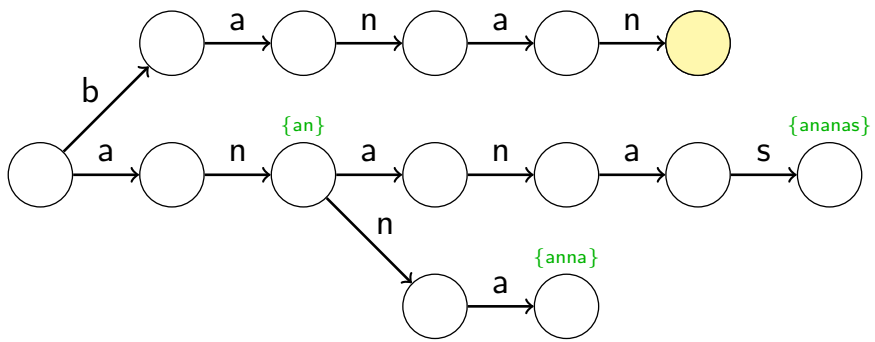


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

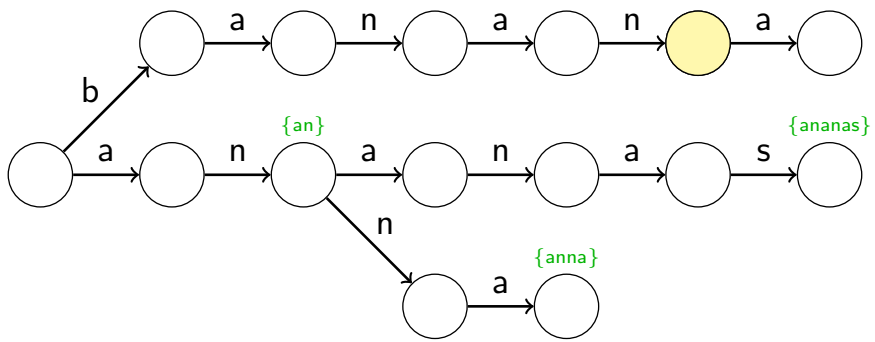


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

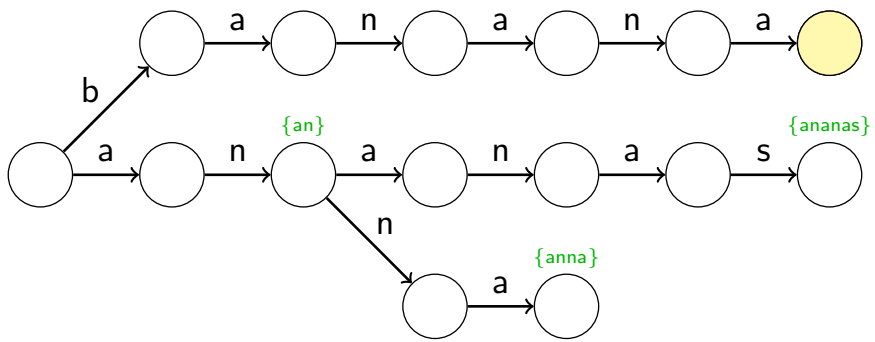


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

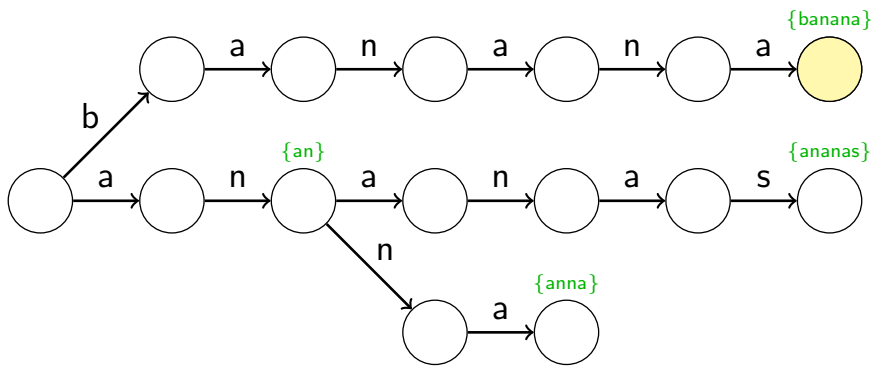


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

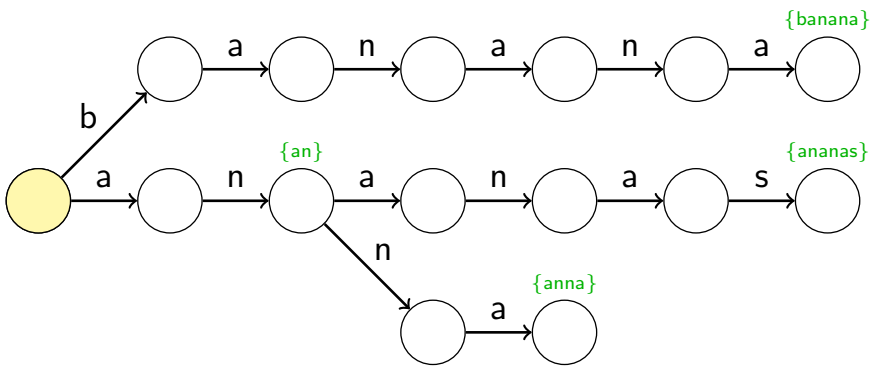


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

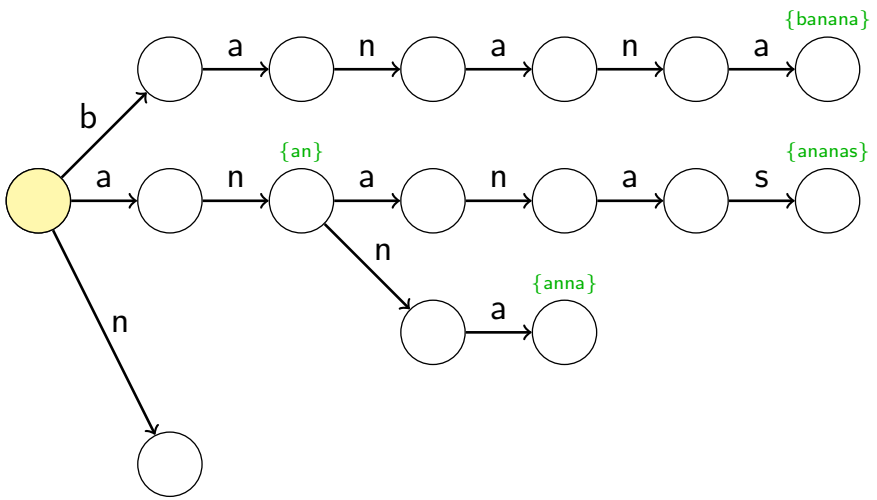


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

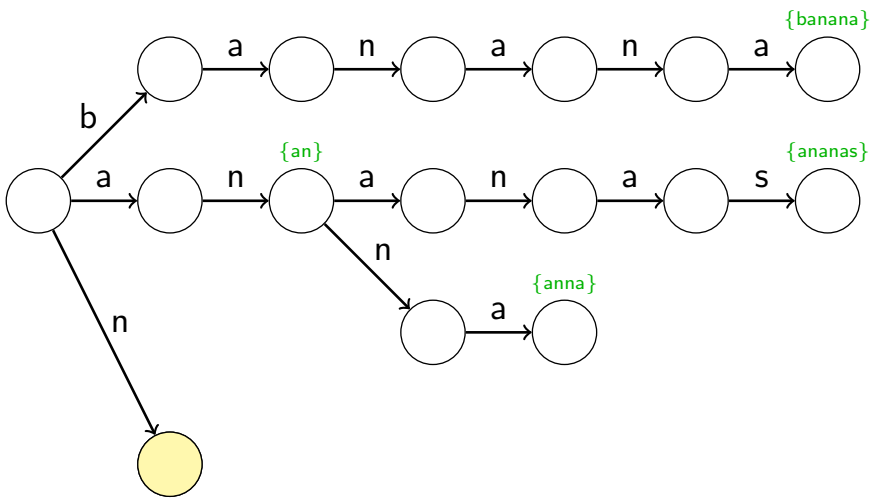


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

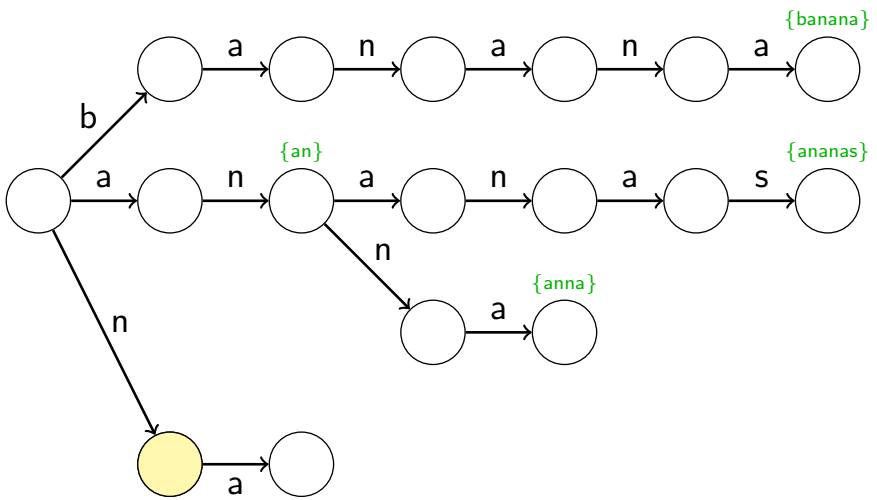


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

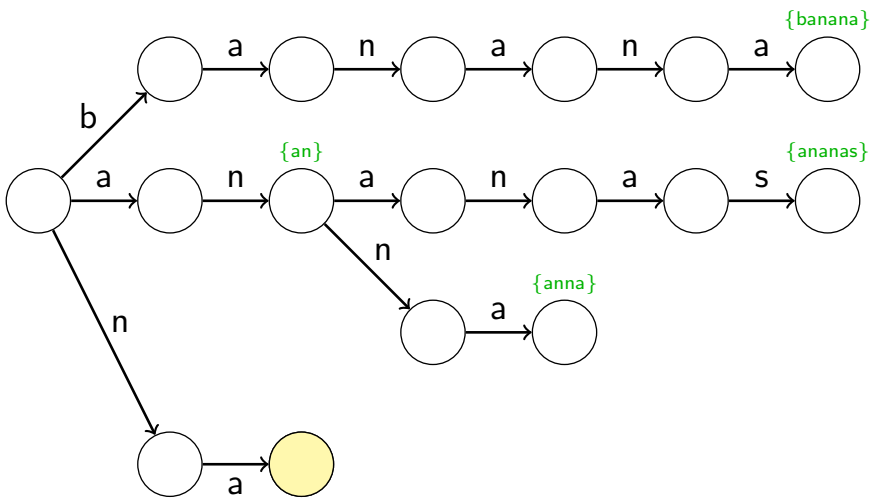


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

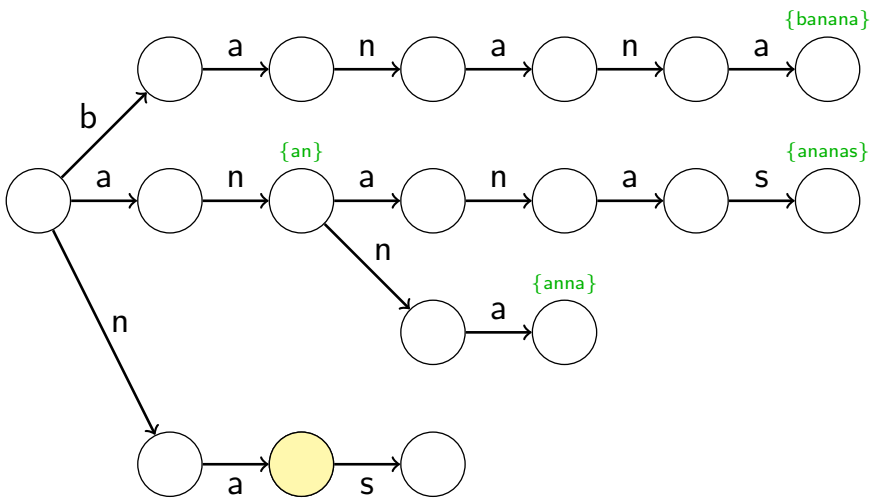


