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.

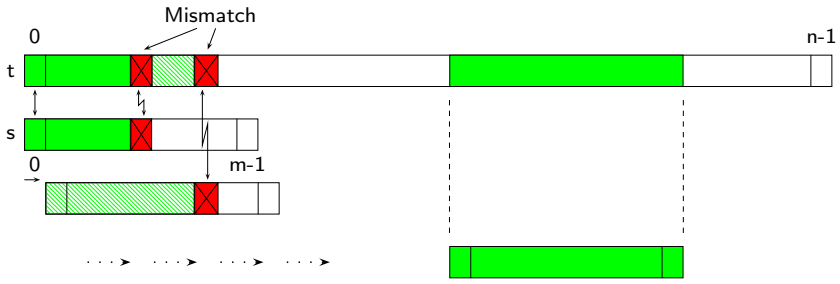


Abbildung 2.1: Skizze: Suchen mit der naiven Methode

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 $= O(n * m)$ $\leftarrow$ Das ist zu viel!

Beispiel:

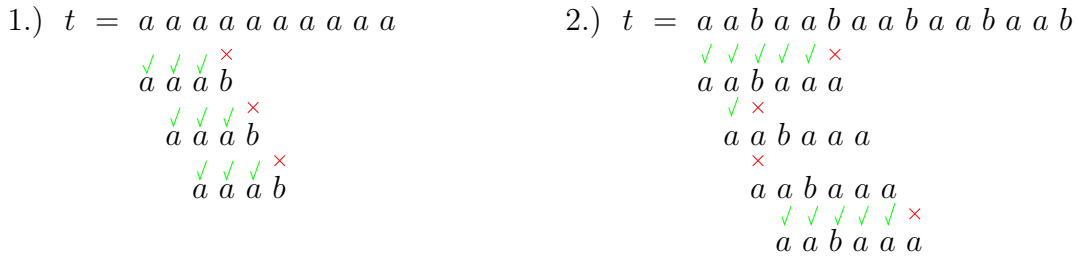


Abbildung 2.3: Beispiel: Suchen mit der naiven Methode

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).

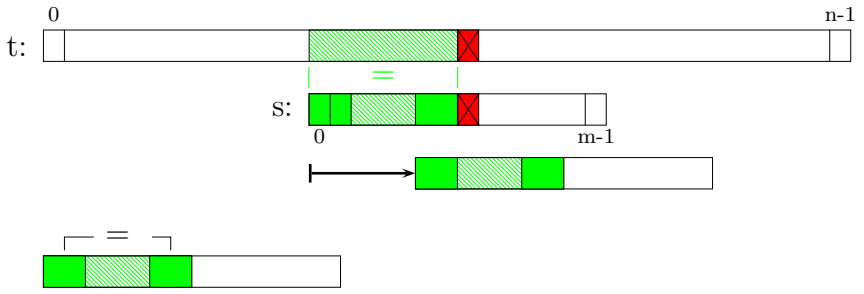


Abbildung 2.4: Skizze: Eine Idee für größere Shifts

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:

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

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. ■

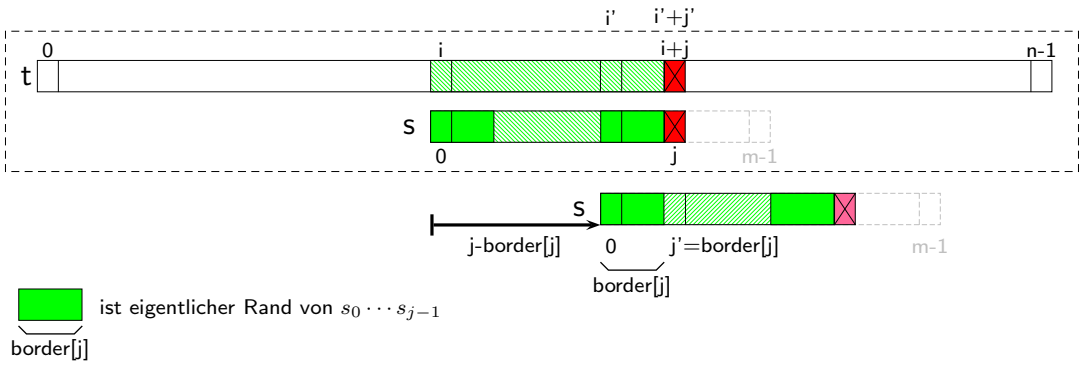


Abbildung 2.5: Skizze: Der Shift um $j - border[j]$

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 ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            j++;
            if (j == m) return TRUE;
        }
        i = i + 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: $\quad i+j$
Wert von $i'+j'$ nachher: $\underbrace{(i+j-border[j])}_{i'} + \underbrace{(max\{0, border[j]\})}_{j'}$

1.Fall: $border[j] \geq 0 \quad \Rightarrow \; i'+j' = i+j$
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

