Kapitel 5

Textalgorithmen

5.1 Grundbegriffe

5.1.1 Das Textsuchproblem

In diesem Kapitel geht es um das Problem der *Textsuche* (engl.: *pattern matching*). Im Hintergrund steht immer ein *Alphabet* Σ, eine endliche Menge, mit $|\Sigma| \geq 2$. *Beispiele*:

- $\{0, 1\}$, das binäre Alphabet;
- $\{\mathrm{A}, \mathrm{G}, \mathrm{C}, \mathrm{T}\}$[1]
- ASCII, ein klassischer Code mit 128 Zeichen, erstmals standardisiert 1963;
- ISO 8859-1 (Latin-1, auch DIN 66303:2000-06) oder Windows CP 1252 beschreiben Zeichensätze („ANSI-Familie“) mit 256 Buchstaben und zugehörige Binärkodierungen mit 8 Bits;
- $\{0, 1\}^8$, die Binärkodierungen zu den ANSI-Alphabeten, und $\{0, 1, \ldots, 255\}$, die entsprechenden numerischen Werte;
- Unicode mit momentan etwa 128 000 Zeichen.

Alphabete gleicher Größe (z. B. die Alphabete der Größe 256) unterscheiden sich nur durch die Bezeichnungen der Buchstaben. Wie die Buchstaben heißen, ist für die

[1] Die Buchstaben stehen für die Namen der Nukleinbasen Adenin (A), Guanin (G), Cytosin (C) und Thymin (T), die bei der Funktionalität der DNA als genetischer Code eine zentrale Rolle spielen. Tatsächlich ist die *Computational Biology* ein zentrales Anwendungsgebiet von Textalgorithmen.

Zwecke der Textsuche irrelevant, und man kann auch annehmen, dass das Alphabet einfach $\{0, 1, \ldots, |\Sigma| - 1\}$ ist. Bei einigen wichtigen Algorithmen ist es sogar ganz gleichgültig, was das Alphabet ist, weil auf Buchstaben a und b nur die Operation „gilt $a = b$?“ ausgeführt wird.

Notation. (a) $\Sigma^* = \{a_1 \ldots a_t \mid t \geq 0, a_1, \ldots, a_t \in \Sigma\}$ bezeichnet die Menge aller *Wörter* (*Strings*, *Zeichenreihen*, *Zeichenketten*) (d. h. aller endlichen Folgen) über Σ.[2]
Dabei steht $a_1 \ldots, a_t$ als bequeme Abkürzung für $(a_1, \ldots, a_t)$. *Beispiel*: $\{\texttt{A}, \texttt{B}, \texttt{C}\}^* = \{\varepsilon, \texttt{A}, \texttt{B}, \texttt{C}, \texttt{AA}, \texttt{AB}, \texttt{AC}, \texttt{BA}, \ldots, \texttt{ACBB}, \ldots\}$.
Beachte: Für $t = 0$ erhält man immer das Wort mit 0 Buchstaben, das *leere Wort* ε.
(b) Arrayschreibweise für Wörter: $W = a_1 \ldots a_t$ wird als $W[1..t]$ geschrieben; $W[i]$ bedeutet a_i, und $W[i..j]$ bedeutet das *Teilwort* $a_i \ldots a_j$.
Beispiel: Wenn $W = \texttt{ACBB}$, ist $W[2] = \texttt{C}$, $W[1..3] = \texttt{ACB}$ und $W[1..0] = W[4..2] = \varepsilon$.
Wir verwenden diese Notation nur für $1 \leq i \leq t+1$ und $0 \leq j \leq t$. Für $i > j$ bedeutet $W[i..j]$ stets das leere Wort ε.
(c) Ein Teilwort $W[1..i] = a_1 \ldots a_i$ mit $0 \leq i \leq t$ heißt ein *Präfix* von $W = a_1 \ldots a_t$.
Beispiel: Die Präfixe von $\texttt{ACBB}$ sind ε, $\texttt{A}$, $\texttt{AC}$, $\texttt{ACB}$ und $\texttt{ACBB}$.
Ein Teilwort $W[i..t] = a_i \ldots a_t$ mit $1 \leq i \leq t + 1$ heißt ein *Suffix* von $W = a_1 \ldots a_t$.
Beispiel: Die Suffixe von $\texttt{ACBB}$ sind $\texttt{ACBB}$, $\texttt{CBB}$, $\texttt{BB}$, $\texttt{B}$ und ε.

Das einfache Textsuchproblem besteht in Folgendem.

Eingabe:
„Muster“ (engl.: *pattern*) $P = P[1..m] \in \Sigma^*$, $m \geq 1$,
„Text“ (engl.: *text*) $S = S[1..n] \in \Sigma^*$, $n \geq 1$.

Aufgabe: Finde erstes (oder einige oder alle) Vorkommen von P in S, d. h. finde das kleinste oder einige oder alle $\ell \in \{1, \ldots, n - m + 1\}$ mit $P = S[\ell..\ell + m - 1]$.

Beispiel:
(Blanks ␣ zählen als Buchstabe; wir wollen alle Vorkommen des Musters finden.)

$S = \texttt{IM␣HEUHAUFEN␣DIE␣NADEL␣FINDEN}$, $P = \texttt{NADEL}$: Ausgabe: $\{18\}$

$S = \texttt{IM␣NADELHAUFEN␣DIE␣NADEL␣FINDEN}$, $P = \texttt{NADEL}$: Ausgabe: $\{4, 20\}$

$S = \texttt{IM␣WALD␣DEN␣BAUM␣FINDEN}$, $P = \texttt{NADEL}$: Ausgabe: $\emptyset$

$S = \texttt{IM␣WALD␣DEN␣BAUM␣FINDEN}$, $P = \texttt{WAL}$: Ausgabe: $\{4\}$.

[2] Dabei steht $a_1 \ldots, a_t$ nur als bequeme Abkürzung für $(a_1, \ldots, a_t)$. Es ist also nie ein Problem, die Buchstaben voneinander abzugrenzen.

5.1.2 Naive Algorithmen

Natürlich lässt sich das Textsuchproblem im Prinzip ganz einfach lösen: Man probiert einfach alle Möglichkeiten durch.

Naive Textsuche: Für jedes $\ell \in \{1, \ldots, n-m+1\}$ tue Folgendes:
Vergleiche $P[1..m]$ Buchstabe für Buchstabe mit $S[\ell..\ell+m-1]$, bis Übereinstimmung aller m Buchstaben festgestellt wird (dann wird ℓ in die Ausgabe geschrieben) oder bis eine Fehlerstelle (engl.: *mismatch*) gefunden wurde (d. h. ein Index $q \in \{0, \ldots, m-1\}$ mit $P[q+1] \neq S[\ell+q]$).

Es gibt einige Varianten dieses Prinzips, je nachdem, in welcher Reihenfolge die Plätze ℓ bearbeitet werden und die Buchstaben der Wörter P und $S[\ell..\ell+m-1]$ verglichen werden. Naheliegend ist es, jeweils von links nach rechts vorzugehen.[3]

Algorithmus 5.1.1 (Naive Textsuche, Vergleiche von links nach rechts).
Naive-PM-left-right(`P[1..m]`,`S[1..n]`)
Eingabe: `P[1..m]`, `S[1..n]` // `P`: Muster, `S`: Text
Ausgabe: `A` $\subseteq \{1, \ldots, n-m+1\}$;
(1) `A` $\leftarrow \emptyset$; // Initialisierung
(2) **for** `l` **from** 1 **to** $n-m+1$ **do**
(3) `q` $\leftarrow 0$;
(4) **while** `q` $< m \wedge$ `P[q+1]` $=$ `S[l+q]` **do** `q` $\leftarrow$ `q+1`;
(5) **if** `q` $= m$ **then** `A` $\leftarrow$ `A` $\cup \{$`l`$\}$;
(6) **return** `A`.

Anschaulich stellt man sich vor, dass man für jedes $\ell = 1, \ldots, n-m+1$ nacheinander das Muster P unter den Abschnitt $S[\ell..\ell+m-1]$ des Textes legt und von links nach rechts gehend Paare von Buchstaben vergleicht. Von Runde zu Runde wird das Muster also um eine Position nach rechts verschoben.

Beispiel: Muster $P =$ `abra` ($m = 4$), Text $S =$ `abracababrabrac` ($n = 15$). Die folgende Tabelle zeigt die 12 Positionen für das Muster und die für die jeweilige Position verglichenen Buchstaben:

[3] Bei den später behandelten Algorithmen der Boyer-Moore-Familie wird es sich als nützlich herausstellen, Positionen ℓ in aufsteigender Reihenfolge zu betrachten, für eine Position ℓ die Buchstaben in $P[1..m]$ jedoch von rechts nach links.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ℓ \ Text	a	b	r	a	c	a	b	a	b	r	a	b	r	a	c
1	a	b	r	a											
2		a	b	r	a										
3			a	b	r	a									
4				a	b	r	a								
5					a	b	r	a							
6						a	b	r	a						
7							a	b	r	a					
8								a	b	r	a				
9									a	b	r	a			
10										a	b	r	a		
11											a	b	r	a	
12												a	b	r	a

Alle 12 Positionierungen des Musters sind angegeben, verglichene Buchstaben sind grau (Übereinstimmung: hellgrau, Fehlerstelle: dunkelgrau). Das Muster kommt dreimal vor. Insgesamt gibt es 24 Vergleiche. Man bemerkt, dass $S[4]$ und $S[11]$ zweimal erfolgreich mit Musterbuchstaben verglichen werden, und dass $S[9]$ und $S[10]$ erfolglos mit $P[1]$ verglichen werden, obwohl man vorher schon gesehen hat, dass an diesen Stellen `b` und `r`, also nicht $P[1] = \texttt{a}$ steht.

Die Laufzeit des naiven Algorithmus beträgt $O((n-m+1)m)$ im schlechtesten Fall; die Anzahl der Buchstabenvergleiche ist maximal $(n-m+1)m$. Folgendes *Beispiel* zeigt, dass der schlechteste Fall auch eintreten kann: Muster $P = \texttt{a}^{m-1}\texttt{b}$ und Text $S = \texttt{a}^{n-1}\texttt{b}$ führen zu genau $(n-m+1)m$ Buchstabenvergleichen, weil in jeder Position ℓ die Fehlerstelle erst nach m Vergleichen gefunden wird. Oft ist die Vergleichszahl geringer, weil in der inneren Schleife schon früh eine Fehlerstelle gefunden wird. Besonders bei natürlichsprachigen Texten wird dies so sein. Man kann dies mit den Beispielen in Abschnitt 5.1.1 ausprobieren. Allgemein gilt: Wenn Muster P oder Text S aus rein zufällig aus Σ^m bzw. Σ^n sind, dann ist die erwartete Anzahl von Buchstabenvergleichen bei der naiven Textsuche kleiner als $(n-m+1) \cdot \frac{|\Sigma|}{|\Sigma|-1} < 2(n-m+1)$.

In diesem Kapitel betrachten wir Algorithmen, die eine lineare Laufzeit von $O(m+n)$ *garantieren*. Dies sind der Algorithmus von Knuth, Morris und Pratt und der Algorithmus von Boyer und Moore. Daneben betrachten wir einen Algorithmus, der nach Vorkommen von Wörtern aus einer ganzen Liste von Mustern sucht (Algorithmus von Aho und Corasick).

5.1.3 Ränder

Grundlegend für viele Textalgorithmen ist der Begriff des **Randes** (engl.: *border*). Anschaulich ist ein Rand eines Wortes W ein kürzeres Teilwort, das man am linken Ende und ebenso am rechten Ende von W findet.