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

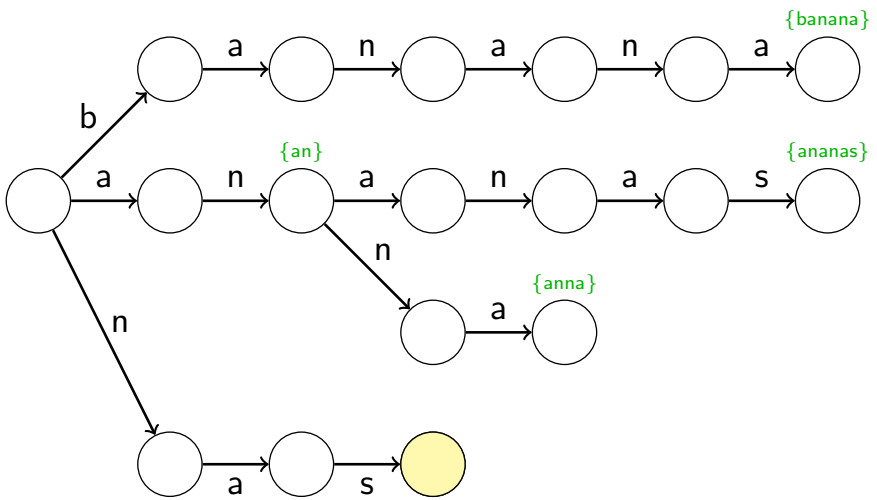


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

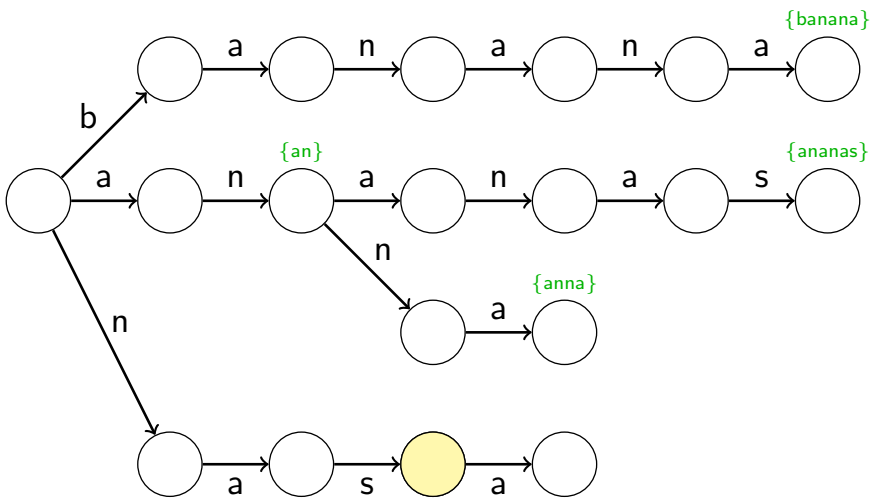


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

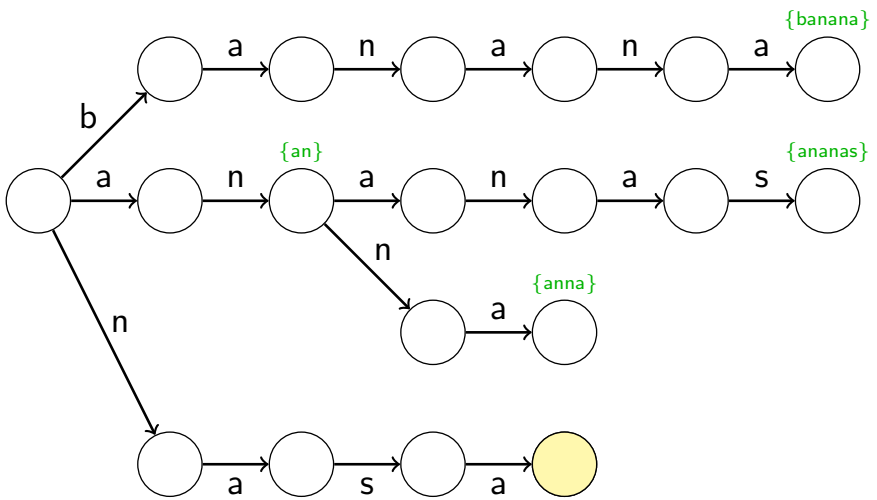


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

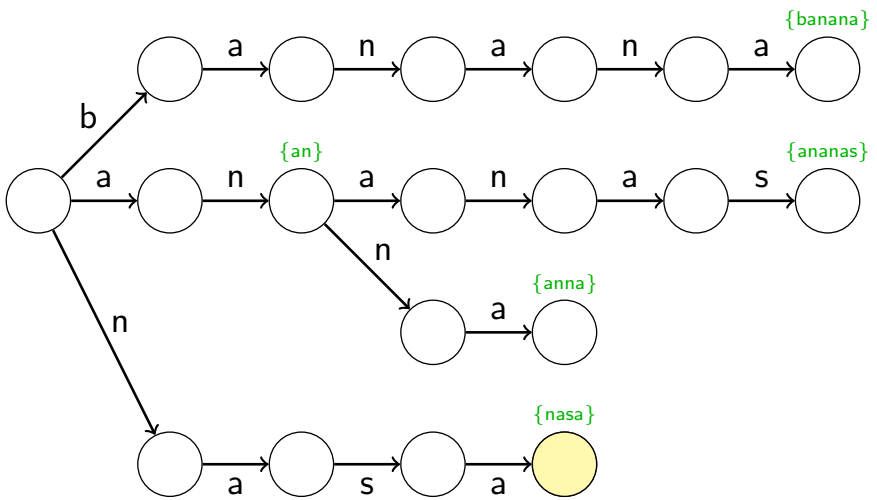


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

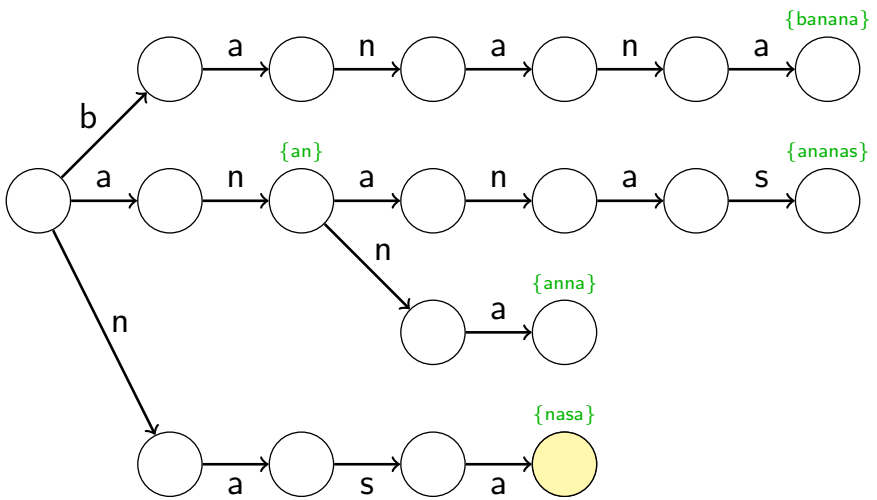


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

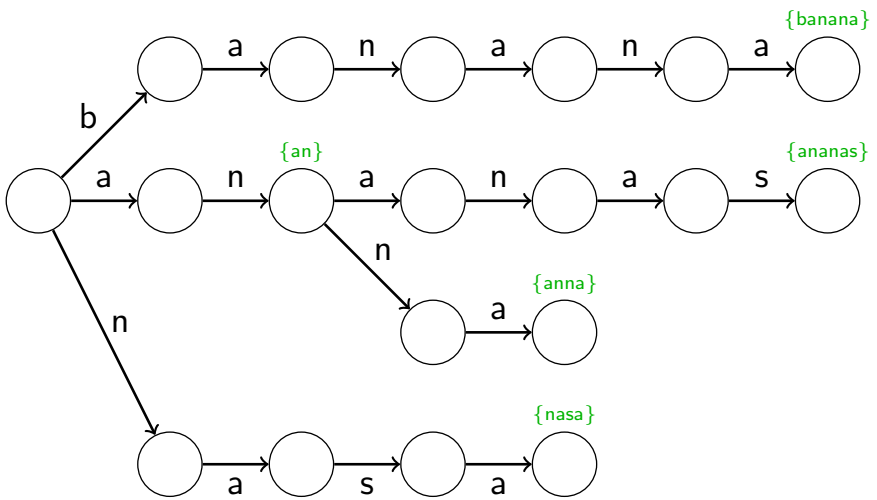


Der Aho-Corasick-Algorithmus

Erstellung des Tries:

```
1  def build_trie(P):
2      root = ACNode()
3      for (i, p) in enumerate(P):
4          node = root
5          for c in p:
6              if c not in node.targets:
7                  node.targets[c] = ACNode()
8              node = node.targets[c]
9          node.out.append(i)
10     return root
```