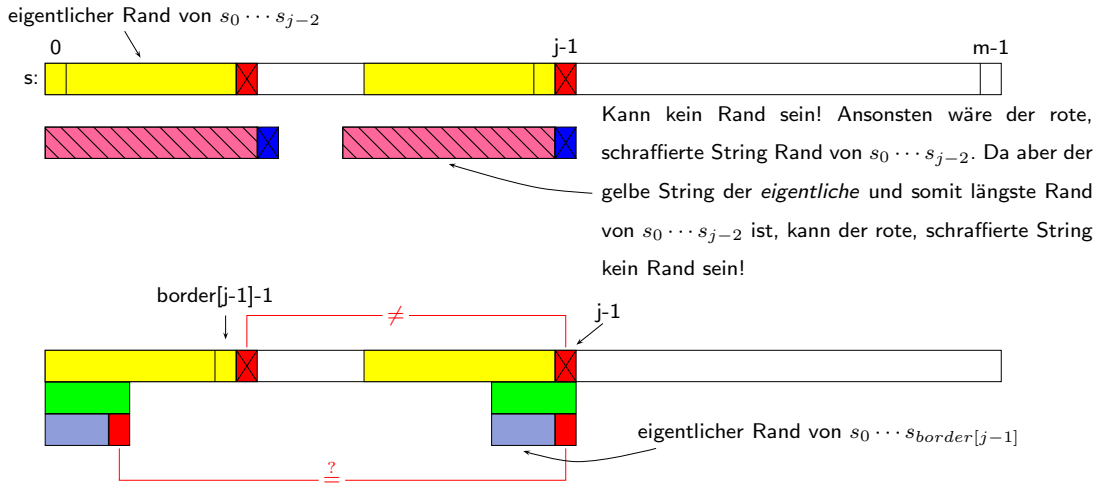


Abbildung 2.7: Skizze: Berechnung von $border[j]$

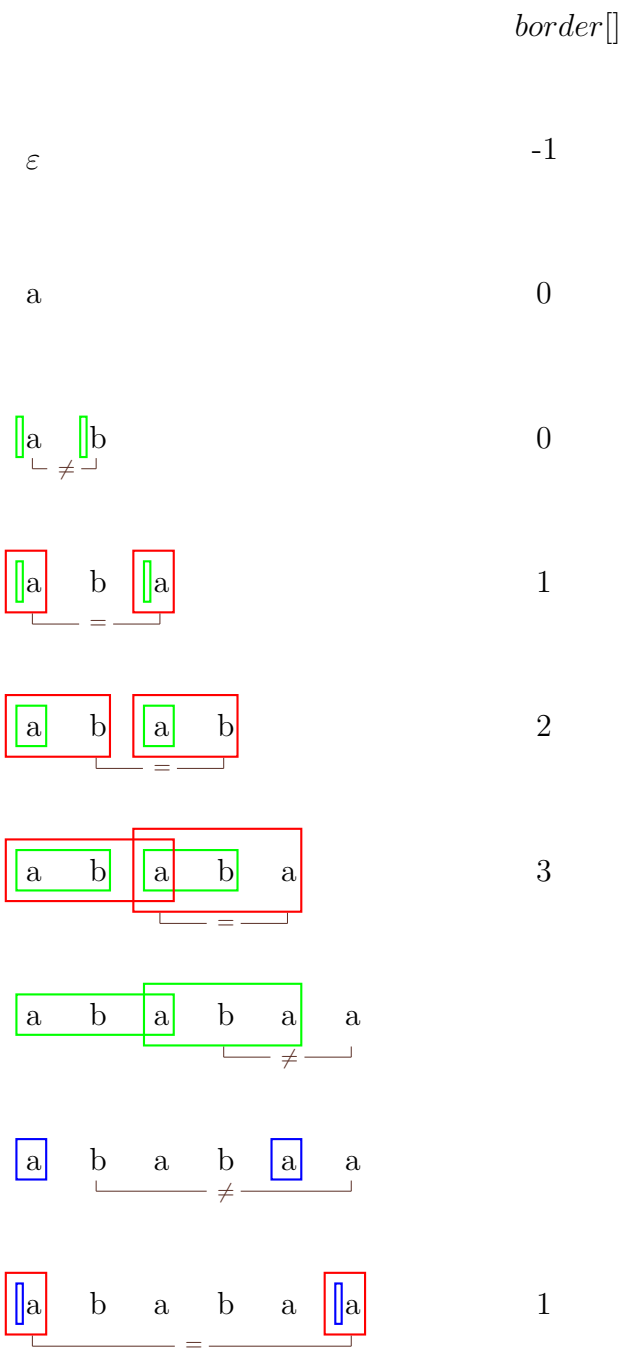
betrachteten Randes dieses Wortes. Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$. Nun testen wir, ob sich dieser Rand von $s_0 \cdots s_{j-2}$ zu einem eigentlichen Rand von $s_0 \cdots s_{j-1}$ erweitern lässt. Dies wiederholen 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.
$\overset{i \geq -1}{\Rightarrow}$ 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.