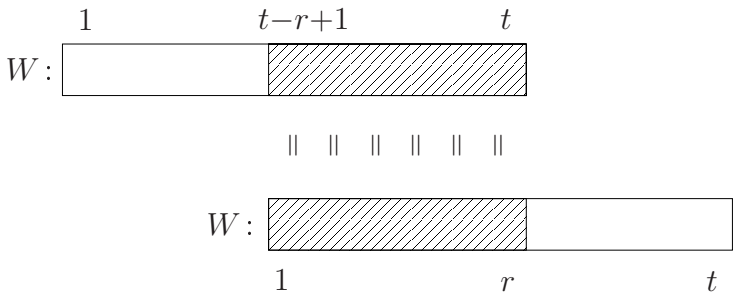

Abbildung 5.1: Ein Teilwort $\neq W$ von W, das sowohl Präfix als auch Suffix von W ist, heißt ein *Rand* von W.

Definition 5.1.2. *Sei $W = a_1 \dots a_t$ ein Wort, $t \geq 0$. $W[1..r]$ heißt ein **Rand** von W, wenn $0 \leq r < t$ und $W[1..r]$ (nicht nur Präfix, sondern auch) Suffix von W ist, d. h. wenn $W[1..r] = W[t-r+1..t]$ gilt* (s. Abb. 5.1).

Man beachte folgende Sonderfälle:
(a) Jedes nichtleere Wort $W = W[1..t]$ hat $\varepsilon = W[1..0] = W[t+1..t]$ als Rand.
(b) W selbst zählt *nicht* als Rand von W, obwohl es Präfix und Suffix von sich selbst ist. (Mitunter wird W als „uneigentlicher/unechter Rand“ von W bezeichnet.)
(c) Das leere Wort ε hat überhaupt keinen Rand, da es kein r mit $0 \leq r < 0$ gibt.

Beispiel: Die Ränder von `abababcabab` sind ε, `ab` und `abab`. Die Ränder von `ababa` sind ε, `a` und `aba`.

Uns interessiert oft der *längste* Rand von W. Der längste Rand von `ababab` ist `abab`, der längste Rand von `ababcabc` ist ε.

Die längsten Ränder der *Präfixe* des Musters $P[1..m]$ sind von besonderem Interesse. Man könnte die Funktion

$$\text{bord}\colon \{P[1..q] \mid 0 \leq q \leq m\} \to \{P[1..q] \mid 0 \leq q < m\} \cup \{-\},$$

$$P[1..q] \mapsto \begin{cases} \text{längster Rand } P[1..q'] \text{ von } P[1..q] & , \text{ falls } 1 \leq q \leq m; \\ - & , \text{ falls } q = 0. \end{cases}$$

betrachten, die jedem Präfix $P[1..q]$ seinen längsten Rand $P[1..q']$ zuordnet, bzw. den Wert „$-$“ (für „undefiniert“) für $P[1..0] = \varepsilon$. Es ist aber technisch bequem, eine Zahlfunktion auf den entsprechenden Längen zu benutzen.

Definition 5.1.3 (Randfunktion). *Gegeben sei das Muster $P[1..m]$. Wir definieren die Funktion $f_{\text{bord}} = f^P_{\text{bord}}$ mit $f_{\text{bord}} \colon \{0, 1, \ldots, m\} \to \{-1, 0, \ldots, m-1\}$ durch:*

$$f_{\text{bord}}(q) := \begin{cases} \text{Länge des längsten Randes von } P[1..q] & , \text{ falls } 1 \le q \le m; \\ -1 & , \text{ falls } q = 0. \end{cases}$$

Offensichtlich ist immer $f_{\text{bord}}(q) < q$. Die Festlegung $f_{\text{bord}}(0) = -1$ wird sich auch (programmier-)technisch als günstig erweisen.

q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`P[`q`]`		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c
$f_{\text{bord}}(q)$	−1	0	0	0	1	0	1	2	1	2	3	4	2	3	4	5

Abbildung 5.2: Randfunktion f_{bord} bei $P[1..15] =$ `abracababrabrac`.

In Abb. 5.2 ist die Randfunktion für das Muster `abracababrabrac` angegeben. Es gilt zum Beispiel $f_{\text{bord}}(10) = 3$, weil $\text{bord}(P[1..10]) = \text{bord}(\texttt{abracababr}) = \texttt{abr} = P[1..3]$ gilt. – Es gibt effiziente Verfahren für die Berechnung einer Tabelle für die Randfunktion. Um diese Algorithmen kümmern wir uns später. (WS 2020/21: Übung.)

Übungsaufgabe. Wie kann man bei gegebener Randfunktion f_{bord} zu gegebenem q *alle* Ränder von $P[1..q]$, d. h. alle r, für die $P[1..r]$ Rand von $P[1..q]$ ist, berechnen? Für jeden auszugebenden Index r soll nur Zeit $O(1)$ aufgewendet werden.

5.2 Der Algorithmus von Knuth, Morris und Pratt

5.2.1 Vorüberlegungen

Der Algorithmus von Knuth, Morris und Pratt (KMP-Algorithmus)[4] geht vom naiven Textsuchalgorithmus 5.1.1 aus, also der Version, in der für jede Stelle $\ell = 1, \ldots, n - m + 1$ im Text $S[1..n]$ die Musterbuchstaben $P[q+1]$, $q = 0, \ldots, m-1$, von links nach rechts mit den Textbuchstaben $S[\ell + q]$, $q = 0, \ldots, m-1$, verglichen werden. Auch der KMP-Algorithmus stellt diese Vergleiche von Wörtern für eine monotone Folge von Werten ℓ an, versucht aber nach Möglichkeit,

- ℓ-Werte zu überspringen (also das Muster in einem Schritt um ein größeres Stück nach rechts zu schieben: „*shift*“ um mehr als eine Position) und

[4] Donald E. Knuth, James H. Morris, Vaughan R. Pratt (1977): Fast Pattern Matching in Strings. SIAM Journal on Computing. **6**(2): 323–350. doi:10.1137/0206024

- überflüssige Buchstabenvergleiche auszulassen.

Wir erklären die Ideen genauer.[5] Angenommen, der Algorithmus ist an Position $\ell \leq n - m + 1$ angekommen, und wir haben das größte $q \in \{0, 1, \ldots, m\}$ mit $P[1..q] = S[\ell..\ell + q - 1]$ ermittelt. Es gibt zwei Fälle:

1. Fall: $q < m$ und $P[q + 1] \neq S[\ell + q]$. – Dann kann an Stelle ℓ das Muster nicht vorkommen. Wir verschieben $P[1..m]$ an eine neue Position $\ell' > \ell$ und testen erneut auf Übereinstimmung (bzw. hören auf, wenn $\ell' > n - m + 1$). Der naive Algorithmus versucht die Stelle $\ell' = \ell+1$ und vergleicht $P[1..m]$ mit $S[\ell'..\ell'+m-1]$ buchstabenweise von links nach rechts. Der KMP-Algorithmus nutzt das Wissen

$$P[1..q] = S[\ell..\ell + q - 1] \quad \text{und} \quad P[q + 1] \neq S[\ell + q] \tag{5.1}$$

geschickt aus, und zwar doppelt! Erstens ist es überflüssig, Positionen $\ell' \in \{\ell + 1, \ldots, \ell + q\}$ zu testen, für die aus (5.1) schon folgt, dass sie nicht als Position des Musters in Frage kommen. Zweitens kann man für eine mögliche neue Position $\ell' \in \{\ell+1, \ldots, \ell+q-1\}$ ausnutzen, dass schon erfolgreiche Vergleiche mit den Buchstaben $S[\ell'..\ell + q - 1]$ im Text durchgeführt worden sind.

In Abb. 5.3 und Abb. 5.4 sind die möglichen Situationen dargestellt. Wir betrachten zuerst einen Verschiebung um einen „Shiftwert“ $s \in \{1, \ldots, q\}$, von Position ℓ zu Position $\ell' = \ell + s$. Wir definieren $q' = q - s \geq 0$. Dann gilt $\ell + q = \ell' + q'$.

Wenn das Muster an Position ℓ' vorkommt, muss

$$P[1..q'] = S[\ell'..\ell + q - 1] \quad \text{und} \quad P[q' + 1] = S[\ell + q]$$

gelten. In Kombination mit (5.1) liefert dies die notwendige Bedingung

$$P[1..q'] = P[q - q' + 1..q] \quad \text{und} \quad P[q' + 1] \neq P[q + 1]. \tag{5.2}$$

Shiftwerte $s = q - q'$, die (5.2) erfüllen, heißen *zulässig*. Die erste Gleichheit in (5.2) bedeutet, dass $P[1..q']$ Rand von $P[1..q]$ sein muss, die zweite ergänzt, dass in P auf $P[1..q']$ und $P[1..q]$ verschiedene Buchstaben folgen müssen. Wenn es ein zulässiges $s \leq q$ gibt, dann verschieben wir das Muster um den *kleinsten* zulässigen Wert s, setzen also $\ell' := \ell + s$. Damit ist sichergestellt, dass keine mögliche Position des Musters übersehen wird. Mit den buchstabenweisen Vergleichen von links nach rechts geht es nun weiter wie im naiven Algorithmus, allerdings brauchen wir in P erst bei $P[q' + 1]$ und in S bei $S[\ell' + q'] = S[\ell + q]$ anzufangen, da die Gleichheit $P[1..q'] = S[\ell'..\ell' + q' - 1]$ wegen (5.1) schon gesichert ist.

[5] Die Entwicklung des KMP-Algorithmus aus dem naiven Algorithmus wird recht detailliert beschrieben, als ein Lehrbeispiel dafür, wie man durch genaue Betrachtung und Verbesserung eines einfachen Algorithmus zu einer sehr effizienten Version kommen kann.

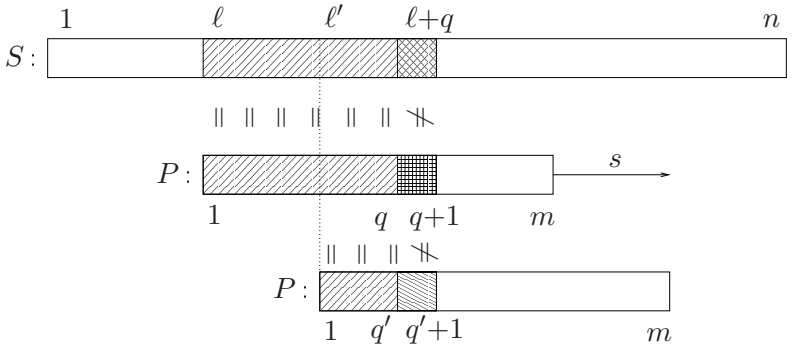

Abbildung 5.3: Grundidee des KMP-Algorithmus, 1. Fall, Standardsituation: Eine Verschiebung um $s = \ell' - \ell = q - q' \leq q$ kann nur sinnvoll sein, wenn $P[1..q'] = P[q - q' + 1..q]$ und $P[q' + 1] \neq P[q + 1]$ gelten. Diese Überlegung gilt auch im Fall $q = 1$ und $q' = 0$. Im Fall einer solchen Verschiebung ist der nächste Vergleich zwischen $S[\ell + q]$ und $P[q' + 1]$ auszuführen.