$P = \{$an,
ananas,
anna,
banana,
nasa$\}$

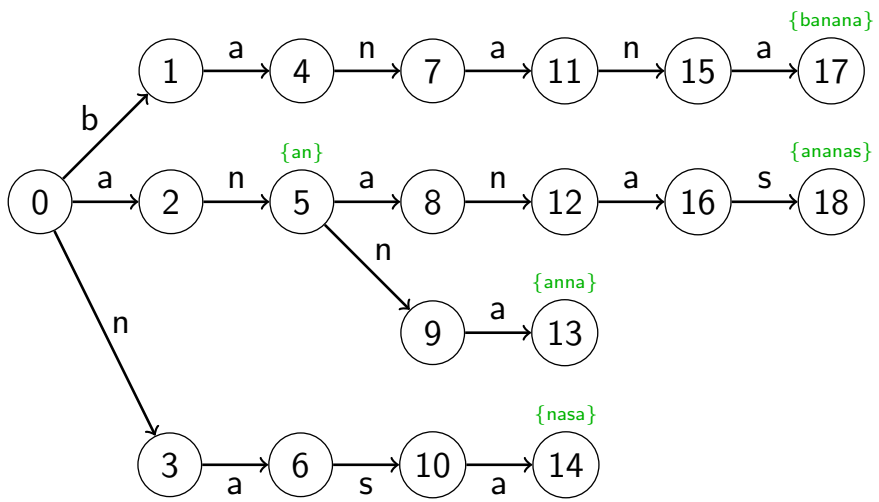


Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q =$ []



node =

letter =

parent =

Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

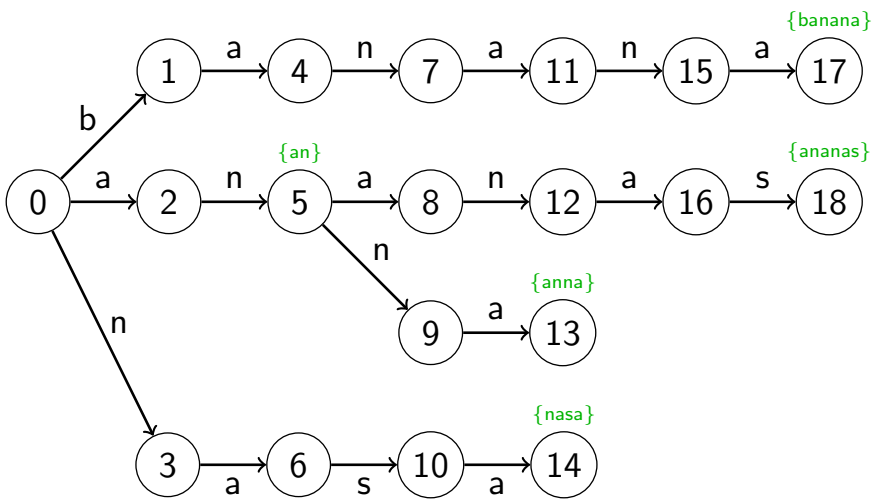
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q =$ $[(0, \emptyset, \emptyset)]$

node =

letter =

parent =



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

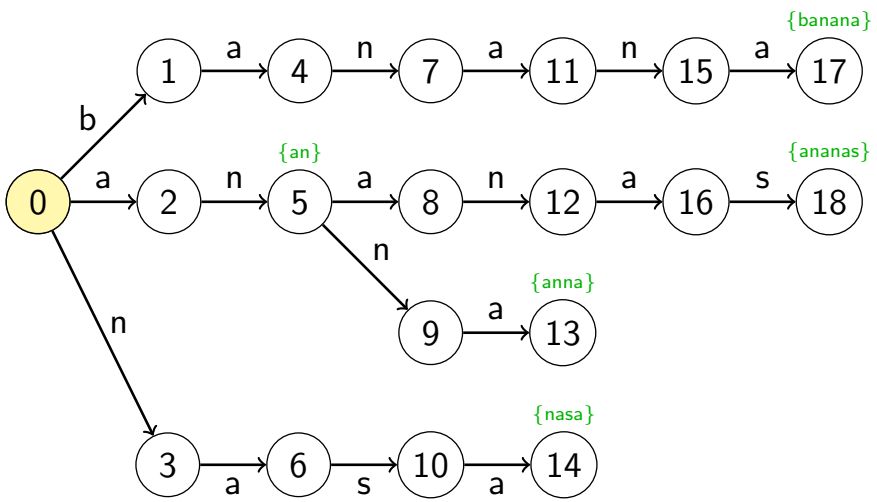
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q =$ []

$node =$ 0

$letter =$ $\emptyset$

$parent =$ $\emptyset$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

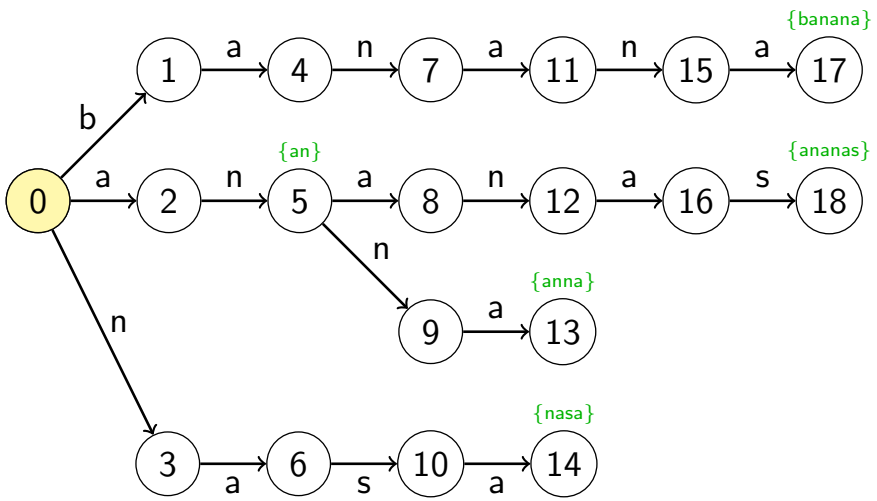
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(1, b, 0), (2, a, 0), (3, n, 0)]$

$node = 0$

$letter = \emptyset$

$parent = \emptyset$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

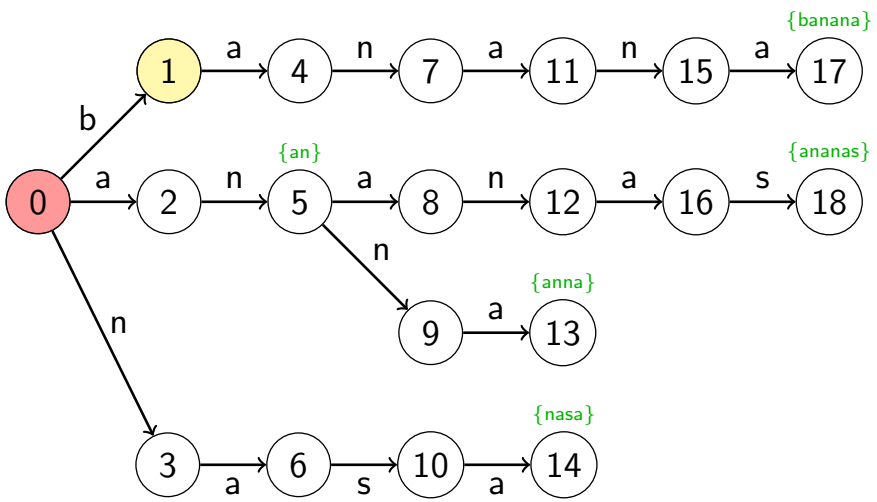
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(2, a, 0), (3, n, 0)]$

$node =$ 1

$letter =$ b

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

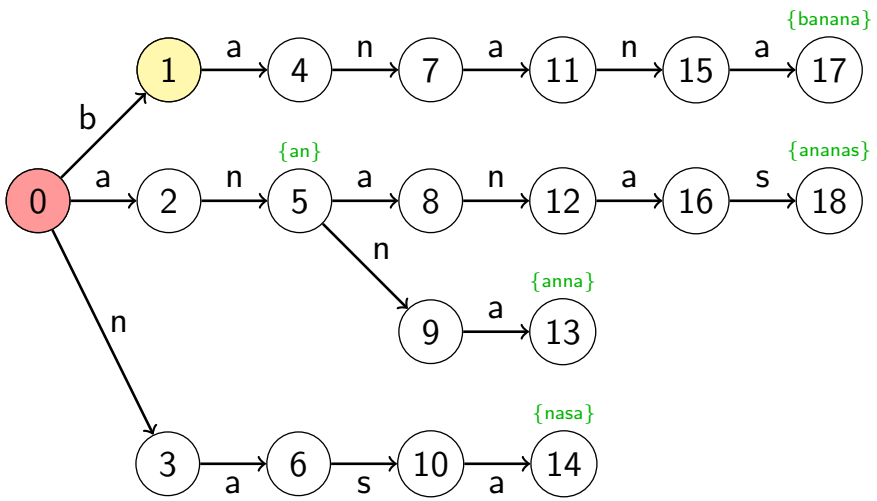
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(2, a, 0), (3, n, 0), (4, a, 1)]$

$node =$ 1

$letter =$ b

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

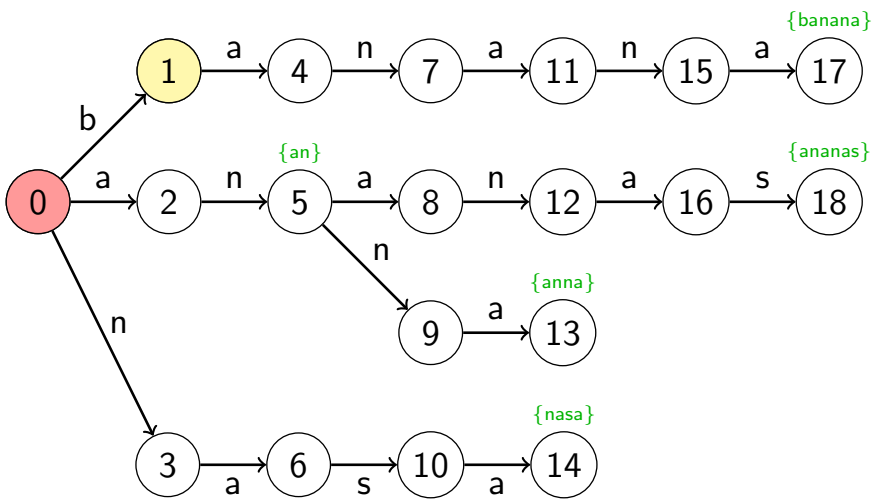
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(2, a, 0), (3, n, 0), (4, a, 1)]$

$node =$ 1

$letter =$ b

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

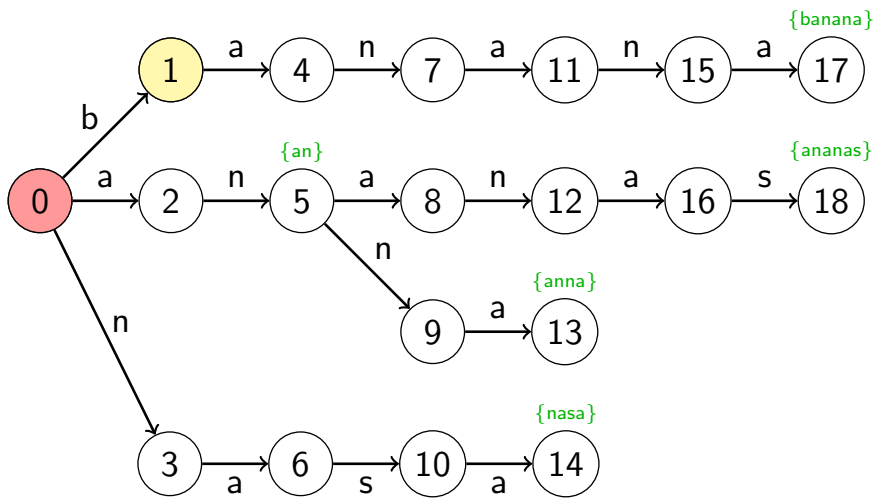
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(2, a, 0), (3, n, 0), (4, a, 1)]$

