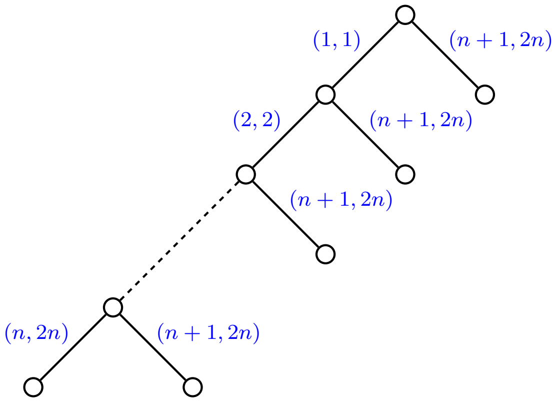


Abbildung 2.45: Beispiel: Echter Suffix-Baum für$t = a^n b^n$

Damit hat der Suffix-Baum nicht nur $O(|t|)$ viele Knoten, sondern er benötigt auch für die Verwaltung der Kantenlabels nur linear viel Platz (anstatt $O(n^2)$).

2.6.6 Ukkonens Online-Algorithmus für Suffix-Bäume

Im Folgenden bezeichnen wir mit $\hat{T}^i$ den Suffix-Baum für $t_1 \cdots t_i$. Wir wollen nun den Suffix-Baum $\hat{T}^i$ aus dem $\hat{T}^{i-1}$ aufbauen. Diese Konstruktion kann man aus der Konstruktion des Suffix-Tries T^i aus dem Suffix-Trie T^{i-1} erhalten.

Dazu folgende Festlegungen:

- Sei $t \in \Sigma^*$, wobei $t = t_1 \cdots t_{i-1}$.
- Sei $s_j = t_j \cdots t_{i-1}$ und sei $\overline{s_j}$ der zugehörige Knoten im Suffix-Trie, dabei sei $\overline{s_i} = \overline{\varepsilon} = root$, und $\overline{s_{i+1}} = \perp$, wobei $\perp$ die virtuelle Wurzel ist, von der für jedes Zeichen $a \in \Sigma$ eine Kante zur eigentlichen Wurzel führt.
- Sei l der größte Index, so dass für alle $j < l$ im Suffix-Trie T^{i-1} eine neue Kante für t_i an den Knoten $\overline{s_j}$ angehängt werden muss ($\rightarrow \overline{s_l}$ heißt Endknoten).
- Sei k der größte Index, so dass für alle $j < k$ $\overline{s_j}$ ein Blatt ist ($\rightarrow \overline{s_k}$ heißt aktiver Knoten).

Da $\overline{s_1}$ immer ein Blatt ist, gilt $k > 1$, und da $\overline{s_{i+1}} = \perp$ immer die Kante mit dem Label t_i besitzt, gilt $l \leq i + 1$.

Sei w ein Teilwort von t, dann heißt ein Knoten $\overline{w}$ des Suffix-Tries *explizit*, wenn es auch im Suffix-Baum einen Knoten gibt, der über die Zeichenreihe w erreichbar ist. Andernfalls heißt er implizit.