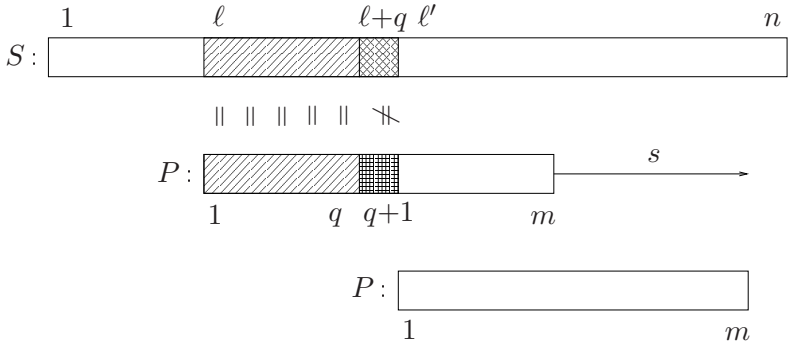

Abbildung 5.4: Grundidee des KMP-Algorithmus, 1. Fall, Sonderfall: Eine Verschiebung zur Position $\ell' = \ell + q + 1$, also mit $s = q + 1$ und $q' = q - s = -1$, kann nie ausgeschlossen werden, da kein Buchstabe in $S[\ell'..\ell' + m - 1]$ bisher gesehen wurde. Im Fall einer solchen Verschiebung ist der nächste Vergleich zwischen $S[\ell + q + 1]$ und $P[1] = P[q' + 2]$ auszuführen.

Es gibt noch einen Sonderfall. Wenn es überhaupt kein zulässiges $s \in \{1, \ldots, q\}$ gibt, dann kommt keine der Positionen $\ell + 1, \ldots, \ell + q$ als neues ℓ' in Frage. Wir setzen dann $\ell' = \ell + q + 1$, also $s = q + 1$, was zu $q' = q - s = -1$ führt. Diese Position ℓ' kann (nach der in (5.1) angegebenen Information) nie ausgeschlossen werden, da kein Buchstabe aus $S[\ell'..n]$ dort erwähnt wird. Der Shiftwert $s = q + 1$ gilt also *immer* als zulässig. Wenn dies der kleinste zulässige Wert ist, wird um $q+1$ verschoben. Der erste neue Vergleich ist in diesem Fall natürlich an der Stelle $P[1]$ und $S[\ell'] = S[\ell + q + 1]$ auszuführen. Dieser Sonderfall tritt im 1. Fall für $q = 0$ (also bei $P[1] \neq S[\ell]$) immer

ein.

2. Fall: $q = m$. – Dann steht das Muster P im Bereich $S[\ell..\ell+q-1]$, und wir geben ℓ aus. Wenn wir alle Vorkommen des Musters in S finden sollen, müssen wir noch weiterarbeiten. Wir suchen also eine neue Position $\ell' > \ell$, indem wir das Wissen

$$P[1..m] = S[\ell..\ell+m-1] \tag{5.3}$$

geschickt ausnutzen. In Abb. 5.5 ist diese Situation dargestellt.

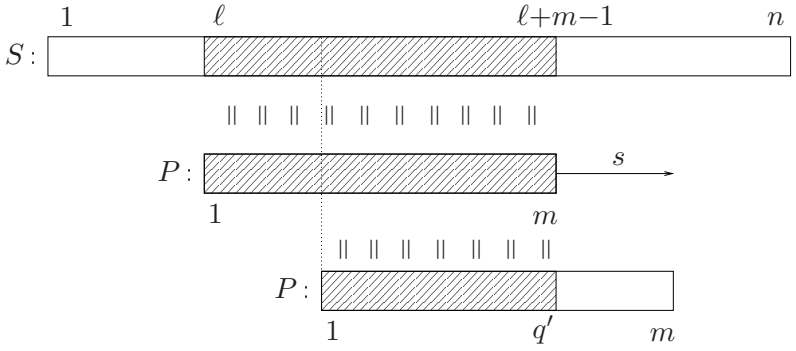

Abbildung 5.5: Grundidee des KMP-Algorithmus, 2. Fall: Wir wissen, dass $P[1..m] = S[\ell..\ell+m-1]$ gilt. Eine Verschiebung des Musters an eine Stelle $\ell' = \ell + s$ kann nur sinnvoll sein, wenn für $q' = q - s$ das Präfix $P[1..q']$ ein Rand von P ist. (Es gibt immer mindestens einen Rand, nämlich ε.)

Weil für eine erfolgreiche Position $\ell' = \ell + s = \ell + q - q'$ die Gleichheit $P[1..q'] = S[\ell'..\ell'+q'-1] = S[\ell'..\ell+m-1]$ gelten muss, folgt die notwendige Bedingung

$$P[1..q'] = P[m-q'+1..m]. \tag{5.4}$$

In diesem Fall gelten also alle Shiftwerte s, für die $P[1..m-s]$ ein Rand des Musters $P[1..m]$ ist, als zulässig. ($s = m$ ist hier immer zulässig.) Von allen zulässigen Werten s wählen wir den kleinsten, damit keine mögliche Position des Musters übersehen wird, und setzen $\ell' := \ell + s$ und $q' := m - s$. In der Bearbeitung der Position ℓ' können wir mit dem Vergleich zwischen $P[q'+1]$ und $S[\ell'+q'] = S[\ell+m]$ fortfahren, weil schon bekannt ist, dass $P[1..q']$ und $S[\ell'..\ell'+q'-1]$ übereinstimmen.

5.2.2 Die Fehlerfunktion

In den Vorüberlegungen zum KMP-Algorithmus muss man nach dem Finden einer Fehlerstelle oder dem Auftreten des Musters die richtige (kleinste) zulässige Shiftweite $s = \ell' - \ell = q - q'$ finden. Aus den Bedingungen (5.2) und (5.4) ist klar, dass diese

Shiftweite nur von P und q abhängt, nicht von S. Wir können also diese Shiftweite für $q = 0, \ldots, m$ aus $P[1..m]$ vorab berechnen. Es ist technisch günstig, nicht die kleinste Shiftweite s, sondern den neuen Wert $q' = q - s$ zu betrachten. Die Funktion $q \mapsto q'$, die *KMP-Fehlerfunktion*, ist eine verschärfte Variante der Randfunktion aus Abschnitt 5.1.3. Für ein Präfix $P[1..q]$ mit $q < m$ sind nur solche Ränder $P[1..q']$ interessant, die $P[q'+1] \neq P[q+1]$ erfüllen.

Definition 5.2.1 (KMP-Fehlerfunktion). *Gegeben sei das Muster $P[1..m]$. Die KMP-Fehlerfunktion $f_{\mathrm{KMP}} \colon \{0, 1, \ldots, m\} \to \{-1, 0, \ldots, m-1\}$ ist wie folgt definiert:*

$$f_{\mathrm{KMP}}(q) := \begin{cases} \text{Länge } q' \text{ des längsten Randes } P[1..q'] \text{ von } P[1..m], \text{ falls } q = m; \\ \text{Länge } q' \text{ des längsten Randes } P[1..q'] \text{ von } P[1..q] \\ \qquad \text{mit } P[q'+1] \neq P[q+1], \text{ falls ein solcher existiert,} \\ \qquad\qquad \text{für } 0 < q < m; \\ -1, \text{ falls kein solcher Rand existiert, für } 0 \leq q < m. \end{cases}$$

Bemerkung: Man kann $f_{\mathrm{KMP}}(q)$ auch kompakt wie folgt beschreiben: $f_{\mathrm{KMP}}(q)$ ist die *größte* Zahl $q' \in \{-1, 0, 1, \ldots, q-1\}$ mit:

$$P[1..q'] = P[q - q' + 1..q] \quad \text{und} \quad (q' \geq 0 \wedge q < m) \Rightarrow P[q'+1] \neq P[q+1]. \qquad (5.5)$$

Die erste Aussage in (5.5) gilt trivialerweise für $q' \in \{-1, 0\}$, die zweite gilt trivialerweise für $q' = -1$ und ebenso für $q = m$. Also ist (5.5) durch $q' = -1$ immer erfüllt, und im Fall $q = m$ durch $q' = 0$ erfüllt.

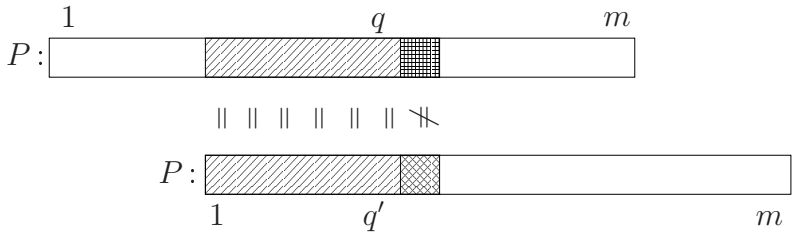

Abbildung 5.6: Für $0 \leq q < m$ ist $f_{\mathrm{KMP}}(q)$ die Länge des längsten Randes $P[1..q']$ von $P[1..q]$, so dass die folgenden Buchstaben $P[q'+1]$ und $P[q+1]$ verschieden sind – falls ein solcher Rand existiert. Sonst ist $f_{\mathrm{KMP}}(q) = -1$. Anmerkung: $f_{\mathrm{KMP}}(0) = -1$, weil $P[1..0] = \varepsilon$ überhaupt keinen Rand hat. $f_{\mathrm{KMP}}(m)$ ist einfach $f_{\mathrm{bord}}(m)$.

In Abb. 5.7 sind die Randfunktion und die KMP-Fehlerfunktion für $P =$ `abracababrabrac` angegeben, zusammen mit Beispielen für die verschiedenen möglichen Situationen. Es gilt zum Beispiel:

- $f_{\mathrm{KMP}}(1) = 0$, weil für den Rand $P[1..0] = \varepsilon$ von $P[1..1]$ die zweite Bedingung erfüllt ist ($P[1] = \mathtt{a} \neq \mathtt{b} = P[2]$);

q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
$P[q]$		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c	
$f_{\text{bord}}(q)$	−1	0	0	0	1	0	1	2	1	2	3	4	2	3	4	5	
$f_{\text{KMP}}(q)$	−1	0	0	−1	1	−1	0	2	0	0	−1	4	0	−1	1	5	
$P[1..0], P[1]$	ε	a															
$P[1..15]$			a	b	r	a	c	a	b	...							
$P[1..1], P[2]$		a	b														
$P[1..15]$			a	b	r	a	c	a	b	...							
$P[1..4], P[5]$		a	b	r	a	c											
$P[1..15]$					a	b	r	a	c	a	b	...					
$P[1..5], P[6]$		a	b	r	a	c	a										
$P[1..15]$								a	b	r	a	c	a	b	...		
$P[1..11], P[12]$		a	b	r	a	c	a	b	a	b	r	a	b				
$P[1..15]$									a	b	r	a	c	a	b	...	
$P[1..14], P[15]$		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c	
$P[1..15]$															a	b	...
$P[1..15]$		a	b	r	a	c	a	b	a	b	r	a	b	r	a	c	
$P[1..15]$												a	b	r	a	c	...

Abbildung 5.7: Randfunktion und KMP-Fehlerfunktion f_{KMP} für $P[1..15] =$ `abracababrabrac`, mit Beispielen für die Positionierung der entsprechenden Ränder.

