Initialisierung der Verschiebefunktionen zur Mustersuche in Texten

Bernhard Ziegler

Institut für Informatik, Universität Stuttgart
Breitwiesenstr. 20-22, D-70565 Stuttgart
Telefax: 0711/7816-370
E-mail: bziegler@informatik.uni-stuttgart.de

Zusammenfassung. Die schnellsten Algorithmen zur Mustersuche in Texten sind Varianten des genialen Algorithmus BoMo von Boyer und Moore. Wenn Text und Muster nicht zusammenpassen, verwendet BoMo Tabellen, um die größten zulässigen Verschiebungen zu ermitteln. Eine effiziente Berechnung dieser Tabellen wurde von Knuth angegeben.

Hier wird sowohl auf die den Algorithmen KMP von Knuth, Morris und Pratt, BoMo und ESS zugrundeliegenden Ideen eingegangen, als auch die Implementierung ihrer Verschiebetabellen detailliert beschrieben.

Schlüsselwörter: Wortsuche, Mustererkennung, Boyer-Moore

Summary. The fastest known algorithms for pattern matching in strings are derivatives of the ingenious algorithm BoMo of Boyer and Moore. On mismatch tables are used by BoMo to ascertain the largest possible pattern shifts. An efficient scheme of computing these tables was given by Knuth.

In this report we recapitulate the key ideas underlying the algorithms KMP of Knuth, Morris and Pratt, BoMo, and ESS, as well as presenting the implementation of their shift tables in detail.

Key words: String Searching, Pattern Matching, Boyer-Moore

Computing Reviews Classification: F.2.2, H.3.3

1. Einleitung

Die Mustersuche in Texten ist nach wie vor eine bedeutende Aufgabe der Datenverarbeitung. Wir haben in [10] einen Algorithmus ESS vorgestellt, der bei Genetischem Code die schnellsten Varianten von BoMo, einem von Boyer und

Moore in [4] publizierten Algorithmus, bis zu einem Faktor 3 übertrifft. Die Initialisierung der Verschiebefunktion $AA0$ von `ESS` (Effiziente String Suche) ist umfangreich. Da in [10] nicht darauf eingegangen wurde, soll sie hier dargestellt werden. Wir beschreiben aber nicht nur allein die Initialisierung von $AA0$, sondern beschäftigen uns mit allen Verschiebefunktionen der angegebenen Mustersuchalgorithmen.

Im Abschnitt „Nomenklatur“ wird die verwendete Notation eingeführt. Im Kapitel „Bausteine der Algorithmen zur Mustersuche in Texten“ wird am Beispiel des naiven Algorithmus `SPM` gezeigt, wie man die verschiedenen Suchverfahren aus geeigneten voneinander (in den meisten Fällen) unabhängigen Bausteinen zusammensetzen kann.

In den drei sich anschließenden Kapiteln werden die Algorithmen `KMP` von Knuth, Morris und Pratt [7], `BoMo` von Boyer und Moore [4] und `ESS` von Ziegler [10] und ihre Verschiebefunktionen eingehend behandelt. Im Vordergrund steht dabei der Versuch, die Verschiebetabelle $Next$ von `KMP` verständlich darzustellen, denn auf ihr basieren die Tabellen D und $D0$ von `BoMo` und DD und $DD0$ von `ESS` .

Auch auf die Berechnung der Tabelle $AA0$ von `ESS` wird detailliert eingegangen; zum einen, weil in [10] nur ihre Definition, nicht aber ihre Implementierung angegeben wurde, zum anderen, weil von ihrer effizienten Implementierung die Konkurrenzfähigkeit von `ESS` wesentlich abhängt.

2. Nomenklatur

Im folgenden wird angenommen, daß der zu durchsuchende Text der Länge Tl in einem Feld $Text\,[*]$ und das zu suchende Muster der Länge Pl in einem Feld $Pat\,[*]$ geeigneter Größe ab Position 1 gespeichert sind. Im Text sollen jeweils alle sich nicht überlappenden Exemplare des Musters gesucht werden. Wird ein Muster gefunden, so wird dies durch den Aufruf einer vom Benutzer frei gestaltbaren Prozedur `Locout(Position)` angezeigt. Ihr wird als Parameter die Position des Musters im Text mitgegeben.

Die an einigen Stellen verwendete Notation $I \ldots J$ bedeutet: alle Indizes K von I bis J ($I \leq K \leq J$). So steht z.B. $Pat\,[Pl-1 \ldots Pl\,] = Text\,[Ke-1 \ldots Ke\,]$ statt $Pat\,[Pl-1] = Text\,[Ke-1]$ & $Pat\,[Pl\,] = Text\,[Ke\,]$.

Mit $K0$ und Ke werden die Stellen im Text indiziert, die $Pat\,[0]$ und $Pat\,[Pl\,]$ gegenüberliegen. Wird auf ein Zeichen im Text zum ersten Mal zugegriffen, so sprechen wir von „aufdecken“. Nur aufgedeckte Zeichen sind bekannt. Nicht aufgedeckte Zeichen werden in unseren Skizzen durch „*“ symbolisiert. Gilt an einer Stelle $Text\,[K] \neq Pat\,[J]$, so sprechen wir davon, daß die Zeichen „nicht passen“ und daher der „Vergleich gescheitert ist“. In den Skizzen liegen die Zeichen, die miteinander verglichen werden sollen, übereinander. Über dem einen Vergleichszeichen steht „=“ oder „$\neq$“, unter dem Partner „^“. „^“ bedeutet ferner in den meisten Fällen, daß auf die beiden Partner beim letzten Schritt zugegriffen und der Vergleich ausgeübt wurde. Alle Zeichen des Textes, die nicht im Muster enthalten sind, bewirken an der jeweils gleichen Position über dem Muster dieselbe Verschiebung. Sie werden durch „$“ symbolisiert.

In redundante runde Klammern gesetzte Ausdrücke in Programmstücken könnten jeweils als Invariante einer eigenen Variablen zugewiesen werden, so daß sie nur einmal zu berechnen sind, oder sie ließen sich in eine Funktionsdefinition

mit aufnehmen. Auf diese Zusammenfassungen haben wir zum Teil, der besseren Verständlichkeit wegen, bei der Darstellung der Algorithmen verzichtet.

Die Verschiebefunktionen, die von den Mustersuchalgorithmen verwendet werden, sind in Tabellen gespeichert. Deshalb werden die Notationen $F(x)$ und $F[x]$ synonym zueinander verwendet.

Wir nehmen an, daß in der Regel das Muster zusätzlich an den Text als Bremse angehängt wurde, so daß die Bedingung „Textende überschritten“ nur dann geprüft werden muß, wenn ein Muster erkannt wurde. Das heißt, die Zahl der benötigten Tests wird stark reduziert, indem man sich darauf beschränkt, nur dann auf Textende zu prüfen, wenn ein Muster gefunden wurde. Dafür muß das Muster einmal zusätzlich erkannt, aber nicht gemeldet werden.

3. Die Bausteine der Algorithmen zur Mustersuche in Texten

Der naive Algorithmus `SPM` (Simple Pattern Matching) beginnt die Suche in Text und Muster von links nach rechts. Sobald ein Zeichen des Musters nicht zum Text paßt, wird das Muster um eine Stelle über den Text nach rechts verschoben, und der nächste Vergleich beginnt wieder links im Muster. Bei natürlichen Sprachen scheitert der Vergleich meist schon am ersten Zeichen. Deshalb ist es zweckmäßig, diesen Fall gesondert zu behandeln und den Algorithmus, wie später alle anderen auch, aus den unabhängigen Bausteinen *Skip-Schleife*, *Test-* und *Shift-Teil* zusammenzusetzen. Wenn sich die Häufigkeiten der Zeichen des Textes wesentlich voneinander unterscheiden, ist es sinnvoll, den Vergleich nicht mit dem ersten, sondern mit dem im Text am seltensten vorkommenden Zeichen des Musters zu beginnen. Steht dieses Zeichen an der Position P, so ergibt sich `SPM` mit Bremse als:

```
SPMF (* SPM with frequency *)

   KF := P; FPat := Pat[P]      (* FPat das seltenste Zeichen *)

   loop

      (* Skip Loop *)
      while Text[KF] # FPat do KF := KF + 1 od;

      K := KF - (P - 1);  J := 1;

      (* Test Part *)
      while (Text[K] = Pat[J]) & (J <= Pl)
       do K := K + 1; J := J + 1 od;

      if J >= Pl
       then (* pattern matched *)
    if K > (Tl+1) then (* text exhausted -> *) exit fi;
         Locout (K-Pl);

      (* Shift Part *)
         KF :+ KF + Pl;
       else KF := KF + 1;
       fi;
    end loop;
 end (* SPMF *);
```