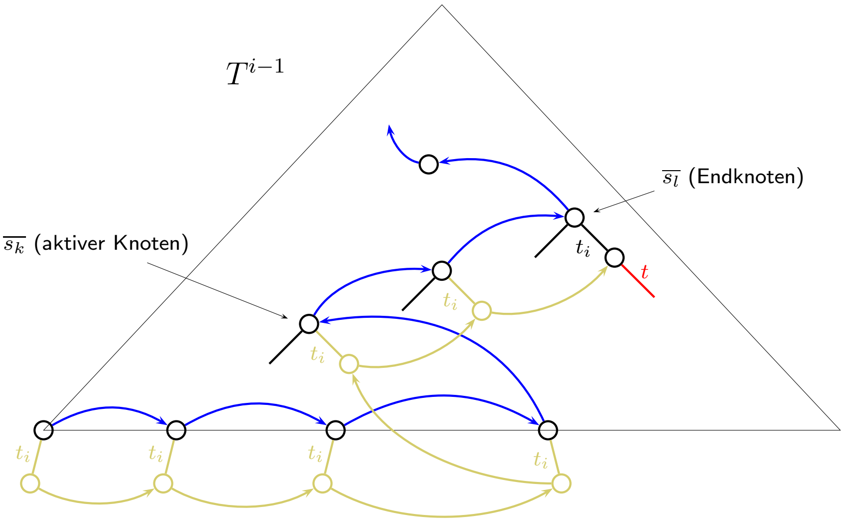


Abbildung 2.46: Skizze: Konstruktion eines Suffix-Tries

Es ist leicht einsehbar, dass die eigentliche Arbeit beim Aufbau des Suffix-Tries zwischen dem aktiven Knoten und dem Endknoten zu verrichten ist. Bei allen Knoten „vor" dem aktiven Knoten, also bei allen bisherigen Blättern, muss einfach ein Kind mit dem Kantenlabel t_i eingefügt werden. Im Suffix-Baum bedeutet dies, dass an das betreffende Kantenlabel einfach das Zeichen t_i angehängt wird. Um sich nun diese Arbeit beim Aufbau des Suffix-Baumes zu sparen, werden statt der Teilwörter als Kantenlabels so genannte (offene) Referenzen benutzt.

Definition 2.20 *Sei w ein Teilwort von $t_1 \cdots t_n$ mit $w = uv = t_j \cdots t_{k-1} t_k \cdots t_p$, wobei $u = t_j \cdots t_{k-1}$ und $v = t_k \cdots t_p$. Ist $s = \overline{u}$ im Suffix-Baum realisiert, dann heißt $(s, (k, p))$ eine* Referenz *für $\overline{w}$ (im Suffix-Trie).*

Beispiele für Referenzen in $t = ababbaa$ (siehe auch Abbildung 2.48):

1. $w = \genfrac{}{}{0pt}{}{babb}{2345} \mathrel{\hat{=}} (\overline{b}, (3, 5)) \mathrel{\hat{=}} (\overline{ba}, (4, 5))$.
2. $w = \genfrac{}{}{0pt}{}{bbaa}{4567} \mathrel{\hat{=}} (\overline{b}, (5, 7)) \mathrel{\hat{=}} (\overline{bb}, (6, 7))$.

Definition 2.21 *Eine Referenz (s,(k,p)) heißt* kanonisch, *wenn $p - k$ minimal ist.*

Für Blätter des Suffix-Baumes werden alle Referenzen *offen* angegeben, d.h. es wird $(s, (k, \infty))$ statt $(s, (k, n))$ in $\hat{T}^n$ verwendet. Dadurch spart man sich dann die Arbeit, die Kantenlabels der Kanten, die zu Blättern führen, während der Konstruktion des Suffixbaumes zu verlängern.

Es stellt sich nun noch die Frage, ob, wenn einmal ein innerer Knoten erreicht wurde, auch wirklich kein Blatt mehr um ein Kind erweitert wird, sondern nur noch innere Knoten. Desweiteren ist noch von Interesse, ob, nachdem der Endknoten erreicht wurde, jeder weitere Knoten auch eine Kante mit dem Label t_i besitzt.

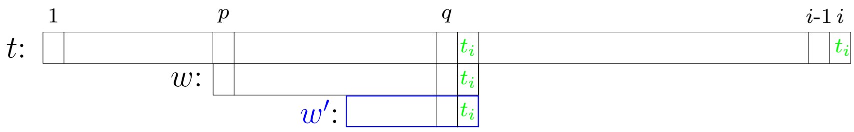


Abbildung 2.47: Skizze: Endknoten beendet Erweiterung von T^{i-1}

Dazu betrachten wir obige Skizze. Zeigt ein Suffix-Link einmal auf einen inneren Knoten w, bedeutet dies, dass die Kantenlabels bis zu diesem Knoten ein Teilwort von t ergeben, das nicht nur Suffix von t ist, sondern auch sonst irgendwo in t auftritt (siehe w in der Skizze). Da nun jeder weitere Suffix-Link auf einen Knoten verweist, der ein Suffix des zuvor gefundenen Teilwortes darstellt ($\rightarrow$ blauer String w' in der Skizze), muss auch dieser Knoten ein innerer sein, da seine Kantenlabels ebenfalls nicht nur Suffix von t sind, sondern auch noch an einer anderen Stelle in t auftauchen müssen.