- $f_{\text{KMP}}(4) = 1$, weil für den längsten Rand $P[1..1] =$ `a` von $P[1..4] =$ `abra` die zweite Bedingung erfüllt ist ($P[2] =$ `b` $\neq$ `c` $= P[5]$);
- $f_{\text{KMP}}(5) = -1$, weil für den einzigen Rand $P[1..0] = \varepsilon$ von $P[1..5] =$ `abrac` die zweite Bedingung nicht erfüllt ist ($P[1] =$ `a` $= P[6]$);
- $f_{\text{KMP}}(11) = 4$, weil für den längsten Rand $P[1..4] =$ `abra` von $P[1..11]$ die zweite Bedingung erfüllt ist ($P[5] =$ `c` $\neq$ `b` $= P[12]$);
- $f_{\text{KMP}}(14) = 1$, weil für den längsten Rand $P[1..4] =$ `abra` von $P[1..14]$ die zweite Bedingung nicht erfüllt ist ($P[5] =$ `c` $= P[15]$), aber für den zweitlängsten Rand $P[1..1] =$ `a` die zweite Bedingung erfüllt ist ($P[2] =$ `b` $\neq$ `c` $= P[15]$);
- $f_{\text{KMP}}(15) = 5$, weil $m = 15$ und $P[1..5] =$ `abrac` der längste echte Rand des Musters ist.

Ein Verfahren zur effizienten Berechnung einer Tabelle für die KMP-Fehlerfunktion wird später vorgestellt. Wir bemerken aber schon hier, dass die KMP-Fehlerfunktion leicht iterativ aus der gewöhnlichen Randfunktion berechnet werden kann.

q	0	1	2	3	4	5	6
`P[`q`]`		a	b	r	a	c	a
$f_{\mathrm{KMP}}(q)$	-1	0	0	-1	1	-1	1

Abbildung 5.8: KMP-Fehlerfunktion des Musters $P =$ `abraca` ($m = 6$).

Übungsaufgabe. Angenommen, f_{bord} ist (z. B. als Tabelle) gegeben. Man beweise, dass die folgende iterative Berechnungsvorschrift für f_{KMP} korrekt ist:

(1) $f_{\mathrm{KMP}}(0) \leftarrow -1$;
(2) **for** q **from** 1 **to** $m-1$ **do**
  $q' \leftarrow f_{\mathrm{bord}}(q)$; // ein Wert in $\{0, \ldots, q-1\}$
  if $P[q'+1] \neq P[q+1]$ **then** $f_{\mathrm{KMP}}(q) \leftarrow q'$ **else** $f_{\mathrm{KMP}}(q) \leftarrow f_{\mathrm{KMP}}(q')$;
(3) $f_{\mathrm{KMP}}(m) \leftarrow f_{\mathrm{bord}}(m)$.

Wir können den Algorithmus von Knuth, Morris und Pratt („KMP-Algorithmus“) jetzt einfach aufschreiben. Für die veränderlichen Werte q und q' genügt eine Variable `q`, für ℓ und ℓ' eine Variable `l`. Für den Shiftwert s benutzen wir eine Variable `s`.

Algorithmus 5.2.2 (Knuth-Morris-Pratt, anschaulich).

KMP-Textsuche(`P[`$1..m$`]`,`S[`$1..n$`]`)

Eingabe: `P[`$1..m$`]`, `S[`$1..n$`]`; // Muster, Text
Ausgabe: `A` $\subseteq \{1, \ldots, n-m+1\}$;
Vorberechnet: `f_KMP[`$0..m$`]`: KMP-Fehlerfunktion von `P[`$1..m$`]` als Tabelle;

(1) `A` $\leftarrow \emptyset$;
(2) `q` $\leftarrow 0$; `l` $\leftarrow 1$;
(3) **while** `l` $\leq n-m+1$ **do**
(4) **while** `q` $< m \wedge$ `P[q+1]` $=$ `S[l+q]` **do** `q` $\leftarrow$ `q+1`;
(5) **if** `q` $= m$ **then** `A` $\leftarrow$ `A` $\cup \{$`l`$\}$; // Muster gefunden
(6) `s` $\leftarrow$ `q - f_KMP[q]`;
(7) `l` $\leftarrow$ `l + s`;
(8) `q` $\leftarrow$ `q - s`; // d. h. `q` $\leftarrow$ `f_KMP[q]`
(9) **if** `q` $= -1$ **then** `q` $\leftarrow 0$;
(10) **return** `A`.

Beispiel: Muster $P =$ `abraca` ($m = 6$), Text $S =$ `babracababradabrab` ($n = 18$). Die KMP-Fehlerfunktion für P ist in Abb. 5.8 angegeben.

ℓ	S: b	a	b	r	a	c	a	b	a	b	r	a	d	a	b	r	a	b	q	q'	s
1	a	b	r	a	c	a													0	−1	1
2		a	b	r	a	c	a												6	1	5
7							a	b	r	a	c	a							2	0	2
9									a	b	r	a	c	a					4	1	3
12												a	b	r	a	c	a		1	0	1
13													a	b	r	a	c	a	0	−1	1
14														a	...				STOP		

Hellgrau sind die Musterbuchstaben, die erfolgreich verglichen werden, dunkelgrau die Fehlerstellen. Dass in jeder Spalte nur ein hellgrauer Buchstabe steht, bedeutet, dass kein Textbuchstabe mehr als einmal erfolgreich verglichen wird. Im Vergleich zum naiven Algorithmus fehlen viele Zeilen. Für jede der getesteten Positionen $\ell \in \{1, \ldots, n-m+1\}$ gibt es maximal einen Fehlerbuchstaben (das ist klar: mit der Entdeckung der Fehlerstelle ist diese Position erledigt). Im Beispiel werden 16 Buchstabenvergleiche durchgeführt; das naive Verfahren würde 26 Vergleiche benötigen.

Satz 5.2.3. *Algorithmus 5.2.2 ist korrekt und hat Rechenzeit $O(n)$. Die Anzahl der durchgeführten Buchstabenvergleiche ist maximal $2n - m + 1$.*

Beweis: Die Korrektheit folgt aus den Vorüberlegungen. Der Ablauf ist ganz genau wie im naiven Algorithmus, mit dem Unterschied, dass einige ℓ-Werte ausgelassen werden und einige Vergleiche eingespart werden, weil die entsprechenden Gleichheiten schon aus früheren Runden bekannt sind.

Für die Rechenzeit überlegt man sich folgendes. Die große **while**-Schleife (Zeilen (3)–(9)) wird höchstens $(n-m+1)$-mal durchlaufen, weil sich der Inhalt ℓ von `l` in jedem Durchlauf erhöht und bei $\ell > n-m+1$ abgebrochen wird. In jedem Durchlauf dieser großen Schleife gibt es maximal einen Test der Schleife in Zeile (4), der zum Abbruch führt (Fehlerstelle). Also gibt es höchstens $n-m+1$ erfolglose Vergleiche. Es bleiben die Durchläufe durch die Schleife in Zeile (4), in denen die Buchstabenvergleiche erfolgreich sind. Hierfür betrachten wir den Wert $\ell + q$, mit ℓ in `l` und q in `q`. Dieser Wert startet bei 1, wird nie erniedrigt und wird bei jedem „erfolgreichen“ Vergleich im Schleifentest in Zeile (4) echt erhöht; der Algorithmus endet auf jeden Fall, wenn $\ell + q$ größer als n wird. Also kann es nicht mehr als n erfolgreiche Vergleiche und nicht mehr als n solche Schleifendurchläufe geben. □

In Abschnitt 5.2.3 werden wir sehen, wie sich die Randfunktion (und damit auch die KMP-Fehlerfunktion) in Zeit $O(m \cdot |\Sigma|)$ berechnen lässt; in Abschnitt 5.2.4 wird dann ein Algorithmus angegeben, der in Zeit $O(m)$ eine Tabelle für die KMP-Fehlerfunktion f_{KMP} berechnet.

5.2.3 Der KMP-Algorithmus in der Präfixautomaten-Auffassung

Wir haben gesehen, dass der KMP-Algorithmus als eine optimierte Version des naiven Textsuchalgorithmus verstanden werden kann, die das Muster in Sprüngen weiterschiebt und Vergleiche einspart, deren Ergebnis schon bekannt ist. Es ist lehrreich und nützlich, insbesondere für Verallgemeinerungen wie in Abschnitt 5.3, den Algorithmus noch auf eine andere, abstraktere Weise zu betrachten, die einen Bezug zur Verarbeitung von Wörtern durch endliche Automaten[6] herstellt. Als Vorbereitung betrachten wir einen recht einfachen endlichen Automaten. Der eigentliche KMP-Automat ist noch etwas subtiler aufgebaut.

Der volle Präfixautomat

Gegeben sei ein Muster $P = P[1..m]$. Wir konstruieren einen deterministischen endlichen Automaten (DFA) $M^P_{\text{voll}} = (Q, \Sigma, q_0, F, \delta)$. Dieser Automat soll den Text S Buchstabe für Buchstabe von links nach rechts lesen und dabei über den Akzeptierungsmechanismus melden, wenn das Muster gefunden worden ist. Die Zustandsmenge Q repräsentiert die Präfixe $P[1..q]$, $0 \leq q \leq m$, des Musters. Ein solches Präfix wird einfach durch seine Länge dargestellt, also wählen wir

$$Q = Q_P = \{0, 1, \ldots, m\}.$$

Das Alphabet $\Sigma = \Sigma_P$ ist fest; es enthält mindestens die Buchstaben, die in P und S vorkommen.[7] Der Startzustand q_0, die Übergangsfunktion $\delta = \delta_P$ mit $\delta\colon Q \times \Sigma \to Q$ und die Menge $F = F_P$ der akzeptierenden Zustände werden im Folgenden festgelegt.

Die Idee ist folgende: Wir lesen den Text $S[1..n]$ Buchstabe für Buchstabe. Nach Leseschritt i, also nachdem $S[1..i]$ gelesen wurde, soll der Zustand $q^{(i)}$ das *längste Präfix* $P[1..q]$ von P benennen, das Suffix von $S[1..i]$ ist. Dies formulieren wir als Invariante:

($\mathbf{I_i}$): Nach Lesen von $S[1..i]$ ist M^P_{voll} in Zustand

$$q^{(i)} = \max\{q \in Q \mid P[1..q] = S[i - q + 1..i]\}.$$

Wie müssen Startzustand und Übergangsfunktion aussehen, damit stets (I_i) gilt? Am Anfang soll $P[1..q^{(0)}]$ das längste Präfix von P sein, das Suffix von $[1..0] = \varepsilon$ ist; das

[6] s. Vorlesungen „Rechnerorganisation“ und „Automaten, Sprachen und Komplexität“ im Bachelorstudium.

[7] Alle Buchstaben, die in S, aber nicht in P vorkommen, werden gleich behandelt. Daher kann man sie im Automaten wie einen einzigen „Fremdbuchstaben“ behandeln, und ihre Anzahl ist unerheblich.

ist natürlich das leere Wort, und wir legen fest:

$$q_0 := 0.$$

Nun betrachten wir einen beliebigen Schritt $i > 0$ und nehmen an, dass die Konstruktion bislang erfolgreich war, dass also $q^{(i-1)}$ Invariante (I_{i-1}) erfüllt. Buchstabe $S[i]$ wird gelesen. Sei $P[1..q']$ das längste Präfix von P, das Suffix von $S[1..i]$ ist. Wir wollen erreichen, dass der neue Zustand $q^{(i)}$ gerade q' ist. Zunächst einmal ist sicher $P[1..q'-1]$ Suffix von $S[1..i-1]$, also folgt aus (I_{i-1}) die Beziehung $q'-1 \leq q^{(i-1)}$ oder $q' \leq q^{(i-1)}+1$. Die Entscheidung darüber, welche Präfixe von P Suffix von $S[1..i]$ sind, hängt also nur von $S[i-q^{(i-1)}..i]$ ab, und das ist das gleiche wie $P[1..q^{(i-1)}] \circ S[i]$, wobei mit $\circ$ die Konkatenation von Wörtern gemeint ist. Daher sollte $P[1..q']$ das längste Präfix von P sein, das Suffix von $P[1..q^{(i-1)}] \circ S[i]$ ist.