4. KMP, der Algorithmus von Knuth, Morris und Pratt

Auch Knuth, Morris und Pratt beginnen in ihrem 1977 in [7] veröffentlichten Algorithmus KMP die Suche links im Muster. Scheitert der Vergleich an der Stelle J im Muster, d. h. gilt $Pat\,[J] \neq Text\,[K0+J]$, so benutzen sie die Tatsache, daß $Pat\,[1 \ldots J-1] = Text\,[K0+1 \ldots K0+J-1]$ ist, dazu, das Muster möglichst weit über den Text zu verschieben, ohne daß eine Fundstelle verloren geht. Dies läßt sich an einem Beispiel aus [7] so anschaulich machen:

```
Index  12345678901234567890123456
Text   babcbabcabcaabcabcabcacabc   ganz aufgedeckt

Text   b************************
       ^
       #                  (a # b)
Muster abcabcacab
       ^
       #                  (. # a)
       .abcabcacab
```

Das erste Zeichen hat nicht gepaßt. Das Muster wird völlig über die aktuelle Position im Text hinweggeschoben.

```
Text   babcb********************
        ^^^^
        ===#               (a # b)
Muster  abcabcacab
           ^
           #               (. # a)
        ....abcabcacab
```

Die ersten drei Zeichen haben gepaßt, das vierte, a, nicht. Wir wissen jetzt, daß die vier Zeichen des Textes abc$x$ waren. Nach einer Verschiebung sollten möglichst viele davon erneut zum Muster passen. Da wir im Text die Position nicht geändert haben, brauchen wir uns das vierte Zeichen nicht zu merken. Wir wissen aber, daß dort nur ein Zeichen B des Musters passen kann, wenn $B \neq$ a ist. Deshalb wird das Muster um 4 Stellen verschoben.

```
       babcbabcabcaabcabcabcacabc
Text   babcbabcabcaa*************
            ^^^^^^^^
            =======#       (c # a)
Muster      abcabcaca
                  ^
               ====#       (b # c)
            ...abcabcaca
```

Sieben Zeichen haben gepaßt, beim achten scheitert der Vergleich. Es wird um drei Stellen verschoben. Aber das Muster paßt immer noch nicht.

```
       babcbabcabcaabcabcabcacabc
Text   babcbabcabcaa*************
                  ^
               ====#       (b # a)
Muster         abcabcaca
                  ^
                  #        (a # b)
               ....abcabcaca
```

Es wird erneut um vier Zeichen verschoben. Jetzt paßt $Pat\,[1]$. Die nächsten Zeichen werden überprüft.

```
Text   babcbabcabcaabcabcabca****
                  ^^^^^^^^
                  ========#      (c # b)
Muster            abcabcacab
                         ^
                     ====#      (b # c)
                  ...abcabcacab
```

Erneut wird um 3 Stellen verschoben. Danach passen Text und Muster zusammen.

```
Text   babcbabcabcaabcabcabcacab*
                        ^^^^^^^
                  ==========
Muster            abcabcacab
```

In allen Fällen wird das Muster so weit nach rechts verschoben, bis gilt:

$$\begin{aligned} Pat\,[1\,...\,L-1] \;=\; & Pat\,[J-L+1\,...\,J-1]\;\&\;(Pat\,[J] \neq Pat\,[L]);\\ & Pat\,[1\,...\,L-1] = Text\,[K0+J-L+1\,...\,K0+J-1]. \end{aligned}$$

Dabei ist L der Index im Muster, bis zu dem verschoben wird. An der Stelle L wird der Vergleich fortgesetzt. Speichert man die Werte von L in einer Tabelle $Next\,[J]$, so ist $J - Next\,[J]$ die Größe der Verschiebung. Das erste Muster im Text findet man mit folgendem Programmstück:

```
    KMPP (* KMP Prototyp *)
1     K := J := 1;
2     while (K <= Tl) & (J <= Pl)
3      do while (Text[K] # Pat[J]) & (J > 0)
          (* verschiebe Pat bis Text[K] = Pat[J] *)
          do J := Next[J] od;
4       K := K + 1;   J := J + 1;
5      od;
6   end (* KMPP *);
```

Die Funktion $Next$ ist definiert als:

$$\begin{aligned} \text{(Next):}\quad Next\,[J] \;=\; & \max(L : (Pat\,[1\,...\,L-1] = Pat\,[J-L+1\,...\,J-1])\\ & \qquad\qquad \&\;(Pat\,[J] \neq Pat\,[L]);\\ & Pat\,[1\,...\,L-1] = Text\,[K0+J-L+1\,...\,K0+J-1]. \end{aligned}$$

In [7] wurde zur Berechnung von $Next$ eine Hilfsfunktion F eingeführt:

$$\begin{aligned} \text{(F):}\quad F[J] \;=\; & \max(L : Pat\,[1\,...\,L-1] = Pat\,[J-L+1\,...\,J-1]);\\ & \qquad\qquad (*\text{ ohne } Pat\,[J] \neq Pat[L]!\ *)\\ & Pat\,[1\,...\,L-1] = Text\,[K0+J-L+1\,...\,K0+J-1]. \end{aligned}$$

Die beiden Funktionen $Next$ und F hängen auf folgende Weise zusammen:

$$\begin{aligned} Next\,[J] \;&=\; \begin{cases} F[J] & : \; Pat\,[F[J]] \neq Pat\,[J] \\ Next\,[F[J]] & : \; Pat\,[F[J]] = Pat\,[J] \end{cases}\\ F[J+1] \;&=\; F[J]+1 \quad : \; Pat\,[F[J+1]] = Pat\,[J+1]. \end{aligned}$$

In [7] wird das folgende Beispiel für sie angegeben:

Index	1	2	3	4	5	6	7	8	9	10
Muster	a	b	c	a	b	c	a	c	a	b
Next	0	1	1	0	1	1	0	5	0	1
F	0	1	1	1	2	3	4	5	1	2

Die Werte lassen sich mit dem oben angegebenen Suchprogramm berechnen, wenn ein zweites Exemplar des Musters den Text simuliert. Gilt $Pat[1] \neq Text[K]$, so wird das Muster völlig über die Textstelle hinweggeschoben bis zur virtuellen Position $Pat[0]$. Daher gilt $Next[1] = F[1] = 0$. Die Prozedur `Init_Next` entsteht aus `KMPP` auf folgende Weise:

```
      Init_Next;
1        (* F[1] := *) Next[1] := 0;   J := 0;   K := 1;
         (* Nachdem das erste Zeichen nicht gepasst hat,
            gilt: K = 1, J = 0 *)
2        while K <= Pl   (* Die anderen Operationen entfallen *)
3         do while (Pat[K] # Pat[J]) & (J > 0) do J = Next[J] od;
          (* Hier gilt der 1. Teil der Definitionsbedingung von
             Next: Pat[K-J+1...K] = Pat[1...J].
             Aus Def (F) -> F[K+1] = J + 1 *)
4          K := K + 1;   J := J + 1;
4.1        (* F[K] := J; *)
4.2        if Pat[K] # Pat[J]
4.3         then Next[K] := J (* == F[K] *)
4.4         else (* Pat[K] = Pat[J]
                    -> das Muster wird weiter verschoben. *)
              Next[K] := Next[J]
4.5         fi;   (* Die Anweisungen 4.1 - 4.5 folgen aus der
                     Definition von F und Next. *)
5         od;
6     end (* Init_Next *);
```

Da die Werte von $F[K]$ nur an einer einzigen Stelle abgerufen werden (`4.3`), nachdem sie unmittelbar vorher (`4.1`) eingetragen wurden, kann auf F völlig verzichtet werden.

