INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
LEHRSTUHL FÜR EFFIZIENTE ALGORITHMEN

Skriptum

zur Vorlesung

Algorithmische Bioinformatik I/II

gehalten im Wintersemester 2001/2002

und im Sommersemester 2002 von

Volker Heun

Erstellt unter Mithilfe von:

Peter Lücke – Hamed Behrouzi – Michael Engelhardt

Sabine Spreer – Hanjo Täubig

Jens Ernst – Moritz Maaß

14. Mai 2003

Version 0.96

Vorwort

Dieses Skript entstand parallel zu den Vorlesungen *Algorithmische Bioinformatik I* und *Algorithmische Bioinformatik II*, die im Wintersemester 2001/2002 sowie im Sommersemester 2002 für Studenten der Bioinformatik und Informatik sowie anderer Fachrichtungen an der Technischen Universität München im Rahmen des von der Ludwig-Maximilians-Universität und der Technischen Universität gemeinsam veranstalteten Studiengangs Bioinformatik gehalten wurde. Einige Teile des Skripts basieren auf der bereits im Sommersemester 2000 an der Technischen Universität München gehaltenen Vorlesung *Algorithmen der Bioinformatik* für Studierende der Informatik.

Das Skript selbst umfasst im Wesentlichen die grundlegenden Themen, die man im Bereich Algorithmische Bioinformatik einmal gehört haben sollte. Die vorliegende Version bedarf allerdings noch einer Ergänzung weiterer wichtiger Themen, die leider nicht in den Vorlesungen behandelt werden konnten.

An dieser Stelle möchte ich insbesondere Hamed Behrouzi, Michael Engelhardt und Peter Lücke danken, die an der Erstellung des ersten Teils dieses Skriptes (Kapitel 2 mit 5) maßgeblich beteiligt waren. Bei Sabine Spreer möchte ich mich für die Unterstützung bei Teilen des siebten Kapitels bedanken. Bei meinen Übungsleitern Jens Ernst und Moritz Maaß für deren Unterstützung der Durchführung des Übungsbetriebs, aus der einige Lösungen von Übungsaufgaben in dieses Text eingeflossen sind. Bei Hanjo Täubig möchte ich mich für die Mithilfe zur Fehlerfindung bedanken, insbesondere bei den biologischen Grundlagen.

Falls sich dennoch weitere (Tipp)Fehler unserer Aufmerksamkeit entzogen haben sollten, so bin ich für jeden Hinweis darauf (an heun@in.tum.de) dankbar.

München, im September 2002 — Volker Heun

Inhaltsverzeichnis

Suchen in Texten

2.1 Grundlagen

- Ein *Alphabet* ist eine endliche Menge von Symbolen.
  Bsp.: $\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{A, C, G, T\}$.
- *Wörter* über Σ sind endliche Folgen von Symbolen aus Σ. Wörter werden manchmal 0 und manchmal von 1 an indiziert, d.h. $w = w_0 \cdots w_{n-1}$ bzw. $w = w_1 \cdots w_n$, je nachdem, was im Kontext praktischer ist.
  Bsp.: $\Sigma = \{a, b\}$, dann ist $w = abba$ ein Wort über Σ.
- Die *Länge* eines Wortes w wird mit $|w|$ bezeichnet und entspricht der Anzahl der Symbole in w.
- Das Wort der Länge 0 heißt leeres Wort und wird mit ε bezeichnet.
- Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Die Menge aller Wörter der Länge größer gleich 1 über Σ wird mit $\Sigma^+ := \Sigma^* \backslash \{\varepsilon\}$ bezeichnet. Die Menge aller Wörter über Σ der Länge k wird mit $\Sigma^k \subseteq \Sigma^*$ bezeichnet.
- Sei $w = w_1 \cdots w_n$ ein Wort der Länge n ($w_i \in \Sigma$). Im Folgenden bezeichne $[a : b] = \{n \in \mathbb{Z} \mid a \leq n \wedge n \leq b\}$ für $a, b \in \mathbb{Z}$.
  - Ist $w' = w_1 \cdots w_l$ mit $l \in [0 : n]$, dann heißt w' *Präfix* von w.
    (für $l = 0 \Rightarrow w' = \varepsilon$)
  - Ist $w' = w_l \cdots w_n$ mit $l \in [1 : n + 1]$, dann heißt w' *Suffix* von w.
    (für $l = n + 1 \Rightarrow w' = \varepsilon$)
  - Ist $w' = w_i \cdots w_j$ mit $i, j \in [1 : n]$, dann heißt w' *Teilwort* von w.
    (wobei $w_i \cdots w_j = \varepsilon$ für $i > j$)

  Das leere Wort ist also Präfix, Suffix und Teilwort eines jeden Wortes über Σ.

2.2 Der Algorithmus von Knuth, Morris und Pratt

Dieser Abschnitt ist dem Suchen in Texten gewidmet, d.h. es soll festgestellt werden, ob ein gegebenes Suchwort s in einem gegebenen Text t enthalten ist oder nicht.

Problem:
Geg.: $s \in \Sigma^*$; $|s| = m$; $t \in \Sigma^*$; $|t| = n \geq m$
Ges.: $\exists\, i \in [0 : n - m]$ mit $t_i \cdots t_{i+m-1} = s$

2.2.1 Ein naiver Ansatz

Das Suchwort s wird Buchstabe für Buchstabe mit dem Text t verglichen. Stimmen zwei Buchstaben nicht überein ($\rightarrow$ Mismatch), so wird s um eine Position „nach rechts“ verschoben und der Vergleich von s mit t beginnt von neuem. Dieser Vorgang wird solange wiederholt, bis s in t gefunden wird oder bis klar ist, dass s in t nicht enthalten ist.

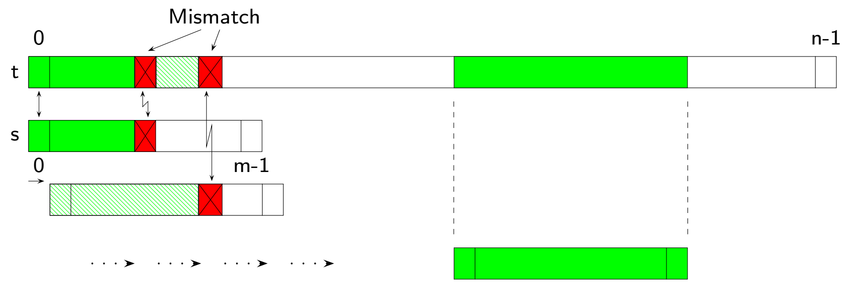


Abbildung 2.1: Skizze: Suchen mit der naiven Methode

Definition 2.1 *Stimmen beim Vergleich zweier Zeichen diese nicht überein, so nennt man dies einen* Mismatch.

In der folgenden Abbildung ist der naive Algorithmus in einem C-ähnlichen Pseudocode angegeben.

```
BOOL NAIV (char t[], int n, char s[], int m)
{
    int i = 0, j = 0;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            j++;
            if (j == m) return TRUE;
        }
        i++;
        j = 0;
    }
    return FALSE;
}
```

Abbildung 2.2: Algorithmus: Die naive Methode

2.2.2 Laufzeitanalyse des naiven Algorithmus:

Um die Laufzeit abzuschätzen, zählen wir die Vergleiche von Symbolen aus Σ:

- Äußere Schleife wird $(n - m + 1)$-mal durchlaufen.
- Innere Schleife wird maximal m-mal durchlaufen.

$\Rightarrow$ Test wird $((n - m + 1) * m)$-mal durchlaufen $= O(n * m) \longleftarrow$ Das ist zu viel!

Beispiel:

1.) $t = a\ a\ a\ a\ a\ a\ a\ a\ a\ a$

√ √ √ ×
$a\ a\ a\ b$
  √ √ √ ×
  $a\ a\ a\ b$
    √ √ √ ×
    $a\ a\ a\ b$

2.) $t = a\ a\ b\ a\ a\ b\ a\ a\ b\ a\ a\ b\ a\ a\ b$

√ √ √ √ √ ×
$a\ a\ b\ a\ a\ a$
  √ ×
  $a\ a\ b\ a\ a\ a$
    ×
    $a\ a\ b\ a\ a\ a$
      √ √ √ √ √ ×
      $a\ a\ b\ a\ a\ a$

Abbildung 2.3: Beispiel: Suchen mit der naiven Methode

2.2.3 Eine bessere Idee

Ein Verbesserung ließe sich vermutlich dadurch erzielen, dass man die früheren erfolgreichen Vergleiche von zwei Zeichen ausnützt. Daraus resultiert die Idee, das Suchwort so weit nach rechts zu verschieben, dass in dem Bereich von t, in dem bereits beim vorherigen Versuch erfolgreiche Zeichenvergleiche durchgeführt wurden, nun nach dem Verschieben auch wieder die Zeichen in diesem Bereich übereinstimmen (siehe auch die folgende Skizze).

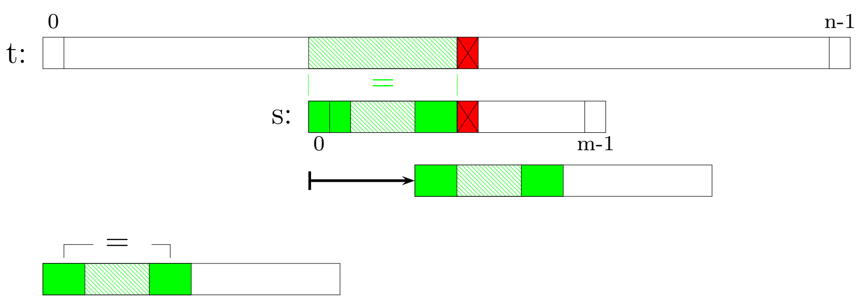


Abbildung 2.4: Skizze: Eine Idee für größere Shifts

Um diese Idee genauer formalisieren zu können, benötigen wir noch einige grundlegende Definitionen.

Definition 2.2 *Ein Wort r heißt* Rand *eines Wortes w, wenn r sowohl Präfix als auch Suffix von w ist.*

Ein Rand r eines Wortes w heißt eigentlicher Rand, *wenn $r \neq w$ und wenn es außer w selbst keinen längeren Rand gibt.*

Bemerkung: ε und w sind immer Ränder von w.

Beispiel: $a\ a\ b\ a\ a\ b\ a\ a$ *Beachte:* Ränder können sich überlappen!

Der eigentliche Rand von *aabaabaa* ist also *aabaa*. *aa* ist ein Rand, aber nicht der eigentliche Rand.

Definition 2.3 *Ein Verschiebung der Anfangsposition i des zu suchenden Wortes (d.h. eine Erhöhung des Index $i \to i'$) heißt* Shift.

Ein Shift von $i \to i'$ heißt sicher, *wenn s nicht als Teilwort von t an der Position $k \in [i+1 : i'-1]$ vorkommt, d.h. $s \neq t_k \cdots t_{k+m-1}$ für alle $k \in [i+1 : i'-1]$.*

Wir definieren:

$$border[j] = \begin{cases} -1 & \textit{für } j = 0 \\ |\partial(s_0 \cdots s_{j-1})| & \textit{für } j \geq 1 \end{cases}$$

wobei $\partial(s)$ den eigentlichen Rand von s bezeichnet.

Lemma 2.4 *Gilt $s_k = t_{i+k}$ für alle $k \in [0 : j-1]$ und $s_j \neq t_{i+j}$, dann ist der Shift $i \to i + j - border[j]$ sicher.*

Beweis: Das Teilwort von s stimmt ab der Position i mit dem zugehörigen Teilwort von t von t_i bzw. s_0 mit t_{i+j-1} bzw. s_{j-1} überein, d.h. $t_{i+j} \neq s_j$ (siehe auch die folgende Skizze in Abbildung 2.5). Der zum Teilwort $s_0 \cdots s_{j-1}$ gehörende eigentliche Rand hat laut Definition die Länge border[j]. Verschiebt man s um $j-$border[j] nach rechts, so kommt der rechte Rand des Teilwortes $s_0 \cdots s_{j-1}$ von s auf dem linken Rand zu liegen, d.h. man schiebt „Gleiches“ auf „Gleiches“. Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt, der ungleich $s_0 \cdots s_{j-1}$ ist, ist dieser Shift sicher. ■

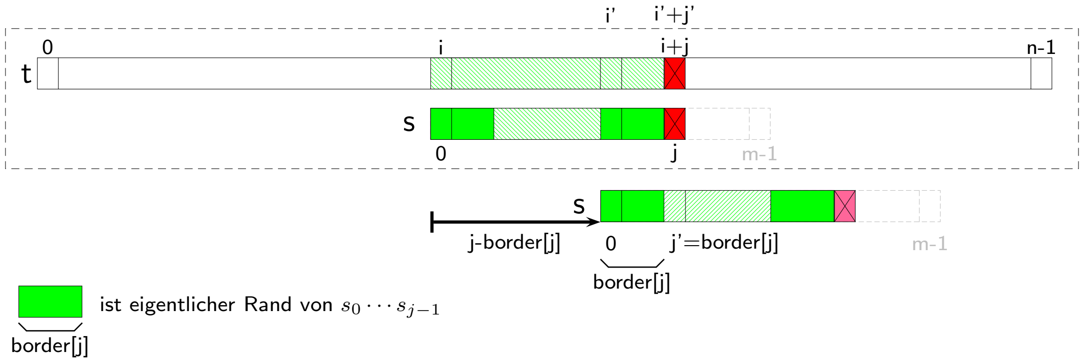


Abbildung 2.5: Skizze: Der Shift um $j - border[j]$

2.2.4 Der Knuth-Morris-Pratt-Algorithmus

Wenn wir die Überlegungen aus dem vorigen Abschnitt in einen Algorithmus übersetzen, erhalten wir den sogenannten KMP-Algorithmus, benannt nach D.E. Knuth, J. Morris und V. Pratt.

BOOL KNUTH-MORRIS-PRATT (char $t[]$, int n, char $s[]$, int m)

{
    int $border[m+1]$;
    compute_borders(int $border[]$, int m, char $s[]$);
    int $i = 0$, $j = 0$;
    while $(i \leq n - m)$
    {
        while $(t[i+j] == s[j])$
        {
            j++;
            if $(j == m)$ **return** TRUE;
        }
        $i = i + \underbrace{j - border[j]}_{>0}$;
        $j = \max\{0, border[j]\}$;
    }
    return FALSE;
}

Abbildung 2.6: Algorithmus: Die Methode von Knuth, Morris und Pratt

2.2.5 Laufzeitanalyse des KMP-Algorithmus:

Strategie: Zähle Anzahl der Vergleiche getrennt nach erfolglosen und erfolgreichen Vergleichen. Ein Vergleich von zwei Zeichen heißt erfolgreich, wenn die beiden Zeichen gleich sind, und *erfolglos* sonst.

erfolglose Vergleiche:

Es werden maximal $n - m + 1$ erfolglose Vergleiche ausgeführt, da nach jedem erfolgten Vergleich $i \in [0 : n - m]$ erhöht und nie erniedrigt wird.

erfolgreiche Vergleiche:

Wir werden zunächst zeigen, dass nach einem erfolglosen Vergleich der Wert von $i + j$ nie erniedrigt wird. Seien dazu i, j die Werte von i, j vor einem erfolglosen Vergleich und i', j' die Werte nach einem erfolglosen Vergleich.

Wert von $i + j$ vorher: $i + j$
Wert von $i' + j'$ nachher: $\underbrace{(i + j - border[j])}_{i'} + \underbrace{(max\{0, border[j]\})}_{j'}$

1.Fall: $border[j] \geq 0 \quad \Rightarrow \quad i' + j' = i + j$
2.Fall: $border[j] = -1$ ($\Leftrightarrow j = 0$) $\Rightarrow i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1$

$\Rightarrow$ Nach einem erfolglosen Vergleich wird $i + j$ nicht kleiner!
Nach einem erfolgreichen Vergleich wird $i + j$ um 1 erhöht!

$\Rightarrow$ Die maximale Anzahl erfolgreicher Vergleiche ist durch n beschränkt, da $i + j \in [0 : n - 1]$.

$\Rightarrow$ Somit werden insgesamt maximal $2n - m + 1$ Vergleiche ausgeführt.

2.2.6 Berechnung der Border-Tabelle

In der border[]-Tabelle wird für jedes Präfix $s_0 \cdots s_{j-1}$ der Länge $j \in [0 : m]$ des Suchstrings s der Länge m gespeichert, wie groß dessen eigentlicher Rand ist.

Initialisierung: $border[0] = -1$
$border[1] = 0$

Annahme: $border[0] \cdots border[j - 1]$ seien bereits berechnet.

Ziel: Berechnung von $border[j]$ = Länge des eigentlichen Randes eines Suffixes der Länge j.

Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j - 1] + 1$. Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist. Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt

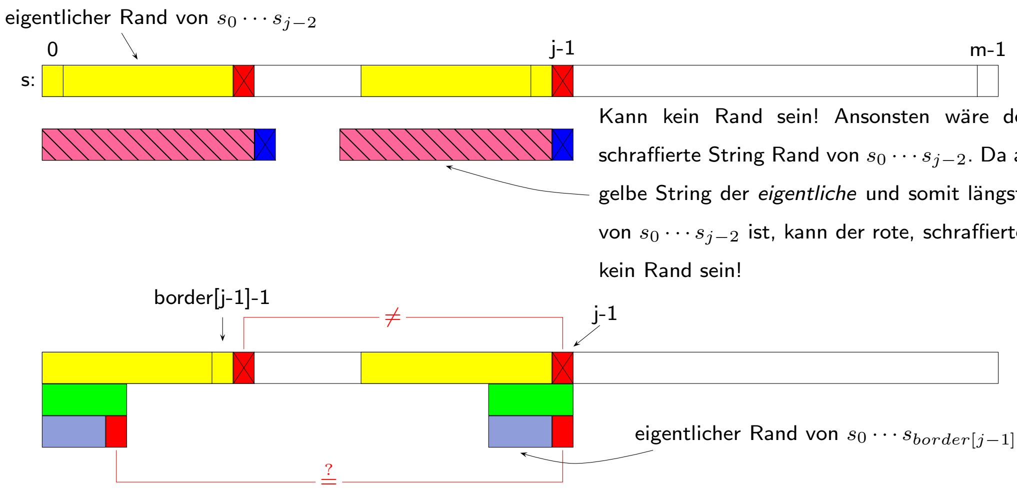


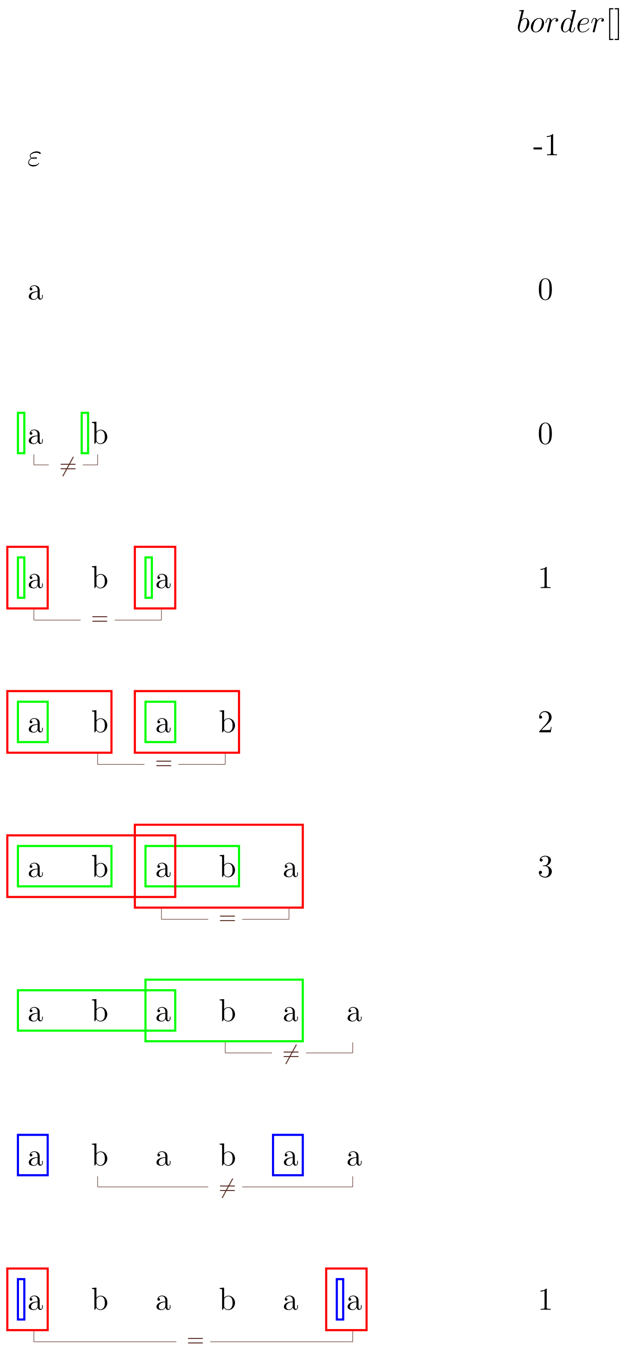
Abbildung 2.7: Skizze: Berechnung von $border[j]$

betrachteten Randes dieses Wortes. Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$. Nun testen wir, ob sich dieser Rand von $s_0 \cdots s_{j-2}$ zu einem eigentlichen Rand von $s_0 \cdots s_{j-1}$ erweitern lässt. Dies wiederholen wir solange, bis wir einen Rand gefunden haben, der sich zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt. Falls sich kein Rand von $s_0 \cdots s_{j-2}$ zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt, so ist der eigentliche Rand von $s_0 \cdots s_{j-1}$ das leere Wort und wir setzen $border[j] = 0$.

Damit erhalten wir den folgenden Algorithmus zur Berechnung der Tabelle *border*.

```
COMPUTE_BORDERS (int border[], int m, char s[])
{
    border[0] = −1;
    int i = border[1] = 0;
    for (int j = 2; j ≤ m; j++)
    { /* Beachte, dass hier gilt: i == border[j − 1] */
        while ((i ≥ 0) && (s[i] ≠ s[j − 1]))
            i = border[i];
        i++;
        border[j] = i;
    }
}
```

Abbildung 2.8: Algorithmus: Berechnung der Tabelle *border*



grün: der bekannte Rand
rot: der verlängerte (neu gefundene) Rand
blau: der "Rand des Randes"

Abbildung 2.9: Beispiel: Berechnung der Tabelle *border* für *ababaa*

2.2.7 Laufzeitanalyse:

Wieder zählen wir die Vergleiche getrennt nach erfolgreichen und erfolglosen Vergleichen.

Anzahl erfolgreicher Vergleiche:
Es kann maximal $m-1$ erfolgreiche Vergleiche geben, da jedes Mal $j \in [2:m]$ um 1 erhöht und nie erniedrigt wird.

Anzahl erfolgloser Vergleiche:
Betrachte i zu Beginn: $i = 0$
Nach jedem erfolgreichen Vergleich wird i inkrementiert
$\Rightarrow$ i wird $(m-1)$ Mal um 1 erhöht, da die for-Schleife $(m-1)$ Mal durchlaufen wird.
$\stackrel{i \geq -1}{\Rightarrow}$ i kann maximal $(m-1)+1 = m$ Mal erniedrigt werden, da $i \geq -1$
(Es kann nur das weggenommen werden, was schon einmal hinzugefügt wurde; das „plus eins“ kommt daher, dass zu Beginn $i = 0$ und ansonsten immer $i \geq -1$ gilt.).

$$\Rightarrow \text{Anzahl der Vergleiche} \leq 2m - 1$$

Theorem 2.5 *Der Algorithmus von Knuth, Morris und Pratt benötigt maximal $2n+m$ Vergleiche, um festzustellen, ob ein Muster s der Länge m in einem Text t der Länge n enthalten ist.*

Der Algorithmus lässt sich leicht derart modifizieren, dass er alle Positionen der Vorkommen von s in t ausgibt, ohne dabei die asymptotische Laufzeit zu erhöhen. Die Details seien dem Leser als Übungsaufgabe überlassen.

2.3 Der Algorithmus von Aho und Corasick

Wir wollen jetzt nach mehreren Suchwörtern gleichzeitig im Text t suchen.

Geg.: Ein Text t der Länge n und eine Menge $S = \{s^1, \ldots, s^l\}$ mit $\sum_{s \in S} |s| = m$.
Ges.: Taucht ein Suchwort $s \in S$ im Text t auf?

Wir nehmen hier zunächst an, dass in S kein Suchwort Teilwort eines anderen Suchwortes aus S ist.

2.3.1 Naiver Lösungsansatz

Wende den KMP-Algorithmus für jedes Suchwort $s \in S$ auf t an.

Kosten des Preprocessing (Erstellen der Border-Tabellen):

$$\sum_{i=1}^{l}(2|s^i| - 2) \leq \sum_{i=1}^{l} 2|s^i| = 2m.$$

Kosten des eigentlichen Suchvorgangs:

$$\sum_{i=1}^{l}(2n - |s^i| + 1) \leq 2l * n - m + l.$$

Somit sind die Gesamtkosten $O(l * n + m)$. Ziel ist die Elimination des Faktors l.

2.3.2 Der Algorithmus von Aho und Corasick

Zuerst werden die Suchwörter in einem so genannten Suchwort-Baum organisiert. In einem *Suchwort-Baum* gilt folgendes:

- Der Suchwort-Baum ist gerichteter Baum mit Wurzel r;
- Jeder Kante ist als Label ein Zeichen aus Σ zugeordnet;
- Die von einem Knoten ausgehenden Kanten besitzen verschiedene Labels;
- Jedes Suchwort $s \in S$ wird auf einen Knoten v abgebildet, so dass s entlang des Pfades von r nach v steht;
- Jedem Blatt ist ein Suchwort zugeordnet.

In der Abbildung auf der nächsten Seite ist ein solcher Suchwort-Baum für die Menge $\{aal, aas, aus, sau\}$ angegeben.

Wie können wir nun mit diesem Suchwort-Baum im Text t suchen? Wir werden die Buchstaben des Textes t im Suchwort-Baum-ablaufen. Sobald wir an einem Blatt gelandet sind, haben wir eines der gesuchten Wörter gefunden. Wir können jedoch auch in Sackgassen landen: Dies sind Knoten, von denen keine Kante mit einem gesuchten Kanten-Label ausgeht.

Damit es zu keinen Sackgassen kommt, werden in den Baum so genannten Failure-Links eingefügt.

Failure-Links: Verweis von einem Knoten v auf einen Knoten w im Baum, so dass die Kantenbezeichnungen von der Wurzel zu dem Knoten w den längsten Suffix des bereits erkannten Teilwortes bilden (= Wort zu Knoten v).

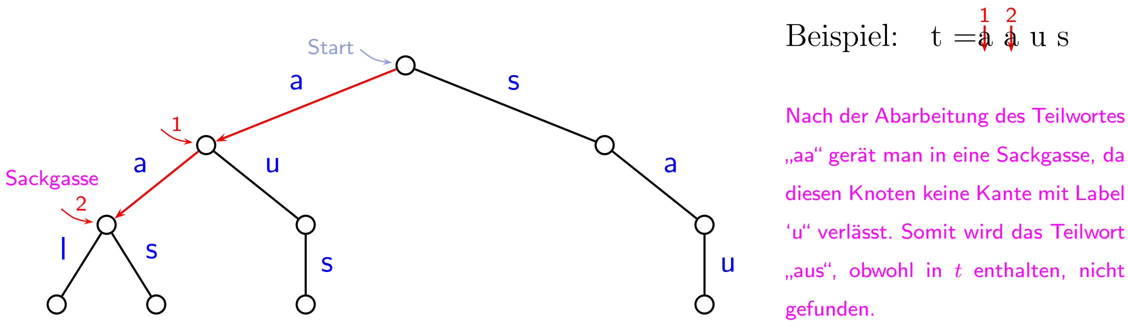


Abbildung 2.10: Beispiel: Aufbau des Suchwort-Baumes für $\{aal, aas, aus, sau\}$

Die Failure-Links der Kinder der Wurzel werden so initialisiert, dass sie direkt zur Wurzel zeigen. Die Failure-Links der restlichen Knoten werden nun Level für Level von oben nach unten berechnet.

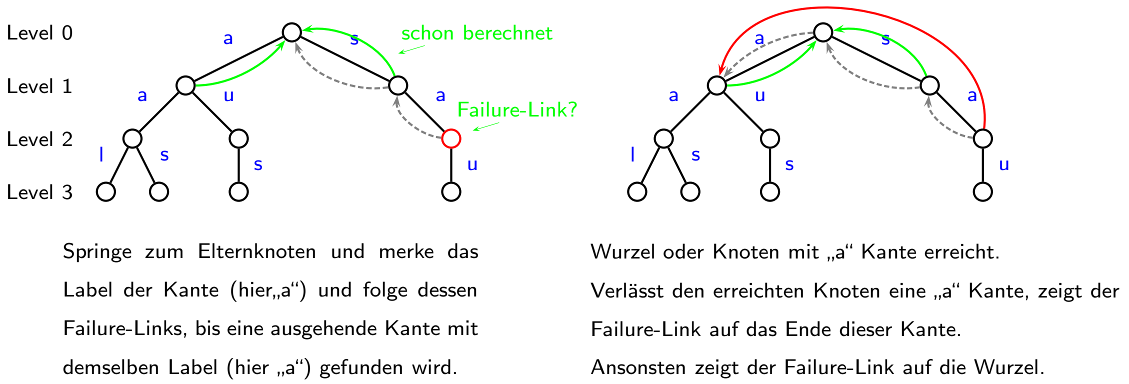


Abbildung 2.11: Beispiel: für die Berechnung eines Failure-Links

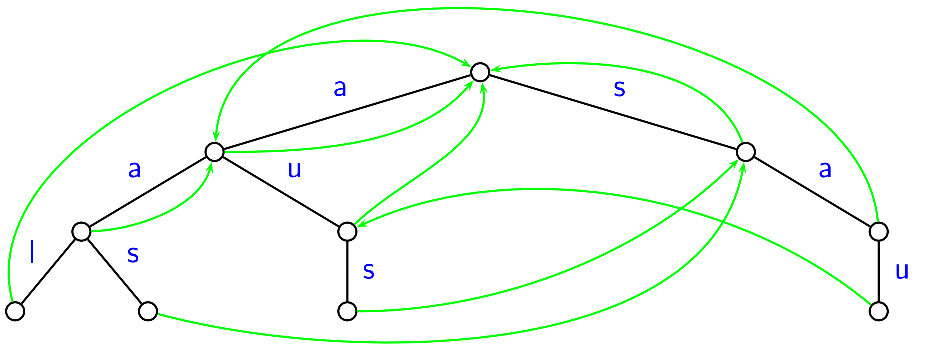


Abbildung 2.12: Beispiel: Der komplette Baum mit allen Failure-Links

Ein Suchwort s ist genau dann im Text t enthalten, wenn man beim Durchlaufen der Buchstaben von t im Suchwort-Baum in einem Blatt ankommt. Sind wir in einer Sackgasse gelandet, d.h. es gibt keine ausgehende Kante mit dem gewünschten

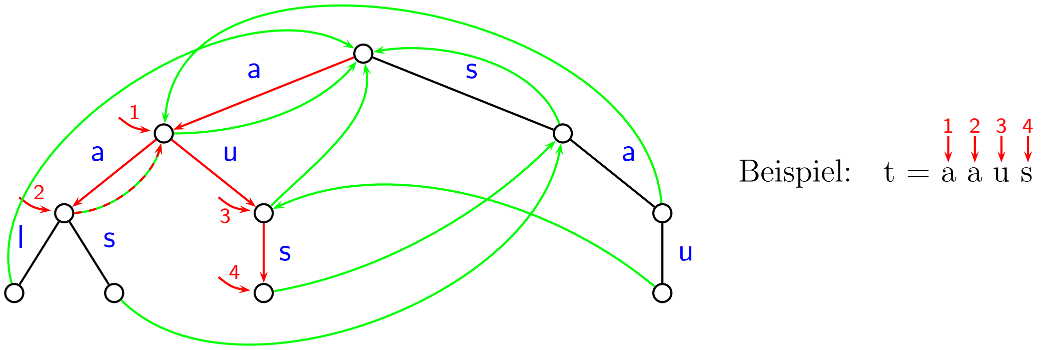
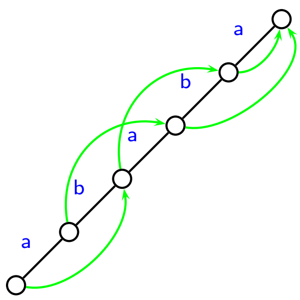


Abbildung 2.13: Beispiel: Suche in $t = aaus$, nur mit Failure-Links

Label, so folgen wir dem Failure-Link und suchen von dem so aufgefundenen Knoten aus weiter.

Ist die Menge S einelementig, so erhalten wir als Spezialfall die Tabelle *border* des KMP-Algorithmus. Im Suchwort-Baum wird dabei auf den entsprechenden längsten Präfix (der auch Suffix ist) verwiesen. In der Tabelle *border* ist hingegen nur die Länge dieses Präfixes gespeichert. Da S einelementig ist, liefern beide Methoden dieselben Informationen.



Level 0, $border[0] = -1$

Level 1, $border[1] = 0$

Level 2, $border[2] = 0$

Level 3, $border[3] = 1$

Level 4, $border[4] = 2$

Level 5, $border[5] = 3$

Abbildung 2.14: Beispiel: $S = \{ababa\}$

Die Laufzeit zur Berechnung der Failure-Links beträgt $O(m)$. Um dies zu zeigen, betrachten wir ein festes Suchwort $s \in S$. Wir zeigen zunächst nur, dass für die Berechnung der Failure-Links der Knoten auf dem Pfad von s im Suchwort-Baum $O(|s|)$ Vergleiche ausgeführt werden.

Wie bei der Analyse der Berechnung der Tabelle *border* des KMP-Algorithmus unterscheiden wir erfolgreiche und erfolglose Vergleiche. Zuerst halten wir fest, dass es maximal $O(|s|)$ erfolgreiche Vergleiche (d.h., es gibt eine Kante $w \xrightarrow{x}$) geben kann, da wir dann zum nächsttieferen Knoten auf dem Pfad von s wechseln. Für die erfolglosen Vergleiche beachten wir, dass Failure-Links immer nur zu Knoten auf

```
Berechnung der Failure-Links (tree T = (V, E))
{
    forall v ∈ V
    {
        Sei v' der Elternknoten von v mit v' --x--> v ∈ E
        w = Failure_Link(v')
        while (!(w --x-->) && (w ≠ root))
            w = Failure_Link(w)
        if (w --x--> w') Failure_Link(v) = w'
        else Failure_Link(v) = root
    }
}
```

Abbildung 2.15: Algorithmus: Berechnung der Failure-Links

einem niedrigeren Level verweisen. Bei jedem erfolglosen Vergleich springen wir also zu einem Knoten auf einem niedrigeren Level. Da wir nur bei einem erfolgreichen Vergleich zu einem höheren Level springen können, kann es nur so viele erfolglose wie erfolgreiche Vergleiche geben.

Somit ist die Anzahl Vergleiche für jedes Wort $s \in S$ durch $O(|s|)$ beschränkt. Damit ergibt sich insgesamt für die Anzahl der Vergleiche

$$\leq \sum_{s \in S} O(|s|) = O(m).$$

In der Abbildung auf der nächsten Seite ist der Algorithmus von A. Aho und M. Corasick im Pseudocode angegeben.

Auch hier verläuft die Laufzeitanalyse ähnlich wie beim KMP-Algorithmus. Da $level(v)$-$level(\text{Failure\_Link}(v)) > 0$ (analog zu $j - border[j] > 0$) ist, wird nach jedem erfolglosen Vergleich i um mindestens 1 erhöht. Also gibt es maximal $n-m+1$ erfolglose Vergleiche. Weiterhin kann man zeigen, dass sich nach einem erfolglosen Vergleich $i + level(v)$ nie erniedrigt und nach jedem erfolgreichen Vergleich um 1 erhöht (da sich $level(v)$ um 1 erhöht). Da $i + j \in [0 : n-1]$ ist, können maximal n erfolgreiche und somit maximal $2n - m + 1$ Vergleiche überhaupt ausgeführt worden sein.

2.3.3 Korrektheit von Aho-Corasick

Es bleibt nur noch, die Korrektheit des vorgestellten Algorithmus von Aho und Corasick nachzuweisen. Wenn kein Muster in t auftritt ist klar, dass der Algorithmus

```
BOOL AHO-CORASICK (char t[], int n, char S[], int m)
{
    int i = 0
    tree T(S) // Suchwort-Baum, der aus den Wörtern in s konstruiert wurde
    node v = root
    while (i < n)
    {
        while ((v --t_{i+level(v)}--> v') in T)
        {
            v = v'
            if (v' ist Blatt) return TRUE;
        }
        i = i + level(v) − level(Failure_Link(v))
        v = Failure_Link(v)
    }
}
```

Abbildung 2.16: Algorithmus: Die Methode von Aho und Corasick

nicht behauptet, dass ein Suchwort auftritt. Wir beschränken uns also auf den Fall, dass eines der Suchwörter aus S in t auftritt.

