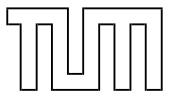
Effiziente Algorithmen 2

Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2009



Übersicht

Alphabet

Definition

Ein Alphabet ist eine endliche Menge von Symbolen.

Wir bezeichnen es meistens mit Σ.

Beispiel

$\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{A, C, G, T\}$

Wörter

Definition

Wörter über Σ sind endliche Folgen von Symbolen aus Σ.

Wir notieren Wörter meist in der Form
$w = w_0 \cdots w_{n-1}$ oder $w = w_1 \cdots w_n$.

Beispiel

$\Sigma = \{a, b\}$, dann ist $w = abba$ ein Wort über Σ.

Wortlänge, Wortmengen

Definition

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.

Definition

Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet.

Die Menge aller Wörter der Länge größer gleich 1 über Σ wird mit $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ bezeichnet.

Die Menge aller Wörter über Σ der Länge k wird mit $\Sigma^k \subseteq \Sigma^*$ bezeichnet.

Präfix, Suffix, Teilwort

Definition

Im Folgenden bezeichne $[a : b] = \{n \in \mathbb{Z} \mid a \leq n \wedge n \leq b\}$ für $a, b \in \mathbb{Z}$.
Sei $w = w_1 \cdots w_n$ ein Wort der Länge n über Σ, dann heißt

- w' Präfix von w, wenn $w' = w_1 \cdots w_\ell$ mit $\ell \in [0 : n]$
- w' Suffix von w, wenn $w' = w_\ell \cdots w_n$ mit $\ell \in [1 : n+1]$
- w' Teilwort von w, wenn $w' = w_i \cdots w_j$ mit $i, j \in [1 : n]$

(Für $w' = w_i \cdots w_j$ mit $i > j$ soll gelten $w' = \epsilon$.)

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

Textsuche

Problem:

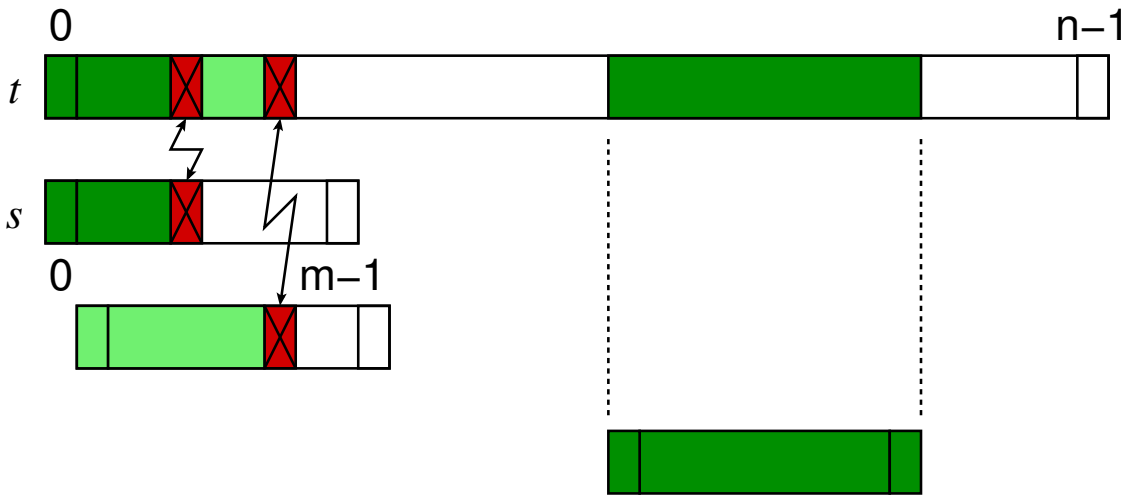
Gegeben: Text $t \in \Sigma^*$; $|t| = n$;
Suchwort $s \in \Sigma^*$; $|s| = m \leq n$

Gesucht: $\exists\ i \in [0 : n - m]$ mit $t_i \cdots t_{i+m-1} = s$?
(bzw. alle solchen Positionen i)

Naiver Algorithmus

- 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.
- Vorgang wird wiederholt, bis s in t gefunden wird oder bis klar ist, dass s in t nicht enthalten ist.

Naiver Algorithmus



Naiver Algorithmus: 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				

Naiver Algorithmus: Beispiel 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					

Naiver Algorithmus: Implementation

Algorithmus 1 : bool Naiv (char $t[]$, int n, char $s[]$, int m)

```
int i := 0, j := 0;
while (i ≤ n − m) do
    while (t[i + j] = s[j]) do
        j++;
        if (j = m) then
            return TRUE;
    i++;
    j := 0;
return FALSE;
```

Match und Mismatch

Definition

Stimmen zwei Zeichen bei einem Vergleich überein, so nennt man dies ein Match, ansonsten ein Mismatch.