Die folgende Version von `KMP` ist aus den im vorausgegangenen Kapitel genannten Bausteinen zusammengesetzt.

```
KMP0
   K := 1;   FPat := Pat[1];
   loop
      (* Skip-Loop *)
      while Text[K] # FPat do K := K + 1 od;
      K := K + 1;   J := 2;
      (* Match-Part *)
      repeat
         while (Text[K] = Pat[J]) & (J <= Pl)
          do K := K + 1;   J := J + 1 od;
         if J > Pl
          then (* pattern matched *)
    if K > (Tl + 1) then (* text exhausted -> *) exit fi;
           Locout(K-Pl);   J := 0;   K := K - 1;
          else (* no match *)
           J := Next[J];
           while (Text[K] # Pat[J]) & (J > 0) do J := Next[J] od;
           if J > 0 then J := J + 1;    K := K + 1 fi;
          fi;
```

```
      until J = 0;
     K := K + 1;
   end loop;
end (* KMP0 *);
```

5. BoMo, der Algorithmus von Boyer und Moore

In [4] stellen Boyer und Moore den Algorithmus KMP von den Füßen auf den Kopf. Sie beginnen einen Vergleich nicht links, sondern rechts im Muster. Da das Muster schon vor Suchbeginn bekannt ist, lassen sich Funktionen bestimmen, die zu jedem nicht passenden Zeichen solche Verschiebungen $S \geq 1$ festlegen, daß kein Muster im Text übersehen wird.

Der Algorithmus von Boyer und Moore hat folgende allgemeine Form:

```
BoMoG   (* BoMo Generalized *)
   loop
      (* Skip-Loop *)
      while Text[Ke] # Pat[Pl] do shift appropriately od;
      (* Test Part *)
      if Text[K0+1...K0+Pl] = Pat[1...Pl]
       then (* pattern matched *)
    if Ke > Tl then (* text exhausted -> *) exit fi;
         output location;   shift by full pattern length;
       else (* no match *)
         shift appropriately
       fi;
    end loop;
 end (* BoMoG *);
```

Boyer und Moore beginnen, wie oben gesagt, den Vergleich am Musterende. Passen dort Text- und Musterzeichen nicht zusammen, so wird das Muster so weit über den Text verschoben, bis das aufgedeckte Zeichen zum ersten Mal zu einem Musterzeichen paßt oder das Muster völlig über die Aufdeckstelle hinweggeglitten ist.

```
    Text      ***************a***d****************
                             ^   ^
                             #   :
    Muster 1  entgegengegangen   :
                             =   #
           2  ....entgegengegangen
                               =
           3      ................entgegengegangen
```

Die Größe der Verschiebung entspricht dem Abstand des aufgedeckten Zeichens vom rechten Rand des Musters. Alle nicht im Muster vorkommenden Zeichen passen erst, wenn dieses um die ganze Musterlänge über sie hinweg geschoben wurde. Die Zeichen, die bereits links vor dem Muster liegen, beeinflussen die Verschiebung nicht mehr. Sie wirken wie Joker. Ihnen darf daher jede gewünschte Eigenschaft zugeschrieben werden. Die Werte dieser Verschiebungen werden in der Skip-Tabelle $A0[B]$ gespeichert. Sie hängen nur vom Zeichen B ab. $A0[B]$ ist definiert als:

(A0): $A0[B] = \min\{S \mid \mathit{Pat}\,[\mathit{Pl} - S] = B,\ 0 \leq S \leq \mathit{Pl}\,\} :\ B = \mathit{Text}\,[\mathit{Ke}\,].$

Hat zunächst nicht nur ein einzelnes Zeichen gepaßt, sondern ein längeres Suffix, und scheitert der Vergleich danach, so wird das Muster so weit verschoben, bis dieses Suffix erneut paßt, die jeweils davor liegenden Musterzeichen sich aber voneinander unterscheiden.

```
Text       ***********egen*****ien**************
                     ^^^^     ^^^
                     #===     :::          (e # n)
Muster 1   entgegengegangen     :::
                     ^^^^       :::
                     #===       #== (n # e) (i # g)
       2   ........entgegengegangen
                               ^^^
                               #==          (g # .)
       3            ..............entgegengegangen
```

Die Werte der Verschiebungen sind in einer Tabelle D[Position] gespeichert. Bei Boyer und Moore heißt diese Tabelle *Delta2*. Ihre Werte hängen nur von der Position im Muster, nicht aber vom Zeichen ab, an dem der Vergleich scheitert. D ist definiert als:

$$\begin{aligned}(\mathrm{D}):\quad D(J) = \min\{S + (Pl - J) \mid\ & Pat\,[J+1-S\ ...\ Pl - S] = Pat\,[J+1\ ...\ Pl\,],\\ & Pat\,[J-S] \neq Pat\,[J]\} :\ 1 \leq J \leq Pl\,;\end{aligned}$$

$$\begin{aligned}& Pat\,[J+1\ ...\ Pl\,] = Text\,[K0+J+1\ ...\ Ke\,],\\ & Pat\,[J] \neq Text\,[K0+J].\end{aligned}$$

Vergleicht man die Bedingungen der Definitionen von D und *Next* , so sieht man, daß im wesenlichen rechts und links vertauscht sind. Diese Einsicht verwendet Knuth in [7] dazu, D durch eine Modifikation von `Init_Next` zu berechnen. Dazu führt er die zu F seitenverkehrte Funktion FR ein, die so definiert ist:

$$\begin{aligned}(\mathrm{FR}):\quad FR\,[J]\ =\ \min\{I \mid\ & Pat\,[I+1\ ...\ Pl\,] = Pat\,[J+1\ ...\ J+Pl-I]\} :\\ & J < I \leq Pl\,.\end{aligned}$$

Hier entspricht I dem Ausdruck $J - S$ aus der Definition von D.
Bei der Definition von D ist:

J: die Stelle im Muster, an der der Vergleich scheitert.
S: die Verschiebung des Musters über den Text.
$Pl - J$: der Abstand des Zeichens vom rechten Rand des Musters, an dem der Vergleich scheitert. Um diesen Betrag muß der Zeiger auf das Muster rückgesetzt werden. Ist $J - S \leq 0$, so liegt das angesprochene Zeichen links vor dem Muster. Es ist ein Joker und beeinflußt die Verschiebung nicht mehr.

Hinweis: Nach jeder Verschiebung, ob mit A oder D, werden alle an sich zugänglichen Informationen über den Text vergessen!

Beispiele:

```
Text     ******cba**      *****caba*******      ****ababa*****
               ^^^              ^^^^                  ^^^^^
               #==              #===                  #====
Muster   babacbaba        babacbaba             babacbaba
```

```
                ^::           ^:::              ^::::
                #==           #===              #====
          ..babacbaba    .......babacbaba    .....babacbaba
```

Mit diesen beiden Verschiebefunktionen ergibt sich folgende BoMo-Version:

```
BoMoO;            (* BoMo Original *)
   Ke := Pl;
   loop
      (* Skip Loop *)
      while (Text[Ke] # Pat[Pl])
       do Ke := Ke + A0[Text[Ke]] od;
      K := Ke - 1;   J := (Pl - 1);
      if J = 0
       then (* pattern matched *)
    if Ke > Tl then (* text exhausted -> *) exit fi;
         Locout(K + 1);   Ke := Ke + Pl;
       else (* no match *)
         Ke := K + max(A0[Text[K]], D[J]);
       fi;
    end loop;
end (* BoMoO *);
```

Knuth gibt in [7] folgende Prozedur zur Berechnung von D an.

```
Init_D   (* Nach Knuth *)
   (* Vorbelegung mit der maximalen Verschiebung an der Stelle J
      verursacht durch $. *)
   for J := 1 to Pl do D[J] := (2 * Pl) - J od;
   K := Pl;   J := Pl + 1;
   while K > 0   (* die beiden Muster ueberlappen sich noch *)
    do FR[K] := J;
      while (J <= Pl) & (Pat[K] # Pat[J])
       do D[J] := min(D[J], Pl - K);
         J := FR[J]
       od;
      K := K - 1;   J := J - 1;
    od;
   (* Korrektur, wenn die Muster sich noch ueberlappen ->
      Praefix(Pat) = Suffix(Pat). *)
   for K := 1 to J do D[K] := min(D[K], Pl - K + J);
 end (* Init_D *);
```

Dieser Algorithmus arbeitet dann nicht korrekt, wenn es mehrere verschieden lange zueinander passende (Präfix, Suffix)-Paare gibt.