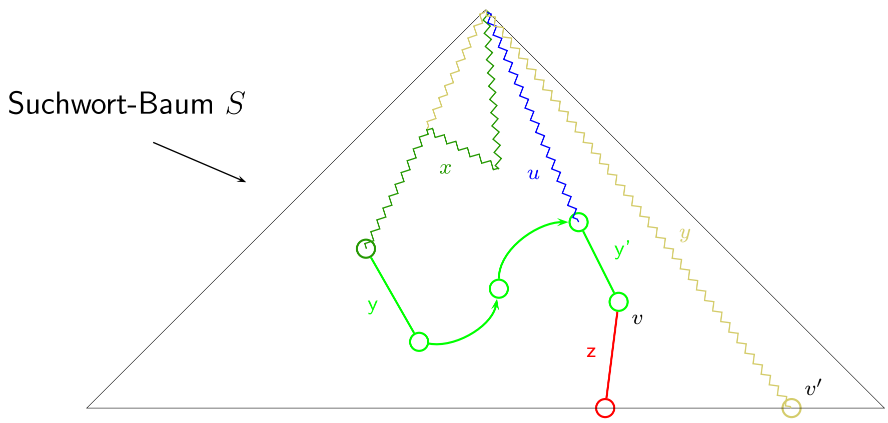


Abbildung 2.17: Skizze: Im Suchwort-Baum wird Treffer von y gemeldet

Was passiert, wenn $y \in S$ als ein Teilwort von t auftritt und sich der Algorithmus unmittelbar nach dem Lesen von y in einem internen Knoten v befindet? Sei uy' das Wort, über den man den Knoten v auf einem einfachen Pfad von der Wurzel aus erreicht, wobei y' das Teilwort des Pfades ist, das während des Algorithmus auf diesem Pfad abgelaufen wurde. Zuerst halten wir fest, dass y' ein Suffix von y ist.

Wir behaupten, dass y ein Suffix von uy' ist. Da $y \in S$, gibt es im Suchwortbaum einen Pfad von der Wurzel zu einem Blatt v', der mit y markiert ist. Somit kann man nach der Verarbeitung von y als Teilwort von t nur an einem Knoten landen, dessen Level mindestens $|y|$ ist (ansonsten müssten wir bei v' gelandet sein). Somit ist $level(v) \geq |y|$. Nach Definition der Failure-Links muss dann die Beschriftung uy' des Pfades von der Wurzel zu v mindestens mit y enden.

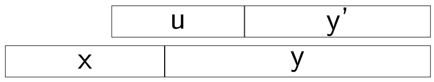


Abbildung 2.18: Skizze: y muss Suffix von uy' sein

Ist z die Beschriftung eines Pfades von der Wurzel zu einem Blatt, das von v aus über normale Baumkanten erreicht werden kann, dann muss y ein echtes Teilwort von $uy'z \in S$ sein. Dies widerspricht aber der Annahme, dass kein Suchwort aus S ein echtes Teilwort eines anderen Suchwortes in S sein kann. Damit befindet sich der Algorithmus von Aho-Corasick nach dem Auffinden eines Suchwortes $s \in S$ in einem Blatt des Baumes. Da y ein Suffix von uy' ist, ist y ein Teilwort von $uy'z \in S$.

Theorem 2.6 *Sei $S \subseteq \Sigma^*$ eine Menge von Suchwörtern, so dass kein Wort $s \in S$ ein echtes Teilwort von $s' \in S$ (mit $s \neq s'$) ist, dann findet der Algorithmus von Aho-Corasick einen Match von $s \in S$ in $t \in \Sigma^*$ in der Zeit $O(n+m)$, wobei $n = |t|$ und $m = \Sigma_{s\in S}|s|$.*

Es stellt sich die Frage, wie der Algorithmus von Aho-Corasick zu erweitern ist, wenn ein Suchwort aus S ein echtes Teilwort eines anderen Suchwortes aus S sein darf. In diesem Fall ist es möglich, dass der Algorithmus beim Auftreten eines Suchwortes $s \in S$ in einem internen Knoten v' des Suchwort-Baumes endet. Sei im Folgenden s' die Kantenbeschriftung des Pfades von der Wurzel zum Knoten v' und, da $s \in S$, sei v der Endpunkt eines einfachen Pfades aus Baumkanten von der Wurzel, dessen Kantenbeschriftungen gerade s ergeben.

Wir überlegen uns zuerst, dass s ein Suffix von s' sein muss. Sei $t's$ der Präfix von t, der gelesen wurde, bis ein Suchwort $s \in S$ gefunden wird. Gemäß der Vorgehensweise des Algorithmus und der Definition der Failure-Links muss s' ein Suffix von $t's$ sein. Ist $|s'| \geq |s|$, dann muss s ein Suffix von s' sein. Andernfalls ist $|s'| < |s|$ und somit $level(v') = |s'| < |s|$. Wir behaupten jetzt, dass dies nicht möglich sein kann. Betrachten wir hierzu die Abarbeitung von $t's$. Sei $\bar{s}$ das längste Präfix von s, so dass sich der Algorithmus nach der Abarbeitung von $t'\bar{s}$ in einem Knoten w mit

$level(w) \geq |\bar{s}|$ befindet. Da mindestens ein solches Präfix die Bedingung erfüllt (z.B. für $\bar{s} = \varepsilon$), muss es auch ein Längstes geben. Anschließend muss der Algorithmus dem Failure-Link von w folgen, da ansonsten $\bar{s}$ nicht das Längste gewesen wäre. Sei also w' = Failure-Link(w). Dann gilt $level(w') < |\bar{s}|$, da ansonsten $\bar{s}$ nicht das Längste gewesen wäre. Dies kann aber nicht sein, da es zu $\bar{s}$ einen Knoten im Suchwort-Baum auf Level $|\bar{s}|$ gibt, wobei die Kantenbeschriftungen des Pfades von der Wurzel zu diesem Knoten gerade $\bar{s}$ ist (da ja $s \in S$ im Suchwort-Baum enthalten ist und $\bar{s}$ ein Präfix von s ist).

Betrachten wir nun den Failure-Link des Knotens v'. Dieser kann auf das Blatt v zeigen (da ja s als Suffix auf dem Pfad zu v' auftreten muss). Andernfalls kann er nach unserer obigen Überlegung nur auf andere Knoten auf einem höheren Level zeigen, wobei die Beschriftung dieses Pfades dann s als Suffix beinhalten muss. Eine Wiederholung dieser Argumentation zeigt, dass letztendlich über die Failure-Links das Blatt v besucht wird. Daher genügt es, den Failure-Links zu folgen, bis ein Blatt erreicht wird. In diesem Fall haben wir ein Suchwort im Text gefunden. Enden wir andernfalls an der Wurzel, so kann an der betreffenden Stelle in t kein Suchwort aus S enden.

Der Algorithmus von Aho-Corasick muss also wie folgt erweitert werden. Wann immer wir einen neuen Knoten über eine Baumkante erreichen, müssen wir testen, ob über eine Folge von Failure-Links (eventuell auch keine) ein Blatt erreichbar ist. Falls ja, haben wir ein Suchwort gefunden, ansonsten nicht.

Anstatt nun jedes Mal den Failure-Links zu folgen, können wir dies auch in einem Vorverarbeitungsschritt durchführen. Zuerst markieren wir alle Blätter als Treffer, die Wurzel als Nicht-Treffer und alle internen Knoten als unbekannt. Dann durchlaufen wir alle als unbekannt markierten Knoten des Baumes. Von jedem solchen Knoten aus folgen wir solange den Failure-Links bis wir auf einen mit Treffer oder Nicht-Treffer markierten Knoten treffen. Anschließend markieren wir alle Knoten auf diesem Pfad genau so wie den gefundenen Knoten. Sobald wir nun im normalen Algorithmus von Aho-Corasick auf einen Knoten treffen, der mit Treffer markiert ist, haben wir ein Suchwort im Text gefunden, ansonsten nicht.

Die Vorverarbeitung lässt sich in Zeit $O(m)$ implementieren, so dass die Gesamtlaufzeit bei $O(m+n)$ bleibt. Wollen wir jedoch zusätzlich auch noch die Endpositionen aller Treffer ausgeben, so müssen wir den Algorithmus noch weiter modifizieren. Wir hängen zusätzlich an die Knoten mit Treffer noch eine Liste mit den Längen der Suchworte an, die an dieser Position enden können. Zu Beginn erhält jedes Blatt eine einelementige Liste, die die Länge des zugehörigen Suchwortes beinhaltet. Alle andere Knoten bekommen eine leere Liste. Finden wir nun einen Knoten, der als Treffer markiert ist, so wird dessen Liste an alle Listen der Knoten auf dem Pfad dorthin als Kopie angefügt. Treffen wir nun beim Algorithmus von Aho-Corasick auf einen als Treffer markierten Knoten, so müssen jetzt mehrere Antworten ausgegeben werden (die Anzahl entspricht der Elemente der Liste). Somit ergibt sich für die Laufzeit $O(m+n+k)$, wobei m die Anzahl der Zeichen in den Suchwörtern ist, n die Länge des Textes t und k die Anzahl der Vorkommen von Suchwörtern aus S in t.

Theorem 2.7 *Sei $S \subseteq \Sigma^*$ eine Menge von Suchwörtern, dann findet der Algorithmus von Aho-Corasick alle Vorkommen von $s \in S$ in $t \in \Sigma^*$ in der Zeit $O(n+m+k)$, wobei $n = |t|$, und $m = \Sigma_{s \in S}|s|$ und k die Anzahl der Vorkommen von Suchwörtern aus s in t ist.*

2.4 Der Algorithmus von Boyer und Moore

In diesem Kapitel werden wir einen weiteren Algorithmus zum exakten Suchen in Texten vorstellen, der zwar im worst-case schlechter als der Knuth-Morris-Pratt-Algorithmus ist, der jedoch in der Praxis meist wesentlich bessere Resultate bzgl. der Laufzeit zeigt.

2.4.1 Ein zweiter naiver Ansatz

Ein weiterer Ansatzpunkt zum Suchen in Texten ist der, das gesuchte Wort mit einem Textstück nicht mehr von links nach rechts, sondern von rechts nach links zu vergleichen. Ein Vorteil dieser Vorgehensweise ist, dass man in Alphabeten mit vielen verschiedenen Symbolen bei einem Mismatch an der Position $i+j$ in t, i auf $i+j+1$ setzen kann, falls das im Text t gesuchte Zeichen t_{i+j} gar nicht im gesuchten Wort s vorkommt. D.h. in so einem Fall kann man das Suchwort gleich um $j+1$ verschieben.

Diese naive Idee ohne den Trick, bei einen Mismatch mit einem Zeichen aus t, das in s gar nicht auftritt, das Muster s möglichst weit zu schieben, ist im Algorithmus auf der nächsten Seite wiedergegeben.

Das folgende Bild zeigt ein Beispiel für den Ablauf des naiven Algorithmus. Dabei erkennt man recht schnell, dass es nutzlos ist (siehe zweiter Versuch von oben), wenn man die Zeichenreihe so verschiebt, dass im Bereich erfolgreicher Vergleiche nach einem Shift keine Übereinstimmung mehr herrscht. Diese Betrachtung ist völlig analog zur Schiebe-Operation im KMP-Algorithmus.

Weiter bemerkt man, dass es ebenfalls keinen Sinn macht, das Muster so zu verschieben, dass an der Position des Mismatches in t im Muster s wiederum dasselbe

```
BOOL NAIV2 (char t[], int n, char s[], int m)
{
    int i = 0, j = m − 1;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            if (j == 0) return TRUE;
            j−−;
        }
        i++;
        j = m − 1;
    }
    return FALSE;
}
```

Abbildung 2.19: Algorithmus: Eine naive Methode mit rechts-nach-links Vergleichen

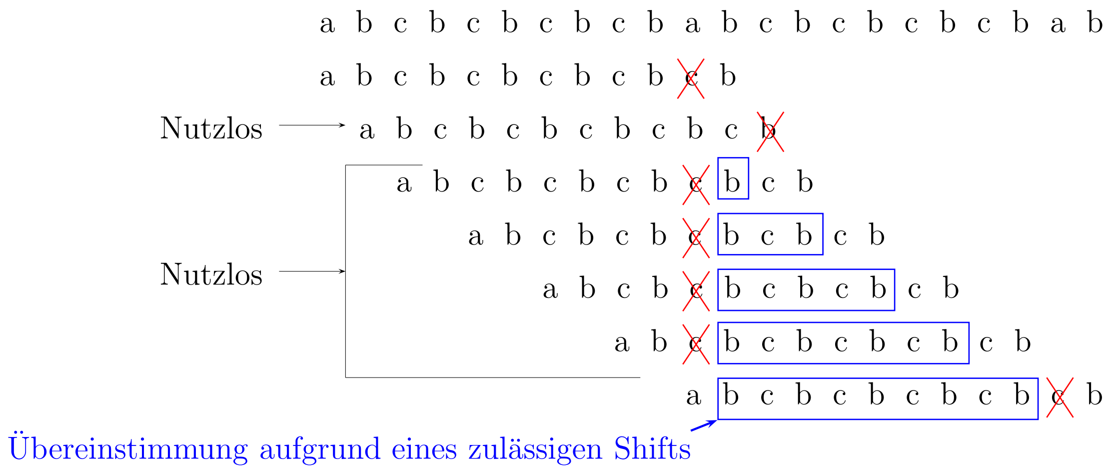


Abbildung 2.20: Beispiel: Suchen mit der zweiten naiven Methode

Zeichen zum Liegen kommt, das schon vorher den Mismatch ausgelöst hat. Daher kann man auch den dritten bis sechsten Versuch als nutzlos bezeichnen.

2.4.2 Der Algorithmus von Boyer-Moore

Der von R.S. Boyer und J.S. Moore vorgeschlagene Algorithmus unterscheidet sich vom zweiten naiven Ansatz nunmehr nur in der Ausführung größerer Shifts, welche in der Shift-Tabelle gespeichert sind. Folgende Skizze zeigt die möglichen Shifts beim Boyer-Moore-Algorithmus:

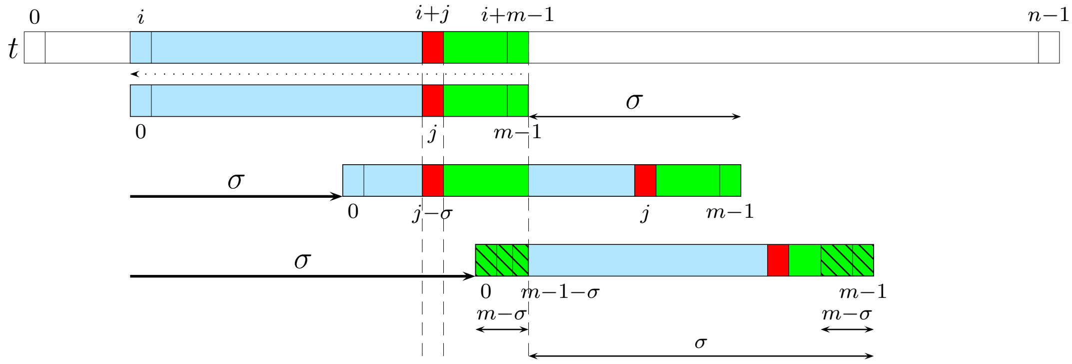


Abbildung 2.21: Skizze: Zulässige Shifts bei Boyer-Moore (Strong-Good-Suffix-Rule)

Prinzipiell gibt es zwei mögliche Arten eines „vernünftigen“ Shifts bei der Variante von Boyer-Moore. Im oberen Teil ist ein „kurzer“ Shift angegeben, bei dem im grünen Bereich die Zeichen nach dem Shift weiterhin übereinstimmen. Das rote Zeichen in t, welches den Mismatch ausgelöst hat, wird nach dem Shift auf ein anderes Zeichen in s treffen, damit überhaupt die Chance auf Übereinstimmung besteht. Im unteren Teil ist ein „langer“ Shift angegeben, bei dem die Zeichenreihe s soweit verschoben wird, so dass an der Position des Mismatches in t gar kein weiterer Vergleich mehr entsteht. Allerdings soll auch hier im schraffierten grünen Bereich wieder Übereinstimmung mit den bereits verglichenen Zeichen aus t herrschen.

Die Kombination der beiden obigen Regeln nennt man die *Good-Suffix-Rule*, da man darauf achtet, die Zeichenreihen so zu verschieben, dass im letzten übereinstimmenden Bereich wieder Übereinstimmung herrscht. Achtet man noch speziell darauf, dass an der Position, in der es zum Mismatch gekommen ist, jetzt in s eine anderes Zeichen liegt, als das, welches den Mismatch ausgelöst hat, so spricht man von der *Strong-Good-Suffix-Rule*, andernfalls *Weak-Good-Suffix-Rule*. Im Folgenden werden wir nur diese Strong-Good-Suffix-Rule betrachten, da ansonsten die worst-case Laufzeit wieder quadratisch werden kann.

Der Boyer-Moore-Algorithmus sieht dann wie in der Abbildung auf der nächsten Seite aus. Hierbei ist $S[]$ die Shift-Tabelle, über deren Einträge wir uns im Detail im Folgenden noch Gedanken machen werden.