$node =$ 1

$letter =$ b

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

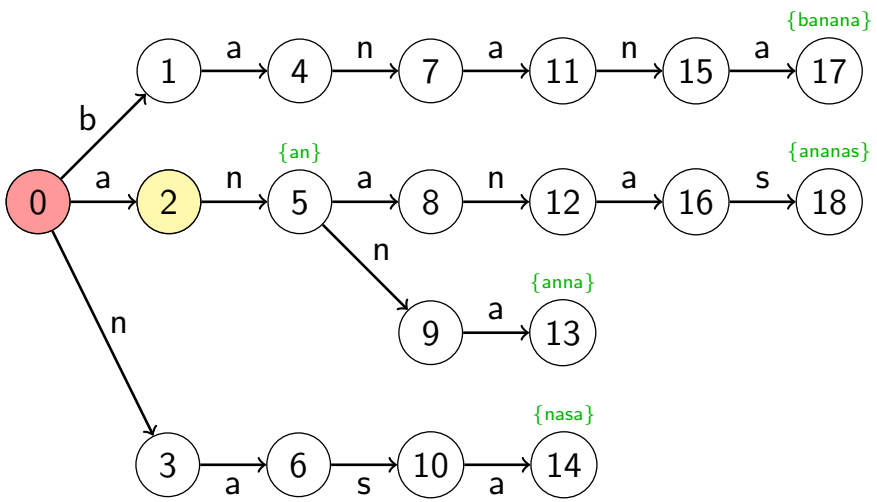
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(3, n, 0), (4, a, 1)]$

$node =$ 2

$letter =$ a

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

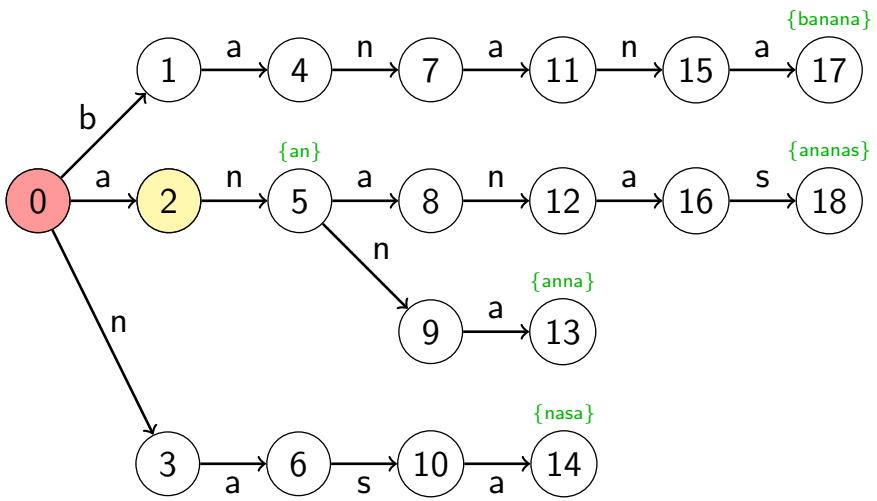
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(3, n, 0), (4, a, 1), (5, n, 2)]$

$node =$ 2

$letter =$ a

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

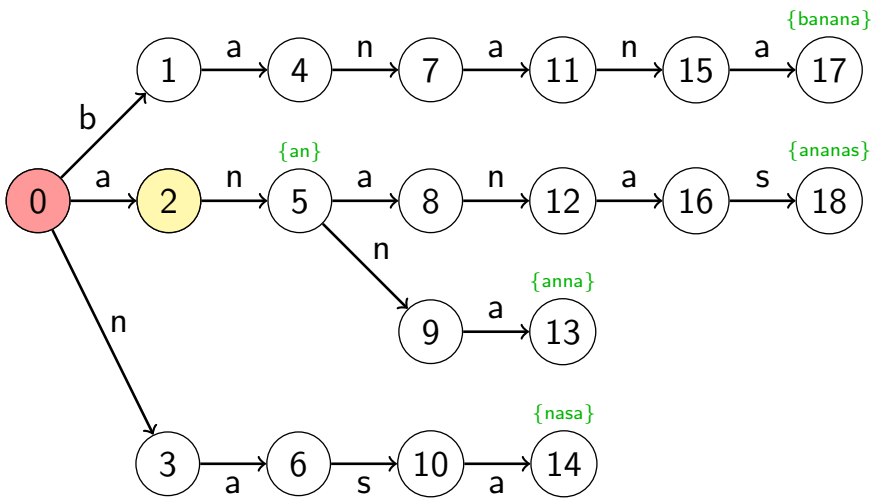
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(3, n, 0), (4, a, 1), (5, n, 2)]$

$node =$ 2

$letter =$ a

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

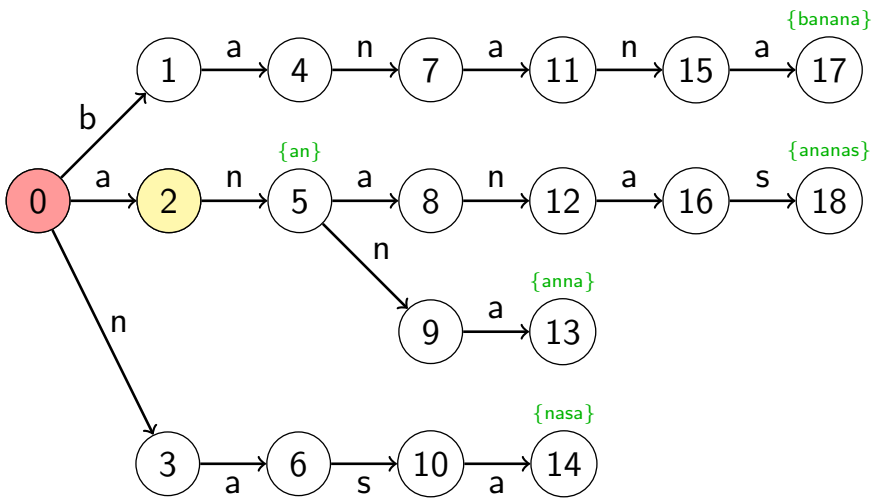
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(3, n, 0), (4, a, 1), (5, n, 2)]$

$node =$ 2

$letter =$ a

$parent =$ 0



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

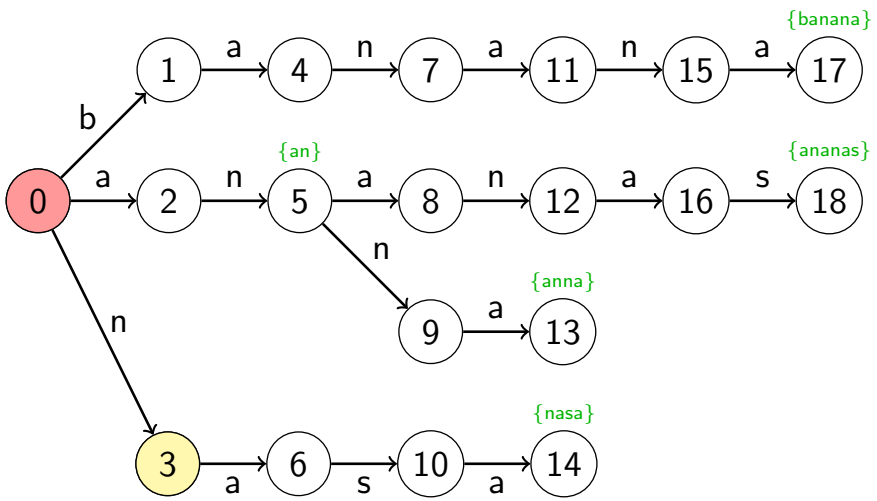
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(4, a, 1), (5, n, 2)]$

$node = 3$

$letter = \text{n}$

$parent = 0$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

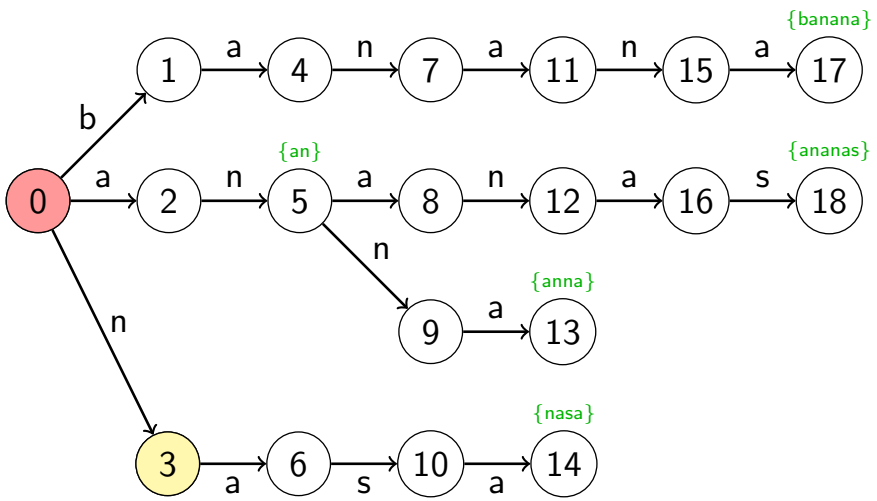
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(4, a, 1), (5, n, 2), (6, a, 3)]$

$node = 3$

$letter = \text{n}$

$parent = 0$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

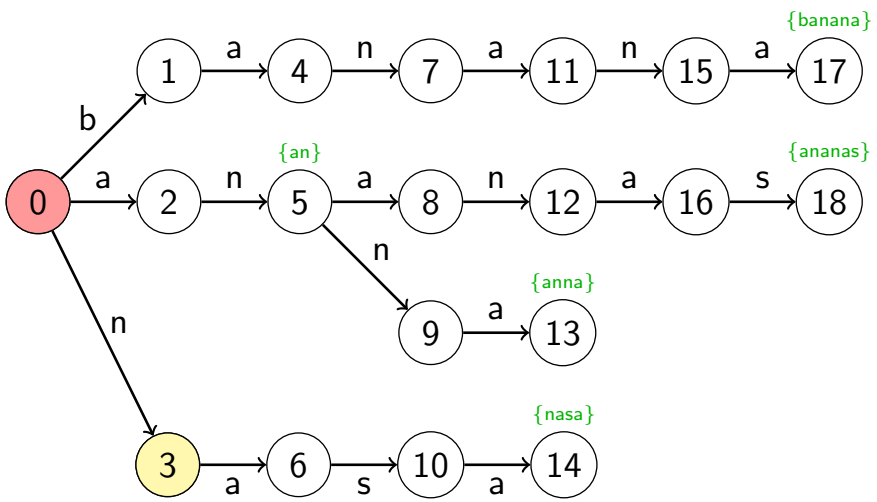
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(4, a, 1), (5, n, 2), (6, a, 3)]$