Wir setzen daher für $q \in Q$ und $a \in \Sigma$:

$$\delta(q,a) := \text{Länge des längsten Präfixes von } P, \text{ das Suffix von } P[1..q] \circ a \text{ ist.}$$

Dann gilt $q^{(i)} = \delta(q^{(i-1)}, S[i]) = q'$, wie gewünscht. Wir beobachten gleich noch Folgendes: Diese Definition von δ führt dazu, dass alle Buchstaben $a \in \Sigma$, die nicht in P vorkommen, $\delta(q,a) = 0$ erfüllen. Man braucht diese im Automaten also gar nicht zu unterscheiden, sondern kann sie zu einer Kategorie „Fremdbuchstabe“ zusammenfassen. Damit hängt δ nur von P ab, und δ kann in einer Vorverarbeitungsphase berechnet werden, ohne Kenntnis von S. (Wir werden gleich sehen, wie diese Vorverarbeitung abläuft.)

Der Automat soll nach dem Lesen von $S[1..i]$ genau dann in einem akzeptierenden Zustand sein, wenn soeben ein Vorkommen des Musters fertig gelesen worden ist, das heißt, wenn $S[i-m+1..i] = P[1..m]$ gilt. Nach der Invarianten gilt dies genau dann wenn $q^{(i)} = m$ ist. Daher legen wir fest:

$$F := \{m\}.$$

Mit diesem Automaten kann die Verarbeitung des Textes „in Echtzeit“ erfolgen, also so, dass in jedem Schritt ein Buchstabe des Textes gelesen wird. Offenbar kann man δ als $(m+1) \times |\Sigma|$-Tabelle darstellen, mit einer Zeile für jeden Zustand und einer Spalte für jeden Buchstaben in Σ. Da alle „Fremdbuchstaben“ in den Zustand $q_0 = 0$ führen, braucht man sie in der Tabelle gar nicht darzustellen. Wir zeigen nun, wie man die Tabelleneinträge in Zeit $O(m \cdot |\Sigma|)$ berechnen kann. Dieser Zeit- und Platzaufwand ist dann tolerierbar, wenn $O(m \times |\Sigma|)$ gegenüber der Länge n des Textes nicht groß ist.

Man erinnere sich, wie in Definition 5.1.3 die *Randfunktion* für ein Muster $P[1..m]$ definiert wurde:

$$f_{\mathrm{bord}}(0) = -1, f_{\mathrm{bord}}(q) = \text{Länge des längsten Randes von } P[1..q], \text{ für } 1 \leq q \leq m.$$

Berechnung der Übergangsfunktion und der Randfunktion zu $P[1..m]$:

Wir berechnen f_{bord} und δ verschränkt, durch Induktion über q, in Zeit $O(m \cdot |\Sigma|)$.

- $f_{\text{bord}}(0) \leftarrow -1$.
- $\delta(0, P[1]) \leftarrow 1$; $\delta(0, a) \leftarrow 0$ für $a \neq P[1]$.
- $f_{\text{bord}}(1) \leftarrow 0$. // ε ist Rand von $P[1]$.
- Für $q = 1, 2, \ldots, m-1$ nacheinander:
  $\delta(q, P[q+1]) \leftarrow q+1$; // passender Buchstabe
  $\delta(q, a) \leftarrow \delta(f_{\text{bord}}(q), a)$ für $a \neq P[q+1]$; // unpassender Buchstabe
  $f_{\text{bord}}(q+1) \leftarrow \delta(f_{\text{bord}}(q), P[q+1])$.
- $\delta(m, a) := \delta(f_{\text{bord}}(m), a)$ für $a \in \Sigma$.

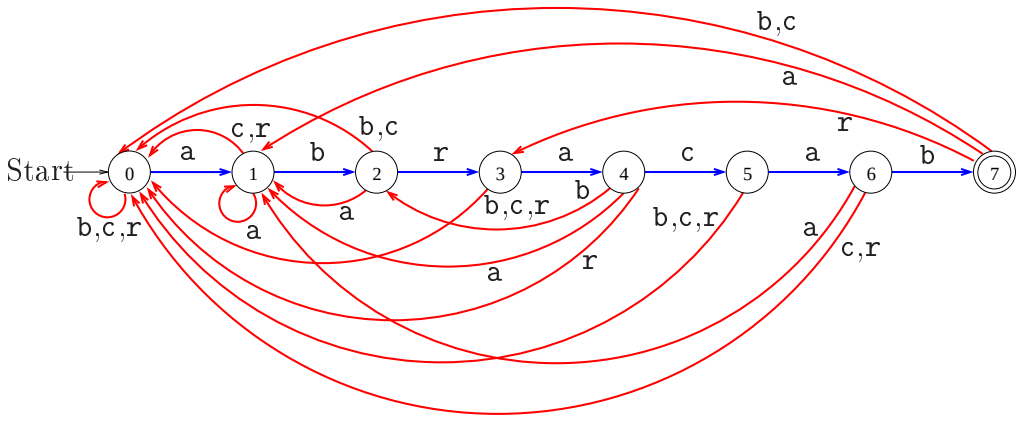

Abbildung 5.9: Der volle Präfixautomat für das Muster `abracab`, mit $m = 7$. (Die Kanten für Fremdbuchstaben führen alle zum Zustand 0; sie sind nicht dargestellt.)

Übungsaufgabe: (a) Führen Sie den Algorithmus am Beispiel $P[1..7] =$ `abracab` durch und vergleichen Sie Ihr Ergebnis mit Abb. 5.9.
(b) Beweisen Sie, dass das Verfahren δ und f_{bord} korrekt berechnet.

Bemerkung: In der Bachelorvorlesung „Automaten, Sprachen und Komplexität“ werden auch nichtdeterministische endliche Automaten (NFAs) behandelt. Man kann ganz leicht einen NFA M angeben, der genau die Wörter akzeptiert, die mit $P = P[1..m]$ enden. Die Zustandsmenge ist Q, der Startzustand ist $q_0 = 0$, die Menge der akzeptierenden Zustände ist $\{m\}$. Die Übergangsfunktion erlaubt es, mit jedem Buchstaben von Zustand 0 zu Zustand 0 zu gehen. Zudem kann man mit Buchstaben $P[q+1]$ von Zustand q in Zustand $q+1$ wechseln, für $0 \leq q < m$. (Die einzige Stelle,

wo „Nichtdeterminismus“ vorkommt, ist im Zustand 0, wo man sich entscheiden kann, ob man beim Lesen von $a = P[1]$ in Zustand 0 bleibt oder nach 1 wechselt.) Wenn man auf diesen NFA die Potenzmengenkonstruktion anwendet, entsteht genau M^P_{voll}. (Dabei gibt es keine Größenexplosion wie sonst bei dieser Konstruktion üblich.)

Der KMP-Präfixautomat

Der große Nachteil des vollen Präfixautomaten M^P_{voll} ist, dass der Platzbedarf und die Zeit für den Aufbau proportional mit $m \cdot \#(\text{Buchstaben in } P)$ wächst. Dies ist allerdings unvermeidlich, wenn in einem Leseschritt für jeden Zustand und jeden Buchstaben eine eigene Aktion vorgeschrieben sein soll. Der KMP-Algorithmus benutzt einen anderen Automaten M^P_{KMP}. Die Zustandsmenge besteht aus $0, 1, \ldots, m$ sowie einem speziellen Zustand -1, dessen Rolle später genauer erklärt wird. Wir setzen also

$$Q := \{-1, 0, 1, \ldots, m-1, m\}.$$

Auch der neue Automat M^P_{KMP} verarbeitet den Text S von links nach rechts. Die „Bedeutung“ (oder der „Informationsgehalt“) der Zustände ist aber gegenüber dem vollen Präfixautomaten abgeändert. Es wird auch zugelassen, dass ein Buchstabe zwar „*inspiziert*“, aber nicht „verbraucht“ wird und damit im nächsten Schritt immer noch verfügbar ist. Hierfür verwenden wir „ε-Züge“.[8]

Im folgenden soll „Buchstabe $S[i]$ wird *gelesen*“ heißen, dass dieser Buchstabe auch „verbraucht“ wird (kein ε-Zug).

Nach wie vor soll Folgendes gelten: Wenn $S[1..i-1]$ gelesen (und verbraucht) worden ist und Buchstabe $S[i]$ als nächstes ansteht, und wenn der Zustand $q \in \{0, \ldots, m-1\}$ ist, dann ist $P[1..q]$ Suffix von $S[1..i-1]$. Daraus ergibt sich, dass es einen „richtigen“ Buchstaben gibt, der in einem solchen Zustand q erwartet wird, nämlich $P[q+1]$. Wenn $S[i]$ dieser richtige Buchstabe ist, werden wir $S[i]$ lesen und in Zustand $q+1$ wechseln. Wenn $S[i] \neq P[q+1]$ ist, gibt es einen ε-Schritt mit einer geeigneten Zustandsänderung. Dieser Ansatz ermöglicht es also, für jeden Zustand q nur noch zwei Alternativen vorzusehen und nicht mehr $|\Sigma|$ viele wie bei M^P_{voll}. Das spart Speicherplatz und Vorberechnungszeit. (Zustände -1 und m sind Sonderfälle. In Zustand m wird *immer* ein „ε-Zug“ ausgeführt, wobei der nächste Buchstabe inspiziert und ignoriert wird; in Zustand -1 wird *immer* der nächste Buchstabe gelesen und verbraucht, wobei es ebenfalls unerheblich ist, was dieser Buchstabe ist.)

Wir wollen Folgendes erreichen: In dem Moment, in dem $S[i]$ gelesen (und verbraucht) wird, geht Automat M^P_{KMP} in einen Zustand $q^{(i)} \in \{0, 1, \ldots, m\}$ über. Für diesen soll dieselbe Invariante wie im vorigen Abschnitt gelten:

[8] ε-Züge kamen in der Vorlesung „Automaten, Sprachen und Komplexität“ bei den Kellerautomaten vor, insbesondere bei den deterministischen Kellerautomaten.

($\mathbf{I_i}$): $P[1..q^{(i)}]$ ist das längste Präfix von P, das Suffix von $S[1..i]$ ist.

In diesen speziellen Schritten verhält sich der neue Automat also exakt wie M^P_{voll}. Nur hat M^P_{KMP} eventuell „Zwischenschritte“, und die Zustände, die in diesen Schritten angenommen werden, haben eine andere, speziellere Bedeutung. Wir formulieren diese „Bedeutung“, d. h., welche Information über $S[1..i]$ im aktuellen Zustand q dargestellt sein soll, wenn der nächste zu verarbeitende Buchstabe $S[i]$ ist, als Invariante:

($\mathbf{Inv_{i,q}}$):
Wenn sich M^P_{KMP} in Zustand q befindet und der nächste zu lesende Buchstabe $S[i]$ ist, dann gilt (s. Abb. 5.10):

> $P[1..q]$ ist Suffix von $S[1..i-1]$ **und**
> für kein $\hat{q} \in \{q+1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i]$.