Man beachte hierbei, dass es nach dem Shift einen Bereich gibt, in dem Übereinstimmung von s und t vorliegt. Allerdings werden auch in diesem Bereich wieder Vergleiche ausgeführt, da es letztendlich doch zu aufwendig ist, sich diesen Bereich explizit zu merken und bei folgenden Vergleichen von s in t zu überspringen.

```
bool Boyer-Moore (char t[], int n, char s[], int m)
{
    int S[m + 1];
    compute_shift_table(S, m, s);
    int i = 0, j = m − 1;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            if (j == 0) return TRUE;
            j−−;
        }
        i = i + S[j];
        j = m − 1;
    }
    return FALSE;
}
```

Abbildung 2.22: Algorithmus: Boyer-Moore mit Strong-Good-Suffix-Rule

Offen ist nun nur noch die Frage nach den in der Shift-Tabelle zu speichernden Werten und wie diese effizient bestimmt werden können. Hierzu sind einige Vorüberlegungen zu treffen. Der erste Mismatch soll im zu durchsuchenden Text t an der Stelle $i+j$ auftreten. Da der Boyer-Moore-Algorithmus das Suchwort von hinten nach vorne vergleicht, ergibt sich folgende Voraussetzung:

$$s_{j+1} \cdots s_{m-1} = t_{i+j+1} \cdots t_{i+m-1} \qquad \wedge \qquad s_j \neq t_{i+j}$$

Um nun einen nicht nutzlosen Shift um σ Positionen zu erhalten, muss gelten:

$$s_{j+1-\sigma} \cdots s_{m-1-\sigma} = t_{i+j+1} \cdots t_{i+m-1} = s_{j+1} \cdots s_{m-1} \qquad \wedge \qquad s_j \neq s_{j-\sigma}$$

Diese Bedingung ist nur für „kleine“ Shifts mit $\sigma \leq j$ sinnvoll. Ein solcher Shift ist im obigen Bild als erster Shift zu finden. Für „große“ Shifts $\sigma > j$ muss gelten, dass das Suffix des übereinstimmenden Bereichs mit dem Präfix des Suchwortes übereinstimmt, d.h.:

$$s_0 \cdots s_{m-1-\sigma} = t_{i+\sigma} \cdots t_{i+m-1} = s_\sigma \cdots s_{m-1}$$

Zusammengefasst ergibt sich für beide Bedingungen folgendes:

$$\begin{array}{lllll} \sigma \leq j & \wedge & s_{j+1} \cdots s_{m-1} \in \mathcal{R}(s_{j+1-\sigma} \cdots s_{m-1}) & \wedge & s_j \neq s_{j-\sigma} \\ \sigma > j & \wedge & s_0 \cdots s_{m-1-\sigma} \in \mathcal{R}(s_0 \cdots s_{m-1}) & & \end{array},$$

wobei $\mathcal{R}(s)$ die Menge aller Ränder bezeichnet. Erfüllt ein Shift nun eine dieser Bedingungen, so nennt man diesen Shift *zulässig*. Um einen sicheren Shift zu erhalten, wählt man das minimale σ, das eine der Bedingungen erfüllt.

2.4.3 Bestimmung der Shift-Tabelle

Zu Beginn wird die Shift-Tabelle an allen Stellen mit der Länge des Suchstrings initialisiert. Im Wesentlichen entsprechen beide Fälle von möglichen Shifts (siehe obige Vorüberlegungen) der Bestimmung von Rändern von Teilwörtern des gesuchten Wortes. Dennoch unterscheiden sich die hier betrachteten Fälle vom KMP-Algorithmus, da hier, zusätzlich zu den Rändern von Präfixen des Suchwortes, auch die Ränder von Suffixen des Suchwortes gesucht sind.

```
COMPUTE_SHIFT_TABLE (int S[], char s[], int m)
{
    /* Initialisierung von S[] */
    for (int j = 0; j ≤ m; j++) S[j] = m;
    /* Teil 1: σ ≤ j */
    int border2[m + 1];
    border2[0] = −1;
    int i = border2[1] = 0;
    for (int j′ = 2; j′ ≤ m; j′++)
    {   /* Beachte, dass hier gilt: i == border2[j′ − 1] */
        while ((i ≥ 0) && (s[m − i − 1] ≠ s[m − j′]))
        {
            int σ = j′ − i − 1;
            S[m − i − 1] = min(S[m − i − 1], σ);
            i = border2[i];
        }
        i++;
        border2[j′] = i;
    }
    /* Teil 2: σ > j */
    int j = 0;
    for (int i = border2[m]; i ≥ 0; i = border2[i])
    {
    int σ = m − i;
        while (j < σ)
        {
            S[j] = min(S[j], σ);
            j++;
        }
    }
}
```

Abbildung 2.23: Algorithmus: Berechnung der Shift-Tabelle für Boyer-Moore

Dies gilt besonders für den ersten Fall ($\sigma \leq j$). Genauer gesagt ist man im ersten Fall besonders daran interessiert, dass das Zeichen unmittelbar vor dem Rand des Suffixes ungleich dem Zeichen unmittelbar vor dem gesamten Suffix sein soll. Diese Situation entspricht dem Fall in der Berechnung eines eigentlichen Randes, bei dem der vorhandene Rand nicht zu einem längeren Rand fortgesetzt werden kann (nur werden Suffixe statt Präfixe betrachtet). Daher sieht die erste for-Schleife genau so aus, wie in der Berechnung von compute_border. Lässt sich ein betrachteter Rand nicht verlängern, so wird die while-Schleife ausgeführt. In den grünen Anweisungen wird zunächst die Länge des Shifts berechnet und dann die entsprechende Position in der Shift-Tabelle (wo der Mismatch in s passiert ist) aktualisiert.

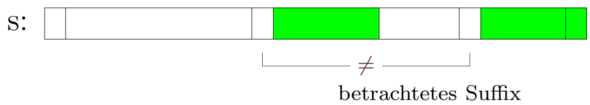


Abbildung 2.24: Skizze: Verlängerung eines Randes eines Suffixes

Im zweiten Fall ($\sigma > j$) müssen wir alle Ränder von s durchlaufen. Der längste Rand ungleich s ist der eigentliche Rand von s. Diesen erhalten wir über $border2[m]$, da ja der Suffix der Länge m von s gerade wieder s ist. Was ist der nächstkürzerere Rand von s? Dieser muss ein Rand des eigentlichen Randes von s sein. Damit es der nächstkürzere Rand ist, muss s der eigentliche Rand des eigentlichen Randes von s sein, also das Suffix der Länge $border2[border2[s]]$. Somit können wir alle Ränder von s durchlaufen, indem wir immer vom aktuell betrachteten Rand der Länge ℓ den Suffix der Länge $border2[\ell]$ wählen. Dies geschieht in der for-Schleife im zweiten Teil. Solange der Shift σ größer als die Position j eines Mismatches ist, wird die Shift-Tabelle aktualisiert. Dies geschieht in der inneren while-Schleife.

Wir haben bei der Aktualisierung der Shift-Tabelle nur zulässige Werte berücksichtigt. Damit sind die Einträge der Shift-Tabelle (d.h. die Länge der Shifts) nie zu klein. Eigentlich müsste jetzt noch gezeigt werden, dass die Werte auch nicht zu groß sind, d.h. dass es keine Situation geben kann, in der ein kleinerer Shift möglich wäre. Dass dies nicht der Fall ist, kann mit einem Widerspruchsbeweis gezeigt werden (Man nehme dazu an, bei einem Mismatch an einer Position j in s gäbe es einen kürzeren Shift gemäß der Strong-Good-Suffix-Rule und leite daraus einen Widerspruch her). Die Details seien dem Leser zur Übung überlassen.

2.4.4 Laufzeitanalyse des Boyer-Moore Algorithmus:

Man sieht, dass die Prozedur *compute_shift_table* hauptsächlich auf die Prozedur *compute_border* des KMP-Algorithmus zurückgreift. Eine nähere Betrachtung der

Die Shift-Tabelle $S[j]$ wird für alle $j \in [0 : m-1]$ mit der Länge m des Suchstrings vorbesetzt.

Teil 1: $\sigma \leq j$

	0	1	2	3	4	5	6	7	8	9		Mismatch
	a	b	a	b	b	a	b	a	b	a		
border2[1]=0									b	a	$\longrightarrow$	S[9] = 1
border2[2]=0								a	b	a		
border2[3]=1							b	a	b	a		
border2[4]=2						a	b	a	b	a		
border2[5]=3					b	a	b	a	b	a		
border2[6]=4				b	b	a	b	a	b	a	$\longrightarrow$	S[5] = 2
				b	b	a	b	a	b	a	$\longrightarrow$	S[7] = 4
border2[7]=0				b	b	a	b	a	b	a	$\longrightarrow$	S[9] = 6*
border2[8]=1		b	a	b	b	a	b	a	b	a		
border2[9]=2	a	b	a	b	b	a	b	a	b	a		
border2[10]=3	a	b	a	b	b	a	b	a	b	a		

Die Besetzung der Shift-Tabelle erfolgt immer nach dem Prinzip, dass das Minimum des gespeicherten Wertes und des neu gefundenen Wertes gespeichert wird, d.h. $S[j] = min\{S[j]; \text{neu gefundener Wert}\}$.

* Dieser Wert wird nicht gespeichert, da S[9] schon mit 1 belegt ist

Abbildung 2.25: Beispiel: $s = ababbababa$ (Teil 1: $\sigma \leq j$)

Teil 2: $\sigma > j$

$\boxed{\text{a b a}}$ b b a b $\boxed{\text{a b a}}$

$\xrightarrow[\sigma = 7]{}$ a b a b b a b a b a

bei Mismatch an Position $j \in [0:6]$ S[j] = 7

$\boxed{\text{a}}$ b a b b a b a b $\boxed{\text{a}}$

$\xrightarrow[\sigma = 9]{}$ a b a b b a b a b a

bei Mismatch an Position $j \in [7:8]$ S[j] = 9

a b a b b a b a b a

$\xrightarrow[\sigma = 10]{}$ a b a b b a b a b a

bei Mismatch an Position $j \in [9]$ S[j] = 10

Zusammenfassung:

	1.Teil	2.Teil	Erg		1.Teil	2.Teil	Erg
S[0] =		7	7	S[5] =	2		2
S[1] =		7	7	S[6] =		7	7
S[2] =		7	7	S[7] =	4		4
S[3] =		7	7	S[8] =		9	9
S[4] =		7	7	S[9] =	1		1

Abbildung 2.26: Beispiel: $s = ababbababa$ (Teil 2: $\sigma > j$)

beiden Schleifen des zweiten Teils ergibt, dass die Schleifen nur m-mal durchlaufen werden und dort überhaupt keine Vergleiche ausgeführt werden.

Lemma 2.8 *Die Shift-Tabelle des Boyer-Moore-Algorithmus lässt sich für eine Zeichenkette der Länge m mit maximal $2m$ Vergleichen berechnen.*

Es bleibt demnach nur noch, die Anzahl der Vergleiche in der Hauptprozedur zu bestimmen. Hierbei wird nur die Anzahl der Vergleiche für eine erfolglose Suche bzw. für das erste Auftreten des Suchwortes s im Text t betrachtet. Wir werden zum Abzählen der Vergleiche wieder einmal zwischen zwei verschiedenen Arten von Vergleichen unterscheiden: *initiale* und *wiederholte* Vergleiche.

Initiale Vergleiche:
Vergleiche von s_j mit t_{i+j}, so dass t_{i+j} zum ersten Mal beteiligt ist.
Wiederholte Vergleiche:
Beim Vergleich s_j mit t_{i+j} war t_{i+j} schon früher bei einem Vergleich beteiligt.

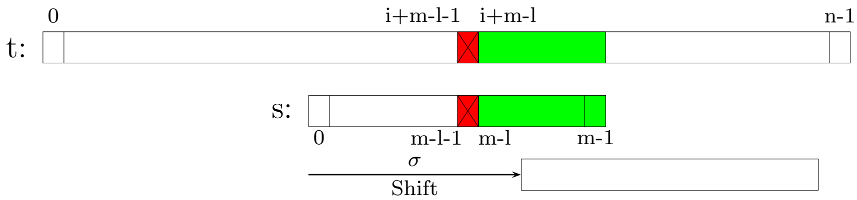


Abbildung 2.27: Skizze: Shift nach ℓ erfolgreichen Vergleichen

Lemma 2.9 *Sei $s_{m-\ell} \cdots s_{m-1} = t_{i+m-\ell} \cdots t_{i+m-1}$ für $i \in [0 : n-m]$ und $\ell \in [0 : m-1]$ sowie $s_{m-\ell-1} \neq t_{i+m-\ell-1}$ (es wurden also ℓ erfolgreiche und ein erfolgloser Vergleich durchgeführt). Dabei seien I initiale Vergleiche durchgeführt worden, und weiter sei σ die Länge des folgenden Shifts gemäß der Strong-Good-Suffix-Rule. Dann gilt:*

$$\ell + 1 \leq I + 4 * \sigma.$$

Aus diesem Lemma folgt, dass maximal $5n$ Vergleiche ausgeführt wurden. Bezeichne dazu $V(n)$ die Anzahl aller Vergleiche, um in einem Text der Länge n zu suchen, und I_i bzw. V_i die Anzahl der initialen bzw. aller Vergleiche, die beim i-ten Versuch ausgeführt wurden. Ferner sei σ_i die Länge des Shifts, der nach dem i-ten Vergleich ausgeführt wird. War der i-te (und somit letzte) Vergleich erfolgreich, so sei $\sigma_i := m$ (Unter anderem deswegen gilt die Analyse nur für einen erfolglosen bzw. den ersten erfolgreichen Versuch).

$$\begin{aligned}
V(n) &= \sum_i V_i \\
&\quad \text{mit Hilfe des obigen Lemmas} \\
&\leq \sum_i (I_i + 4\sigma_i) \\
&\quad \text{da es maximal } n \text{ initiale Vergleiche geben kann} \\
&\leq n + 4\sum_i \sigma_i \\
&\quad \text{da die Summe der Shifts maximal } n \text{ sein kann} \\
&\quad \text{(der Anfang von } s \text{ bleibt innerhalb von } t\text{)} \\
&\leq n + 4n \;=\; 5n
\end{aligned}$$

Zum Beweis des obigen Lemmas nehmen wir an, dass ℓ erfolgreiche und ein erfolgloser Vergleich stattgefunden haben. Wir unterscheiden jetzt den darauf folgenden Shift in Abhängigkeit seiner Länge im Verhältnis zu ℓ. Ist $\sigma \geq \lceil \ell/4 \rceil$, dann heißt der Shift *lang*, andernfalls ist $\sigma \leq \lceil \ell/4 \rceil - 1$ und der Shift heißt *kurz*.

<u>1.Fall:</u> Shift σ ist lang, d.h. $\sigma \geq \lceil \ell/4 \rceil$:

$$\begin{aligned} \text{Anzahl der ausgeführten Vergleiche} &= \ell + 1 \\ &\leq 1 + 4 * \lceil \ell/4 \rceil \\ &\leq 1 + 4 * \sigma \\ &\leq I + 4 * \sigma \end{aligned}$$

Die letzte Ungleichung folgt aus der Tatsache, dass es bei jedem Versuch immer mindestens einen initialen Vergleich geben muss (zu Beginn und nach einem Shift wird dass letzte Zeichen von s mit einem Zeichen aus t verglichen, das vorher noch an keinem Vergleich beteiligt war).

<u>2.Fall:</u> Shift σ ist kurz, d.h. $\sigma \leq \ell/4$.
Wir zeigen zuerst, dass dann das Ende von s periodisch sein muss, d.h. es gibt ein $\alpha \in \Sigma^+$, so dass s mit α^5 enden muss. Betrachten wir dazu die folgende Skizze:

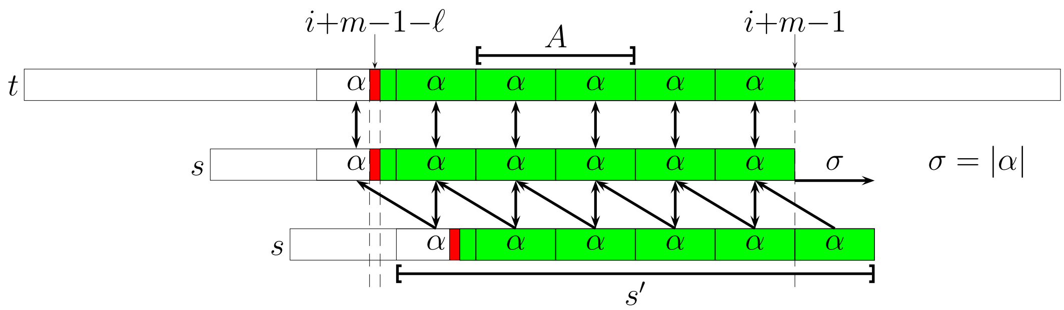


Abbildung 2.28: Skizze: Periodisches Ende von s bei kurzen Shifts

Wir wissen, dass der Shift der Länge σ kurz ist. Wegen der Strong-Good-Suffix-Rule wissen wir, dass im Intervall $[i + m - \ell : i + m - 1]$ in t die Zeichen mit der verschobenen Zeichenreihe s übereinstimmen. Aufgrund der Kürze des Shifts der Länge σ folgt, dass das Suffix α von s der Länge σ mindestens $\lfloor \ell/\sigma \rfloor$ mal am Ende der Übereinstimmung von s und t vorkommen muss. Damit muss α also mindestens $k := 1 + \lfloor \ell/\sigma \rfloor \geq 5$ mal am Ende von s auftreten (siehe auch obige Abbildung). Nur falls das Wort s kürzer als $k \cdot \sigma$ ist, ist s ein Suffix von α^k. Sei im Folgenden s' das Suffix der Länge $k \cdot \sigma$ von s, falls $|s| \geq k \cdot \sigma$ ist, und $s' = s$ sonst. Wir halten noch fest, dass sich im Suffix s' die Zeichen im Abstand von σ Positionen wiederholen.

Wir werden jetzt mit Hilfe eines Widerspruchbeweises zeigen, dass die Zeichen in t an den Positionen im Intervall $A := [i + m - (k - 2)\sigma : i + m - 2\sigma - 1]$ bislang noch an keinem Vergleich beteiligt waren. Dazu betrachten wir frühere Versuche, die Vergleiche im Abschnitt A hätten ausführen können.

Zuerst betrachten wir einen früheren Versuch, bei dem mindestens σ erfolgreiche Vergleiche ausgeführt worden sind. Dazu betrachten wir die folgende Abbildung, wobei der betrachtete Versuch oben dargestellt ist. Da mindestens σ erfolgreiche Vergleiche ausgeführt wurden, folgt aus der Periodizität von s', dass alle Vergleiche bis zur Position $i+m-\ell-2$ erfolgreich sein müssen. Der Vergleich an Position $i+m-\ell-1$ muss hingegen erfolglos sein, da auch der ursprüngliche Versuch, s an Position i in t zu finden, an Position $i+m-\ell-1$ erfolglos war. Wir behaupten nun, dass dann jeder sichere Shift gemäß der Strong-Good-Suffix-Rule die Zeichenreihe s auf eine Position größer als i verschoben hätte.