Hat man nun den Endknoten erreicht, bedeutet dies, dass der betreffende Knoten bereits eine t_i Kante besitzt. Somit ergeben die Kantenlabels von der Wurzel bis zu diesem Knoten ein Teilwort von t, das nicht nur Suffix von t ist, sondern auch sonst irgendwo mitten in t auftritt. Außerdem kann aufgrund der t_i Kante an das Teilwort das Zeichen t_i angehängt werden. Da nun jeder weitere Suffix-Link auf einen Knoten zeigt, der ein Suffix des vom Endknoten repräsentierten Wortes darstellt, muss auch an das neue Teilwort das Zeichen t_i angehängt werden können. Dies hat zur Folge, dass der betreffende Knoten eine t_i Kante besitzen muss.

Somit haben wir das folgende für die Konstruktion von Suffix-Bäumen wichtige Resultat bewiesen:

Lemma 2.22 *Ist (s,(k,i-1)) der Endknoten von $\hat{T}^{i-1}$ (Suffix-Baum $t_1 \cdots t_{i-1}$), dann ist (s,(k,i)) der aktive Knoten von $\hat{T}^i$.*

Die Beispiele zu den folgenden Prozeduren beziehen sich alle auf den unten abgebildeten Suffix-Baum für das Wort $t = ababbaa$.

Ukkonens Algorithmus für Suffix-Bäume ist in der folgenden Abbildung angegeben. In der for-Schleife des Algorithmus wird jeweils $\hat{T}^i$ aus $\hat{T}^{i-1}$ mit Hilfe der Prozedur

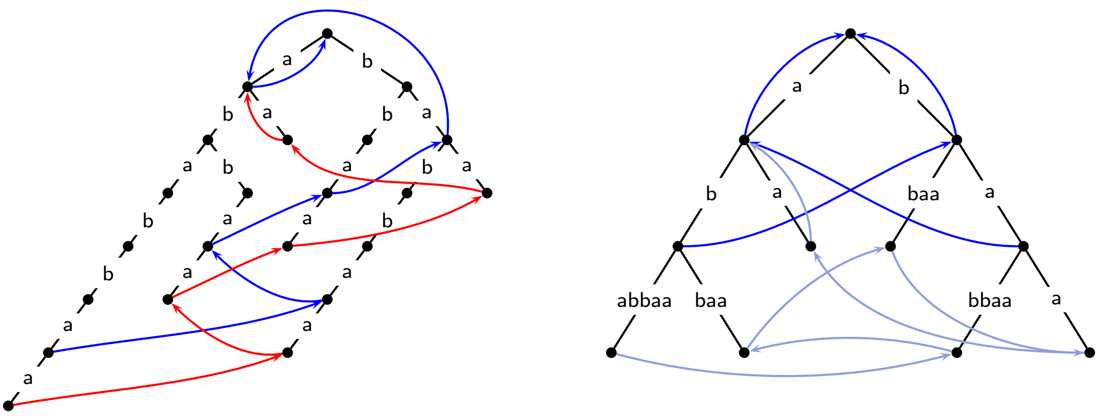


Abbildung 2.48: Beispiel: Suffix-Trie und Suffix-Baum für $t = ababbaa$

Update konstruiert. Dabei ist jeweils $(s, (k, i-1))$ der aktive Knoten. Zu Beginn für $i = 1$ ist dies für $(\overline{\varepsilon}, (1, 0))$ klar, da dies der einzige Knoten ist (außer dem Virtuellen). Wir bemerken hier, dass die Wurzel eines Baumes für uns per Definition kein Blatt ist, obwohl sie hier keine Kinder besitzt. Zum Ende liefert die Prozedur Update, die

BuildSuffixTree (char $t[]$, int n)