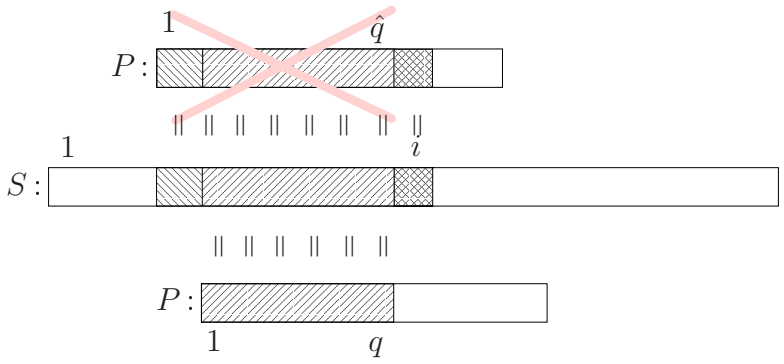

Abbildung 5.10: Illustration von ($\text{Inv}_{i,q}$): $P[1..q]$ ist Suffix von $S[1..i-1]$, und es gibt kein $\hat{q} > q$ mit der Eigenschaft, dass $P[1..\hat{q}+1]$ Suffix von $S[1..i]$ ist.

In Zustand q ist also eventuell Information über $S[i]$ gespeichert! Wie kann das sein, wenn doch $S[i]$ noch gar nicht „gelesen“ worden ist? Nun, in vorangegangenen ε-Zügen kann $S[i]$ schon „inspiziert“ und für Zustandsübergänge verwendet worden sein (sogar mehrfach)!

Die Fälle $q = m$ und $q = -1$ verdienen spezielle Aufmerksamkeit.

- Zustand $q = m$ wird nur erreicht, wenn im selben Moment ein Buchstabe $S[i]$ „gelesen“ wird. Dieser Zustand signalisiert (wegen (I_i)), dass $S[i-m+1..i] = P[1..m]$ gilt, dass also das Muster an Position $i-m+1$ gefunden wurde. In Zustand m gibt es nur eine einzige nächste Aktion: Ohne den nächsten Buchstaben in S anzusehen, erfolgt *immer* ein ε-Übergang zum Zustand $q' = f_{\text{bord}(m)}$, und eine Erhöhung von i um 1. Man sieht leicht, dass dann ($\text{Inv}_{i,q'}$) wieder gilt.

- Wenn Zustand $q = -1$ erreicht wurde, ist die erste Bedingung in ($\text{Inv}_{i,q}$) immer erfüllt (weil $P[1..-1] = \varepsilon$ Suffix von $S[1..i-1]$ ist), und es kommt nur auf die zweite Bedingung an: „Für kein $\hat{q} \in \{0, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i]$." Das heißt, dass kein Präfix $\neq \varepsilon$ von P Suffix von $S[1..i]$ ist. In diesem Fall liest (und verbraucht) man $S[i]$, ohne es weiter zu beachten, erhöht i um 1, und setzt den Zustand q auf 0. Es ist klar, dass ($\text{Inv}_{i,0}$) erfüllt ist.

Wir können nun den Automaten M^P_{KMP} angeben. Die Zustandsmenge Q ist schon festgelegt. Explizit benannt werden nur die Buchstaben von P. In jedem Zustand $q \in \{0, \ldots, m-1\}$ wird der Buchstabe $P[q+1]$ konkret angesprochen; die einzige Alternative ist „$\neq P[q+1]$". Dadurch können vom Automaten auch Buchstaben in S verarbeitet werden, die in P nicht vorkommen. In Zustand -1 wird ein Buchstabe gelesen, aber ignoriert; in Zustand m erfolgt ein ε-Zug ohne Verbrauch eines Zeichens.

Als Startzustand wählen wir $q_0 = 0$. Dann ist die Invariante (Inv_{1,q_0}) erfüllt: $P[1..0] = \varepsilon$ ist Suffix von $S[1..0] = \varepsilon$ und für kein $\hat{q} \in \{1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1]$ (weil $P[1..\hat{q}+1]$ länger als $S[1]$ ist).

Als Menge akzeptierender Zustände wählen wir $F = \{m\}$.

Nun fehlt nur noch die Übergangsfunktion δ. Wir schreiben $\delta(q, a) = q'$, wenn es sich um einen Leseschritt handelt, der Buchstabe a also „verbraucht" wird, und $\delta(q, \langle a \rangle) = q'$, wenn es sich um einen ε-Schritt handelt.

Nehmen wir an, der Automat ist in Zustand $q = q^{(i-1)}$, das nächste zu verarbeitende Zeichen in der Eingabe ist $a = S[i]$, und die Invariante ($\text{Inv}_{i,q}$) gilt. Wir wollen den nun auszuführenden Schritt so festlegen, dass die Invariante weiter gilt. Es gibt einige Fälle.

Fall 1: $0 \leq q < m$ und $P[q+1] = a$. – Nun sollte der Automat den nächsten Buchstaben $S[i]$ *lesen* und im Zustand vermerken, dass ein längeres Präfix von P beobachtet worden ist. Wir setzen also:

$$\delta(q, P[q+1]) = q+1,$$

und damit $q^{(i)} = q+1$. Wegen des ersten Teils von ($\text{Inv}_{i,q}$) gilt $P[1..q+1] = S[i-q..i]$. Wegen des zweiten Teils von ($\text{Inv}_{i,q}$) gilt auch, dass für kein $\hat{q} > q$ die Gleichheit $P[1..\hat{q}+1] = S[i-\hat{q}..i]$ gilt. Das bedeutet, dass $P[1..q+1]$ das *längste* Präfix von P ist, das Suffix von $S[1..i]$ ist, also (I_i). Nun gibt es zwei Unterfälle:
(1a) Wenn $q+1 < m$ ist, dann gilt wegen (I_i) auch:

für kein $\hat{q} \in \{q+2, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i+1]$;

es gilt also die Invariante ($\text{Inv}_{i+1,q+1}$), und wir können im neuen Zustand $q+1$ den nächsten Buchstaben $S[i+1]$ bearbeiten.

(1b) Wenn $q+1=m$ ist, dann ist $P[1..m]$ Suffix von $S[1..i]$. Dies wird im Zustand registriert (mit der Festlegung $F := \{m\}$), aber es erfolgt auch sofort ein Wechsel in einen neuen Zustand, um auf die Bearbeitung von $S[i+1]$ vorzubereiten. Wir suchen das längste echte Präfix $P[1..q']$ von P, das Suffix von $S[1..i]$ ist. Wir können unsere Suche auf Präfixe von P beschränken, die kürzer als m sind und Suffixe von $S[1..i]$ sind. Das sind aber genau die Ränder von P. Hiervon betrachten wir den längsten, und setzen

$$\delta(m, \langle a \rangle) = q', \text{ mit } q' = f_{\text{bord}}(m).$$

(Der nächste Buchstabe $S[i+1]$, falls er existiert, wird ignoriert.) Dann gilt die Invariante ($\text{Inv}_{i+1,q'}$): $P[1..q']$ ist Suffix von $P[1..m]$, also von $S[1..i]$. Für $\hat{q} > q'$ ist $P[1..\hat{q}]$ kein Suffix von $P[1..m]$, also auch keines von $S[1..i]$, also ist $P[1..\hat{q}+1]$ kein Suffix von $S[1..i+1]$. Daher gilt auch jetzt die Invariante ($\text{Inv}_{i+1,q'}$), und wir können im neuen Zustand q' den nächsten Buchstaben $S[i+1]$ bearbeiten.

Fall 2: $0 \leq q < m$ und $P[q+1] \neq a$. – Wegen Invariante ($\text{Inv}_{i,q}$), zweiter Teil, sind die einzigen Präfixe $P[1..q'+1]$ des Musters, die Suffix von $S[1..i]$ sein können, kürzer als $q+1$. Welche q' kommen hierfür in Frage? Es muss $P[1..q'] = S[i-q'..i-1]$, also $P[1..q'] = P[q-q'+1..q]$ gelten (das heißt: $P[1..q']$ ist Rand von $P[1..q]$) und es muss $P[q'+1] = S[i] = a$ gelten, also $P[q'+1] \neq P[q+1]$. Von allen q', die diese Bedingung erfüllen, sollten wir das größte wählen, dann gilt die Invariante ($\text{Inv}_{i,q'}$). Dieses q' hatten wir früher $f_{\text{KMP}}(q)$ genannt.
Achtung: Es könnte $f_{\text{KMP}}(q) = -1$ gelten, was bedeutet, dass es überhaupt keinen Rand $P[1..q']$ von $P[1..q]$ mit $P[q'+1] \neq P[q+1]$ gibt. Daraus folgt aber sofort, dass es kein Präfix $P[1..q'+1]$ von P gibt, das Suffix von $S[1..i]$ ist. Damit ist ($\text{Inv}_{i,-1}$) erfüllt (vgl. auch die obige Diskussion zu Zustand -1). Wir können also in jedem Fall

$$\delta(q, \langle a \rangle) := f_{\text{KMP}}(q)$$

setzen, und die Invariante ($\text{Inv}_{i,q'}$) gilt.

Fall 3: $q = -1$. – Wir haben oben schon gesehen, dass die Invariante ($\text{Inv}_{i,-1}$) impliziert, dass kein Präfix von P Suffix von $S[1..i]$ sein kann. Wir lassen den Automaten den Buchstaben $S[i]$ lesen und verbrauchen, und in Zustand 0 gehen:

$$\delta(-1, a) = 0.$$

Ist die Invariante ($\text{Inv}_{i+1,0}$) erfüllt? Tatsächlich ist $P[1..0] = \varepsilon$ Suffix von $S[1..i]$ und für kein $\hat{q} \in \{1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i+1]$ (weil $P[1..\hat{q}]$ kein Suffix von $S[1..i]$ ist). Außerdem gilt (I_i) mit $q^{(i)} = 0$, wie gewünscht, weil $P[1..0] = \varepsilon$ das längste Präfix von P ist, das Suffix von $S[1..i]$ ist.

In Abb. 5.11 ist der entsprechende Automat für das Muster $P[1..7] =$ `abracab` dargestellt. Man beachte, dass zur Herstellung des Automaten im Wesentlichen die Fehlerfunktion f_{KMP} des Musters benötigt wird.

Wir fassen zusammen: $M^P_{\text{KMP}} = (Q, \Sigma, q_0, F, \delta)$, ein deterministischer ε-Automat für das Muster $P[1..m]$, ist durch die folgenden Komponenten gegeben. Dabei kann der Automat auch Buchstaben verarbeiten, die nicht in Σ liegen (so genannte „Fremdbuchstaben“).

- $Q = \{-1, 0, 1, \ldots, m\}$.
- $\Sigma = \{a \mid \text{Buchstabe } a \text{ kommt in } P[1..m] \text{ vor}\}$.
- $q_0 = 0$.
- $F = \{m\}$.
- $\delta(-1, a) = 0$, für beliebige Buchstaben a.

  Für $0 \leq q < m$: $\delta(q, a) = \begin{cases} q+1 & \text{für } a = P[q+1], \\ f_{\text{KMP}}(q) & \text{für beliebiges } a \neq P[q+1]. \end{cases}$

  $\delta(m, \langle a \rangle) = f_{\text{bord}}(m)$, für beliebige Buchstaben a.

Lemma 5.2.4. *Wann immer M^P_{KMP} in einem Zustand q ist und der nächste zu lesende Buchstabe $S[i]$ ist, gilt die Invariante* ($\text{Inv}_{i,q}$).

Beweis: Wir haben M^P_{KMP} so konstruiert, dass die Invariante ($\text{Inv}_{i,q}$) ständig gilt. □

Wir müssen noch zeigen, dass die Invariante stark genug ist, um zu erzwingen, dass der Automat alle Vorkommen von P erkennt.