$node = 3$

$letter = \text{n}$

$parent = 0$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

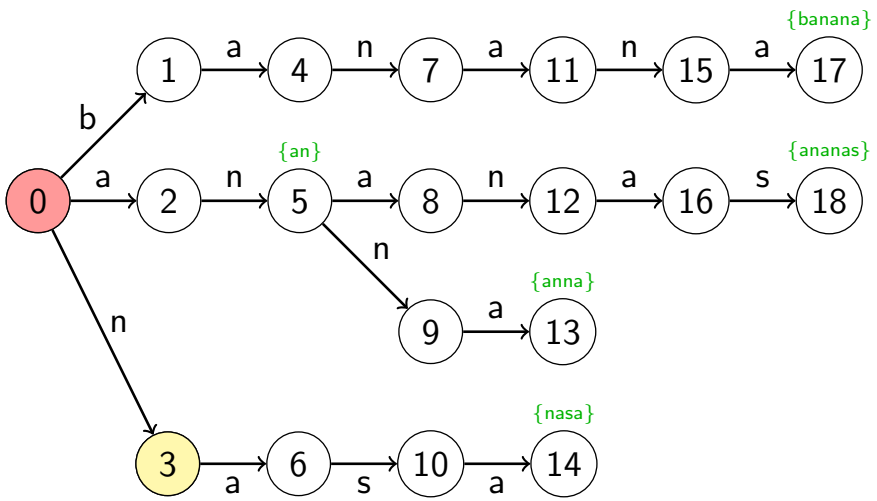
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(4, a, 1), (5, n, 2), (6, a, 3)]$

$node = 3$

$letter = n$

$parent = 0$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

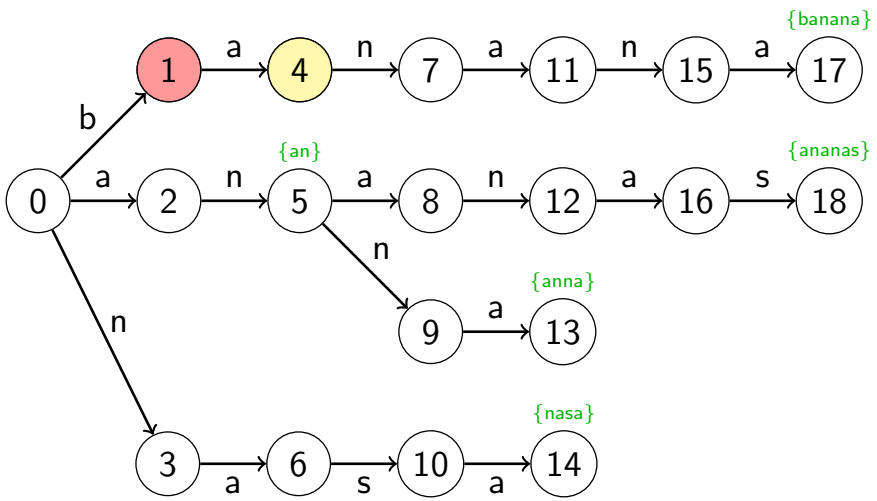
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(5, n, 2), (6, a, 3)]$

$node =$ 4

$letter =$ a

$parent =$ 1



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

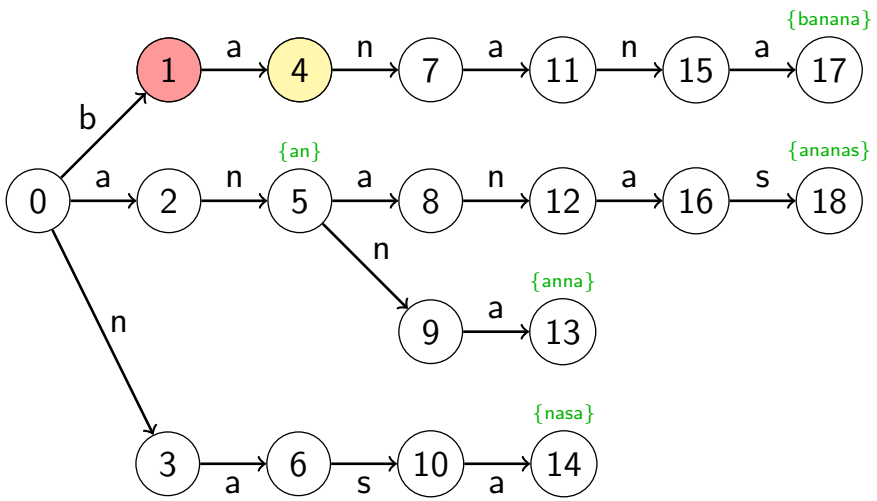
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(5, n, 2), (6, a, 3), (7, n, 4)]$

$node =$ 4

$letter =$ a

$parent =$ 1



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

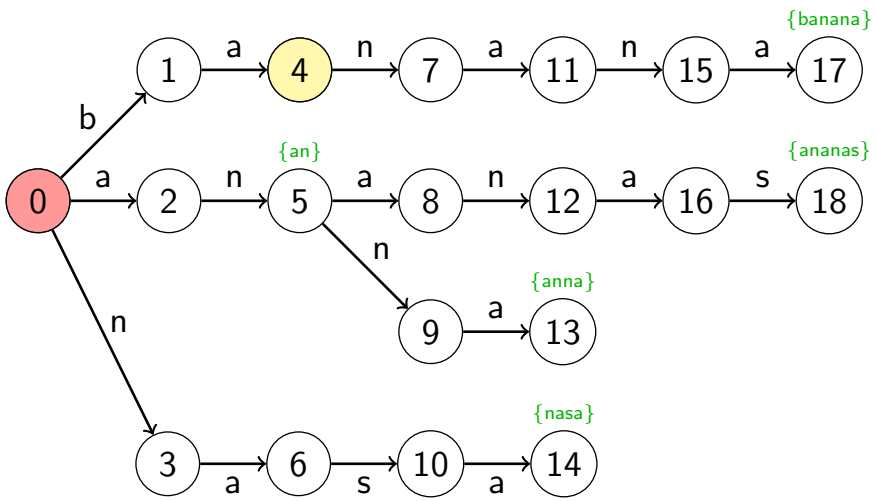
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(5, n, 2), (6, a, 3), (7, n, 4)]$

$node =$ 4

$letter =$ a

$parent =$ 1



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

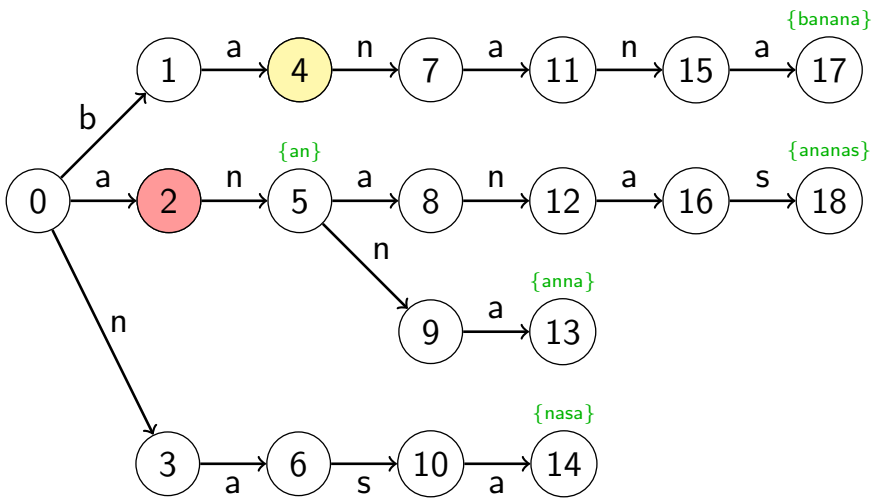
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(5, n, 2), (6, a, 3), (7, n, 4)]$

$node =$ 4

$letter =$ a

$parent =$ 1



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

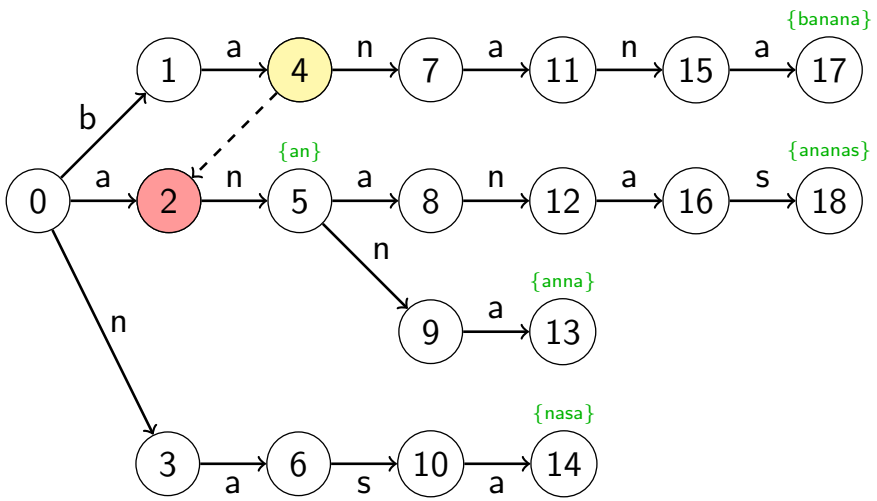
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(5, n, 2), (6, a, 3), (7, n, 4)]$

$node =$ 4

$letter =$ a

$parent =$ 1



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

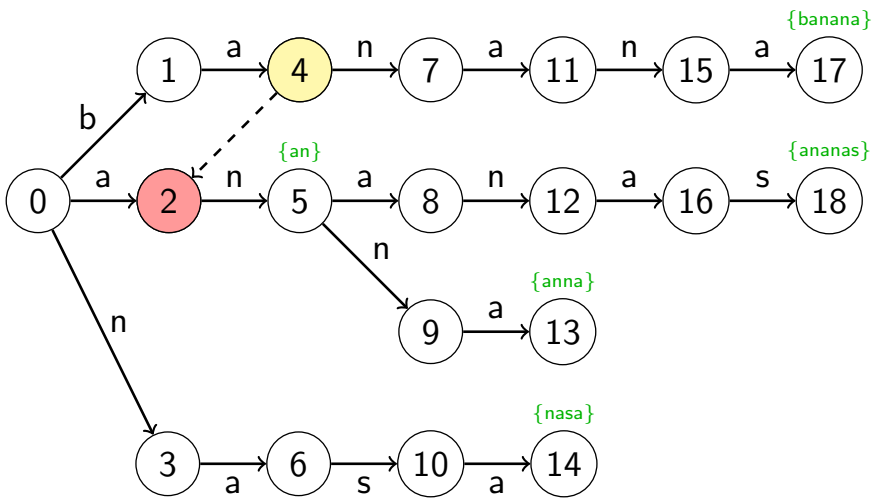
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(5, n, 2), (6, a, 3), (7, n, 4)]$