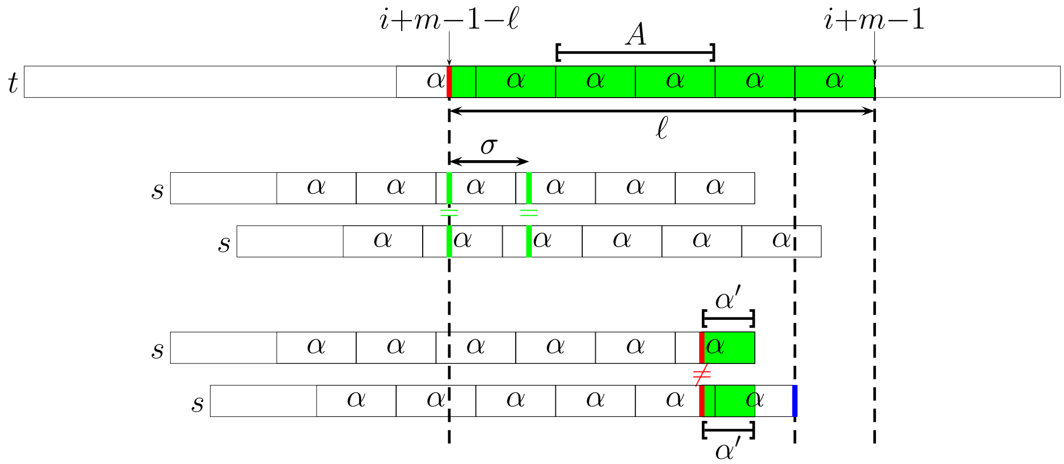


Abbildung 2.29: Skizze: Mögliche frühere initiale Vergleich im Bereich A

Nehmen wir an, es gäbe einen kürzeren sicheren Shift (wie in der Abbildung darunter dargestellt). Dann müsste das Zeichen an Position $i+m-\ell-1$ in t mit einem Zeichen aus s' verglichen werden. Dort steht im verschobenen s' aber dasselbe Zeichen wie beim erfolglosen Vergleich des früheren Versuchs, da sich in s' die Zeichen alle σ Zeichen wiederholen. Damit erhalten wir einen Widerspruch zur Strong-Good-Suffix-Rule. Falls der Shift um s Zeichen zu kurz ist, damit es zu einem Vergleich mit dem Zeichen an Position $i+m-\ell-1$ kommen kann, hätten wir s in t gefunden und wir hätten keinen Versuch an Position i unternommen.

Es bleiben noch die früheren Versuche, die weniger als σ Vergleiche ausgeführt haben. Da wir nur Versuche betrachten, die Vergleiche im Abschnitt A ausführen, muss der Versuch an einer Position im Intervall $[i-(k-2)\sigma : i-\sigma-1]$ von rechts nach links begonnen haben. Nach Wahl von A muss der erfolglose Vergleich auf einer Position größer als $i+m-\ell-1$ erfolgen. Betrachte hierzu die erste Zeile im unteren Teil der obigen Abbildung. Sei α' das Suffix von α (und damit von s' bzw. s), in dem die erfolgreichen Vergleiche stattgefunden haben. Seien nun $x \neq y$ die Zeichen, die den Mismatch ausgelöst haben, wobei x unmittelbar vor dem Suffix α' in s' steht. Offensichtlich liegt α' völlig im Intervall $[i - m - \ell : i + m - \sigma - 1]$.

Wir werden jetzt zeigen, dass ein Shift auf $i - \sigma$ (wie im unteren Teil der Abbildung darunter dargestellt) zulässig ist. Die alten erfolgreichen Vergleiche stimmen mit dem Teilwort in s' überein. Nach Voraussetzung steht an der Position unmittelbar vor α' in s' das Zeichen x und an der Position des Mismatches in s' das Zeichen y. Damit ist ein Shift auf $i - \sigma$ zulässig. Da dies aber nicht notwendigerweise der Kürzeste sein muss, erfolgt ein sicherer Shift auf eine Position kleiner gleich $i - \sigma$.

Erfolgt ein Shift genau auf Position $i - \sigma$, dann ist der nächste Versuch bis zur Position $i + m - \ell - 1$ in t erfolgreich. Da wir dann also mindestens σ erfolgreiche Vergleiche ausführen, folgt, wie oben erläutert, ein Shift auf eine Position größer als i (oder der Versuch wäre erfolgreich abgeschlossen worden). Andernfalls haben wir einen Shift auf Position kleiner als $i - \sigma$ und wir können dieselbe Argumentation wiederholen. Also erhalten wir letztendlich immer einen Shift auf eine Position größer als i, aber nie einen Shift auf die Position i.

Damit ist bewiesen, dass bei einem kurzen Shift die Zeichen im Abschnitt A von t zum ersten Mal verglichen worden sind. Da auch der Vergleich des letzten Zeichens von s mit einem Zeichen aus t ein initialer gewesen sein musste, folgt dass mindestens $1 + |A|$ initiale Vergleiche ausgeführt wurden. Da $|A| \geq \ell - 4\sigma$, erhalten wir die gewünschte Abschätzung:

$$1 + \ell = (1 + |A|) + (\ell - |A|) \leq (1 + |A|) + 4\sigma.$$

Da wie schon weiter oben angemerkt, der Vergleich des letzten Zeichens von s immer initial sein muss, und alle Vergleiche im Bereich A initial sind, folgt damit die Behauptung des Lemmas.

Theorem 2.10 *Der Boyer-Moore Algorithmus benötigt für das erste Auffinden des Musters s in t maximal $5n$ Vergleiche ($|s| \leq |t| = n$)*

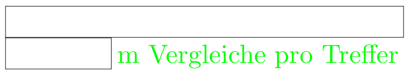
Mit Hilfe einer besseren, allerdings für die Vorlesung zu aufwendigen Analyse, kann man folgendes Resultat herleiten.

Theorem 2.11 *Der Boyer-Moore Algorithmus benötigt maximal $3(n + m)$ Vergleiche um zu entscheiden, ob eine Zeichenreihe der Länge m in einem Text der Länge n enthalten ist.*

Die Behauptung im obigen Satz ist scharf, da man Beispiele konstruieren kann, bei denen mindestens $3n - o(n)$ Vergleiche bei einer Suche mit der Strong-Good-Suffix-Rule beim Boyer-Moore-Algorithmus ausgeführt werden müssen. Damit ist der Boyer-Moore-Algorithmus zwar im worst-case etwas schlechter als der Knuth-Morris-Pratt-Algorithmus, dafür jedoch im average-case deutlich besser.

Wir erinnern nochmals, dass unsere Analyse nur für den Fall einer erfolglosen Suche oder dem ersten Auffinden von s in t gültig ist. Folgendes Beispiel zeigt, dass der Boyer-Moore-Algorithmus beim Auffinden aller Vorkommen von s in t wieder quadratischen Zeitbedarf bekommen kann (im Gegensatz zum KMP-Algorithmus, der immer seine Laufzeit von $2n + m$ beibehält).

Suche a^m in a^n



n-m Vorkommen von a^m
$\Rightarrow$ Laufzeit $\Theta(n * m)$

Abbildung 2.30: Skizze: Anzahl Vergleiche bei Boyer-Moore bei Suche a^m in a^n

Diese worst-case Laufzeit lässt sich jedoch vermeiden, wenn man einen Trick anwendet. Nach einem erfolgreichen Vergleich verschieben wir das Muster jetzt wie beim KMP-Algorithmus. Da ja alle Vergleiche erfolgreich waren, gibt es ja nun keinen Präfix des Suchwortes, das noch unverglichen wäre. Dann merken wir uns (ähnlich wie beim KMP-Algorithmus),bis wohin wir beim Vergleichen von rechts nach links im Folgenden noch vergleichen müssen, da wir im Präfix des Suchwortes, das einen Rand darstellt, nach dem erfolgreichen Vergleich sicher sein können, dass dort eine Übereinstimmung herrscht.

2.4.5 Bad-Character-Rule

Eine andere mögliche Beschleunigung beim Boyer-Moore Algorithmus stellt, wie in der Einleitung zu diesem Abschnitt bereits angedeutet, die so genannte *Bad-Character-Rule* dar. Bei einem Mismatch von s_j mit t_{i+j} verschieben wir das Such-

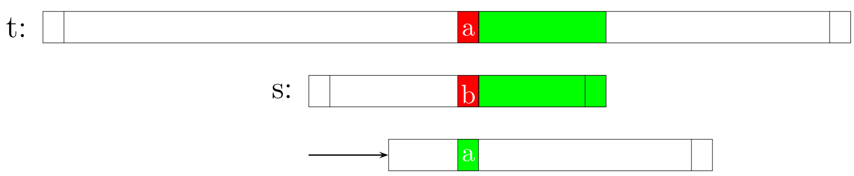


Abbildung 2.31: Skizze: Bad-Character-Rule

wort s so, dass das rechteste Vorkommen von t_{i+j} in s genau unter t_{i+j} zu liegen kommt. Falls wir die Zeichenreihe s dazu nach links verschieben müssten, schieben wir s stattdessen einfach um eine Position nach rechts. Falls das Zeichen in s gar nicht auftritt, so verschieben wir s so, dass das erste Zeichen von s auf der Position $i + j + 1$ in t zu liegen kommt. Hierfür benötigt man eine separate Tabelle mit $|\Sigma|$ Einträgen.

Diese Regel macht insbesondere bei kleinen Alphabeten Probleme, wenn beim vorigen Vergleich schon einige Zeichenvergleiche ausgeführt wurden. Denn dann ist das gesuchte Zeichen in s meistens rechts von der aktuell betrachteten Position. Hier kann man die so genannte *Extended-Bad-Character-Rule* verwenden. Bei einem Mismatch von s_j mit t_{i+j} sucht man nun das rechteste Vorkommen von t_{i+j} im Präfix $s_0 \cdots s_{j-1}$ (also links von s_j) und verschiebt nun s so nach rechts, dass die beiden Zeichen übereinander zu liegen kommen. Damit erhält man immer recht große Shifts. Einen Nachteil erkauft man sich dadurch, dass man für diese Shift-Tabelle nun $m \cdot |\Sigma|$ Einträge benötigt, was aber bei kleinen Alphabetgrößen nicht sonderlich ins Gewicht fällt.

In der Praxis wird man in der Regel sowohl die Strong-Good-Suffix-Rule als auch die Extended-Bad-Character-Rule verwenden. Hierbei darf der größere der beiden vorgeschlagenen Shifts ausgeführt werden. Das worst-case Laufzeitverhalten verändert sich dadurch nicht, aber die average-case Laufzeit wird besser. Wir weisen noch darauf hin, dass sich unser Beweis für die Laufzeit nicht einfach auf diese Kombination erweitern lässt.

2.5 Der Algorithmus von Karp und Rabin

Im Folgenden wollen wir einen Algorithmus zum Suchen in Texten vorstellen, der nicht auf dem Vergleichen von einzelnen Zeichen beruht. Wir werden vielmehr das Suchwort bzw. Teile des zu durchsuchenden Textes als Zahlen interpretieren und dann aufgrund von numerischer Gleichheit auf das Vorhandensein des Suchwortes schließen.

2.5.1 Ein numerischer Ansatz

Der Einfachheit halber nehmen wir im Folgenden an, dass $\Sigma = \{0, 1\}$ ist. Die Verallgemeinerung auf beliebige k-elementige Alphabete, die wir ohne Beschränkung der Allgemeinheit als $\Sigma = \{0, \ldots, k-1\}$ annehmen dürfen, sei dem Leser überlassen.

Da $\Sigma = \{0, 1\}$ kann jedes beliebige Wort aus Σ^* als Binärzahl interpretiert werden. Somit kann eine Funktion V angegeben werden, welche jedem Wort aus Σ^* ihre zugehörige Dezimalzahl zuordnet.

$$V:\ \{0,1\}^* \longrightarrow \mathbb{N}_0:\ V(s) = \sum_{i=0}^{|s|-1} 2^i \cdot s_i$$

Für $\Sigma = \{0, \ldots, k-1\}$ gilt dementsprechend:

$$V_k:\ [0:k-1]^* \longrightarrow \mathbb{N}_0:\ V_k(s) = \sum_{i=0}^{|s|-1} k^i \cdot s_i$$

Bsp.: $V_2(0011) = 12$; $V_2(1100) = 3$;

Fakt: s ist ein Teilwort von t an der Stelle i, falls es ein i gibt, $i \in [0:n-m]$, so dass $V(s) = V(t_{i,m})$, wobei $t_{i,m} = t_i \cdots t_{i+m-1}$

Wird also s in t gesucht, müssen immer nur zwei Dezimalzahlen miteinander verglichen werden. Dabei wird $V(s)$ mit jeder Dezimalzahl $V(t_{i,m})$ verglichen, die durch das Teilwort von t der Länge m an der Stelle i repräsentiert wird. Stimmen die beiden Zahlen überein, ist s in t an der Stelle i enthalten.

Diese Idee ist im folgenden einfachen Algorithmus wiedergegeben:

```
BOOL NAIV3 (char s[], int m, char t[], int n)
{
    for (i = 0; i ≤ n − m; i++) {
        if (V(s) == V(t_i, m)) return TRUE;
    }
}
```

Abbildung 2.32: Algorithmus: Naive numerische Textsuche

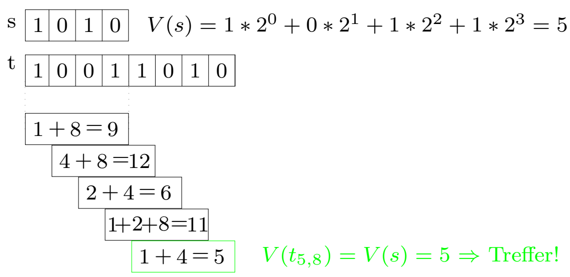


Abbildung 2.33: Beispiel: Numerische Suche von 1010 in 100110101

Wie groß ist der Aufwand, um $V(x)$ mit $x \in \Sigma^m$, $x = x_0 ... x_{m-1}$ zu berechnen?

$$V(x) = \underbrace{\sum_{i=0}^{m-1} x_i \cdot 2^i}_{1} = \underbrace{x_0 + 2(x_1 + 2(x_2 + 2(x_3 ... x_{m-2} + 2x_{m-1})))}_{2}$$

Die zweite Variante zur Berechnung folgt aus einer geschickten Ausklammerung der einzelnen Terme und ist auch als *Horner-Schema* bekannt.

Aufwand für 1: $\sum_{i=0}^{m-1} i + (m-1) = \Theta(m^2)$ arithmetische Operationen.

Aufwand für 2: $\Theta(m)$ arithmetische Operationen.

Somit ist die Laufzeit für die gesamte Suche selbst mit Hilfe der zweiten Methode immer noch $\Theta(n*m)$. Wiederum wollen wir versuchen, dieses quadratische Verhalten zu unterbieten.

Wir versuchen zuerst festzustellen, ob uns die Kenntnis des Wertes für $V(t_{i,m})$ für die Berechnung des Wertes $V(t_{i+1,m})$ hilft:

$$\begin{aligned}
V(t_{i+1,m}) &= \sum_{j=0}^{m-1} t_{i+1+j} \cdot 2^j \\
&= \sum_{j=1}^{m} t_{i+1+j-1} \cdot 2^{j-1} \\
&= \frac{1}{2} \sum_{j=1}^{m} t_{i+j} \cdot 2^j \\
&= \frac{1}{2} \left(\sum_{j=0}^{m-1} t_{i+j} \cdot 2^j - t_{i+0} \cdot 2^0 + t_{i+m} \cdot 2^m \right) \\
&= \frac{1}{2} * (V(t_{i,m}) - t_i + 2^m t_{i+m})
\end{aligned}$$

$V(t_{i+1,m})$ lässt sich somit mit dem Vorgänger $V(t,m)$ sehr einfach und effizient berechnen, nämlich in konstanter Zeit.

Damit ergibt sich für die gesamte Laufzeit $O(n+m)$: Für das Preprocessing zur Berechnung von $V(s)$ und 2^m benötigt man mit dem Horner-Schema $O(m)$. Die Schleife selbst wird dann n-Mal durchlaufen und benötigt jeweils eine konstante Anzahl arithmetischer Operationen, also $O(n)$.

Allerdings haben wir hier etwas geschummelt, da die Werte ja sehr groß werden können und damit die Kosten einer arithmetischen Operation sehr teuer werden können.

2.5.2 Der Algorithmus von Karp und Rabin

Nun kümmern wir uns um das Problem, dass die Werte von $V(s)$ bzw. von $V(t_{i,m})$ sehr groß werden können. Um dies zu verhindern, werden die Werte von V einfach modulo einer geeignet gewählten Zahl gerechnet. Dabei tritt allerdings das Problem auf, dass der Algorithmus Treffer auswirft, die eigentlich gar nicht vorhanden sind (siehe folgendes Beispiel).

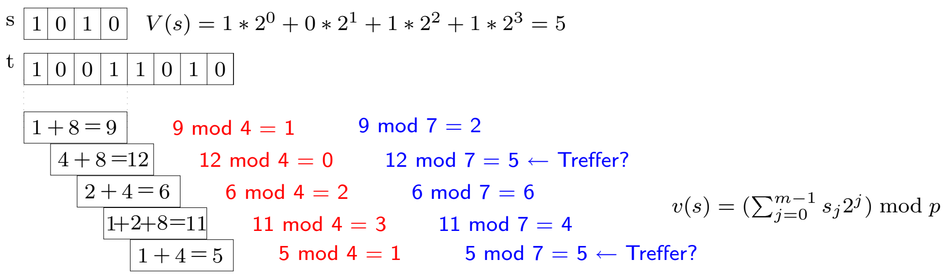


Abbildung 2.34: Beispiel: Numerische Suche von 1010 in 10011010 modulo 7

Die Zahl, durch welche modulo gerechnet wird, sollte teilerfremd zu $|\Sigma|$ sein, da ansonsten Anomalien auftreten können. Wählt man beispielsweise eine Zweierpotenz, so werden nur die ersten Zeichen des Suchwortes berücksichtigt, wo hingegen die letzten Zeichen des Suchwortes bei dieser Suche überhaupt keine Relevanz haben. Um unabhängig von der Alphabetgröße die Teilerfremdheit garantieren zu können, wählt man p als eine Primzahl.

Um nun sicher zu sein, dass man wirklich einen Treffer gefunden hat, muss zusätzlich noch im Falle eines Treffers überprüft werden, ob die beiden Zeichenreihen auch wirklich identisch sind. Damit erhalten wir den Algorithmus von R. Karp und M. Rabin, der im Detail auf der nächsten Seite angegeben ist. Den Wert $V(s) \bmod p$ nennt man auch einen *Fingerabdruck* (engl. *fingerprint*) des Wortes s.

2.5.3 Bestimmung der optimalen Primzahl

Es stellt sich nun die Frage, wie die Primzahl, mit der modulo gerechnet wird, auszusehen hat, damit möglichst selten der Fall auftritt, dass ein „falscher Treffer“ vom Algorithmus gefunden wird.