{

  $T = (\{root, \perp\}, \{\perp \xrightarrow{x} root : x \in \Sigma\});$

  $suffix\_link(root) = \perp;$

  $s = root;$

  $k = 1;$

  for $(i = 1;\ i \leq n;\ i{+}{+})$

    // Constructing T^i from T^{i-1}

    // $(s, (k, i-1))$ is active point in T^{i-1}

    $(s, k) = \text{Update}(s, (k, i-1), i);$

    // Now $(s, (k, i-1))$ is endpoint of T^{i-1}

}

Abbildung 2.49: Algorithmus: Ukkonens Online-Algorithmus

aus T^{i-1} den Suffix-Baum T^i konstruiert, den Endknoten von T^{i-1} zurück, nämlich $(s, (k, i-1))$. Nach Lemma 2.22 ist dann der aktive Knoten von T^i gerade $(s, (k, i))$. Da in der for-Schleife i um eins erhöht wird, erfolgt der nächste Aufruf von Update wieder korrekt mit dem aktiven Knoten von T^i, der jetzt ja T^{i-1} ist, wieder mit der korrekten Referenz $(s, (k, i-1))$.

Bevor wir die Prozedur Update erläutern, geben wir noch die Prozedur Canonize an, die aus einer übergegebenen Referenz eine kanonische Referenz konstruiert. Wir wollen ja eigentlich immer nur mit kanonischen Referenzen arbeiten.

Canonize (node s, ref (k, p))

$\{$

while $(|t_k \cdots t_p| > 0)$

$\{$

let $e = s \xrightarrow{w} s'$ s.t. $w_1 = t_k$;

if $(|w| > |t_k \cdots t_p|)$ **break**;

$k = k + |w|$;

$s = s'$

$\}$

return (s, k);

$\}$

Abbildung 2.50: Algorithmus: Die Prozedur Canonize

Der Prozedur Canonize wird eine gegebene Referenz $(\overline{s}, (k, p))$ übergeben. Die Prozedur durchläuft dann den Suffix-Baum ab dem Knoten $\overline{s}$ entlang der Zeichenreihe $t_k \cdots t_p$. Kommt man nun mitten auf einer Kante zum Stehen, entspricht der zuletzt besuchte Knoten dem Knoten, der für die kanonische Referenz verwendet wird. Dies ist in der folgenden Abbildung noch einmal am Beispiel für die Referenz $(\overline{ab}, (6, 7))$ illustriert.