Lemma 5.2.5. *P ist Suffix von $S[1..i]$ genau dann wenn der Automat M^P_{KMP} unmittelbar nach dem Lesen von $S[i]$ in Zustand m ist.*

Beweis: „⇒“: Sei P Suffix von $S[1..i]$. Betrachte die Situation unmittelbar bevor $S[i]$ gelesen wird. Der Automat ist in einem Zustand $q < m$. Wegen der Invarianten ($\text{Inv}_{i,q}$) gilt: $P[1..q] = S[i-q..i-1]$ und für kein $\hat{q} \in \{q+1, \ldots, m-1\}$ ist $P[1..\hat{q}+1]$ Suffix von $S[1..i]$. Nun gilt aber, dass $P[1..m]$ Suffix von $S[1..i]$ ist. Dies entspricht dem Wert $\hat{q} = m-1$, und die einzige Möglichkeit dafür, dass die Invariante gilt, ist, dass $m-1 \notin \{q+1, \ldots, m-1\}$ ist. Das bedeutet, dass $q+1 > m-1$ ist, also $q = m-1$ ist. Da $S[i] = P[m]$, trifft Fall 1 zu, und der Automat wechselt in den Zustand m.

„⇐“: Nehmen wir an, der Automat geht beim Lesen von $S[i]$ in Zustand m. Das kann nur passieren, wenn Fall 1 zutrifft, der vorige Zustand $m-1$ war und der gelesene Buchstabe $S[i]$ gleich $P[m]$ ist. Nach der Invarianten ($\text{Inv}_{i,m-1}$) gilt auch $S[i-m+1..i-1] = P[1..m-1]$. Damit ist P Suffix von $S[1..i]$. □

In Abb. 5.12 ist die Arbeitsweise des Automaten an einem Beispiel angegeben. Wir verfolgen die Zustände des Automaten anhand der Präfixe selbst. Von Zeile zu Zeile

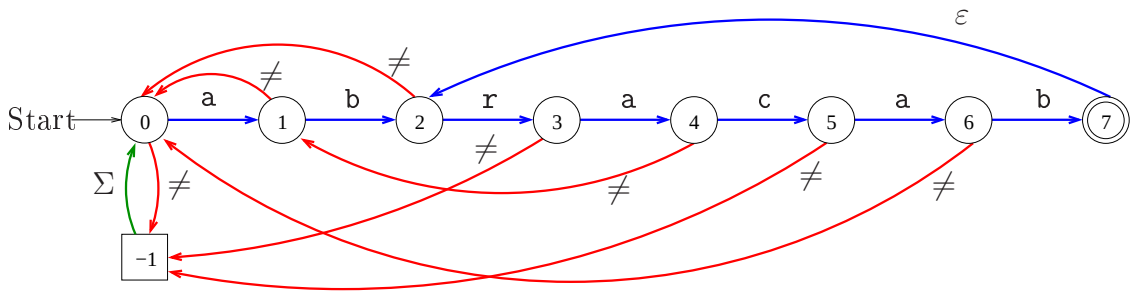

Abbildung 5.11: Der KMP-Präfixautomat für das Muster `abracab`, mit $m = 7$. Die Zustandsmenge ist $\{-1, 0, \ldots, m\}$. Dabei entsprechen die Zustände $0, \ldots, m$ den $m + 1 = 8$ Präfixen $P[1..q]$ des Musters, Zustand -1 spielt eine Sonderrolle. Man beginnt in Zustand 0. Der Text $S[1..n]$ wird Buchstabe für Buchstabe gelesen. Die blauen Zustandsübergänge bestimmen, was bei erfolgreichen Vergleichen passiert. Von Zustand $q \in \{0, \ldots, m-1\}$ gelangt man unter Lesen des Buchstabens $P[q+1]$ in den Zustand $q+1$, von Zustand $q = m-1$ gelangt man unter Lesen des Buchstabens $P[m]$ zunächst in den akzeptierenden Zustand m (Ausgabe „Muster gefunden“) und dann, ohne einen Buchstaben anzusehen, in den Zustand $f_{\mathrm{KMP}}(m)$ (hier: 2). Die roten Zustandsübergänge („*mismatch*“) bestimmen das Verhalten, wenn in Zustand $q \in \{0, \ldots, m-1\}$ ein Buchstabe $\neq P[q+1]$ gesehen wird. Ohne diesen Buchstaben zu verbrauchen, also mit einem ε-Übergang, wird in den Zustand $f_{\mathrm{KMP}}(q)$ gegangen. Im Zustand -1 wird der nächste Buchstabe im Text gelesen und es wird in den Zustand 0 gewechselt (grüner Übergang).

wächst das Suffix $P[1..q]$ entweder um einen Buchstaben an oder es wird verkürzt, manchmal auf Länge 0, manchmal auf eine Länge > 0. Im Beispiel tritt die Bedingung von Lemma 5.2.5 für $i = 12$ ein.

Schritt \ Text		a	b	c	a	b	a	b	r	a	c	a	b	r	a	b	a	b	r
i		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	ε																		
$0 \to 1$	a																		
$1 \to 2$	a	b																	
$2 \xrightarrow{\varepsilon} 0$		ε																	
$0 \to -1 \xrightarrow{\varepsilon} 0$			ε																
$0 \to 1$				a															
$1 \to 2$				a	b														
$2 \xrightarrow{\varepsilon} 0$					ε														
$0 \to 1$						a													
$1 \to 2$						a	b												
$2 \to 3$						a	b	r											
$3 \to 4$						a	b	r	a										
$4 \to 5$						a	b	r	a	c									
$5 \to 6$						a	b	r	a	c	a								
$6 \to 7$						a	b	r	a	c	a	b							
$7 \xrightarrow{\varepsilon} 2$											a	b							
$2 \to 3$											a	b	r						
$3 \to 4$											a	b	r	a					
$4 \xrightarrow{\varepsilon} 1$														a					
$1 \to 2$														a	b				
$2 \xrightarrow{\varepsilon} 0$																ε			
$0 \to 1$																	a		
$1 \to 2$																	a	b	
$2 \to 3$																	a	b	r

Abbildung 5.12: *Beispiel* für die Arbeitsweise des KMP-Präfixautomaten. Muster: $P[1..7] =$ `abracab`; Text: $S[1..18] =$ `abcababracabrababr`.

Programmtechnische Umsetzung: Der Algorithmus verläuft in Runden $i = 1, 2, \ldots, n$. In Runde i wird Buchstabe $S[i]$ des Textes eventuell mehrmals inspiziert (Fall 2) und schließlich gelesen (Fall 1). Wenn bei Betrachten von $S[i]$ das Muster gefunden wird und man Zustand m betritt, wird die Aktion aus Fall 3, also der ε-Schritt zu Zustand $f_{\mathrm{KMP}}(m)$, noch im gleichen Schleifendurchlauf ausgeführt. Die Runden werden über eine (äußere) **for**-Schleife realisiert. Der Inhalt der Schleifenvariablen `i` gibt dabei immer die aktuelle Runde an. Der aktuelle Zustand q wird in einer Variablen `q` gehalten. Die ε-Übergänge im *mismatch*-Fall (Fall 2) finden in einer inneren **while**-Schleife statt. Wenn diese Schleife in den Zustand $q = -1$ führt, wird die in Fall 3 vorgesehene Erhöhung von `q` auf 0 ausgeführt.

Text		c	a	b	a	b	r	a	d	a	c	a	r
i	0	1	2	3	4	5	6	7	8	9	10	11	12
$q_7 = 4$					a	b	r	a	c	...			
$q'_1 = 1$								a	b	...			
$q'_2 = 0$									a	...			
$q'_3 = -1$										a	...		

Abbildung 5.13: Beispiel für wiederholte ε-Übergänge im *mismatch*-Fall: Wir suchen im Text `cababradacar` nach dem Muster `abraca` (für die KMP-Fehlerfunktion dieses Musters s. Abb. 5.8). Es ist $q^{(7)} = 4$. Nach einigen ε-Zügen durch Zustände q'_1, q'_2, q'_3 findet man $q^{(8)} = q'_3 + 1 = 0$.

Man beachte (und staune über) die extrem kompakte Umsetzung als Programm.

Algorithmus 5.2.6 (Knuth-Morris-Pratt)**.**
KMP-Textsuche(`P[1..`m`]`,`S[1..`n`]`)
Eingabe: `P[1..`m`]`, `S[1..`n`]` // Muster, Text
Ausgabe: `A` $\subseteq \{1, \ldots, n - m + 1\}$;
Vorberechnen: `f_KMP[0..`m`]`: KMP-Fehlerfunktion von `P[1..`m`]` als Tabelle;

```
(1)   A ← ∅;
(2)   q ← 0;
(3)   for i from 1 to n do
(4)     while q ≥ 0 ∧ P[q+1] ≠ S[i] do q ← f_KMP[q];
(5)     q ← q+1;
(6)     if q = m
(7)       then A ← A ∪ {i-m+1}; q ← f_KMP[m];
(8)   return A.
```

Wo findet man die Runden und die Fälle im Programm? Die Initialisierung, Runde 0, besteht aus der Zuweisung `q` $\leftarrow 0$, und $q_0 = 0$ (Zeile (2)). Der Zustand q steht stets in der Variablen `q`. Runde i ist der Durchlauf der **for**-Schleife, in dem `i` die Zahl i enthält. Hier wird Buchstabe $S[i]$ behandelt. Man geht davon aus, dass $q < m$ gilt. Die Bedingung der **while**-Schleife ist der Test darauf, ob Fall 2 eintritt. Solange dies so ist, wird q durch $f_{\mathrm{KMP}}[q]$ ersetzt. Wenn die Schleife endet, ist einer von zwei Fällen eingetreten.

(i) $q = -1$: Dies ist Fall 3. Die Zuweisung in Zeile (4) stellt den Zustand $q = 0$ ein. Der Test in Zeile (6) ergibt *false*. Man geht zur nächsten Runde über.

(ii) $q \geq 0$ und $P[q + 1] = S[i]$: Dies ist Fall 1. Zeile (4) bewirkt die nötige Erhöhung von `q` um 1. Die Aktionen für Fall 1b werden unmittelbar angeschlossen (Zeilen (6)-(7)). Wenn $q = m$ gilt, wurde das Muster gefunden, und man kann den Startindex

$i - m + 1$ in der Menge A vermerken. Der ε-Übergang von Fall 1b wird ebenfalls in Zeile (7) durchgeführt.

Wir fassen die Eigenschaften des Algorithmus von Knuth, Morris und Pratt zusammen.

Satz 5.2.7. (a) *Algorithmus 5.2.6 liefert in* `A` *alle Positionen ℓ in $S[1..n]$ zurück, an denen das Muster $P[1..m]$ vorkommt.*

(b) *Die Rechenzeit von Algorithmus 5.2.6 ist $O(n)$. Es werden nicht mehr als $2n$ Buchstabenvergleiche durchgeführt.*

Beweis: Die Korrektheit folgt aus der Vorüberlegung, insbesondere Lemma 5.2.5. Für die Anzahl der Vergleiche betrachten wir die Zahlen i und q in `i` und `q`. Für jeden erfolgreichen Vergleich mit Ergebnis $P[q+1] = S[i]$ steigt i um 1 (man geht zur nächsten Runde); dies kann nicht öfter als n-mal geschehen. Für jeden erfolglosen Vergleich (Ergebnis $P[q+1] \neq S[i]$) steigt die Zahl $i - q$ strikt an (man ersetzt q durch $f_{\mathrm{KMP}}(q) < q$); diese Zahl startet mit dem Wert 1, sie sinkt nie und sie kann nicht größer als $n + 1$ werden. Also gibt es nicht mehr als n erfolglose Vergleiche. □

Wenn man die beiden Versionen 5.2.2 und 5.2.6 des KMP-Algorithmus vergleicht, erkennt man, dass in beiden genau die gleichen Buchstabenvergleiche stattfinden, es sich also tatsächlich im Wesentlichen um denselben Algorithmus handelt. Der erste orientiert sich, wie der naive Algorithmus, an einer Position ℓ des Musters im Text. Dieses ℓ wird in 5.2.2 in der äußeren Schleife mitgeführt und in Sprüngen erhöht. Version 5.2.6 löst sich von dieser Vorstellung. In der äußeren Schleife geht es um die Verarbeitung von Buchstaben von S, mit dem kontinuierlich laufenden Index i, wie bei Automaten üblich. Nur in der inneren Schleife ist mit der sprungweisen Verringerung von q noch eine Ahnung vom Verschieben des Musters nach rechts übrig.