Beispiel:

```
    Text          xaaa*        *xaa**        **xa***        ***x****
                  ^^^^          ^^^            ^^             ^
                  #===          #==            #=             #
    Muster        aaaa          aaaa           aaaa           aaaa
                  ^:::          ^::            ^:             ^
                  #===          #==            #=             #
                  .aaaa        ..aaaa         ...aaaa        ....aaaa
    S          =     1             2              3              4
    J          =     1             2              3              4
    S + Pl - J =     4             4              4              4
```

In diesem Fall gilt: $D[1,2,3,4] = (4,4,4,4)$. `Init_D` in der Version von Knuth liefert die Vorbelegung $D = (7,6,5,4)$. Die innere Schleife wird nie durchlaufen, da stets $Pat\,[K] = Pat\,[J]$ für jedes Paar (K,J), $1 \leq K,\ J \leq Pl$ ist. D wird nur in der Korrekturschleife am Programmende geändert. Da K und J synchron verkleinert werden, gilt unmittelbar vor ihr $J = 1$, d.h. sie wird genau einmal durchlaufen und berechnet: $D[1] = \min(D[1], Pl - J + S) = \min(7, 4-1+1) = 4$. Danach ist $D = (4,6,5,4)$, d. h. offensichtlich falsch.

Bei der Bestimmung von D berechnet Knuth zunächst die Funktion FR , die die passende Verschiebung angibt, wenn $Pat\,[K] = Pat\,[J]$ ist. Für alle Fälle, bei denen das nicht galt, ist die korrekte Belegung nachzutragen, d.h. immer dann, wenn so weit verschoben wurde, daß die Fehlstelle bereits vor dem Muster liegt. Ist dabei $S < Pl$, dann gibt es sich überlappende (Präfix, Suffix)-Paare, für die die Verschiebung korrigiert werden muß. Knuth macht dies nur für das kürzeste Paar statt für alle Paare.

Bevor wir den Fehler beheben, soll auf eine Eigenschaft von `Init_D` hingewiesen werden, die bisher nur in [11] erwähnt wurde: Schon beim ersten Zugriff nach der Vorbelegung wird in $D[J]$ der Endwert eingetragen! Das kann verwendet werden, um ohne die Berechnung des Minimums auszukommen. Dies sieht man so ein. Es gilt immer $FR\,[J] > J$. In der inneren Schleife von `Init_D` wird $\min(D[J], Pl - K)$ berechnet. K wird in dieser Schleife nicht geändert und außerhalb von ihr verkleinert, so daß sich $Pl - K$ nur noch vergrößert. Daraus folgt die Behauptung und folgende Version von `Init_D`:

```
Init_D;
   (* Vorbelegung *)
   for J := 1 to Pl do D[J] := (2 * Pl) od;
   K := Pl;   J := (Pl+1);
   while K > 0
    do FR[K] := J;
      while (J <= Pl) & (Pat[K] # Pat[J])
       do if D[J] = (2 * Pl) then D[J] := Pl - K fi;
         J := FR[J]
       od,
      K := K - 1;   J := J - 1;
    od;
   (* Korrektur fuer alle (Praefix, Suffix)-Paare *)
   S := 1;
   while K <= Pl
    do for J := S to K
      do if D[J] = (2 * Pl) then D[J] := K + Pl - J fi od;
      S := K + 1;   K := FR[K];
    od;
 end (* Init_D *);
```

D gibt die Größe der Zeigerveränderung auf dem Text an, also Verschiebung des Musters über den Text + Rücksetzen des Zeigers auf das Muster an das Musterende. Will man nur die Musterverschiebung berechnen, so ergibt sich:

(D0): $\quad D0[J] = D[J] - (Pl - J) :\ 1 \leq J \leq Pl\,.$

In `Init_D` kommt die Funktion $Next$ nicht vor, weil auf sie nur in dem Fall $Pat\,[K] \neq Pat\,[J]$ zugegriffen würde. Dabei gilt $F[J] = Next\,[J]$!

Beispiel für FR , D und $D0$:

```
    Index :   1   2   3   4   5   6   7   8   9
    Muster:   b   a   b   a   c   b   a   b   a
    FR    :   6   7   8   9   7   8   9   9  10
    D     :  13  12  11  10   9  10   4  10   1
    D0    :   5   5   5   5   5   7   2   9   1
```

A0 läßt sich einfacher berechnen.

```
Init_A0;
   for B := FirstLetter to LastLetter do A0[B] := Pl od;
   for J := 1 to Pl - 1 do A0[B] := Pl - J od;
   CShift := A0[Pat[Pl]];   A0[Pat[Pl]] := 0;
 end (* Init_A0 *);
```

CShift gibt den minimalen Abstand > 0 an, den das Zeichen $Pat\,[Pl\,]$ vom rechten Rand hat, wenn es im Muster mehrfach enthalten ist. Dieser Wert wird in der BoMo-Variante von Horspool [10] verwendet.

Beispiel für $A0$:

```
    Index :   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
    Muster:   e   n   t   g   e   g   e   n   g   e   g   a   n   g   e   n
    Pl - J:  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0

    B     :   a   e   g   n   t   $        CShift = 3
    A0[B] :   4   1   2   0  13  16
```

Hume und Sunday haben in [6] vorgeschlagen, BoMo ausrollbar zu formulieren und auf diese Weise zu beschleunigen. Ausrollbar heißt: die Schleife darf (teilweise) durch linearen Code ersetzt werden, womit die Abbruchbedingung seltener zu überprüfen ist. Die ausrollbare Version von BoMo sieht so aus:

```
BoMoUr1;     (* BoMo unrollable *)

K := 0;   S := Pl;
loop

   (*** skip loop ***)
   while S # 0 do K := K + S;    S := A0[Text[K]]  od;

   (*** test equal part ***)
   K := K - 1;   J := (Pl - 1);
   while (Text[K] = Pat[J]) & (J > 0)
    do K := K - 1;   J := J - 1 od;

   if J = 0
    then (* pattern matched *)
 if K >= (Tl - Pl) then (* text exhausted *) exit;
      Locout(K + 1);   S := (2 * Pl);
    else (* no match *)

    (*** shift part ***)
      S := max(A0[Text[K]], D[J]);
      (* shift-function: max(A0, D) *)
    fi;
 end loop;
end (* BoMoUr1 *);
```

Schon Boyer und Moore haben überlegt, ob sie die Funktionen $A0$ und D nicht zu einer einzigen von zwei Parametern abhängenden Funktion $DD\,[B, J]$ zusammenfassen sollten, um größere Verschiebungen zu erhalten. Sie haben darauf verzichtet, nachdem Knuth bewiesen hatte, daß schon D in jedem Fall lineares Suchen garantiert. Die Idee wurde erst wieder bei der Suche in Genetischem Code aufgegriffen und hat sich dort als außerordentlich effizient erwiesen [6].

DD läßt sich analog zu D so definieren:

$$\begin{aligned}\text{(DD):}\quad DD\,[B,J] = \min\{S + (Pl - J) \mid\ & Pat\,[J+1-S\ \ldots\ Pl\ -S] = Pat\,[J+1\ \ldots\ Pl\,],\\ & Pat\,[J-S] = B\} :\\ & Pat\,[J+1 \ldots Pl\,] = Text\,[K0+J+1 \ldots Ke\,],\\ & B = Text\,[K0+J].\end{aligned}$$

(DD0): $DD0[B,J] = DD\,[B,J] - (Pl\ - J)$ für jedes B und jedes J.

DD und D unterscheiden sich nur im zweiten Teil der Definitionsbedingung. $Pat\,[J-S]$ muß in DD der schärferen Bedingung $Pat\,[J-S] = B = Text\,[K0+J]$ genügen, statt der schwächeren von D: $Pat\,[J-S] \neq Pat\,[J]$.

```
Init_DD0;
   for K := 1 to Pl
    do for B := FirstLetter to LastLetter
        do DD0[B, K] := Pl od
     od;
   K := Pl;   J := Pl - 1;
   while K > 0
    do FR[K] := J;
      while (J <= Pl) & (Pat[K] # Pat[J])
       do if DD0[Pat[K], J] = Pl
   then DD0[Pat[K], J] := J - K fi;
         J := FR[J]
       od;
      K := K - 1;   J := J -  1;
    od;
   K := 1;
   while K <= Pl
    do for T := K to J
       do for B := FirstLetter to LastLetter
          do if DD0[B, T] = Pl then DD0[B, T] := J fi od;
       od;
      K := K + 1;   J := FR[J];
    od;
 end (* Init_DD0 *);
```

6. Der Algorithmus ESS

Die Skip-Schleife wird in BoMo stets dann verlassen, wenn gilt: $Text\,[Ke\,] = Pat\,[Pl\,]$. Danach wird in natürlichen Sprachen Test- und Shift-Teil in den meisten Fällen wegen $Text\,[Ke\,-1] \neq Pat\,[Pl\,-1]$ nur einmal durchlaufen und, nach einer Reihe zum Teil redundanter Operationen, in die Skipschleife zurückgekehrt. Gelingt es, dort länger zu bleiben, so steigt die Effizienz des Algorithmus.