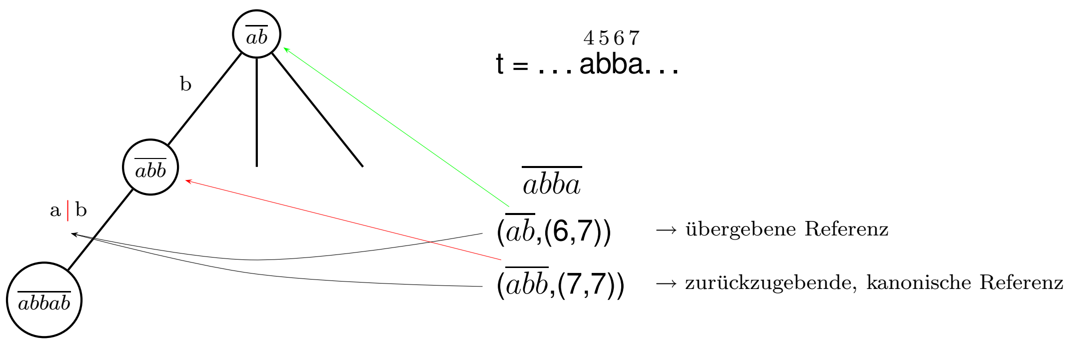


Abbildung 2.51: Beispiel: Erstellung kanonischer Referenzen mittels Canonize

In der Prozedur Update wird nun der Suffix-Baum $\hat{T}^i$ aus dem $\hat{T}^{i-1}$ aufgebaut. Die Prozedur Update bekommt eine nicht zwingenderweise kanonische Referenz des Knotens übergeben, an welchen das neue Zeichen t_i, dessen Index i ebenfalls übergeben wird, angehängt werden muss. Dabei hilft die Prozedur Canonize, welche die übergebene Referenz kanonisch macht, und der Prozedur TestAndSplit, die den tatsächlichen Knoten zurückgibt, an welchen die neue t_i Kante angehängt werden muss, falls sie noch nicht vorhanden ist. Bevor wir jetzt die Prozedur Update weiter erläutern, gehen wir zunächst auf die Hilfsprozedur TestAndSplit ein. Die Prozedur TestAndSplit hat, wie oben bereits angesprochen, die Aufgabe, den Knoten auszugeben, an welchen die neue t_i Kante, falls noch nicht vorhanden, anzuhängen ist.

```
UPDATE (node s, ref (k,p), int i)
{
    // (s,(k,p)) is active point
    old_r = root;
    (s,k) = Canonize(s,(k,p));
    (done,r) = TestAndSplit(s,(k,p),t_i);
    while ( ! done )
    {
        let m be a new node and add r --(i,∞)--> m;
        if (old_r ≠ root)
            suffix_link(old_r) = r;
        old_r = r;
        (s,k) = Canonize(suffix_link(s),(k,p));
        (done,r) = TestAndSplit(s,(k,p),t_i);
    }
    if (old_r ≠ root)
        suffix_link(old_r) = s;
    return (s, k);
}
```

Abbildung 2.52: Algorithmus: Die Prozedur Update

Die Prozedur TestAndSplit geht dabei so vor, dass zunächst überprüft wird, ob die betreffende Kante bereits vorhanden ist oder nicht. Falls dies der Fall ist, wird der Knoten, von welchem diese Kante ausgeht, ausgegeben. Ist die Kante jedoch noch nicht vorhanden, aber der Knoten, von welchem sie ausgehen sollte, existiert bereits im Suffix-Baum, wird einfach dieser explizite Knoten zurückgegeben. Nun kann noch der Fall auftreten, dass der Knoten, an welchen die neue Kante angehängt werden muss, nur implizit im Suffix-Baum vorhanden ist. In diesem Fall muss die betreffende Kante aufgebrochen werden, und der benötigte Knoten als expliziter Knoten eingefügt werden. Anschließend wird dieser dann ausgegeben, und in der Prozedur Update wird die fehlende Kante eingefügt.

In der Prozedur TestAndSplit ist die erste Fallunterscheidung, ob die Referenz auf einen expliziten oder impliziten Knoten zeigt ($|t_k \cdots t_p| == 0$). Ist der Knoten explizit, muss in jedem Falle kein neuer Knoten generiert werden und die Kante mit Label t_i kann an den expliziten Knoten s im Suffix-Baum angehängt werden.

Ist andernfalls der referenzierte Knoten implizit, so wird zuerst getestet, ob an diesem eine Kante mit Label t_i hängt ($t_i == w_{|t_k \cdots t_p|+1}$). Falls ja, ist ja nichts zu tun. Ansonsten muss diese Kante mit dem langen Label aufgebrochen werden, wie dies im folgenden Beispiel illustriert ist. Dann wird der neue Knoten r, der in die lan-

TestAndSplit (node s, ref (k,p), char x)

{
    if $(|t_k \cdots t_p| == 0)$
        if $(\exists\ s \xrightarrow{x\cdots})$ **return** (TRUE, s);
        else return (FALSE, s);
    else
    {
        let $e = s \xrightarrow{w} s'$ s.t. $w_1 = t_k$;
        if $(x == w_{|t_k \cdots t_p|+1})$ **return** (TRUE, s);
        else
        {
            split $e = s \xrightarrow{w} s'$ s.t.:
            $s \xrightarrow{w_1 \cdots w_{|t_k \cdots t_p|}} m \xrightarrow{w_{|t_k \cdots t_p|+1} \cdots w_{|w|}} s'$
            return (FALSE, m)
        }
    }
}

Abbildung 2.53: Algorithmus: Die Prozedur TestAndSplit

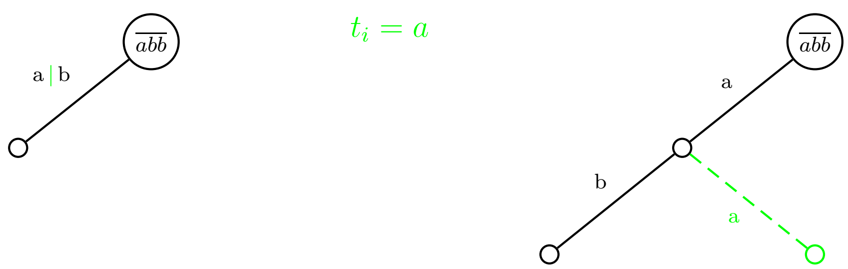


Abbildung 2.54: Beispiel: Vorgehensweise von TestAndSplit

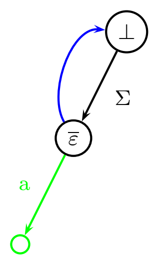
gen Kante eingefügt wurde, von TestAndSplit zurückgegeben, damit die Prozedur Update daran die neue Kante mit Label t_i anhängen kann.