Sei $\mathbb{P}$ ={2,3,5,7...} die Menge aller Primzahlen. Weiter sei $\pi(k) := |P \cap [1:k]|$ die Anzahl aller Primzahlen kleiner oder gleich k. Wir stellen nun erst einmal ein paar nützliche Tatsachen aus der Zahlentheorie zu Verfügung, die wir hier nur teilweise beweisen wollen.

BOOL KARP-RABIN (char $t[]$,int n, char $s[]s$, int m)

```
{
    int p;  // hinreichend große Primzahl, die nicht zu groß ist ;-)
    int vs = v(s) mod p = ∑_{j=0}^{m-1} s_j · 2^j mod p;
    int vt = (∑_{j=0}^{m-1} t_j · 2^j) mod p;
    for (i = 0; i ≤ n − m; i++)
    {
        if ((vs == vt) && (s == t_{i,m})) // Zusätzlicher Test auf Gleichheit
            return TRUE;
        vt = 1/2 (vt − t_i + (2^m mod p) − t_{i+m}) mod p;
    }
}
```

Abbildung 2.35: Algorithmus: Die Methode von Karp und Rabin

Theorem 2.12 *Für alle $k \in \mathbb{N}$ gilt:*

$$\frac{k}{\ln(k)} \leq \pi(k) \leq 1.26\frac{k}{\ln(k)}.$$

Theorem 2.13 *Sei $k \geq 29$, dann gilt*

$$\prod_{\substack{p\in\mathbb{P}\\p\leq k}} p \geq 2^k.$$

Beweis: Mit Hilfe des vorherigen Satzes und der Stirlingschen Abschätzung, d.h. mit $n! \geq \left(\frac{n}{e}\right)^n$, folgt:

$$\prod_{\substack{p\in\mathbb{P}\\p\leq k}} p \geq \prod_{p=1}^{\pi(k)} p = \pi(k)! \geq \left(\frac{k}{\ln(k)}\right)! \geq \left(\frac{k}{e\cdot\ln(k)}\right)^{k/\ln(k)} \geq \left(\left(\frac{k}{e\cdot\ln(k)}\right)^{1/\ln(k)}\right)^k.$$

Da ferner gilt, dass

$$\lim_{k\to\infty}\left(\frac{k}{e\cdot\ln(k)}\right)^{1/\ln(k)} = \exp\left(\lim_{k\to\infty}\frac{\ln(k)-1-\ln(\ln(k))}{\ln(k)}\right) = e,$$

folgt, dass $\left(\frac{k}{e\cdot\ln(k)}\right)^{1/\ln(k)} \geq 2$ für große k sein muss und somit auch $\prod_{\substack{p\in\mathbb{P}\\p\leq k}} p > 2^k$. Eine genaue Analyse zeigt, dass dies bereits für $k \geq 29$ gilt. ∎

Theorem 2.14 *Seien $k, x \in \mathbb{N}$ mit $k \geq 29$ und mit $x \leq 2^k$, dann besitzt x maximal $\pi(k)$ verschiedene Primfaktoren.*

Beweis: Wir führen den Beweis durch Widerspruch. Dazu seien $k, x \in \mathbb{N}$ mit $x \leq 2^k$ und mit $k \geq 29$. Wir nehmen an, dass x mehr als $\pi(k)$ verschiedene Primteiler besitzt. Seien $p_1, \ldots, p_\ell$ die verschiedenen Primteiler von x mit Vielfachheit $m_1, \ldots, m_\ell$. Nach Annahme ist $\ell > \pi(k)$. Dann gilt mit Hilfe des vorherigen Satzes:

$$x = \prod_{i=1}^{\ell} p_i^{m_i} \geq \prod_{i=1}^{\ell} p_i \geq \prod_{\substack{p \in \mathbb{P} \\ p \leq \ell}} p \geq 2^\ell > 2^{\pi(k)}.$$

Dies liefert den gewünschten Widerspruch. ■

Theorem 2.15 *Sei $s, t \in \{0,1\}^*$ mit $|s| = m$, $|t| = n$, mit $n * m \geq 29$. Sei $P \in \mathbb{N}$ und $p \in [1 : P] \cap \mathbb{P}$ eine zufällig gewählte Primzahl, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers bei Karp-Rabin von s in t*

$$\mathrm{Ws}(\textit{irrtümlicher Treffer}) \leq \frac{\pi(n * m)}{\pi(P)}.$$

Damit folgt für die Wahl von P unmittelbar, dass $P > n * m$ sein sollte.

Beweis: Sei $I = \{i \in [0 : n - m] \mid s \neq t_{i,m}\}$ die Menge der Positionen, an denen s in t nicht vorkommt. Dann gilt für alle $i \in I$, dass $V(s) \neq V(t_{i,m})$.

Offensichtlich gilt

$$\prod_{i \in I} |V(s) - V(t_{i,m})| \;\leq\; \prod_{i \in I} 2^m \;\leq\; 2^{nm}.$$

Mit Satz 2.14 folgt dann, dass $\prod_{i \in I} |V(s) - V(t_{i,m})|$ maximal $\pi(nm)$ verschiedene Primteiler besitzt.

Sei s ein irrtümlicher Treffer an Position j, d.h. es gilt $V(s) \equiv V(t_{j,m}) \bmod p$ und $V(s) \neq V(t_j, m)$. Dann ist $j \in I$.

Da $V(s) \equiv V(t_{j,m}) \bmod p$, folgt, dass p ein Teiler von $|V(s) - V(t_{i,m})|$ ist. Dann muss p aber auch $\prod_{i \in I} |V(s) - V(t_{i,m})|$ teilen.

Für p gibt es $\pi(P)$ mögliche Kandidaten, aber es kann nur einer der $\pi(n * m)$ Primfaktoren ausgewählt worden sein. Damit gilt

$$\text{Ws(irrtümlicher Treffer)} \leq \frac{\pi(n * m)}{\pi(P)}.$$

■

Zum Abschluss zwei Lemmas, die verschiedene Wahlen von P begründen.

Lemma 2.16 *Wählt man $P = m * n^2$, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers begrenzt durch $3/n$.*

Beweis: Nach dem vorherigen Satz gilt, dass die Wahrscheinlichkeit eines irrtümlicher Treffers durch $\frac{\pi(n*m)}{\pi(P)}$ beschränkt ist. Damit folgt mit dem Satz 2.12:

$$\frac{\pi(n * m)}{\pi(n^2 m)} \leq 1.26 \cdot \frac{nm \ln(n^2 m)}{n^2 m \ln(nm)} \leq \frac{1.26}{n} \cdot \frac{2 \ln(nm)}{\ln(nm)} \leq \frac{2.52}{n}.$$

■

Ein Beispiel hierfür mit $n = 4000$ und $m = 250$. Dann ist $P = n * m^2 < 2^{32}$. Somit erhält man mit einem Fingerabdruck, der sich mit 4 Byte darstellen lässt, eine Fehlerwahrscheinlichkeit von unter 0,1%.

Lemma 2.17 *Wählt man $P = nm^2$, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers begrenzt durch $3/m$.*

Beweis: Nach dem vorherigen Satz gilt, dass die Wahrscheinlichkeit eines irrtümlicher Treffers durch $\frac{\pi(n*m)}{\pi(P)}$ beschränkt ist. Damit folgt mit dem Satz 2.12:

$$\frac{\pi(n * m)}{\pi(nm^2)} \leq 1.26 \cdot \frac{nm \ln(nm^2)}{nm^2 \ln(nm)} \leq \frac{1.26}{m} \cdot \frac{2 \ln(nm)}{\ln(nm)} \leq \frac{2.52}{m}.$$

■

Nun ist die Wahrscheinlichkeit eines irrtümlichen Treffers $O(\frac{1}{m})$. Da für jeden irrtümlichen Treffer ein Vergleich von s mit dem entsprechenden Teilwort von t ausgeführt werden muss (mit m Vergleichen), ist nun die erwartete Anzahl von Vergleichen bei einem irrtümlichen Treffer $O(m\frac{1}{m}) = O(1)$. Somit ist die erwartete Anzahl von Zeichenvergleichen bei dieser Variante wieder linear, d.h. $O(n)$.

Zum Schluss wollen wir noch anmerken, dass wir ja durchaus verschiedene Fingerabdrücke testen können. Nur wenn alle einen Treffer melden, kann es sich dann um einen Treffer handeln. Somit lässt sich die Fehlerwahrscheinlichkeit noch weiter senken bzw. bei gleicher Fehlerwahrscheinlichkeit kann man den kürzeren Fingerabdruck verwenden.

2.6 Suffix-Tries und Suffix-Bäume

In diesem Abschnitt wollen wir noch schnellere Verfahren zum Suchen in Texten vorstellen. Hier wollen wir den zu durchsuchenden Text vorverarbeiten (möglichst in Zeit und Platz $O(|t|)$), um dann sehr schnell ein Suchwort s (möglichst in Zeit $O(|s|)$) suchen zu können.

2.6.1 Suffix-Tries

Ein *Trie* ist eine Datenstruktur, in welcher eine Menge von Wörtern abgespeichert werden kann. Tries haben wir eigentlich schon kennen gelernt. Der Suchwort-Baum des Aho-Corasick-Algorithmus ist nichts anderes als ein Trie für die Menge der Suchwörter.

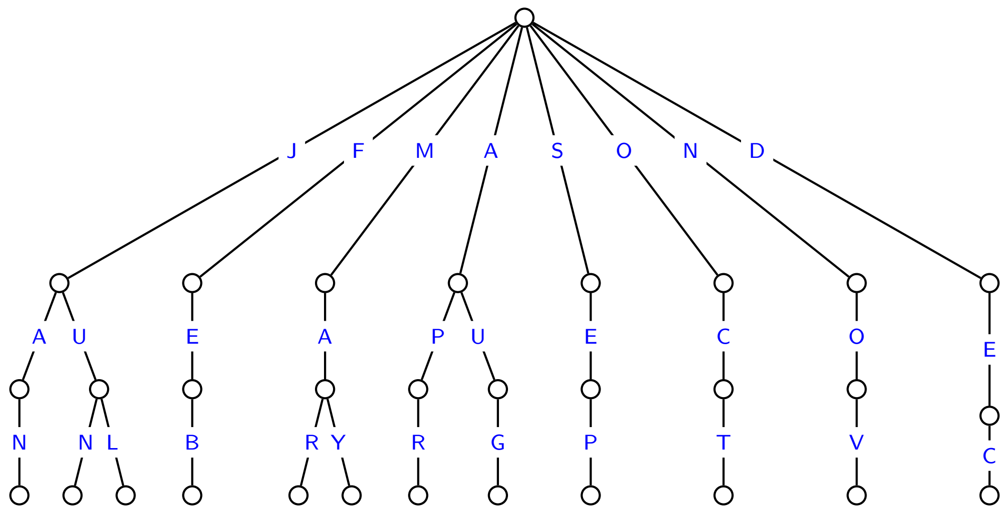


Abbildung 2.36: Beispiel: Trie für die dreibuchstabigen Monatsnamensabk.

Definition 2.18 *Ein* Trie *für ein gegebenes $M \subseteq \Sigma^*$ ist ein Baum mit Kantenmarkierungen, wobei alle Kantenbeschriftungen von Pfaden von der Wurzel zu einem Blatt gerade die Menge M ergeben und es von jedem Knoten aus für ein Zeichen $a \in \Sigma$ maximal eine ausgehende Kante mit diesem Zeichen als Markierung geben darf.*

Vorteil: Schnelles Suchen mit der Komplexität $O(m)$.

Definition 2.19 *Ein* Suffix-Trie *für ein Wort $t \in \Sigma^*$ ist ein Trie für alle Suffixe von t, d.h. $M = \{t_i \cdots t_n \mid i \in [1 : n+1]\}$ mit $t = t_1 \cdots t_n$ $(t_{n+1} \cdots t_n = \varepsilon)$.*

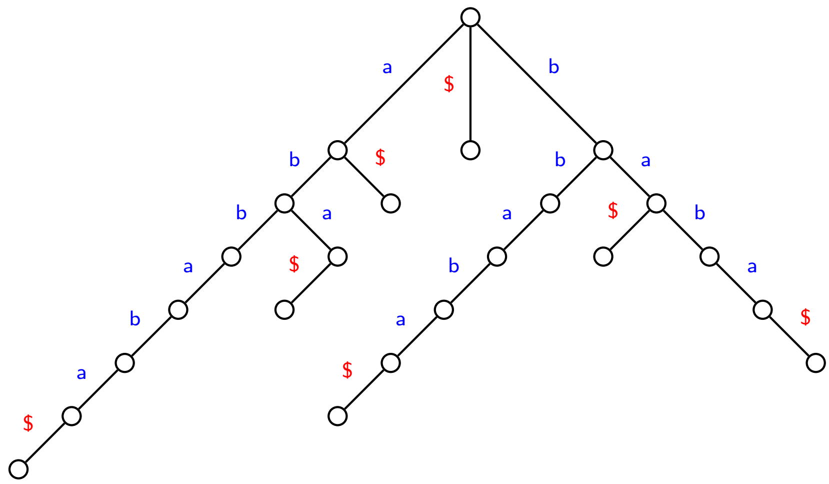


Abbildung 2.37: Beispiel: $t = abbaba\$$

Damit kein Suffix von t ein Präfix eines anderen Suffixes ist und somit jedes Suffix von t in einem Blatt endet, hängt man an das Ende von t oft ein Sonderzeichen \$ (\$ $\notin \Sigma$) an.

Der Suffix-Trie kann also so aufgebaut werden, dass nach und nach die einzelnen Suffixe von t in den Trie eingefügt werden, beginnend mit dem längsten Suffix. Der folgende einfache Algorithmus beschreibt genau dieses Verfahren.

Im Folgenden zählen wir als Elementaroperationen die Anzahl der besuchten und modifizierten Knoten der zugrunde liegenden Bäume.

Laufzeit: Da das Einfügen des Suffixes $t_i \cdots t_n$ genau $O(|t_i \cdots t_n|) = O(n - i + 1)$ Operationen benötigt, folgt für die Laufzeit:

$$\sum_{i=1}^{n}(n-i+1) \;=\; \sum_{i=1}^{n} i \;=\; O(n^2).$$

```
BuildSuffixTrie (char t[], int n)
{
    tree T;
    for (i = 0; i ≤ n; i++)
        insert(T, t_i ··· t_n)
}
```

Abbildung 2.38: Algorithmus: Simple Methode zum Aufbau eines Suffix-Tries

Fakt: w ist genau dann ein Teilwort von t, wenn w ein Präfix eines Suffixes von t ist.

Somit können wir in Suffix-Tries sehr schnell nach einem Teilwort w von t suchen. Wir laufen im Suffix-Trie die Buchstabenfolge $(w_1, \ldots, w_{|w|})$ ab. Sobald wir auf einen Knoten treffen, von dem wir nicht mehr weiter können, wissen wir, dass w dann nicht in t enthalten sein kann. Andernfalls ist w ein Teilwort von t.

2.6.2 Ukkonens Online-Algorithmus für Suffix-Tries

Sei $t \in \Sigma^n$ und sei T^i der Suffix-Trie für $t_1..t_i$. Ziel ist es nun, den Suffix-Trie T^i aus aus dem Trie T^{i-1} zu konstruieren. T^0 ist einfach zu konstruieren, da $t_1 t_0 = \varepsilon$ ist und somit T^0 der Baum ist, der aus nur einem Knoten (der Wurzel) besteht.

Auf der nächsten Seite ist der sukzessive Aufbau eines Suffix-Tries für *ababba* aus dem Suffix-Trie für *ababb* ausführlich beschrieben. Dabei wird auch von so genannten Suffix-Links Gebrauch gemacht, die wir gleich noch formal einführen werden.

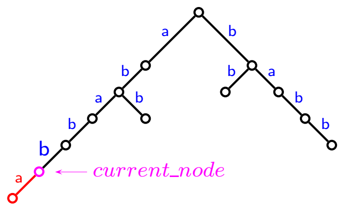
Das darauf folgende Bild zeigt den kompletten Suffix-Trie T^7 mit allen Suffix-Links (auch die Suffix-Links zur Konstruktion des T^6 sind eingezeichnet).

Sei $w \in \Sigma^*$ ein Teilwort von t. Den Knoten, der über w in einem Suffix-Trie erreichbar ist, bezeichnen wir mit $\overline{w}$. Sei $\overline{aw}$ der Knoten, der über aw von der Wurzel aus erreichbar ist, wobei $a \in \Sigma$ und $w \in \Sigma^*$. Dann zeigt der *Suffix-Link* von $\overline{aw}$ auf $\overline{w}$.

Damit auch die Wurzel einen Suffix-Link besitzt, ergänzen wir den Suffix-Trie um eine virtuelle Wurzel $\perp$ und setzen $\mathit{suffix\_link}(\mathit{root}) = \perp$. Diese virtuelle Wurzel $\perp$ besitzt für jedes Zeichen $a \in \Sigma$ eine Kante zur eigentlichen Wurzel. Dies wird in Zukunft aus algorithmischer Sicht hilfreich sein, da für diesen Knoten dann keine Suffix-Links benötigt werden.

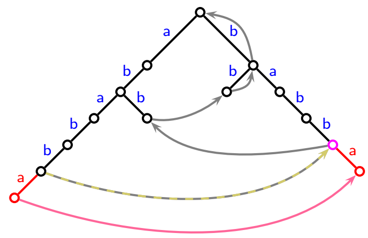
Beispiel: Aufbau eines Suffix-Tries

1. Schritt



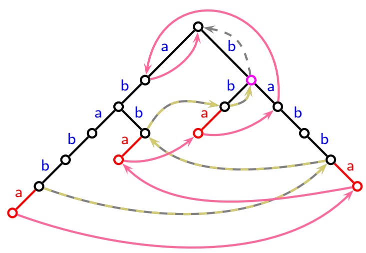
Der Trie für $t = ababb$ ist bereits konstruiert. An den Knoten, der am Ende des Pfades steht, dessen Labels das bisher längste Suffix ergibt, muss nun zuerst das neue Zeichen a angehängt werden. Anschließend muss an jedes kürzere Präfix ein a angehängt werden.

2. Schritt



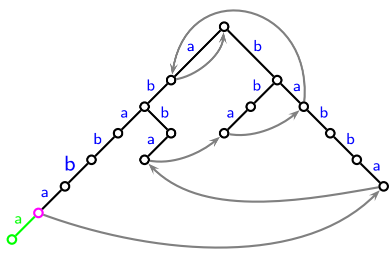
Um nun den Knoten schnell zu finden, der das nächst kürzere Suffix repräsentiert, sind im Trie so genannte *Suffix-Links* vorhanden. Diese zeigen immer auf den Knoten, der das nächst kürzere Suffix darstellt. Folge nun dem Suffix-Link des aktuellen Knotens und füge an dessen Endknoten wiederum die neue a-Kante ein und aktualisiere den Suffix-Link.