Horspool hat in [5] gezeigt, daß die Laufzeiten von SPM beträchtlich sinken, wenn man statt des ersten Zeichens im Muster ein anderes, im Text seltener vorkommendes zuerst testet. Dieses Vorgehen setzt voraus, daß die Zeichenhäufigkeiten im Text wenigstens annähernd bekannt sind, und daß sie sich deutlich voneinander unterscheiden. Verwendet man in der Skip-Schleife von BoMo ein anderes als das letzte Zeichen des Musters beim ersten Vergleichsschritt, so hat man eine geänderte Skip-Funktion zu verwenden. Ihre Werte werden kleiner. Daher sollte zunächst geprüft werden, ob der Verlust an Verschiebung aufgewogen wird durch den Gewinn, den man erzielt, wenn Test- und Shift-Teil seltener durchlaufen werden. Wir haben auf diese Art von Verbesserung verzichtet, da die Zeichen bei Genetischem Code nahezu gleich häufig auftreten.

Wie schon in der Einleitung gesagt, vergißt BoMo in der Skip-Schleife nach jeder Verschiebung die Vorgeschichte und fährt im Zustand „weiß nichts“ fort. Dabei ist durch eine Verschiebung L das Zeichen, das danach zum Text paßt, und seine Position im Muster festgelegt. Wird dieser Sachverhalt berücksichtigt, so läßt sich eine im Durchschnitt größere Verschiebung berechnen. Außerdem wird die Skip-Schleife erst dann verlassen, wenn die beiden letzten Zeichen passen, wenn also gilt: $Pat\,[Pl\,-1\,...\,Pl\,] = Text\,[Ke\,-1\,...\,Ke\,]$.

Bevor wir die neue, von zwei Parametern abhängende Skip-Funktion $AA0(B, L)$ formal definieren, soll sie an einem Beispiel mit $A0(B)$ verglichen werden.

```
Text     Sie waren ihnen dem Vorschlag entgegen den Hang entlang entgegengegangen
                       ^              ^^^              ^^  ^            ^^^^
                       #              :::              ::  ^^^      ^^^^^^^^^^^^^^^^
Muster 1 entgegengegangen             :::              ::  :::      :   ::::        :
                       :              :::              ::  :::      :   ::::        :
                       =              #==              ::  :::      :   ::::        :
       2 ................entgegengegangen              ::  :::      :   ::::        :
                                      ^::              ::  :::      :   ::::        :
                                      #                #=  :::      :   ::::        :
       3                  ..............entgegengegangen  :::      :   ::::        :
                                                          :::      :   ::::        :
                                                          :#:      :   ::::        :
       4                                ...entgegengegangen:      :   ::::        :
                                                          :::      :   ::::        :
                                                          #==      :   ::::        :
       5                                   .entgegengegangen      :   ::::        :
                                                          ^::      :   ::::        :
                                                          #==      :   ::::        :
       6                                    ..............entgegengegangen        :
                                                                   ::::::::::::::::
                                                                   ================
       7                                                    ........entgegengegangen
```

Nach 14 Textzugriffen ist das Muster entgegengegangen mit $A0$ und $D0$ bis zur Position verschoben, an der es erkannt wird. Dabei kommt es zu folgenden Verschiebungen: 1. $A0[\sqcup] = 16$; 2. $D0[14] = 14$; 3. $D0[15] = 3$; 4. $A0[\mathrm{e}] = 1$; 5. $D0[14] = 14$; 6. $D0[13] = 8$; 7. Muster wird erkannt.

```
Text     Sie waren ihnen dem Vorschlag entgegen den Hang entlang entgegengegangen.
                       ^             ^^^            ^^  ^           ^^^^
                       #             :::            ::  :       ^^^^^^^^^^^^^^^^
Muster 1 entgegengegangen            :::            ::  :       :   ::::       :
                       :             :::            ::  :       :   ::::       :
                       =             #==            ::  :       :   ::::       :
       2 ...............entgegengegangen            ::  :       :   ::::       :
                                     ^::            ::  :       :   ::::       :
                                     #==            #=  :       :   ::::       :
       3                 ..............entgegengegangen  :       .   ::::       :
                                                    ::  :       :   ::::       :
                                                    ==  #       :   ::::       :
       4                                 ...entgegengegangen       :   ::::       :
                                                    :   :       :   ::::       :
                                                    =   =       :   #===       :
       5                                    ..............entgegengegangen       :
                                                                ::::::::::::::::
                                                                ================
       6                                                    ........entgegengegangen
```

Hier ist mit $AA0$ und $D0$ die Endposition bereits nach 11 Textzugriffen erreicht. Dabei wird so verschoben: 1. $AA0[\sqcup, 16] = 16$; 2. $D0[14] = 14$; 3. $AA0[\mathtt{a}, -1] = 3$; 4. $AA0[\mathtt{e}, 4] = 15$; 5. $D0[13] = 8$; 6. Muster wird erkannt.

Die Skip-Schleife läßt sich besonders effizient implementieren, wenn die Skip-Funktion nicht die Verschiebung des Musters, sondern die Änderung des Verweises auf den Text angibt. Die beiden sind nur dann voneinander verschieden, wenn $Text\,[Ke\,]$ und $Pat\,[Pl\,]$ gepaßt haben. Danach wird $Text\,[Ke\,-1]$ mit $Pat\,[Pl\,-1]$ verglichen. Der Verweis bewegt sich rückwärts, das Muster verändert seine Position nicht. Passen in dieser Situation die Zeichen nicht, so muß dieses Rücksetzen ausgeglichen werden, so daß die Verweisänderung um eins größer ist als die Verschiebung des Musters und den Wert $Pl\,+1$ erreichen kann.

```
    Text   **************an******************
                         ^::::
                         #=  :
    Muster entgegengegangen  :
                         ==  :
           ...entgegengegangen
```

Wie oben erwähnt, soll $AA0$ eine Funktion sein, die es erlaubt, die Skip-Schleife auszurollen. Deshalb muß ihr Wert stets dann 0 sein, wenn die beiden letzten Zeichen des Musters passen. Diese Situation kann auf zwei Wegen erreicht werden: Zuerst wird $Pat\,[Pl\,]$ als passend erkannt und danach $Pat\,[Pl\,-1]$, oder die Reihenfolge ist umgekehrt. Abhängig vom Weg zeigt der Verweis auf verschiedene Stellen im Text. Der Verschiebung 0 ist aber nicht anzusehen, auf welchem Weg sie erreicht wurde. Deshalb haben wir den ersten Fall, in dem das letzte Zeichen zuerst als passend erkannt wurde, auf den zweiten Fall zurückgeführt, allerdings auf Kosten eines zusätzlichen redundanten Textzugriffs auf $Text\,[Ke\,]$.

Jetzt läßt sich $AA0(B, L)$ definieren. Dabei ist B das zuletzt aufgedeckte Zeichen, L die unmittelbar vorausgegangene Verweisänderung:

(AA0):

$AA0(B, L)$

$= \min\{S \mid\ Pat\,[Pl\ -S] = B,$
$Pat\,[(Pl\ -L) - S] = Pat\,[Pl\ -L]\} :$
$B \neq Pat\,[Pl\,], 1 \leq L \leq Pl\,;$
$B = Text\,[Ke\,].$
Die Zeichen B und $Pat\,[Pl\ -L]$ müssen auch nach der Verschiebung um S passen. Ist der Index von Pat kleiner 1, so sind die dadurch angesprochenen Zeichen Joker.

$= -1 :$ $B = Pat\,[Pl\,],\ L > 1.$
Das aufgedeckte Zeichen paßt ohne Verschiebung.

$= \ \ 0 :$ $B = Pat\,[Pl\,],\ L = 0, 1.$
Die beiden letzten Zeichen passen. $AA0(B, 0)$ wird nur in einer ausgerollten Skip-Schleife angesprochen. Bei $L = 0$ ist dabei aber immer $B = Pat\,[Pl\,].$

$= \min\{S \mid\ B = Pat\,[Pl\ -S],$
$Pat\,[(Pl\ -L) - S] = Pat\,[Pl\ -L]\} + 1 :$
$B \neq Pat\,[Pl\ -1],\ L = -1;$
$B = Text\,[Ke\ -1].$
$Text\,[Ke\,]$ hat gepaßt, $Text\,[Ke\ -1]$ paßt nicht!