Zurück zur Beschreibung der Prozedur Update. Auch hier folgen wir wieder vom aktiven Knoten aus den Suffix-Links und hängen eine neue Kante mit Label t_i ein. Wir bemerken hier, dass wir im Suffix-Baum nur für interne Knoten die Suffix-Links berechnen und nicht für die Blätter, da wir vom aktiven Knoten starten, der ja der erste interne Knoten ist. Mit *old_r* merken wir uns immer den zuletzt neu eingefügten internen Knoten, für den der Suffix-Link noch zu konstruieren ist. Zu Beginn setzen wir *old_r* auf *root* um damit anzuzeigen, dass wir noch keinen neuen Knoten eingefügt haben. Wann immer wir dem Suffix-Link gefolgt sind und im Schritt vorher einen neuen Knoten eingefügt haben, setzen wir für diesen zuvor eingefügten Knoten den Suffix-Link jetzt mittels Suffix-Link(old_r) $= r$. Der Knoten r ist gerade der Knoten, an den wir jetzt aktuell die Kante mit Label t_i anhängen wollen und somit der Elter des neuen Blattes. Somit wird der Suffix-Link von den korrespondierenden Eltern korrekt gesetzt.

Folgendes längeres *Beispiel* zeigt, wie nach und nach der Suffix-Baum für das bereits öfter verwendete Wort $t = ababbaa$ aufgebaut wird:

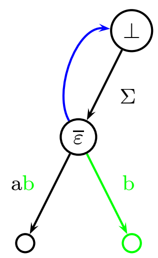
	1	2	3	4	5	6	7
t=	a	b	a	b	b	a	a

Buchstabe a wird in den noch leeren Tree eingefügt:



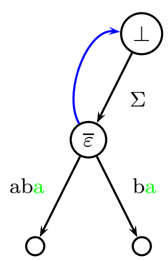
($\overline{\varepsilon}$,(1,0)) $i = 1; t_i = a$
$\leadsto$ TestAndSplit = (FALSE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(1,0))
$\downarrow$ Suffix-Link
($\perp$,(1,0))
$\downarrow$ Canonize
($\perp$,(1,0))

Buchstabe b wird in den $\hat{T}^1$ eingefügt:



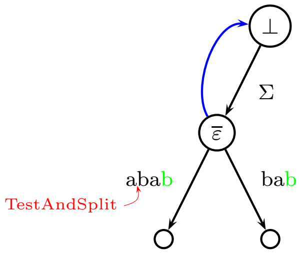
($\perp$,(1,1)) $i = 2; t_i = b$
$\downarrow$ Canonize
($\overline{\varepsilon}$,(2,1))
$\leadsto$ TestAndSplit = (FALSE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(2,1))
$\downarrow$ Suffix-Link
($\perp$,(2,1))
$\downarrow$ Canonize
($\perp$,(2,1))

Buchstabe a wird in den $\hat{T}^2$ eingefügt:



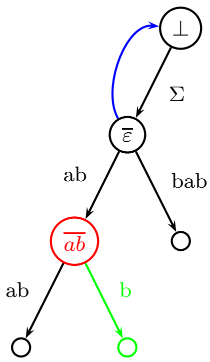
($\perp$,(2,2)) $i = 3; t_i = a$
$\downarrow$ Canonize
($\overline{\varepsilon}$,(3,2))
$\leadsto$ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(3,2))

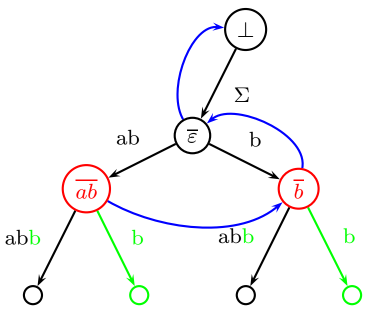
Buchstabe b wird in den $\hat{T}^3$ eingefügt:



$(\overline{\varepsilon}$,(3,3)) $i = 4$; $t_i = b$
$\downarrow$ Canonize
$(\overline{\varepsilon}$,(3,3))
$\rightsquigarrow$ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
$(\overline{\varepsilon}$,(3,3))

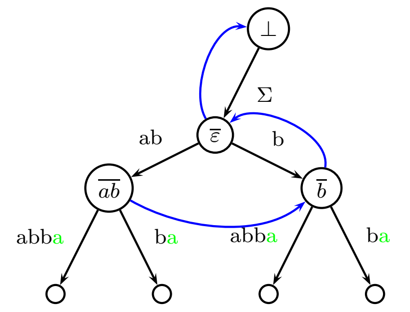
Buchstabe b wird in den $\hat{T}^4$ eingefügt:





$(\overline{\varepsilon}$,(3,4)) $i = 5$; $t_i = b$
$\downarrow$ Canonize
$(\overline{\varepsilon}$,(3,4))
$\rightsquigarrow$ TestAndSplit = (FALSE,$\overline{ab}$)
$(\overline{\varepsilon}$,(3,4))
$\downarrow$ Suffix-Link
$(\perp$,(3,4))
$\downarrow$ Canonize
$(\overline{\varepsilon}$,(4,4))
$\rightsquigarrow$ TestAndSplit = (FALSE,$\overline{b}$)
$(\overline{\varepsilon}$,(4,4))
$\downarrow$ Suffix-Link
$(\perp$,(4,4))
$\downarrow$ Canonize
$(\overline{\varepsilon}$,(5,4))
$\rightsquigarrow$ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
$(\overline{\varepsilon}$,(5,4))

Buchstabe a wird in den $\hat{T}^5$ eingefügt:



($\bar{\varepsilon}$,(5,5)) $i = 6; t_i = a$

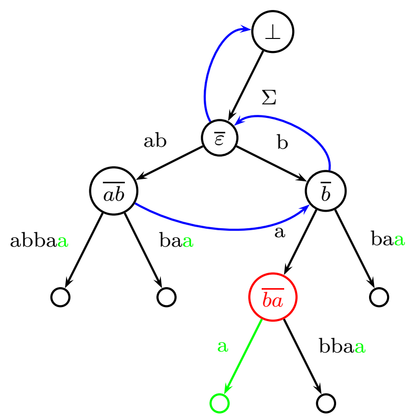
↓ Canonize

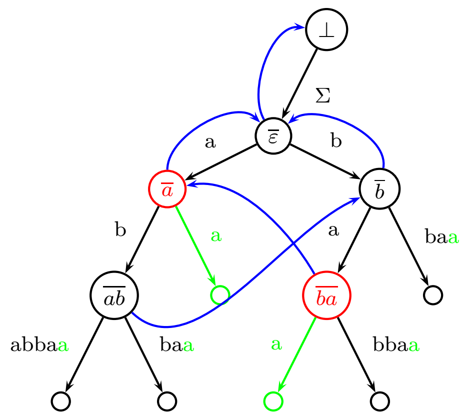
($\bar{b}$,(6,5))

⦚ TestAndSplit = (TRUE,$\bar{b}$)

($\bar{b}$,(6,5))

Buchstabe a wird in den $\hat{T}^6$ eingefügt:





($\bar{b}$,(6,6)) $i = 7; t_i = a$

↓ Canonize

($\bar{b}$,(6,6))

⦚ TestAndSplit = (FALSE,$\overline{ba}$)

($\bar{b}$,(6,6))

↓ Suffix-Link

($\bar{\varepsilon}$,(6,6))

↓ Canonize

($\bar{\varepsilon}$,(6,6))

⦚ TestAndSplit = (FALSE,$\bar{a}$)

($\bar{\varepsilon}$,(6,6))

↓ Suffix-Link

($\perp$,(6,6))

↓ Canonize

($\bar{\varepsilon}$,(7,6))

⦚ TestAndSplit = (TRUE,$\bar{\varepsilon}$)

($\bar{\varepsilon}$,(7,6))

Jetzt ist der Suffix-Baum für $t = ababbaa$ vollständig aufgebaut.