$node =$ 4

$letter =$ a

$parent =$ 1



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

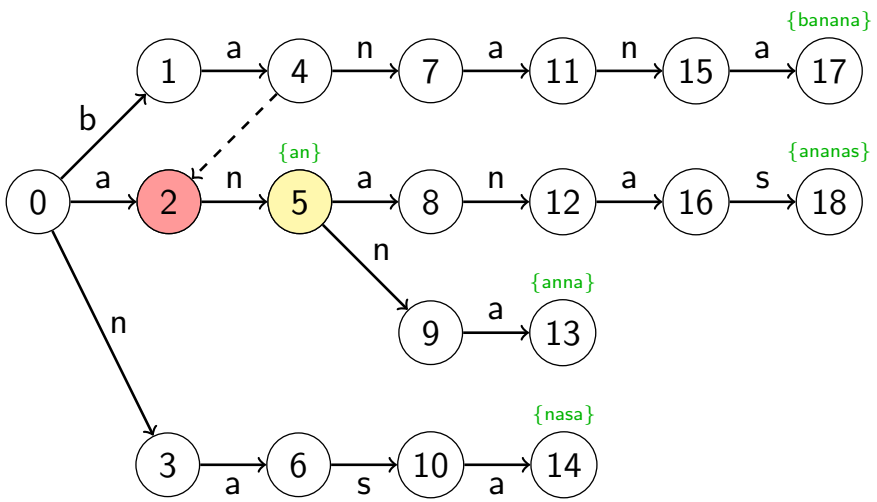
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(6, a, 3), (7, n, 4)]$

$node =$ 5

$letter =$ n

$parent =$ 2



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

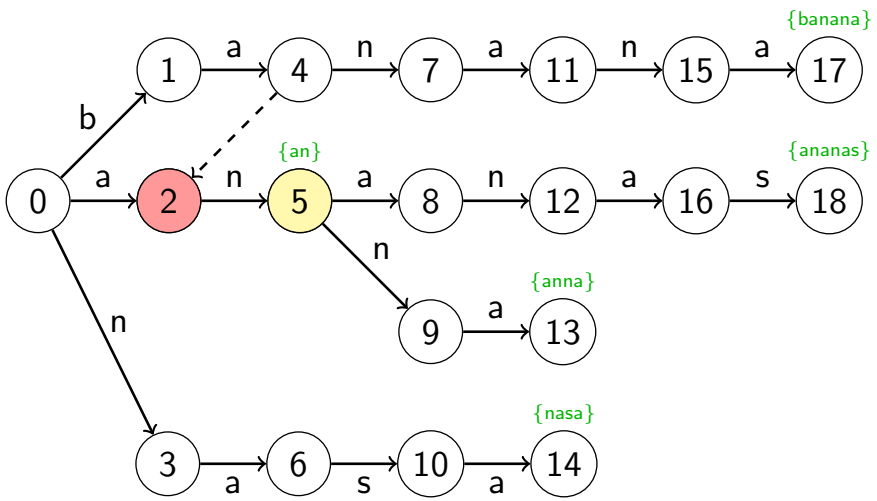
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(6, a, 3), (7, n, 4), (8, a, 5), (9, n, 5)]$

$node =$ 5

$letter =$ n

$parent =$ 2



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

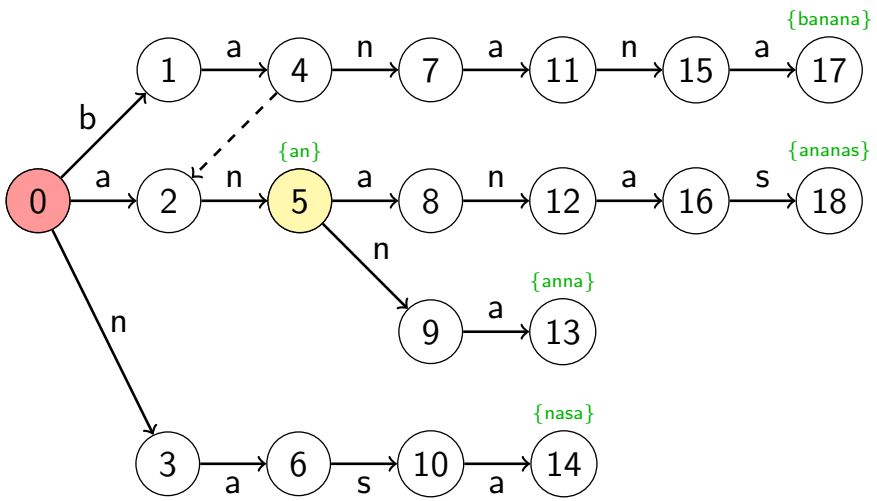
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q =$ $[(6, a, 3),$
$(7, n, 4),$
$(8, a, 5),$
$(9, n, 5)]$

$node =$ 5

$letter =$ n

$parent =$ 2



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

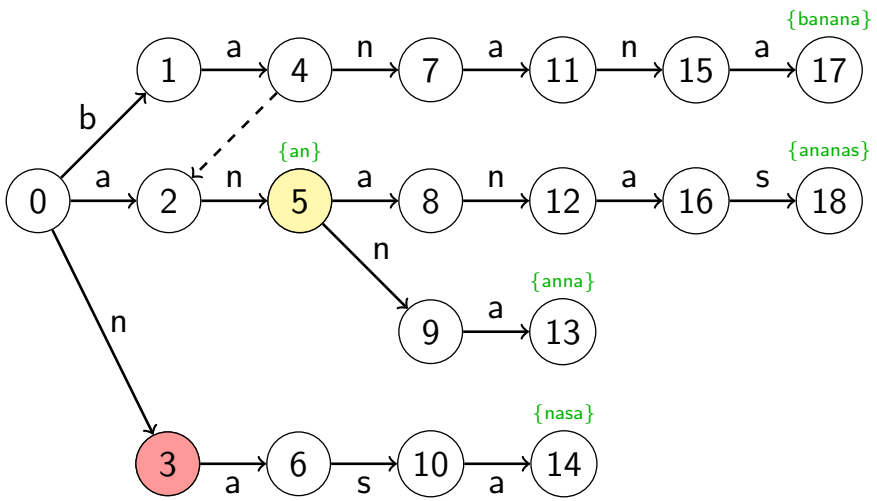
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(6, a, 3), (7, n, 4), (8, a, 5), (9, n, 5)]$

$node =$ 5

$letter =$ n

$parent =$ 2



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

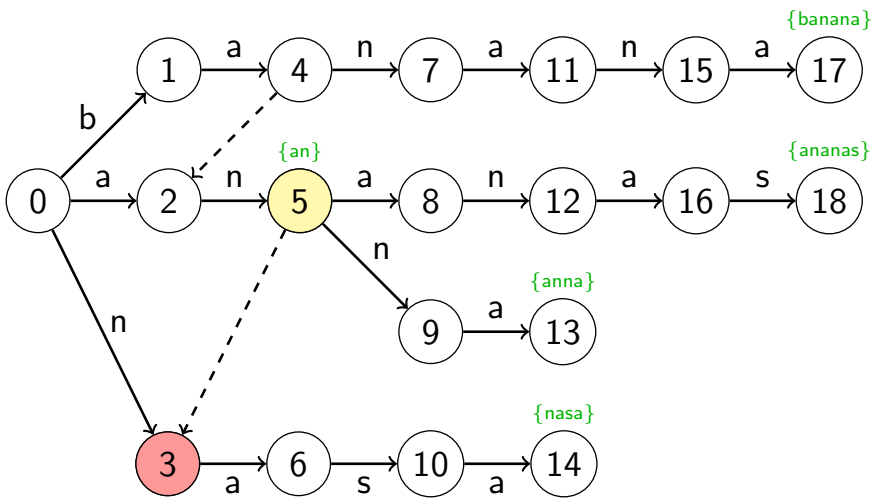
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(6, a, 3), (7, n, 4), (8, a, 5), (9, n, 5)]$

$node =$ 5

$letter =$ n

$parent =$ 2



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

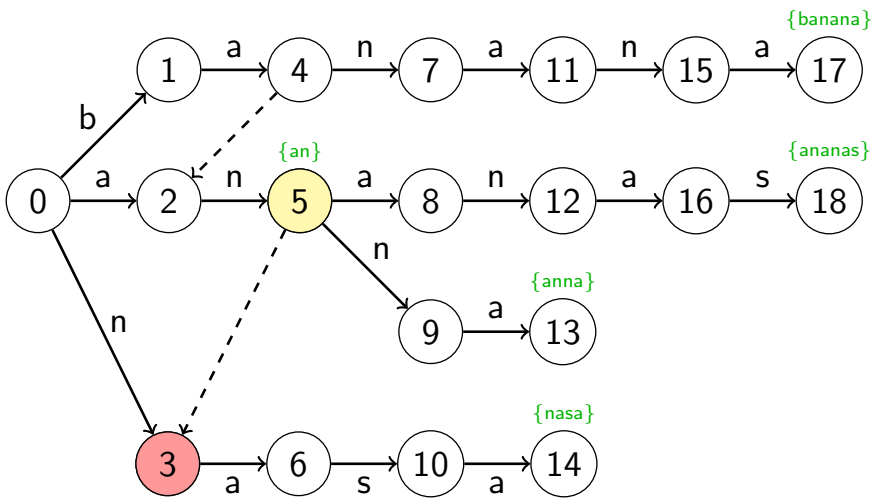
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(6, a, 3), (7, n, 4), (8, a, 5), (9, n, 5)]$

$node =$ 5

$letter =$ n

$parent =$ 2



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

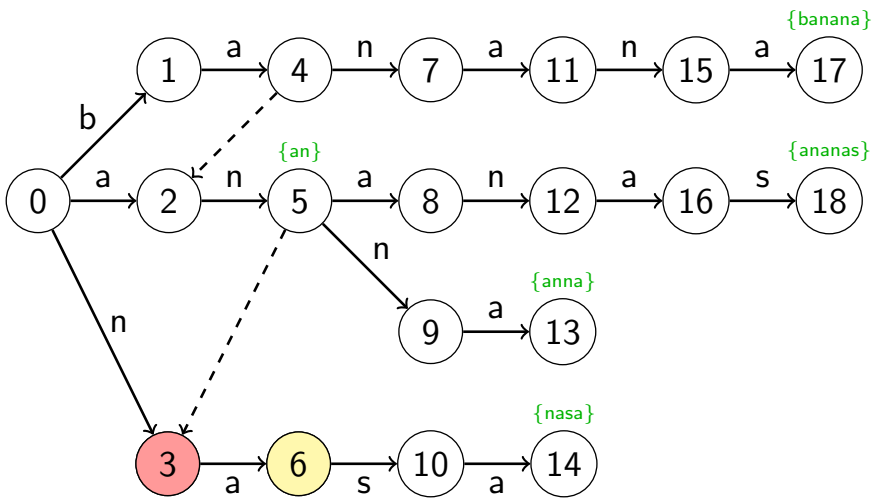
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(7, n, 4), (8, a, 5), (9, n, 5)]$