Analyse des naiven Algorithmus

- zähle Vergleiche von Zeichen,
- äußere Schleife wird $(n - m + 1)$-mal durchlaufen,
- die innere Schleife wird maximal m-mal durchlaufen.
- maximale Anzahl von Vergleichen: $(n - m + 1)m$,
- Laufzeit: $\mathcal{O}(nm)$

Bessere Idee

- frühere erfolgreiche Vergleiche von zwei Zeichen ausnutzen

- Idee:

  Suchwort so weit nach rechts 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

Rand und eigentlicher Rand

Definition

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

Bemerkung: Für jedes Wort w sind damit immer auch das leere Wort ε und w selbst Ränder von w.

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

Rand und eigentlicher Rand

Beispiel

Das Wort $w =$aabaabaa besitzt folgende Ränder:

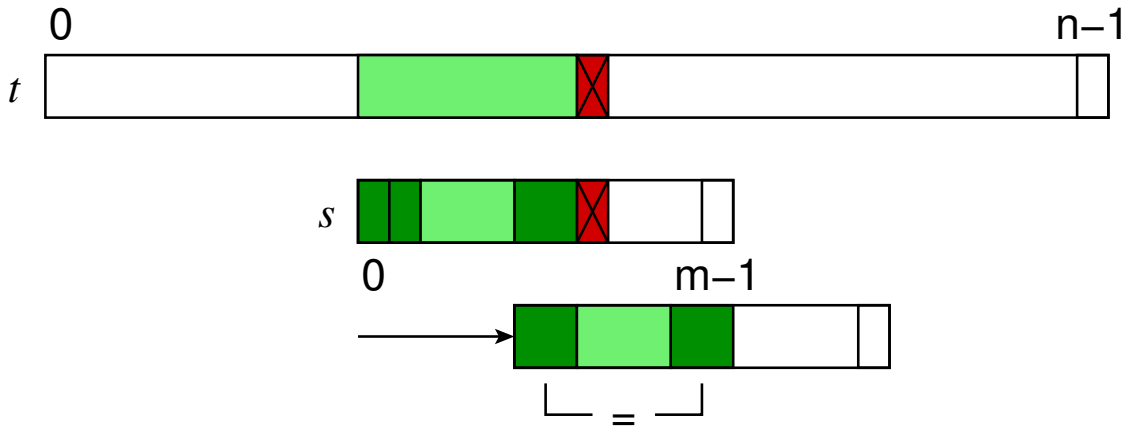
- ε
- a
- aa
- aabaa
- aabaabaa$= w$

Der eigentlichen Rand ist aabaa.

Man beachte, dass sich bei der Darstellung eines Rands im Wort das entsprechende Präfix und Suffix in der Mitte des Worts überlappen können.

Shift-Idee

- Pattern s so verschieben, dass im bereits gematchten Bereich wieder Übereinstimmung herrscht.
- Dazu müssen überlappendes Präfix und Suffix dieses Bereichs übereinstimmen.



Shifts und sichere Shifts

Definition

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

Ein Shift von $i \longrightarrow i'$ heißt sicherer Shift, 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 $k \in [i+1 : i'-1]$.

- Sinn eines sicheren Shifts: dass man beim Verschieben des Suchworts kein eventuell vorhandenes Vorkommen von s in t überspringt.