$= \ 1 :$ $B = Pat\,[Pl\ -1],\ L = -1;$
$B = Text\,[Ke\ -1].$
Zuerst hat das letzte Zeichen gepaßt, erst danach das vorletzte; jetzt wird die Reihenfolge umgedreht.

$AA0(B, Pl\ +1) = AA0(B, Pl\,)$ für jedes zulässige B. Das Muster wurde über alle Aufdeckstellen hinweggeschoben.

Berechnet man dieser Definition entsprechend die Skip-Funktion $AA0$ zum Muster `entgegengegangen`, so erhält man folgende Tabelle:

Tab. 1. Funktionswerte von $AA0$ für das Muster `entgegengegangen`

				×				×		×	×				×				
L	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	−1
B			e	n	t	g	e	g	e	n	g	e	g	a	n	g	e	n	e
n	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	0	0	17
e	1	1	1	6	6	6	6	6	6	6	9	9	11	15	11	15	15		1
g	2	2	2	2	5	5	5	2	2	10	5	5	2	12	5	5	5		17
a	4	4	4	4	4	4	4	4	16	16	16	16	16	16	16	16	16		4
t	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	16	16		17
$	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16		17

Wie man leicht sieht, gilt stets: $AA0(B,S) \geq A0(B)$ für jedes S und jedes B, $B \neq Pat\,[Pl\,]$. Das heißt, $AA0$ verschiebt nie weniger als $A0$. Außerdem wird die Skip-Schleife seltener verlassen. Allerdings muß auf ein zweidimensionales Feld zugegriffen werden, dessen Ansteuerung länger dauern kann als die des eindimensionalen. Die mit „$\times$“ gekennzeichneten Spalten der Tabelle werden nie erreicht, wenn nur Verschiebungen aus $AA0$ verwendet werden.

Im allgemeinen steigt der Aufwand zur Berechnung der vollständigen Matrix $AA0$ mit $O(Pl^{\,2})$. Bei langen Mustern und kleinen Alphabeten, etwa bei Genetischem Code, wird in der Skip-Schleife nur ein Teil der Werte von $AA0$ verwendet. Wir beschränken uns hier darauf, nur diese zu berechnen.

$AA0$ kann man als einen endlichen Automaten auffassen, bei dem die Identifikationsnummern der Zustände so geschickt gewählt wurden, daß sie auch als Verschiebungen des Textzeigers interpretiert werden dürfen. Es ergibt sich die BoMo-Variante ESS .

```
ESS (* BoMo unrollable *);
Ke := 0;   S := Pl;
loop
   while S # 0 do Ke := Ke + S;   S := AA0[Text[Ke], S] od;
   J := (Pl - 2);   K := Ke - 2;
   while Text[K] = Pat[J] & J > 0
    do K := K - 1;   J := J - 1 od;
   if J = 0
    then (* pattern matched *)
 if K >= (Tl - Pl) then (* text exhausted -> *) exit fi;
      Locout(K + 1);   Ke := Ke + Pl;
    else (* no match *)
      Ke := Ke + DD0[Text[K], J]
    fi;
   S := AA0[Text[Ke], Pl];
 end loop;
end (* ESS *);
```

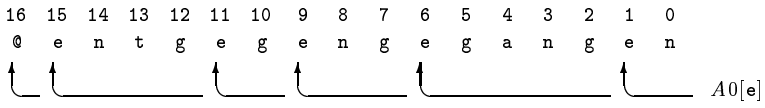
In dieser Version darf für $DD0$ jede andere zulässige Verschiebefunktion stehen. Dabei wird allerdings Information nach der Verschiebung vergessen.

7. Init_AA0

In diesem Kapitel wird die Initialisierung von $AA0$ detailliert beschrieben. Dazu ist es notwendig, Begriffe einzuführen, die entweder die Beschreibung von Init_AA0 vereinfachen oder seine Effizienz steigern. Hier werden die Variablen in der Schreibweise eingeführt, in der sie später im Programmtext vorkommen. Da unser PASCAL-Compiler nur Integer-Variable als Indizes zuläßt, mußten wir mit den Funktionen *ord* (Zeichen) und *chr* (Integer) arbeiten. Auf diese Umwandlungen wird bei der Beschreibung der Programmbedeutung verzichtet.

Weg durch das Muster. Wir verstehen unter dem Weg eines Zeichens B durch das Muster die Folge von Positionen in *InvPat* , an denen B steht. Dabei ist für alle Zeichen die Position Pl , an der in *InvPat* ein Joker @ steht, das Ende des Weges, d. h. jeder Weg hat eine Richtung. Der Weges beginnt an der durch $A0[B]$ festgelegten Stelle.

Beispiel des Weges von `e` durch `entgegengegangen`:



Weg(`e`, `entgegengegangen`) = (1, 6, 9, 11, 15, 16).

InvPat [0 ... $Pl - 1$] = *Pat* [Pl ...1] enthält das Muster invers, d.h. seitenverkehrt beim Index 0 beginnend. Der Index von *InvPat* gibt den Abstand des entsprechenden Zeichens vom rechten Musterrand an. Dieser muß nicht mehr eigens berechnet werden. *InvPat* ist so dimensioniert, daß noch zusätzliche Zeichen, z.B. als Bremsen, eingefügt werden können.

PatAlpha [0 ... LL] enthält das Alphabet aller im Muster vorkommenden Zeichen. Für jedes dieser Zeichen muß zu jedem Zustand ein Folgezustand berechnet werden. Wird an der Stelle *Text* [Ke] das Zeichen *Pat* [Pl] aufgedeckt, so paßt dieses immer. Der Folgezustand steht ohne Verschiebung fest, so daß er nie auf dem Weg von *Pat* [Pl] durch das Muster gesucht werden muß. Wir speichern *Pat* [Pl] an der festen Stelle *PatAlpha* [0], so daß sich ein überflüssiger Zugriff auf *Pat* [Pl] in *PatAlpha* leicht vermeiden läßt.

$LL + 1$ ist die Zahl der Zeichen dieses Alphabets. Alle Zeichen, die nicht im Muster vorkommen, bewirken in allen Zuständen des Automaten $AA0$ die maximale Verschiebung Pl. Diese wird als Voreinstellung verwendet. Nur für die Zeichen in *PatAlpha* müssen zu jedem erreichbaren Zustand die Folgezustände ermittelt werden.

PatPath [0 ... Pl] enthält für jedes Zeichen aus *PatAlpha* dessen Weg durch das Muster.
Beispiel zu *PatPath*:

```
16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 @  e  n  t  g  e  g  e  n  g  e  g  a  n  g  e  n
17 16 16 16 16 15 12 11 14 10  9  7 16  8  5  6  3
```

Die Bedingung, daß zwei gegebene Zeichen mit festem Abstand L voneinander zum Muster passen, läßt sich effizient prüfen, indem man den Weg des einen durch das Muster geht und testet, ob das andere im Abstand L im Muster liegt.

UsedCol [$*$] enthält die Nummern der Zustände, die in $AA0$ aufgenommen werden müssen. Zu jedem Zustand in *UsedCol* werden alle Folgezustände ermittelt und, wenn nötig, dort hinzugefügt.

*IsInAA*0[S] gibt an, ob der Zustand S schon in *UsedCol* eingetragen wurde (*true*) oder nicht (*false*). Dies steigert die Effizienz, weil ein und derselbe Zustand Folgezustand mehrerer verschiedener Zustände sein kann.

Im folgenden werden nicht auf Anhieb einsichtige Programmteile von `Init_AA0` besprochen. Das komplette lauffähige PASCAL-Quellprogramm steht im Anhang.