$node =$ 6

$letter =$ a

$parent =$ 3



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

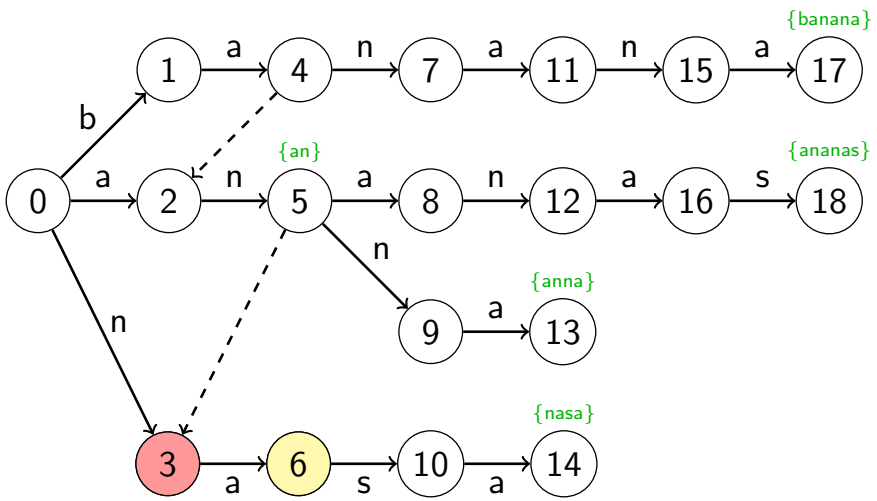
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(7, n, 4), (8, a, 5), (9, n, 5), (10, s, 6)]$

$node = 6$

$letter = a$

$parent = 3$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

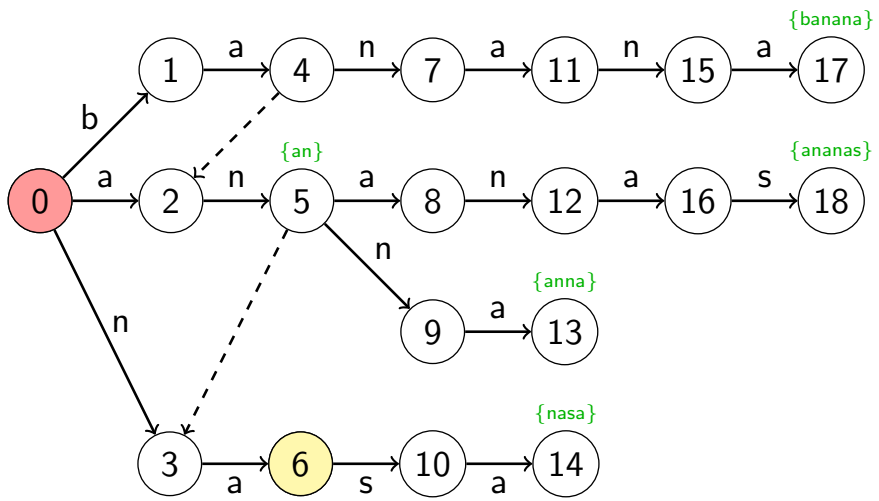
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(7, n, 4), (8, a, 5), (9, n, 5), (10, s, 6)]$

$node = 6$

$letter = a$

$parent = 3$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

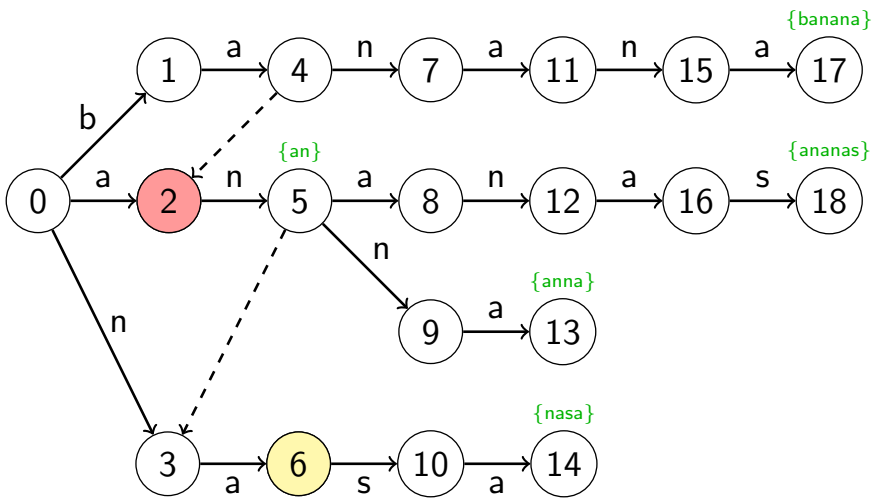
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(7, n, 4), (8, a, 5), (9, n, 5), (10, s, 6)]$

$node = 6$

$letter = a$

$parent = 3$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

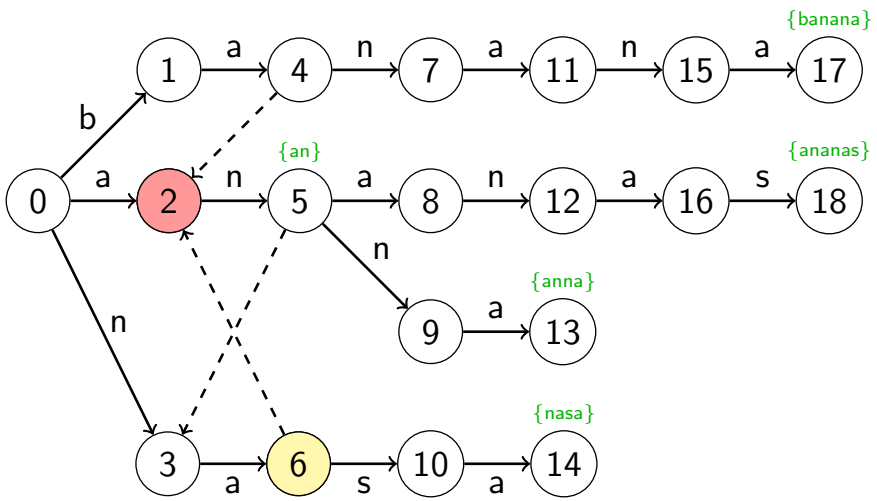
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(7, n, 4), (8, a, 5), (9, n, 5), (10, s, 6)]$

$node = 6$

$letter = a$

$parent = 3$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

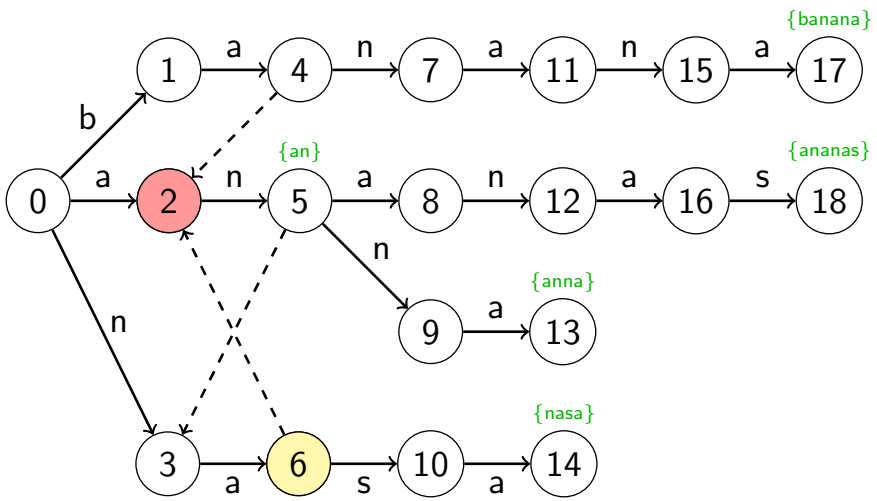
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(7, n, 4), (8, a, 5), (9, n, 5), (10, s, 6)]$

$node =$ 6

$letter =$ a

$parent =$ 3



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

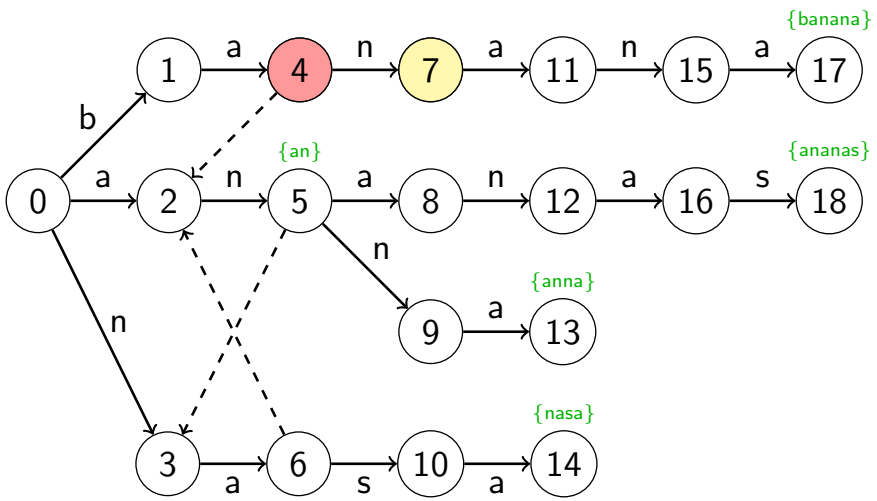
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(8, a, 5), (9, n, 5), (10, s, 6)]$

$node = 7$

$letter = \mathrm{n}$

$parent = 4$



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

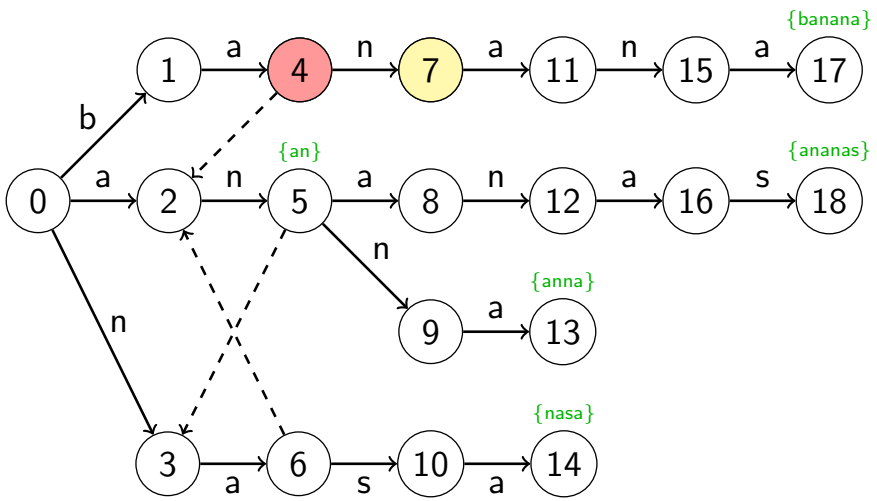
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(8, a, 5), (9, n, 5), (10, s, 6), (11, a, 7)]$

$node =$ 7

$letter =$ n

$parent =$ 4



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

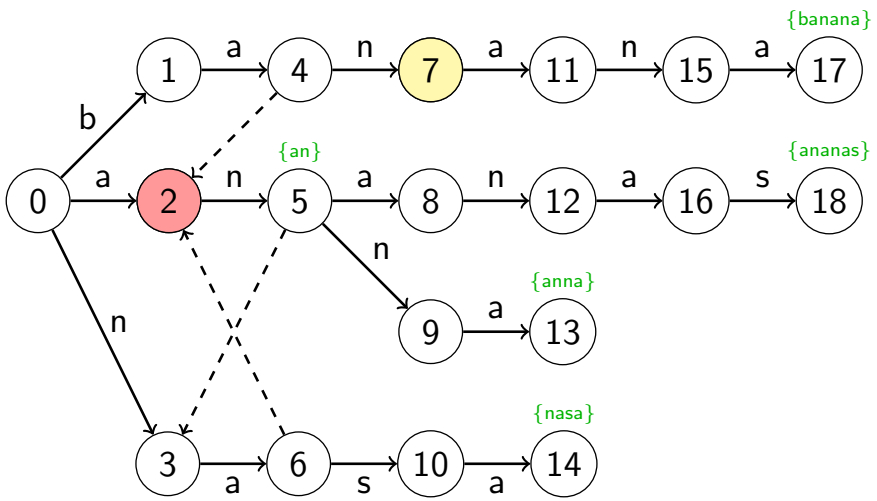
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(8, a, 5), (9, n, 5), (10, s, 6), (11, a, 7)]$