Sichere Shifts

Definition

Sei $\partial(s)$ der eigentliche Rand von s und sei

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

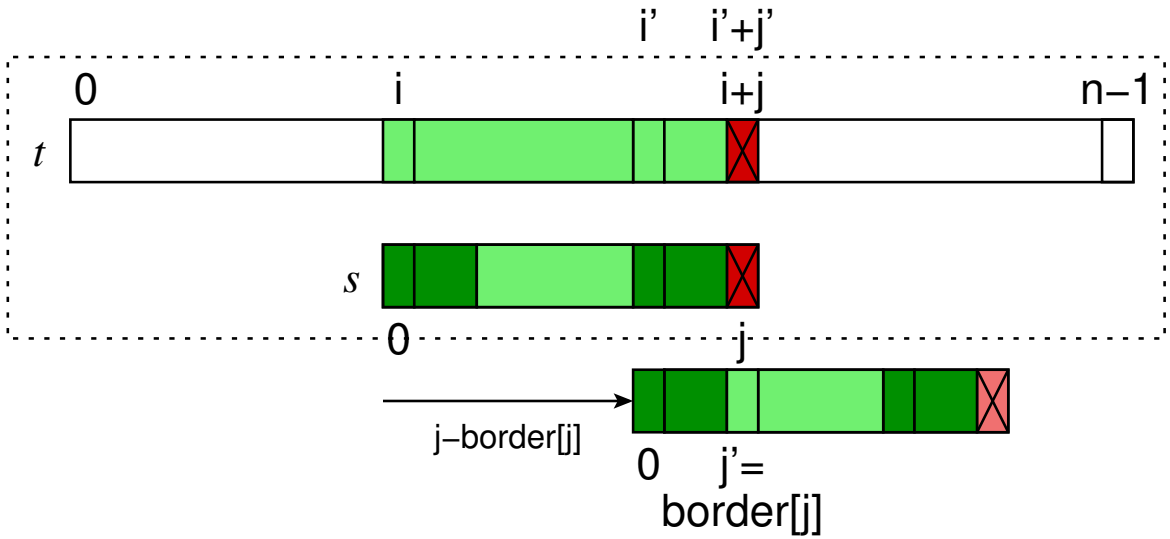
die Länge des eigentlichen Rands des Präfixes der Länge j.

Lemma

Ist das Präfix der Länge j gematcht (also gilt $s_k = t_{i+k}$ für alle $k \in [0 : j-1]$) und haben wir ein Mismatch an der nächsten Position j ($s_j \neq t_{i+j}$), dann ist der Shift $i \to i + j - border[j]$ sicher.

Sichere Shifts

Shift um $j - \text{border}[j]$:



Sichere Shifts

Beweis.

- (siehe Skizze)

$$\begin{aligned} s_0 \cdots s_{j-1} &= t_i \cdots t_{i+j-1}, \\ s_j &\neq t_{i+j} \end{aligned}$$

- Der eigentliche Rand von $s_0 \cdots s_{j-1}$ hat die Länge $\mathrm{border}[j]$.
- Verschiebt man s um $j - \mathrm{border}[j]$ nach rechts, so liegt der linke Rand von $s_0 \cdots s_{j-1}$ nun genau da, wo vorher der rechte Rand lag, d.h. im Präfix/Suffix-Überlappungsbereich besteht Übereinstimmung zwischen Präfix, Suffix und Text.
- Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt (außer $s_0 \cdots s_{j-1}$ selbst), ist dieser Shift sicher.