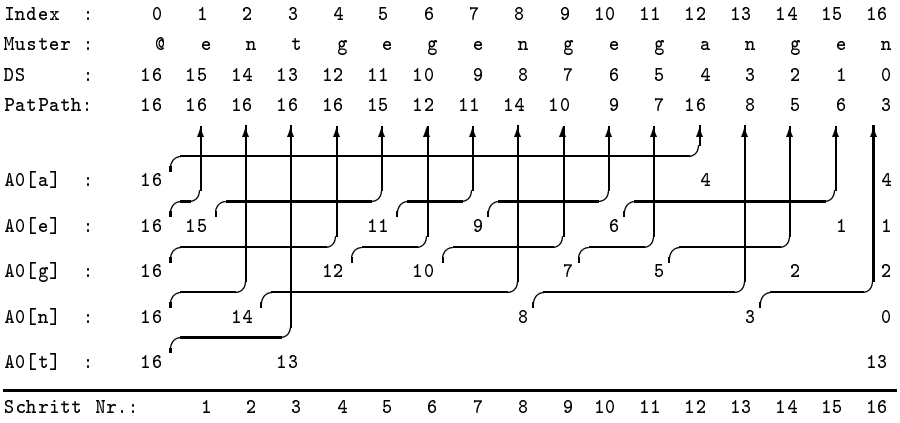
7.1. Initialisierung von A0, InvPat, IsInAA0, PatPath und PatAlpha

```
LL := -1;               (* LL: Zahl der Zeichen in PatAlpha                  *)
DS := Pl;               (* Index von InvPat == Abstand vom Rand              *)
for J := 1 to Pl - 1
 do B = Pat[J];
   DS := DS - 1;
   InvPat[DS] := B;     (* InvPat[Pl-J] := Pat[J]                            *)
   IsInAA0[DS]:= false;
   PatPath[DS]:= A0[B]; (* Weg wird von seinem Ende her erzeugt, -> Verweis auf
                           direkten Nachfolger                               *)
   if A0[B] = Pl        (* true -> Zeichen B wurde noch nicht angesprochen   *)
    then LL := LL + 1;
     PatAlpha[LL] := B  (* PatAlpha + B                                      *)
    fi;
   A0[B] := DS;         (* verzoegerte Uebernahme des Abstands Pl - J        *)
 od;
InvPat[0] := Pat[Pl];
```

Wenn, was bei Genetischem Code häufig der Fall sein dürfte, alle Zeichen des Textalphabets auch im Muster vorkommen, kann *PatAlpha* einfacher initialisiert werden (wie, lieber Leser?).

$A0[B]$ wird hier nur verwendet, weil es eindimensional ist, man könnte es durch $AA0[B, Pl\,]$ substituieren.

Beispiel zur Erzeugung von *PatPath* :



Die Belegung von *PatPath* bricht beim $(Pl - 1)$–ten Schritt ab. In $A0[Pat\,[Pl\,]]$ steht *CShift* . Der Schritt *Pl* wird nachgeholt.

```
CShift := A0[Pat[Pl]];  A0[Pat[Pl]] := 0;
PatPath[0] := CShift;   PatPath[Pl] := PatPath[Pl+1] := Pl + 1;

(* Garantieren, dass PatAlpha[0] = Pat[Pl] ist. *)
PatAlpha[Pl+1] := Pat[Pl];              (* Bremse                            *)
J := 0;
while PatAlpha[J] # Pat[Pl] do J := J + 1 od;
PatAlpha[0] :=: PatAlpha[J];            (* Pat[Pl] war noch nicht in PatAlpha *)
if J > LL then LL := LL + 1;
```

7.2. Belegen der Spalten $Pl+1$, Pl, $Pl-1$ und 0

Die Zustände $Pl+1$, Pl und $Pl-1$ führen zu den gleichen Verschiebungen wie bei $A0$ und werden genauso berechnet.

```
for B := FirstLetter to LastLetter
 do AA0[B, PL+1] := AA0[B, Pl] := AA0[B, Pl-1] := A0[B, Pl-1] od;
AA0[Pat[Pl], Pl+1] := AA0[Pat[Pl], Pl] := AA0[Pat[Pl], Pl-1] := -1;
(* Das letzte Zeichen passt: Pat[Pl] = Text[Ke] *)
```

Die Spalte $Pl-1$ wird bereits belegt, weil sie hier einfach zu berechnen ist.

```
AA0[Pat[Pl], 0] := 0;
```

Der Zustand 0 wird nur über den Zustand 1 ($Pat\,[Pl-1] = Text\,[Ke-1]$) erreicht, wenn $Pat\,[Pl-1\,...Pl\,] = Text\,[Ke-1\,...\,Ke\,]$ ist. Danach ändert sich der Zeiger Ke auf den Text nicht mehr. Die Operationen in der ausgerollten Skip-Schleife sind:

```
S := AA0[Text[Ke], 1] == S := AA0[Pat[Pl],  1] == 0;
Ke: = Ke + S == Ke + 0;
S := AA0[Text[Ke], S] == AA0[Pat[Pl],  0] == 0;
Ke := Ke + S == Ke + 0; ...
```

Deshalb kann der Rest der Spalte 0 leer bleiben.

```
for J := 1 to LL
 do S := A0[PatAlpha[J]];
   UsedCol[SMax+J] := S;   IsInAA0[S] := true;
 od;
SMax := SMax + LL;
IsInAA0[Pl+1] := IsInAA0[Pl] := IsInAA0[Pl-1] := true;
```

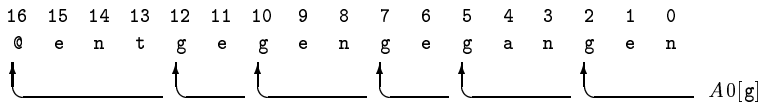
Alle Folgezustände des Anfangszustands Pl wurden in *UsedCol* aufgenommen und als schon bekannt in $IsInAA0$ markiert.

7.3. Berechnung der Folgezustände aller Zustände in UsedCol

Wenn der Zustand L erreicht wird, gilt nach der Verschiebung des Musters $InvPat\,[L] = Text\,[K0 + Pl - L]$. Von L ausgehend wird dem Weg S von $Z = InvPat\,[L]$ durch $InvPat$ gefolgt. Die Folgezustände werden durch die im Abstand L rechts vom Weg liegenden Zeichen $InvPat\,[S-L]$ bestimmt. Sind auf dem Weg alle Zeichen von *PatAlpha* angesprochen worden, so bricht das Verfahren ab; wenn nicht, dann wird von $InvPat\,[Pl-L]$ bis zum Musteranfang nach den restlichen Zeichen von *PatAlpha* gesucht. Spätestens an der Stelle $InvPat\,[Pl] =$ `@` werden die letzten gefunden.

Die Berechnung der Folgezustände eines Zustands L beginnt mit der Vorbelegung der Spalte L mit der maximalen Verschiebung Pl. $AA0[B,L] \neq Pl$ bedeutet also: zum Zeichen B wurde der Folgezustand schon gefunden.

Beispiel: `entgegengegangen`, Folgezustände von Zustand 2



Weg von 2 beginnend und Zeichen im Abstand 2: (2,**n**; 5,**n**; 7,**g**; 10,**n**; 12,**g**; 16,**n**)

Es gibt die Folgezustände zu (2,n) $\rightarrow -1$; (7,g) $\rightarrow 5$. Es fehlen noch die Folgezustände zu den Zeichen `a`, `e` und `t`. Zwischen 14 und 16 wird nur das Zeichen `e` gefunden, der Folgezustand ist 15. Für `a` und `t` bleibt die maximale Verschiebung 16 bestehen.

```
SC := 0;                (* Zeiger auf den Zustand, dessen Folgezustaende
                           gesucht werden                                  *)
while SC < SMax         (* es gibt noch unbehandelte Zustaende             *)
 do SC := SC + 1;
   S := UseCol[SC];     (* aktueller Zustand                               *)
   B := InvPat[S];      (* zu S gehoerendes Musterzeichen                  *)
   for B := FirstLetter to LastLetter do AA0[B, S] := Pl;
   AA0[Pat[Pl], S] := -1;

   (* Weg von B ab S durch das Muster                                      *)
   NextS := PatPat[S];   (* naechste Station auf dem Weg                   *)
   LB := 0;             (* LB = LL -> alle Zeichen aus PatAlpha gefunden   *)
   while (NextS <= Pl) & (LB < LL) (* NextS > Pl -> Ende des Wegs erreicht *)
    do DS := NextS - S  (* Index des moeglichen Folgezustands              *)
      DB := InvPat[DS]; (* Zeichen das den Folgezustand erzeugt            *)
      if AA0[B, S] >= Pl
       then             (* Folgezustand zu B noch nicht in AA0 eingetragen.*)
         AA0[B, S] := DS;
         LB := LB + 1;
         if not IsInAA0[DS]
          then          (* Folgezustand in UsedCol eintragen               *)
            SMax := SMax + 1;   UsedCol[SMax] := DS;
            IsInAA0[DS] := true;
          fi
       fi;
      NextS := PatPath[NextS];  (* naechste Station auf dem Weg           *)
    od;                 (* Am Ende des Weges angelangt                     *)

   (* Folgezustaende noch nicht gefundener Zeichen aus PatAlpha            *)
   DS := Pl + 1 - S;
   while (DS < Pl) & ( LB <= LL)
    do DB := InvPat[DS];(* Zeichen das den Folgezustand erzeugt            *)
      if AA0[B, S] >= Pl
       then             (* Folgezustand zu B noch nicht in AA0 eingetragen.*)
         AA0[B, S] := DS;
         LB := LB + 1;
         if not IsInAA0[DS]
          then          (* Folgezustand in UsedCol eintragen               *)
            SMax := SMax + 1;   UsedCol[SMax] := DS;
            IsInAA0[DS] := true;
          fi
       fi;
      DS := DS + 1;     (* naechste Station auf dem Weg                    *)
    od;                 (* Am Ende des Weges angelangt                     *)
   AA0[Pat[Pl], 1] := 0;(* Die beiden letzten Zeichen passen.              *)
 end (* Init_AA0 *)
```