$node =$ 7

$letter =$ n

$parent =$ 4



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

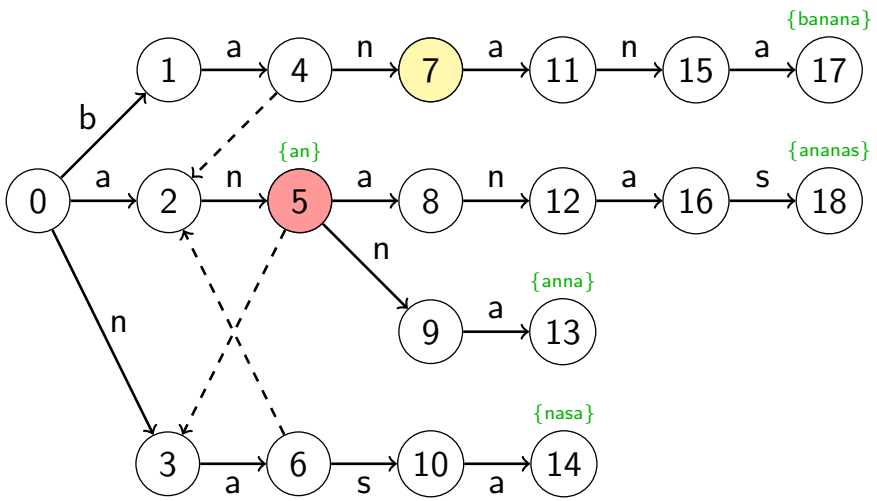
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(8, a, 5), (9, n, 5), (10, s, 6), (11, a, 7)]$

$node =$ 7

$letter =$ n

$parent =$ 4



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

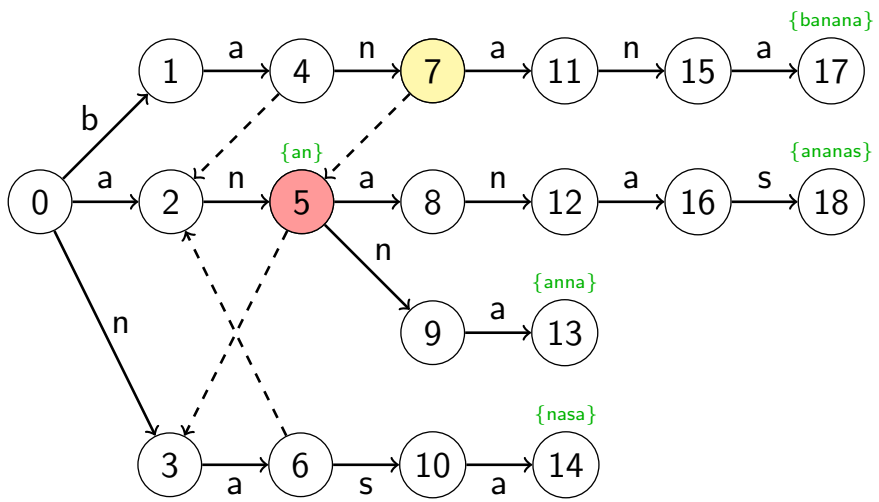
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q =$ $[(8, a, 5),$
$(9, n, 5),$
$(10, s, 6),$
$(11, a, 7)]$

$node =$ 7

$letter =$ n

$parent =$ 4



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

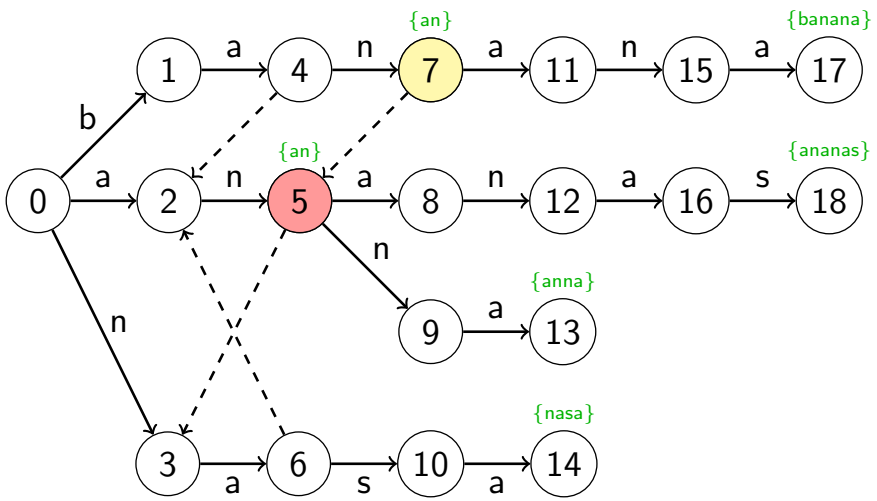
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q = [(8, a, 5), (9, n, 5), (10, s, 6), (11, a, 7)]$

$node =$ 7

$letter =$ n

$parent =$ 4



Der Aho-Corasick-Algorithmus

Erstellung der *lps*-Kanten:

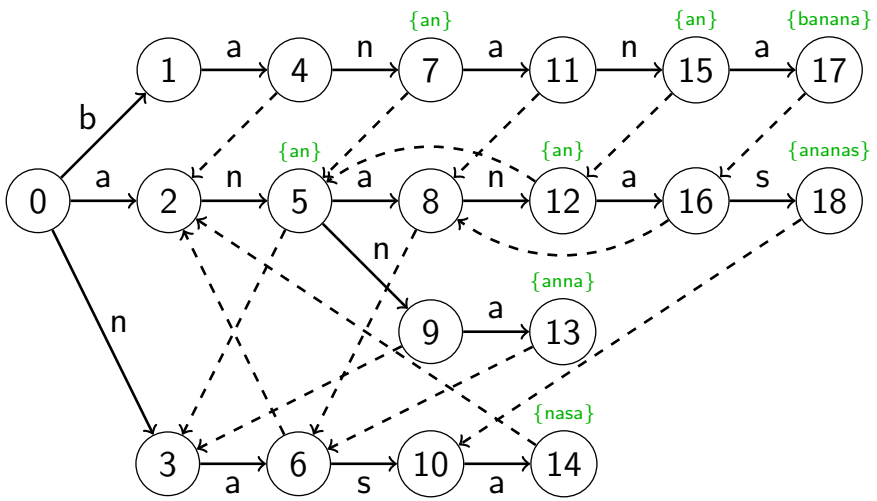
```
1  def build_lps(root):
2      # tuple with (child node, letter to child node, parent node)
3      Q = [(root, None, None)]
4      while len(Q):
5          node, letter, parent = Q.pop(0)
6          Q += [(v, k, node) for k, v in node.targets.items()]
7          if parent == None: continue
8          node.lps = delta(parent.lps, letter) if parent != root else root
9          node.out += node.lps.out
10     return root
```

$Q =$

$node =$

$letter =$

$parent =$



Der Aho-Corasick-Algorithmus

Mustersuche mit dem Aho-Corasick-Algorithmus:

```
1 def aho_corasick(P, T):
2     q = build_AC(P)
3     for (i, c) in enumerate(T):
4         q = delta(q, c)
5         for x in q.out:
6             yield i - len(P[x]) + 1, i + 1, x
```

Laufzeitanalyse

- Ähnlich wie beim KMP wird die while-Schleife in der *delta*-Funktion amortisiert beim Erstellen des Automaten nur $\mathcal{O}(m)$ und bei der Mustersuche nur $\mathcal{O}(n)$ mal betreten.

Laufzeitanalyse

- Ähnlich wie beim KMP wird die while-Schleife in der *delta*-Funktion amortisiert beim Erstellen des Automaten nur $\mathcal{O}(m)$ und bei der Mustersuche nur $\mathcal{O}(n)$ mal betreten.
- Beim Verknüpfen der Ausgabelisten sind alle erkannten Pattern in der Liste des lps-Vorgängers kürzer als die Pattern in der Liste des aktuellen Knotens $\Rightarrow$ Beide Listen sind disjunkt, Verknüpfung in $\mathcal{O}(1)$.

Laufzeitanalyse

- Ähnlich wie beim KMP wird die while-Schleife in der *delta*-Funktion amortisiert beim Erstellen des Automaten nur $\mathcal{O}(m)$ und bei der Mustersuche nur $\mathcal{O}(n)$ mal betreten.
- Beim Verknüpfen der Ausgabelisten sind alle erkannten Pattern in der Liste des lps-Vorgängers kürzer als die Pattern in der Liste des aktuellen Knotens $\Rightarrow$ Beide Listen sind disjunkt, Verknüpfung in $\mathcal{O}(1)$.
- Der AC-Algorithmus braucht für die Mustersuche und das Erkennen(!!!) eines Zustandes mit nichtleerer Ausgabemenge $\mathcal{O}(m+n)$.

Laufzeitanalyse

- Im ungünstigsten Fall werden pro Buchstabe im Text bis zu $\mathcal{O}(k)$ Pattern erkannt, somit beträgt die komplette Worst-case-Laufzeit samt Ausgabe $\mathcal{O}(m + kn)$.

Laufzeitanalyse

- Im ungünstigsten Fall werden pro Buchstabe im Text bis zu $\mathcal{O}(k)$ Pattern erkannt, somit beträgt die komplette Worst-case-Laufzeit samt Ausgabe $\mathcal{O}(m + kn)$.
- Da bei jedem multiplen Pattern-Matching-Algorithmus dieses Problem auftritt, wird angenommen, dass im Durchschnitt pro Knoten maximal ein Pattern erkannt wird.
- Somit beträgt die erwartete Laufzeit $\mathcal{O}(m + n)$, welche oft angegeben wird.

Zusammenfassung

- Der Aho-Corasick-Algorithmus ist eine Erweiterung des KMP-Algorithmus auf eine Patternmenge.
- Die erwartete Laufzeit samt Ausgabe beträgt $\mathcal{O}(m + n)$.

Zusammenfassung

- Der Aho-Corasick-Algorithmus ist eine Erweiterung des KMP-Algorithmus auf eine Patternmenge.
- Die erwartete Laufzeit samt Ausgabe beträgt $\mathcal{O}(m+n)$.
- Bei der gezeigten konzeptionellen Implementierung beträgt der Speicherplatz im Worst-case $O(n+km)$.
- Wenn die Ausgabelisten nicht konkateniert, sondern nur Verweise auf die Vorgängerlisten gespeichert werden, sinkt der Speicherplatz auf $O(n+m)$.