3. Schritt



Der zuvor beschriebene Vorgang wird nun so lange wiederholt, bis man auf einen Knoten trifft, der bereits eine a-Kante besitzt. Nun muss natürlich kein neuer Knoten mehr eingefügt werden, da dieser ja bereits existiert. Allerdings müssen noch die restlichen Suffix-Links aktualisiert werden.

4. Schritt



Nachdem T^6 vollständig aufgebaut ist, beginnt das ganze Spiel von vorne für den nächsten Buchstaben, im Beispiel a.

Abbildung 2.39: Beispiel: $t = ababbaa$; Konstruktion von T^6 aus T^5

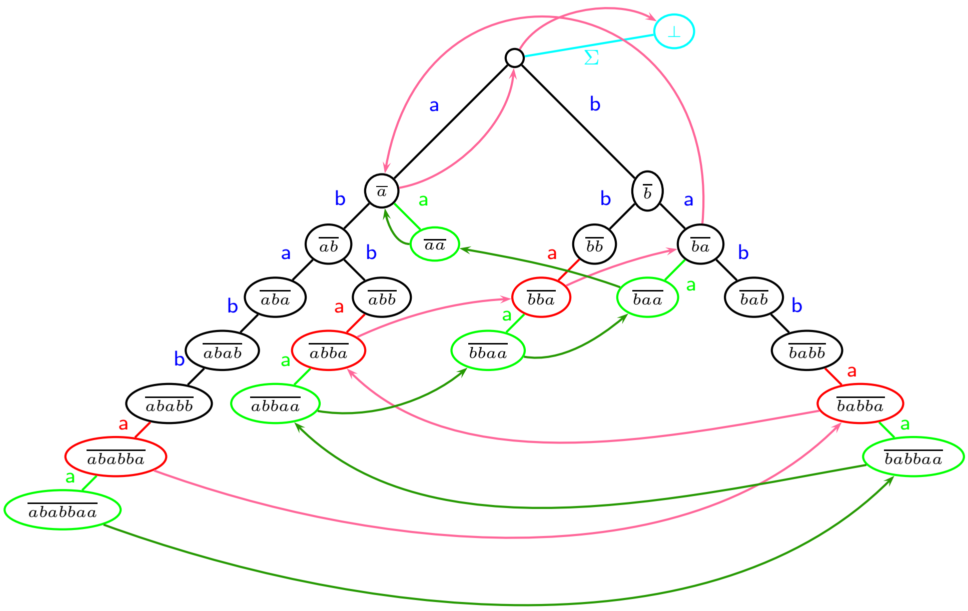


Abbildung 2.40: Beispiel: Suffix-Trie für *ababbaa* mit Suffix-Links

2.6.3 Laufzeitanalyse für die Konstruktion von T^n

Naive Methode: Bei der naiven Methode muss zur Verlängerung eines Suffixes der ganze Suffix durchlaufen werden. Daher sind zum Anhängen von t_i an $t_j \cdots t_{i-1}$ wiederum $O(i-j+1)$ Operationen erforderlich,

$$\sum_{i=1}^{n} \underbrace{\sum_{j=1}^{i} O(i-j+1)}_{\text{Zeit für } T^{i-1} \to T^{i}} = O\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right) = O\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right) = O\left(\sum_{i=1}^{n} i^2\right) = O(n^3).$$

Laufzeit mit Suffix-Links: Durch die Suffix-Links benötigt das Verlängern des Suffixes $t_j \cdots t_{i-1}$ um t_i nur noch $O(1)$ Operationen.

$$\sum_{i=1}^{n}\sum_{j=1}^{i} O(1) = \sum_{i=1}^{n} O(i) = O(n^2)$$

2.6.4 Wie groß kann ein Suffix-Trie werden?

Es stellt sich die Frage, ob wir unser Ziel aus der Einleitung zu diesem Abschnitt bereits erreicht haben. Betrachten wir das folgende Beispiel für $t = a^n b^n$:

```
BuildSuffixTrie (char t[], int n)
{
    T = ({root, ⊥}, {⊥ -x-> root : x ∈ Σ});
    suffix_link(root) = ⊥;
    longest_suffix = prev_node = root;
    for (i = 1; i ≤ n; i++)
    {
        curr_node = longest_suffix;
        while (curr_node -t_i-> some_node does not exist)
        {
            Add curr_node -t_i-> new_node to T;
            if (curr_node == longest_suffix)
                longest_suffix = new_node;
            else
                suffix_link(prev_node) = new_node;
            prev_node = new_node;
            curr_node = suffix_link(curr_node);
        }
        suffix_link(prev_node) = some_node;
    }
}
```

Abbildung 2.41: Algorithmus: Konstruktion von Suffix-Tries mit Hilfe von Suffix-Links

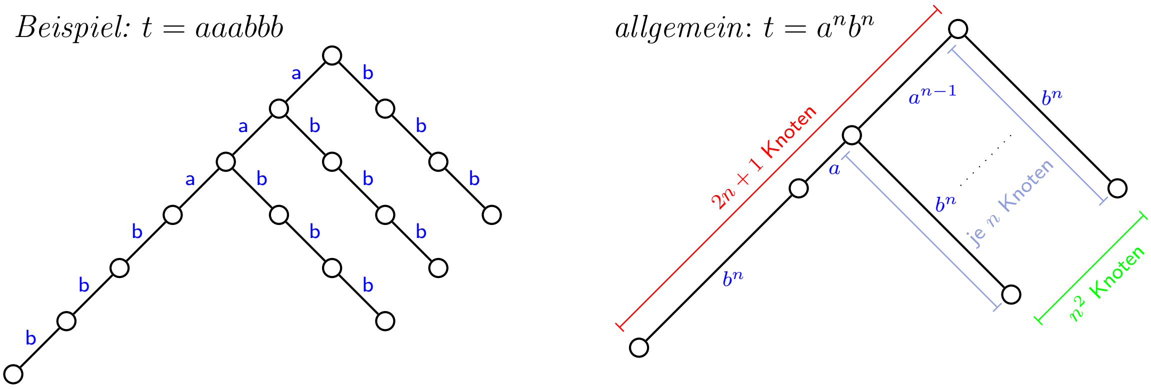


Abbildung 2.42: Beispiel: Potentielle Größe von Suffix-Tries (für $a^n b^n$)

Der Suffix-Trie für ein Wort t der Form $t = a^n b^n$ hat damit insgesamt

$$(2n + 1) + n^2 = (n + 1)^2 = O(n^2)$$

viele Knoten, was optimiert werden muss.

Die Idee hierfür ist, möglichst viele Kanten sinnvoll zusammenzufassen, um den Trie zu kompaktifizieren. So können alle Pfade, die bis auf den Endknoten nur aus Knoten mit jeweils einem Kind bestehen, zu einer Kante zusammengefasst werden. Diese kompaktifizierten Tries werden *Patricia-Tries* genannt (für *Practical Algorithm To Retrieve Information Coded In Alphanumeric*), siehe auch folgendes Beispiel.

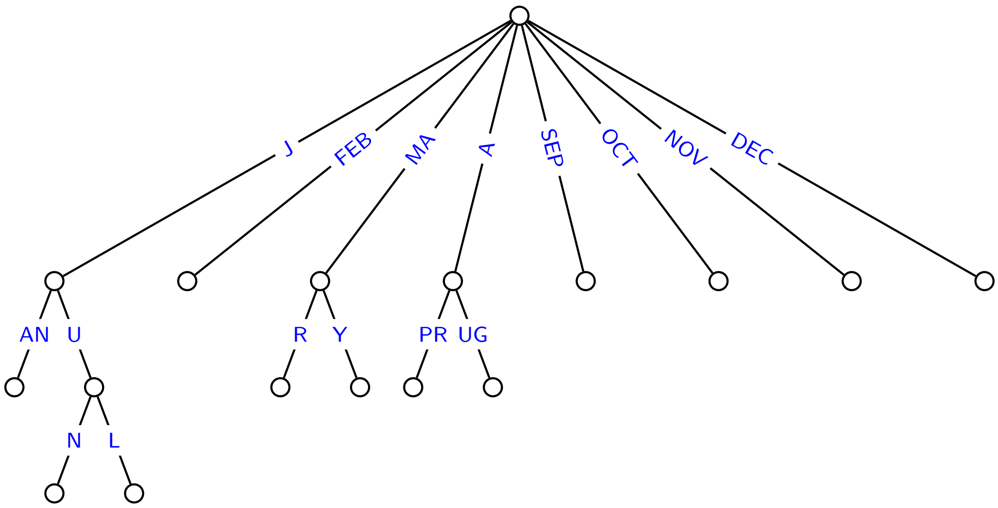


Abbildung 2.43: Beispiel: Patricia-Trie für die dreibuchstabigen Monatsnamensabk.

2.6.5 Suffix-Bäume

So kompaktifizierte Suffix-Tries werden *Suffix-Bäume* (engl. *suffix-trees*) genannt. Durch das Zusammenfassen mehrerer Kanten wurde zwar die Anzahl der Knoten reduziert, dafür sind aber die Kantenlabels länger geworden. Deswegen werden als Labels der zusammengefassten Kanten nicht die entsprechenden Teilwörter verwendet, sondern die Position, an welcher das Teilwort im Gesamtwort auftritt. Für das Teilwort $t_i \cdots t_j$ wird die Referenz (i, j) verwendet.

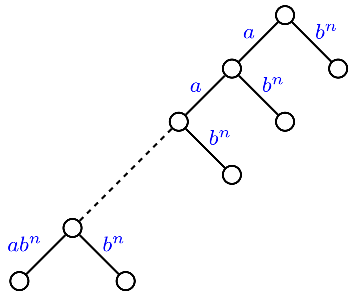


Abbildung 2.44: Beispiel: Kompaktifizierter Trie für $t = a^n b^n$

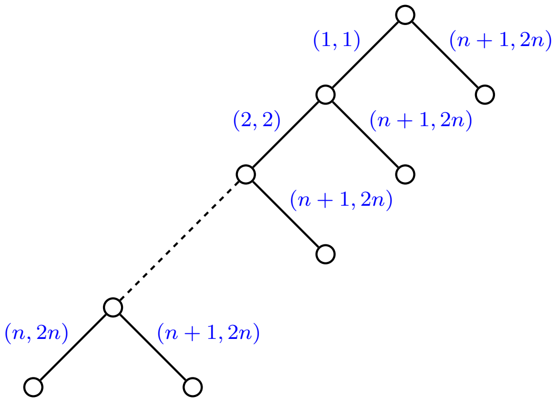


Abbildung 2.45: Beispiel: Echter Suffix-Baum für$t = a^n b^n$

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 $\overline{s_i} = \overline{\varepsilon} = \mathit{root}$, 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.

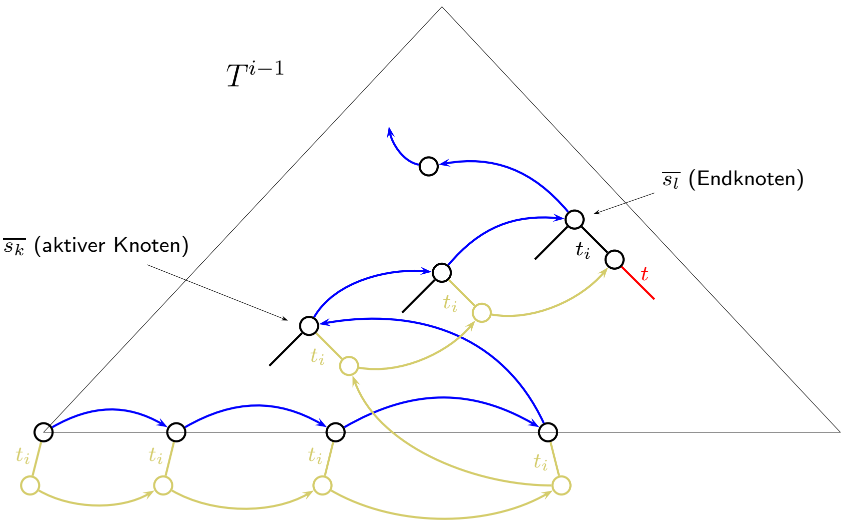


Abbildung 2.46: Skizze: Konstruktion eines Suffix-Tries

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

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.

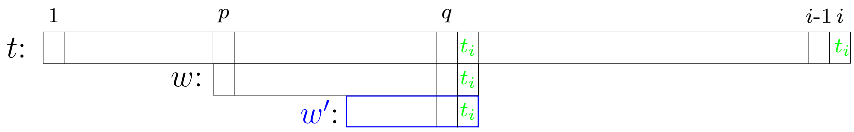


Abbildung 2.47: Skizze: Endknoten beendet Erweiterung von T^{i-1}

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

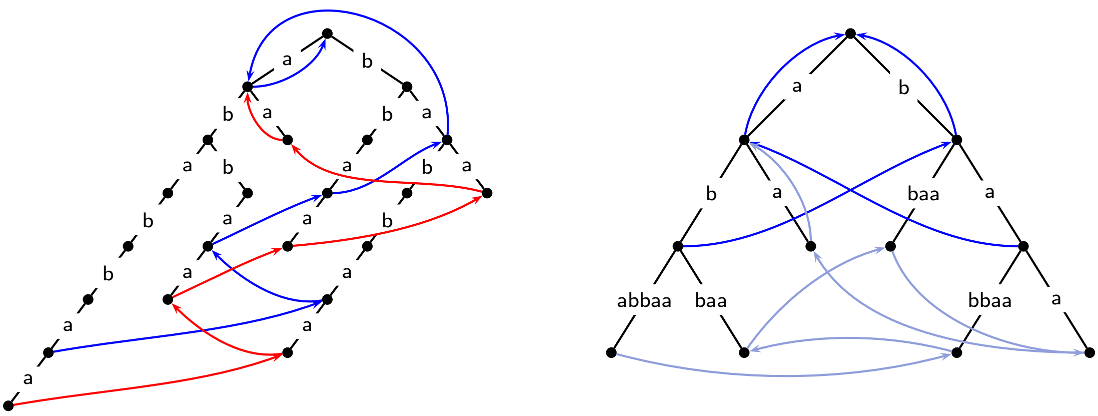


Abbildung 2.48: Beispiel: Suffix-Trie und Suffix-Baum für $t = ababbaa$

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

<u>BuildSuffixTree (char t[], int n)</u>

{

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

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

  $s = root$;

  $k = 1$;

  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.

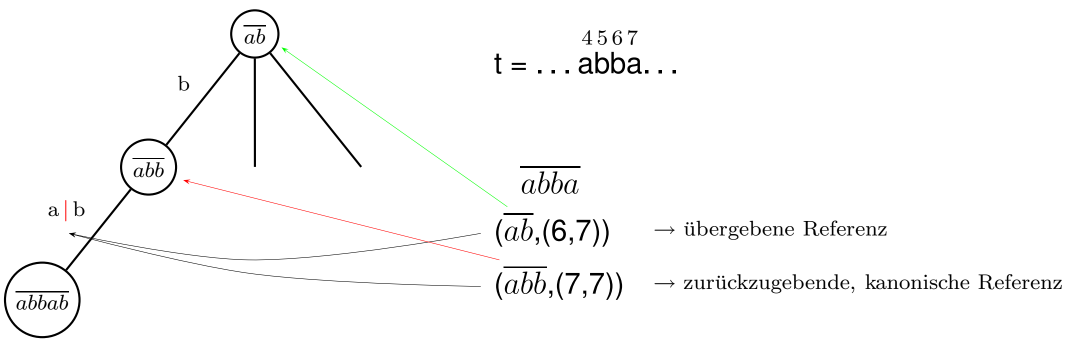


Abbildung 2.51: Beispiel: Erstellung kanonischer Referenzen mittels Canonize

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

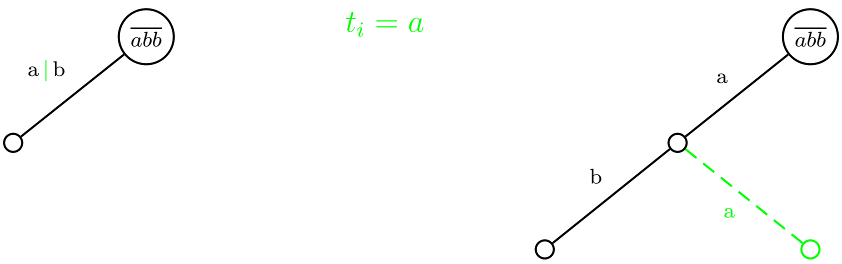


Abbildung 2.54: Beispiel: Vorgehensweise von TestAndSplit

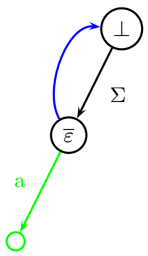
gen Kante eingefügt wurde, von TestAndSplit zurückgegeben, damit die Prozedur Update daran die neue Kante mit Label t_i anhängen kann.