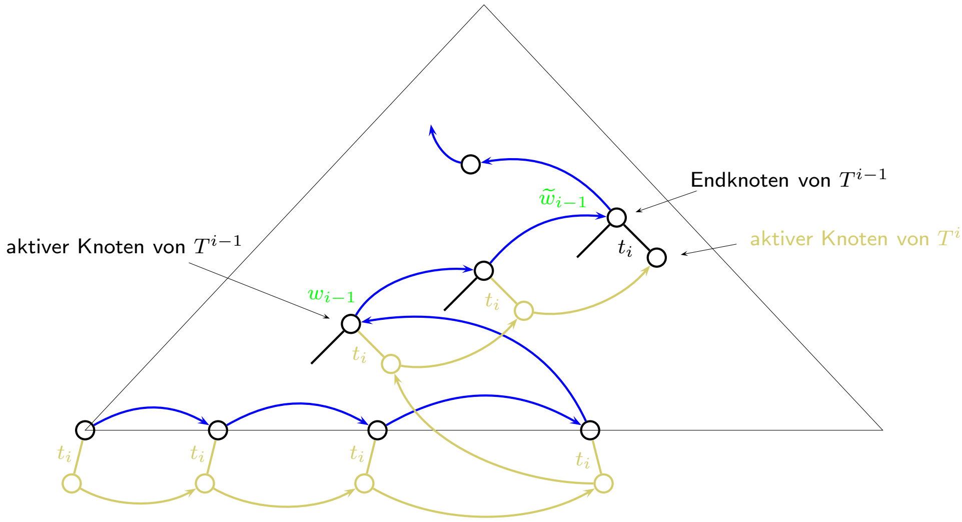


Abbildung 2.55: Skizze: Erweiterung von T^{i-1} auf T^i

2.6.7 Laufzeitanalyse

Zum Schluss kommen wir noch zur Laufzeitanalyse des Online-Algorithmus von Ukkonen für Suffix-Bäume. Wir teilen die Zeitanalyse in zwei Teile auf. Zunächst analysieren wir den Zeitbedarf für alle Ausführungen der while-Schleife in der Prozedur Canonize. Ein Durchlauf der while-Schleife kann offensichtlich in Zeit $O(1)$ ausgeführt werden. Wir stellen fest, dass nur hier der Wert von k verändert wird, und zwar wird er immer nur vergrößert. Da k zu Beginn 1 ist und nie größer als n werden kann, wird die while-Schleife der Prozedur Canonize maximal n-mal durchlaufen. Somit ist auch der Zeitbedarf höchstens $O(n)$.

Nun betrachten wir alle übrigen Kosten. Die verbleibenden Kosten für einen Aufruf der Prozedur Canonize sind jetzt noch $O(1)$. Ebenso benötigt jeder Aufruf der Prozedur TestAndSplit Zeit $O(1)$. Damit sind die verbleibenden Kosten durch die Anzahl der betrachteten Knoten des Suffix-Baumes beschränkt.

Sei w_{i-1} die Zeichenreihe, um den aktiven Knoten von T^{i-1} zu erreichen und $\widetilde{w}_{i-1}$ die Zeichenreihe, um den Endknoten von T^{i-1} zu erreichen. Dann ist $\overline{w_i}$ der aktive Knoten von T^i und $\overline{\widetilde{w}_i}$ der Endknoten von T^i. Betrachte hierzu auch die folgende Skizze. Wie viele Knoten werden nun bei der Konstruktion von $\hat{T}^i$ aus $\hat{T}^{i-1}$ besucht? Ist $\overline{w_i}$ der aktive Knoten von T^i, dann war der Endknoten von T^{i-1} gerade $\overline{\widetilde{w}_i}$, wobei $w_i = \tilde{w}_{i-1} \cdot t_i$. Die Anzahl der Durchläufe einer *while-Schleife* entspricht $(|w_{i-1}| - |\widetilde{w}_{i-1}| + 1)$. Weiterhin ist

$$|w_i| - |\widetilde{w}_{i-1}| + 1 = (|w_{i-1}| - (|w_i| - 1) + 1) = |w_{i-1}| - |w_i| + 2.$$