KMP-Algorithmus

Algorithmus 2 : bool KMP(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 do
    while t[i + j] = s[j] do
        j++;
        if j = m then
            return TRUE;
    i := i + (j − border[j]) ;          // Es gilt j − border[j] > 0
    j := max{0, border[j]};
return FALSE;
```

Laufzeit des KMP-Algorithmus

Definition

Ein Vergleich von zwei Zeichen heißt erfolgreich, wenn die beiden Zeichen gleich sind (Match), und erfolglos sonst (Mismatch).

Laufzeit des KMP-Algorithmus: erfolglose Vergleiche

Nach einem erfolglosen Vergleich wird der Wert von $i + j$ nie kleiner:

- Seien dazu i und j die Werte vor einem erfolglosen Vergleich und i' und j' die Werte nach einem erfolglosen Vergleich.
- Wert vor dem Vergleich: $i + j$
- Wert nach dem Vergleich:
  $i' + j' = (i + j - border[j]) + (max\{0, border[j]\})$.
- Fallunterscheidung: $border[j]$ negativ oder nicht.

$border[j] < 0$ Ist $border[j] = -1$, dann muss $j = 0$ sein. Das bedeutet
$i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1$.

$border[j] \geq 0$ Ist $border[j] \geq 0$, dann gilt $i' + j' = i + j$.

- Also wird $i + j$ nach einem erfolglosen Vergleich nicht kleiner.

Laufzeit des KMP-Algorithmus: erfolglose Vergleiche

- Nach jedem erfolglosen Vergleich wird $i \in [0 : n - m]$ erhöht.
- Im Verlaufe des Algorithmus wird i nie erniedrigt wird.

$\Rightarrow$ maximal $n - m + 1$ erfolglose Vergleiche

Laufzeit des KMP-Algorithmus: erfolgreiche Vergleiche

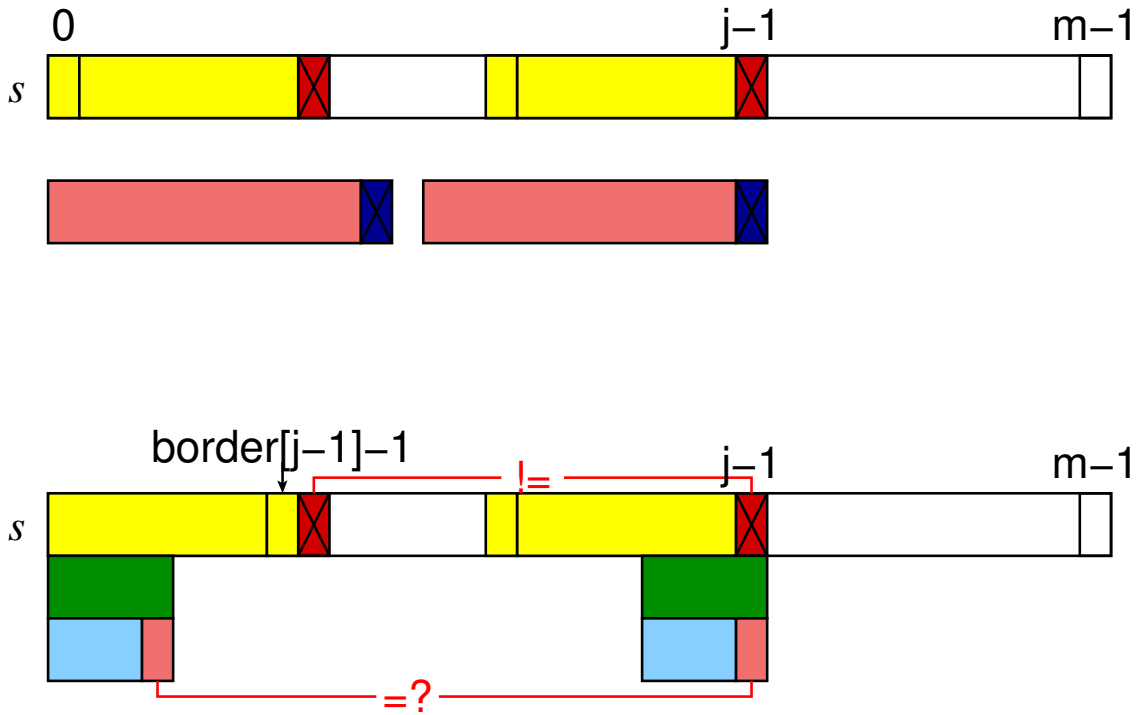
- Nach einem erfolgreichen Vergleich wird $i + j$ um 1 erhöht! Die maximale Anzahl erfolgreicher Vergleiche ist somit durch n beschränkt, da $i + j \in [0 : n - 1]$.

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

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.
- Wir initialisieren die Tabelle zunächst mit $border[0] = -1$ und $border[1] = 0$.
- Im Folgenden nehmen wir an, dass $border[0], \ldots, border[j-1]$ bereits berechnet sind.
- Wir wollen dann den Wert $border[j]$, also die Länge des eigentlichen Randes eines Präfixes der Länge j, berechnen.

Berechnung der border-Tabelle



Berechnung der border-Tabelle

- Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$, da nach Definition $s_0 \cdots s_{k-1}$ auch ein Rand von $s_0 \cdots s_{j-2}$ ist (oberer Teil der Abbildung).
- 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 betrachteten Randes dieses Wortes.
- Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$.

Berechnung der border-Tabelle

- 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 $border[j] = 0$.

Algorithmus zur Berechnung der border-Tabelle

Algorithmus 3 : compute_borders(int *border*[], int *m*, char *s*[])

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

Laufzeit der Berechnung der border-Tabelle

- Zähle erfolgreiche und erfolglose Ausführungen der while-Schleifenbedingung

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

Laufzeit der Berechnung der border-Tabelle

- Für die Anzahl erfolgloser Vergleiche betrachten wir den Wert i. Zu Beginn ist $i = 0$.
- i wird genau $(m-1)$ Mal um 1 erhöht, da die for-Schleife $(m-1)$ Mal durchlaufen wird.
- Im Fall eines erfolglosen Vergleichs wird i um mindestens 1 erniedrigt.
- i kann maximal $(m-1)+1 = m$ Mal erniedrigt werden, da immer $i \geq -1$ gilt. Es gibt also höchstens m erfolglose Vergleiche.

- Also ist die Anzahl der Vergleiche durch $2m-1$ beschränkt.

Laufzeit des KMP-Algorithmus

Theorem

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.

Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt
Fast Pattern Matching in Strings

SIAM Journal on Computing 6(2):323–350, 1977.