5.2.4 Die Berechnung der Randfunktion und der KMP-Fehlerfunktion

Es bleibt noch zu zeigen, wie man zu einem gegebenen Muster $P[1..m]$ in Zeit $O(m)$ eine Tabelle für die KMP-Fehlerfunktion f_{KMP} (Definition 5.2.1) berechnen kann. Es ist nützlich, hierfür zunächst die Berechnung der Randfunktion f_{bord} (Definition 5.1.3) zu betrachten.

Nach Definition ist $f_{\mathrm{bord}}(0) = -1$ und $f_{\mathrm{bord}}(i) =$ die Länge des längsten Randes von $P[1..i]$, für $0 \leq i \leq m$. Die Grundidee für die Berechnung von f_{bord} ist, den KMP-Automaten mit $P[1..m]$ als Text ablaufen zu lassen. Das ist natürlich ziemlich langweilig, weil man nur feststellt, dass das Muster genau einmal vorkommt. Dieses

„triviale“ Vorkommen können wir abschalten, indem wir $P[1]$ im Text durch einen „Fremdbuchstaben“ $* \notin \Sigma$ ersetzen. Wir lassen also den KMP-Automaten auf Eingabe $P^*[1..m] = * \circ P[2..m]$ ablaufen.

Wir betrachten die angepasste Version der Invarianten (I_i) aus dem vorigen Abschnitt. Für $1 \leq i \leq m$ gilt:

(I_i^*) Nach dem Lesen von $P^*[i]$ ist der KMP-Automat in Zustand $q^{(i)}$, wobei $P[1..q^{(i)}]$ das längste Präfix von P ist, das Suffix von $P^*[1..i]$ ist.

Das längste Präfix von P, das Suffix von $P^*[1..i] = * \circ P[2..i]$ ist, ist aber genau der längste Rand von $P[1..i]$! Unsere Invariante verkürzt sich zu:

(I_i^*) Nach dem Lesen von $P^*[i]$ ist der KMP-Automat in Zustand $f_{\mathrm{bord}}(i)$.

Aha! Wenn wir f_{KMP} haben, können wir den KMP-Automaten ablaufen lassen und die Werte der Randfunktion ablesen.

Wie schon in Abschnitt 5.2.2 (Seite 12, Übungsaufgabe) bemerkt, können wir dann leicht die Fehlerfunktion f_{KMP} berechnen, wie folgt:

(i) $f_{\mathrm{KMP}}(0) \leftarrow -1$.

(ii) Für $i = 1, \ldots, m-1$ tue nacheinander:

$$q' \leftarrow f_{\mathrm{bord}}(i) \quad // \text{ ein Wert in } \{0, \ldots, i-1\}.$$
$$f_{\mathrm{KMP}}(i) \leftarrow \begin{cases} q' & , \text{ wenn } P[i+1] \neq P[q'+1], \\ f_{\mathrm{KMP}}(q') & , \text{ wenn } P[i+1] = P[q'+1]. \end{cases}$$

(iii) $f_{\mathrm{KMP}}(m) \leftarrow f_{\mathrm{bord}}(m)$.

Es ist leicht zu sehen, dass man mit dieser Methode in Kombination mit einem Verfahren, das nacheinander $f_{\mathrm{bord}}(0), f_{\mathrm{bord}}(1), \ldots$ liefert, unmittelbar nach der Berechnung von $f_{\mathrm{bord}}(i)$ auch $f_{\mathrm{KMP}}(i)$ ausrechnen kann.

Leider ist das ganze Verfahren zirkulär, weil man zur Berechnung von f_{KMP} eine Tabelle von f_{KMP} benötigt. Oder nicht? Man beobachtet, dass bei der Bearbeitung des Buchstabens $P^*[i]$ in der Eingabe (ε-Schritte oder Leseschritt) der aktuelle Zustand q kleiner als i sein muss. (Nach dem Lesen von $P^*[i-1]$ ist der Zustand $q = f_{\mathrm{bord}}(i-1) < i-1$. Während ε-Schritten auf $P^*[i]$ wird q höchstens kleiner; beim Lesen von $P^*[i]$ kann q höchstens um 1 wachsen, ist also immer noch $< i$.) Daher können wir den KMP-Automaten *gleichzeitig aufbauen und benutzen*, da in jedem Schritt der benötigte Teil von f_{KMP} schon bekannt ist.

Der Ablauf aus Automatensicht, der Arrays `f_bord`$[0..m]$ und `f_KMP`$[0..m]$ beschriftet und liest, ist wie folgt.

- `f_bord`$[0] \leftarrow -1$; `f_KMP`$[0] \leftarrow -1$;
- Startzustand: $q \leftarrow 0$.
- ε-Schritt mit $P^*[1] = *$ führt in Zustand $q = -1$.
- Lese $P^*[1]$, gehe in Zustand $q = q^{(1)} = 0$.
- `f_bord`$[1] \leftarrow 0$; `f_KMP`$[1] \leftarrow \left\{ \begin{array}{ll} 0 & \text{, wenn } P[2] \neq P[1], \\ -1 & \text{, wenn } P[2] = P[1]. \end{array} \right.$.
- Setze $i \leftarrow 2$.
- Wiederhole, bis `f_bord`$[m]$ und `f_KMP`$[m]$ berechnet sind:

  Fall 1: $q \geq 0$ und $P[i] = P[q+1]$:

  erhöhe q um 1; `f_bord`$[i] \leftarrow q$;

  `f_KMP`$[i] \leftarrow \left\{ \begin{array}{ll} q & \text{, wenn } i = m \text{ oder } P[i+1] \neq P[q+1], \\ \texttt{f\_KMP[}q\texttt{]} & \text{, sonst.} \end{array} \right\}$;

  erhöhe i um 1;

  Fall 2: $q \geq 0$ und $P[i] \neq P[q+1]$: Setze q auf `f_KMP`$[q]$.

  Fall 3: $q = -1$:

  setze q auf 0; `f_bord`$[i] \leftarrow q$;

  `f_KMP`$[i] \leftarrow \left\{ \begin{array}{ll} 0 & \text{, wenn } i = m \text{ oder } P[i+1] \neq P[1], \\ -1 & \text{, sonst.} \end{array} \right\}$;

  erhöhe i um 1;

Fall (1b) aus dem KMP-Algorithmus („Muster P in Text P^* gefunden“) kann nicht vorkommen.

Aus der Vorüberlegung folgt, dass dieser Automat die Tabellen `f_bord`$[0..m]$ und `f_KMP`$[0..m]$ korrekt berechnet.

Wir können den Ablauf genau wie beim KMP-Algorithmus ganz leicht in ein Programm umsetzen. Der Buchstabe $P^*[1] = *$ muss natürlich nicht benannt werden; es genügt, den Automaten anfangs in den Zustand -1 zu setzen, damit das Zeichen $P[1]$ als „Fremdzeichen“ gelesen wird. Eine Sonderbehandlung für $i = 1$ ist dann nicht nötig. Die Aktionen für Fall (1b) fallen weg; nach einem erfolgreichen Vergleich

Abbildung 5.14: Randfunktion f_{bord} und KMP-Fehlerfunktion f_{KMP} bei $P[1..12] =$ `abababcababa`:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
`P[`i`]`		a	b	a	b	a	b	c	a	b	a	b	a
$f_{\text{bord}}(i)$	-1	0	0	1	2	3	4	0	1	2	3	4	5
$f_{\text{KMP}}(i)$	-1	0	-1	0	-1	0	4	-1	0	-1	0	-1	5

„$P[i] = P[q+1]$“ wird in beiden Arrays ein neuer Eintrag berechnet. Fälle 1 und 3 können identisch behandelt werden, genau wie im KMP-Algorithmus.

Algorithmus 5.2.8 (Berechnung der Randfunktion und der KMP-Fehlerfunktion)**.**
KMP-Rand-Preprocessing(`P[`$1..m$`]`)
Eingabe: `P[`$1..m$`]`: Muster;
Ausgabe: `f_bord[`$0..m$`]`: Randfunktion als Tabelle;
`f_KMP[`$0..m$`]`: KMP-Fehlerfunktion als Tabelle;

```
(1)   f_bord[0] ← −1; f_KMP[0] ← −1;
(2)   q ← −1;
(3)   for i from 1 to m do
(4)     while q ≥ 0 ∧ P[q+1] ≠ P[i] do q ← f_KMP[q];
(5)     q ← q+1;
(6)     f_bord[i] ← q;
(7)     if i = m or P[q+1] ≠ P[i+1]
(8)       then f_KMP[i] ← q
(9)       else f_KMP[i] ← f_KMP[q];
(10)  return f_bord[0..m], f_KMP[0..m].
```

Ein Beispielergebnis ist in Abb. 5.14 dargestellt. Man prüfe den Ablauf des Algorithmus anhand dieses Beispiels und vergewissere sich, dass nach dem Lesen von `P[`i`]` und Erhöhen von `q` stets die Invariante (I_i^*) gilt.

Satz 5.2.9. *Algorithmus 5.2.8 berechnet f_{bord} und f_{KMP} korrekt und in Zeit $O(m)$.*

Beweis: Wie beim KMP-Algorithmus selbst überprüft man leicht, dass der Algorithmus nichts anderes tut als den KMP-Automaten auf $P^*[1..m]$ ablaufen zu lassen. Dabei werden die Werte $q^{(i)}$ ausgelesen und als $f_{\text{bord}}(i)$ eingetragen (Zeile (6)); weiter wird $f_{\text{KMP}}(i)$ ausgerechnet und eingetragen (Zeilen (7)–(9)). Die Rechenzeitanalyse ist identisch zu der des KMP-Algorithmus selbst. □

Bemerkung: Man kann sich überlegen, dass der KMP-Algorithmus auch dann korrekt abläuft, wenn man anstelle der KMP-Fehlerfunktion die Randfunktion benutzt.

Das führt zu einer Abwandlung von Algorithmus 5.2.8, in der nur die Randfunktion berechnet wird. Alternativ kann man in Algorithmus 5.2.8 auf die explizite Darstellung und Ausgabe der Randfunktion verzichten und nur die KMP-Fehlerfunktion berechnen. Dies führt zu folgendem wiederum sehr kompakten Algorithmus, der in linearer Zeit die KMP-Fehlerfunktion berechnet:

Algorithmus 5.2.10 (Berechnung der KMP-Fehlerfunktion)**.**
KMP-Preprocessing(`P[1..m]`)
Eingabe: `P[1..m]`: Muster;
Ausgabe: `f[0..m]`: KMP-Fehlerfunktion als Tabelle;

```
(1)   f[0] ← −1;
(2)   q ← −1;
(3)   for i from 1 to m do
(4)     while q ≥ 0 ∧ P[q+1] ≠ P[i] do q ← f[q];
(5)     q ← q+1;
(6)     if i = m or P[q+1] ≠ P[i+1]
(7)       then f[i] ← q
(8)       else f[i] ← f[q];
(9)   return f[0..m].
```