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:

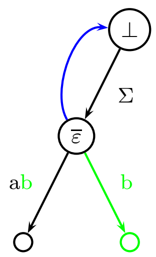
	1	2	3	4	5	6	7
t=	a	b	a	b	b	a	a

Buchstabe a wird in den noch leeren Tree eingefügt:



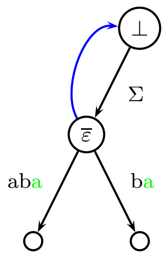
($\bar{\varepsilon}$,(1,0)) $i = 1; t_i = a$
 $\rightsquigarrow$ TestAndSplit = (FALSE,$\bar{\varepsilon}$)
($\bar{\varepsilon}$,(1,0))
 $\downarrow$ Suffix-Link
($\perp$,(1,0))
 $\downarrow$ Canonize
($\perp$,(1,0))

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



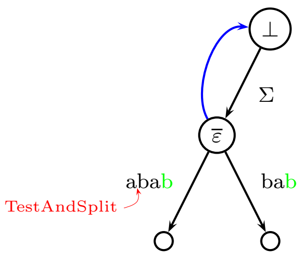
($\perp$,(1,1)) $i = 2; t_i = b$
 $\downarrow$ Canonize
($\bar{\varepsilon}$,(2,1))
 $\rightsquigarrow$ TestAndSplit = (FALSE,$\bar{\varepsilon}$)
($\bar{\varepsilon}$,(2,1))
 $\downarrow$ Suffix-Link
($\perp$,(2,1))
 $\downarrow$ Canonize
($\perp$,(2,1))

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



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

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



($\overline{\varepsilon}$,(3,3)) $i = 4$; $t_i = b$

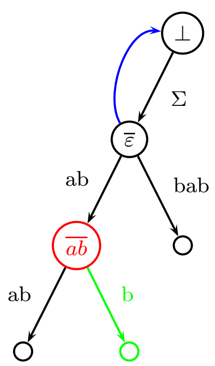
$\downarrow$ Canonize

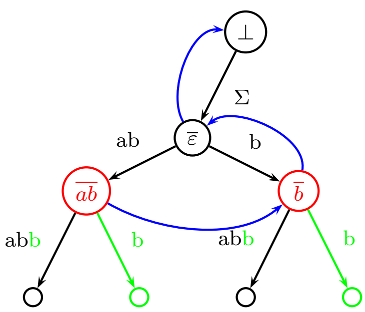
($\overline{\varepsilon}$,(3,3))

$\rightsquigarrow$ 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))

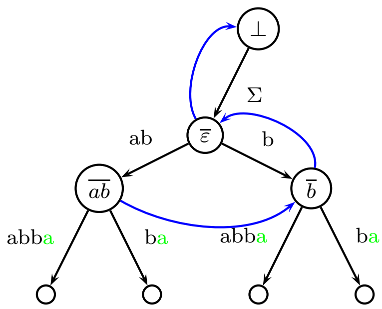
$\downarrow$ Canonize

($\overline{\varepsilon}$,(5,4))

$\rightsquigarrow$ TestAndSplit = (TRUE,$\overline{\varepsilon}$)

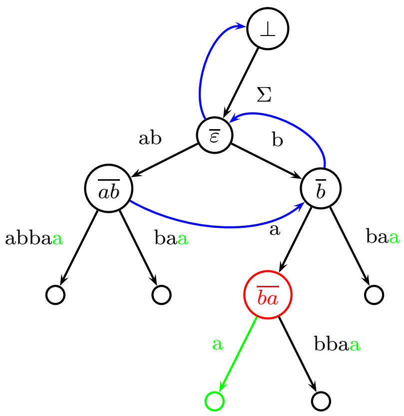
($\overline{\varepsilon}$,(5,4))

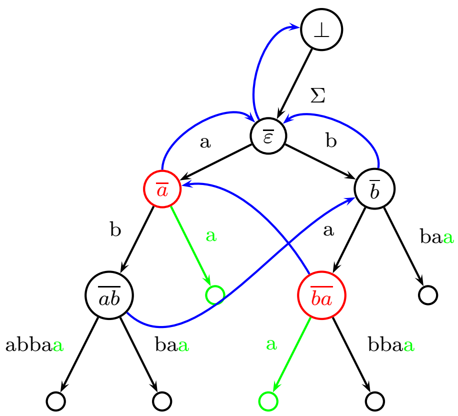
Buchstabe a wird in den $\hat{T}^5$ eingefügt:



($\overline{\varepsilon}$,(5,5)) $i = 6$; $t_i = a$
↓ Canonize
($\overline{b}$,(6,5))
$\wr$ TestAndSplit = (TRUE,$\overline{b}$)
($\overline{b}$,(6,5))

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





($\overline{b}$,(6,6)) $i = 7$; $t_i = a$
↓ Canonize
($\overline{b}$,(6,6))
$\wr$ TestAndSplit = (FALSE,$\overline{ba}$)
($\overline{b}$,(6,6))
↓ Suffix-Link
($\overline{\varepsilon}$,(6,6))
↓ Canonize
($\overline{\varepsilon}$,(6,6))
$\wr$ TestAndSplit = (FALSE,$\overline{a}$)
($\overline{\varepsilon}$,(6,6))
↓ Suffix-Link
($\perp$,(6,6))
↓ Canonize
($\overline{\varepsilon}$,(7,6))
$\wr$ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(7,6))

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

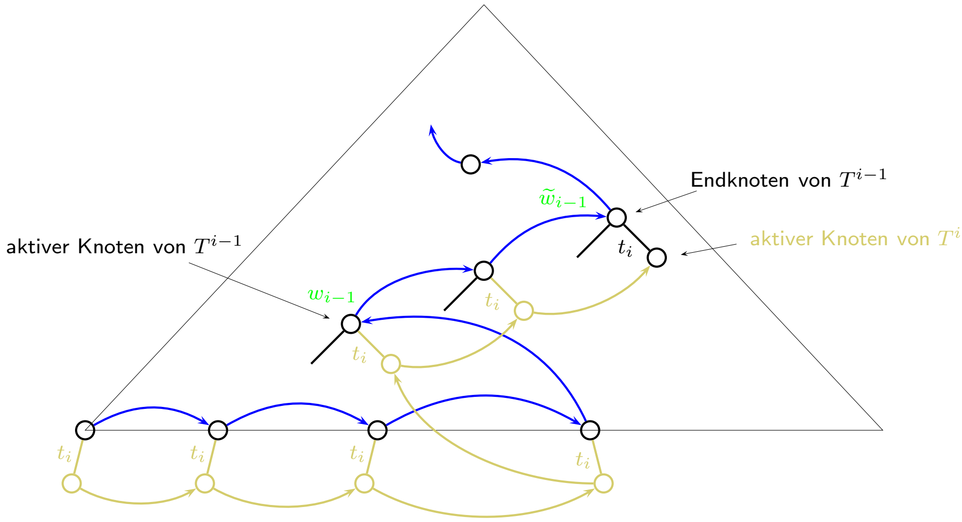


Abbildung 2.55: Skizze: Erweiterung von T^{i-1} auf T^i

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

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

Wie viele Knoten werden für alle Abläufe der *while-Schleife* in Update getestet?

$$\begin{aligned}
&\sum_{i=1}^{n}(|w_{i-1}| - |w_i| + 2) \\
&= |w_0| - |w_1| + |w_1| - |w_2| + |w_2| - |w_3| + \cdots + |w_{n-1}| - |w_n| + \sum_{i=1}^{n} 2 \\
&= |w_0| - |w_n| + \sum_{i=1}^{n} 2 \\
&= 2n - |w_n| \\
&\leq 2n
\end{aligned}$$

Die Laufzeit für *while-Schleife* bei allen Updates entspricht $O(n)$. Die Laufzeit für *nicht-while-Schleife* ist ebenfalls insgesamt $O(n)$. Somit ist die Gesamtlaufzeit $O(n)$.

Theorem 2.23 *Ein Suffix-Baum für $t \in \Sigma^n$ lässt sich in Zeit $O(n)$ und Platz $O(n)$ mit Hilfe des Algorithmus von Ukkonen konstruieren.*

2.6.8 Problem: Verwaltung der Kinder eines Knotens

Zum Schluss wollen wir uns noch kurz mit der Problematik des Abspeicherns der Verweise auf die Kinder eines Knotens in einem Suffix-Baum widmen. Ein Knoten kann entweder sehr wenige Kinder besitzen oder sehr viele, nämlich bis zu $|\Sigma|$ viele. Wir schauen uns die verschiedenen Methoden einmal an und bewerten sie nach Platz- und Zeitbedarf. Wir bewerten den Zeitbedarf danach, wie lange es dauert, bis wir für ein Zeichen aus Σ das entsprechende Kind gefunden haben. Für den Platzbedarf berechnen wir den Bedarf für den gesamten Suffix-Baum für einen Text der Länge m.

Realisierungsmöglichkeiten:

Felder Die Kinder eines Knotens lassen sich sehr einfach mit Hilfe eines Feldes der Größe $|\Sigma|$ darstellen.

- Platz: $O(m \cdot |\Sigma|)$.
  Dies folgt daraus, dass für jeden Knoten ein Feld mit Platzbedarf $O(|\Sigma|)$ benötigt wird.
- Zeit: $O(1)$.
  Übliche Realisierungen von Feldern erlauben einen Zugriff in konstanter Zeit.

Der Zugriff ist also sehr schnell, wo hingegen der Platzbedarf, insbesondere bei großen Alphabeten doch sehr groß werden kann.

Lineare Listen: Wir verwalten die Kinder eines Knotens in einer linearen Liste, diese kann entweder sortiert sein (wenn es eine Ordnung, auch eine künstliche, auf dem Alphabet gibt) oder auch nicht.

- Platz: $O(m)$.

  Für jeden Knoten ist der Platzbedarf proportional zur Anzahl seiner Kinder. Damit ist Platzbedarf insgesamt proportional zur Anzahl der Knoten des Suffix-Baumes, da jeder Knoten (mit Ausnahme der Wurzel) das Kind eines Knotens ist. Im Suffix-Baum gilt, dass jeder Knoten entweder kein oder mindestens zwei Kinder hat. Für solche Bäume ist bekannt, dass die Anzahl der inneren Knoten kleiner ist als die Anzahl der Blätter. Da ein Suffix-Baum für einen Text der Länge m maximal m Blätter besitzt, folgt daraus die Behauptung für den Platzbedarf.
- Zeit: $O(|\Sigma|)$.

  Leider ist hier die Zugriffszeit auf ein Kind sehr groß, da im schlimmsten Fall (aber auch im Mittel) die gesamte Kinderliste eines Knotens durchlaufen werden muss und diese bis zu $|\Sigma|$ Elemente umfassen kann.

Balancierte Bäume : Die Kinder lassen sich auch mit Hilfe von balancierten Suchbäumen (AVL-, Rot-Schwarz-, B-Bäume, etc.) verwalten:

- Platz: $O(n)$

  Da der Platzbedarf für einen Knoten ebenso wie bei linearen Listen proportional zur Anzahl der Kinder ist, folgt die Behauptung für den Platzbedarf unmittelbar.
- Zeit: $O(\log(|\Sigma|))$. Da die Tiefe von balancierten Suchbäumen logarithmisch in der Anzahl der abzuspeichernden Schlüssel ist, folgt die Behauptung unmittelbar.

Hashfunktion Eine weitere Möglichkeit ist die Verwaltung der Kinder aller Knoten in einem einzigen großen Feld der Größe $O(m)$. Um nun für ein Knoten auf ein spezielles Kind zuzugreifen wird dann eine Hashfunktion verwendet:

$$h : V \times \Sigma \to \mathbb{N} : (v, a) \mapsto h(v, a)$$

Zu jedem Knoten und dem Symbol, die das Kind identifizieren, wird ein Index des globales Feldes bestimmt, an der die gewünschte Information enthalten ist.

Leider bilden Hashfunktionen ein relativ großes Universum von potentiellen Referenzen (hier Paare von Knoten und Symbolen aus Σ also $\Theta(m \cdot |\Sigma|)$) auf ein kleines Intervall ab (hier Indizes aus $[1 : \ell]$ mit $\ell = \Theta(m)$). Daher sind so genannte *Kollisionen* prinzipiell nicht auszuschließen. Ein Beispiel ist das so genannte *Geburtstagsparadoxon*. Ordnet man jeder Person in einem Raum eine Zahl aus dem Intervall [1 : 366] zu (nämlich ihren Geburtstag), dann ist ab 23 Personen die Wahrscheinlichkeit größer als 50%, dass zwei Personen denselben Werrt erhalten. Also muss man beim Hashing mit diesen Kollisionen leben und diese geeignet auflösen. Für die Details wird auf andere Vorlesungen (wie etwa Informatik IV) verwiesen.

Um solche Kollisionen überhaupt festzustellen, enthält jeder Feldeintrag i neben den normalen Informationen noch die Informationen, wessen Kind er ist und über welches Symbol er von seinem Elter erreichbar ist. Somit lassen sich Kollisionen leicht feststellen und die üblichen Operationen zur Kollisionsauflösung anwenden.

Abbildung 2.56: Skizze: Realisierung mittels eines Feldes und Hashing

- Platz: $O(m)$

  Das folgt unmittelbar aus der obigen Diskussion.

- Zeit: $O(1)$

  Im Wesentlichen erfolgt der Zugriff in konstanter Zeit, wenn man voraussetzt, dass sich die Hashfunktion einfach (d.h. in konstanter Zeit) berechnen lässt und dass sich Kollisionen effizient auflösen lassen.

Schaut man sich einen Suffix-Baum genauer an, so wird man feststellen, dass die Knoten auf niedrigem Level, d.h. nah bei der Wurzel, einen sehr großen Verzweigungsgrad haben. Dort sind also fast alle potentiellen Kinder auch wirklich vorhanden. Knoten auf recht großem Level haben hingegen relativ wenige Kinder. Aus diesem Grunde bietet sich auch eine hybride Implementierung an. Für Knoten auf niedrigem Level verwendet man für die Verwaltung der Kinder einfache Felder, während man bei Knoten auf großem Level auf eine der anderen Varianten umsteigt.

Literaturhinweise

A.1 Lehrbücher zur Vorlesung

Peter Clote, Rolf Backofen: *Introduction to Computational Biology*; John Wiley and Sons, 2000.

Richard Durbin, Sean Eddy, Anders Krogh, Graeme Mitchison: *Biological Sequence Analysis*; Cambridge University Press, 1998.

Dan Gusfield: *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*; Cambridge University Press, 1997.

David W. Mount: *Bioinformatics — Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2001.

Pavel A. Pevzner: *Computational Molecular Biology - An Algorithmic Approach*; MIT Press, 2000.

João Carlos Setubal, João Meidanis: *Introduction to Computational Molecular Biology*; PWS Publishing Company, 1997.

Michael S. Waterman: *Introduction to Computational Biology: Maps, Sequences, and Genomes*; Chapman and Hall, 1995.

A.2 Skripten anderer Universitäten

Bonnie Berger: *Introduction to Computational Molecular Biology*, Massachusetts Institute of Technology, http://theory.lcs.mit.edu/~bab/01-18.417-home.html;

Bonnie Berger, *Topics in Computational Molecular Biology*, Massachusetts Institute of Technology, Spring 2001, http://theory.lcs.mit.edu/~bab/01-18.418-home.html;

Paul Fischer: *Einführung in die Bioinformatik* Universität Dortmund, Lehrstuhl II, WS2001/2002, http://ls2-www.cs.uni-dortmund.de/lehre/winter200102/bioinf/

Richard Karp, Larry Ruzzo: *Algorithms in Molecular Biology*; CSE 590BI, University of Washington, Winter 1998. http://www.cs.washington.edu/education/courses/590bi/98wi/

Larry Ruzzo: *Computational Biology*, CSE 527, University of Washington, Fall 2001; http://www.cs.washington.edu/education/courses/527/01au/

Georg Schnittger: *Algorithmen der Bioinformatik*, Johann Wolfgang Goethe-Universität Frankfurt am Main, Theoretische Informatik, WS 2000/2001, http://www.thi.informatik.uni-frankfurt.de/BIO/skript2.ps.

Ron Shamir: *Algorithms in Molecular Biology* Tel Aviv University, http://www.math.tau.ac.il/~rshamir/algmb.html; http://www.math.tau.ac.il/~rshamir/algmb/01/algmb01.html.

Ron Shamir: *Analysis of Gene Expression Data, DNA Chips and Gene Networks*, Tel Aviv University, 2002; http://www.math.tau.ac.il/~rshamir/ge/02/ge02.html;

Martin Tompa: *Computational Biology*, CSE 527, University of Washington, Winter 2000. http://www.cs.washington.edu/education/courses/527/00wi/

A.3 Lehrbücher zu angrenzenden Themen

Teresa K. Attwood, David J. Parry-Smith; *Introduction to Bioinformatics*; Prentice Hall, 1999.

Maxime Crochemore, Wojciech Rytter: *Text Algorithms*; Oxford University Press: New York, Oxford, 1994.

Martin C. Golumbic: *Algorithmic Graph Theory and perfect Graphs*; Academic Press, 1980.

Benjamin Lewin: *Genes*; Oxford University Press, 2000.

Milton B. Ormerod: *Struktur und Eigenschaften chemischer Verbindungen*; Verlag Chemie, 1976.

Hooman H. Rashidi, Lukas K. Bühler: *Grundriss der Bioinformatik — Anwendungen in den Biowissenschaften und der Medizin*,

Klaus Simon: *Effiziente Algorithmen für perfekte Graphen*; Teubner, 1992.

Maxine Singer, Paul Berg: *Gene und Genome*; Spektrum Akademischer Verlag, 2000.

Lubert Stryer: *Biochemie*, Spektrum Akademischer Verlag, 4. Auflage, 1996.

A.4 Originalarbeiten

Kellogg S. Booth, George S. Lueker: Testing for the Consecutive Ones property, Interval Graphs, and Graph Planarity Using PS-Tree Algorithms; *Journal of Computer and System Science*, Vol.13, 335–379, 1976.

Ting Chen, Ming-Yang Kao: On the Informational Asymmetry Between Upper and Lower Bounds for Ultrametric Evolutionary Trees, *Proceedings of the 7th Annual European Symposium on Algorithms, ESA'99*, Lecture Notes in Computer Science 1643, 248–256, Springer-Verlag, 1999.

Richard Cole: Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm; *SIAM Journal on Computing*, Vol. 23, No. 5, 1075–1091, 1994.
s.a. *Technical Report*, Department of Computer Science, Courant Institute for Mathematical Sciences, New York University, TR1990-512, June, 1990, http://csdocs.cs.nyu.edu/Dienst/UI/2.0/Describe/ncstrl.nyu_cs%2fTR1990-512

Martin Farach, Sampath Kannan, Tandy Warnow: A Robust Model for Finding Optimal Evolutionary Trees, *Algorithmica*, Vol. 13, 155–179, 1995.

Wen-Lian Hsu: PC-Trees vs. PQ-Trees; *Proceedings of the 7th Annual International Conference on Computing and Combinatorics, COCOON 2001*, Lecture Notes in Computer Science 2108, 207–217, Springer-Verlag, 2001.

Wen-Lian Hsu: A Simple Test for the Consecutive Ones Property; *Journal of Algorithms*, Vol.43, No.1, 1–16, 2002.

Haim Kaplan, Ron Shamir: Bounded Degree Interval Sandwich Problems; *Algorithmica*, Vol. 24, 96–104, 1999.

Edward M. McCreight: A Space-Economical Suffix Tree Construction Algorithm; *Journal of the ACM*, Vol. 23, 262–272,1976.

Moritz Maaß: *Suffix Trees and Their Applications*, Ausarbeitung von der Ferienakademie '99, Kurs 2, Bäume: Algorithmik und Kombinatorik, 1999. http://www14.in.tum.de/konferenzen/Ferienakademie99/

Esko Ukkonen: On-Line Construction of Suffix Tress, *Algorithmica*, Vol. 14, 149–260, 1995.

Index

F

G

H

P

Q

R