Literatur

1. Baeza-Yates, Ricardo A.: Improved string searching. Software - Practice and Experience 19.3, 257 - 271 (1989).
2. Baeza-Yates, Ricardo A.: String searching algorithms revisited. Lecture Notes in Comp. Sci. 382, Springer-Verlag (1989).
3. Baeza-Yates, Ricardo A.; Krogh, Fred T.; Ziegler, Bernhard; Sibbald, Peter R.; Sunday, Daniel M.: Notes on a very fast substring search algorithm. In <Technical Correspondence> Comm. ACM 35.4, 132 - 137 (1992).
4. Boyer, Robert S.; Moore, J. Strother: A fast string search algorithm. Comm. ACM 20.10, 762 - 772 (1977).
5. Horspool, R. Nigel: Practical fast searching in strings. Software - Practice and Experience 10.8, 501 - 506 (1980).
6. Hume, Andrew; Sunday, Daniel: Fast string searching. Software - Practice and Experience 21.11, 1221 - 1248 (1991).
7. Knuth, Donald E.; Morris, James H.; Pratt, Vaugham R.: Fast pattern matching in strings. SIAM J. Comput. 6.2, 323 - 350 (1977).
8. Smith, P. D.: Experiments with a very fast substring search algorithm. Software - Practice and Experience 21.10, 1065 - 1074 (1991).
9. Sunday, Daniel M.: A very fast substring search algorithm. Comm. ACM 33.8, 132 - 142 (1990).
10. Ziegler, Bernhard: QuickSearch – Ein schneller Algorithmus zur Mustersuche in Texten. Report Nr. 1993/14 (Dezember 1993). Inst. für Informatik, Breitwiesenstr. 20-22, D-70565 Stuttgart.
11. Ziegler, Bernhard: Anmerkungen zu einem Algorithmus von Knuth. Bericht Nr. 1/82 (1982). Inst. für Informatik, Azenbergstr. 12, D-7000 Stuttgart.

A. Anhang

```
procedure INIT_AA0_nackt; (*** == entkoppelt von Shift-Funktionen ***)


   (* Globale Groessen:

      Import:
         PL         : Musterlaenge,
         MAXPL      : groesste zulaessige Musterlaenge: integer;
         FIRSTLETTER: Index des ersten,
         LASTLETTER : des letzten Zeichens im Alphabet: char;

      Export:
         LPAT       : == PAT[PL]: char;
        CSHIFT, CCSHIFT:
                      invariante Verschiebung im Shift-Teil: integer;
         AA0[B,S]   : Skip-Tabelle: B in Alphabet, -1 <= S <= PL + 1.
    *)
   const Nabla = '%';         (*    Repraesentant fuer alle nicht
                                    im Muster vorkommenden Zeichen.   *)
   type  AlphaRange = FIRSTLETTER..LASTLETTER;
         PATRange   = 0..MAXPL;

   var   InvPAT     : array[PATRange] of char;
                            (*    inverses Muster,
                                  InvPAT[0] = PAT[PL], ... ,
                                  InvPAT[PL] = '%' !                *)
         PATPath    : array[PATRange] of integer;
                            (*    enthaelt zu jedem Zeichen
                                  den zugehoerenden Weg durch
                                  das Muster                        *)
         InvPATPath : array[PATRange] of integer;
         PATAlpha   : array[PATRange] of char;
                            (*    Alphabet der Zeichen im Muster    *)
         IsInAA0    : array[PATRange] of Boolean;
                            (*    true == Spalte wird verwendet.    *)
         UsedCol    : array[PATRange] of integer;
                            (*    enthaelt die Spaltennummern       *)
         A0         : array[AlphaRange] of integer;
                            (*    Skip-Funktion von BoMo            *)
         LPAT1,             (*    == PAT[PL-1]                      *)
         B, DB      : char;
         PLp1, PLm1,        (*    Invariante: PL + 1, PL - 1        *)
         LL,                (*    /PATAlpha/ - 1                    *)
         S, DS, NextS, S0,
                            (*    Positionen auf Wegen durch PAT    *)
         SMax,              (*    Zahl der Spalten in UsedCol       *)
         SC,                (*    Zeiger auf UsedCol                *)
```

```
        LB,                 (*    Gibt an, zu wieviel Zeichen S
                                  gefunden wurde                    *)
        J, JB      : integer;


  procedure Drucke_PATPath;
   var J: integer;
  begin writeln('begin PATPath');
   for J := PL + 1 downto 0 do write(J:2, ' ');   writeln;
   for J := PL + 1 downto 0 do write(InvPAT[J]:2, ' ');   writeln;
   for J := PL + 1 downto 0 do write(PATPath[J]:2, ' ');   writeln;
   writeln('end PATPath');
  end (* Drucke_PATPath *);


  procedure Drucke_UsedCol;
   var J: integer;
  begin writeln('begin UsedCol');
   for J := PL + 1 downto 0 do write(J:2, ' ');   writeln;
   for J := PL + 1 downto 0 do write(InvPAT[J]:2, ' ');   writeln;
   for J := PL + 1 downto 0 do write(UsedCol[J]:2, ' ');   writeln;
   for J := PL + 1 downto 0


do if IsInAA0[J] then write(' t ') else write(' f ');   writeln;
   writeln('end UsedCol');   readln;
  end (* Drucke_PATPath *);



begin
   (*** Belegung der Invarianten ***)
   PLp1 := PL + 1;
   PLm1 := PL - 1;
   LPAT1 := PAT[PLm1];
   LPAT := PAT[PL];          (*++  ZF_I := ZF_I+2;  ++*)

   (***   Berechnung von A0, InvPAT, PATPath, PATAlpha,
                                  CCSHIFT, CSHIFT, LL            ***)
   LL := -1;
   InvPAT[PL+1] := Nabla;
   PAT[0] := LPAT;           (*    => PATAlpha[0]=PAT[PL]
                                  wird garantiert!                *)
                             (*++  ZF_I := ZF_I+2;  ++*)

   for JB := FIRSTLETTER to LASTLETTER do A0[JB] := PL;
      ZF_I := ZF_I + (LASTLETTER-FIRSTLETTER+1);

   DS:= PL;
   for J := 1 to PLm1
```

```
 do begin B := PAT[J];
   JB := ord(B);
   DS  := DS - 1;
   InvPAT[DS] := B;        (*    Muster umgedreht                  *)
   IsInAA0[DS] := false;   (*    Spalte DS noch nicht in AA0       *)
   PATPath[DS] := A0[JB]; (*    Verweis auf direkten
                                 Nachfolger auf dem Weg            *)
   if A0[JB] = PL
    then begin             (*    Zeichen noch nicht im
                                 Alphabet, einsetzen!              *)
      LL := LL + 1;
     PATAlpha[LL] := B;    (*++  ZF_I := ZF_I+1;  ++*)
    end (* if *);

   A0[JB] := DS;           (*++  ZF_I := ZF_I+6;  ++*)

 end (* for J := 1 to PLm1 *);
CSHIFT := A0[ord(LPAT)];
A0[ord(LPAT)] := 0;
PATPath[0] := CSHIFT;
PATPath[PL] := PLp1;
PATPath[PLp1] := PLp1;
InvPAT[0] := LPAT;
PATAlpha[LL+1] := LPAT;    (*    Bremse in PATAlpha                *)

                           (*++  ZF_I := ZF_I+7;  ++*)

(* PATAlpha <= LPAT *)
J := 0;                    (*++  ZF_I := ZF_I+1;  ++*)
while PATAlpha[J] <> LPAT
  do begin J := J+1;       (*++  ZF_I := ZF_I+1   ++*)
  end;
if J > LL then LL := J;
PATAlpha[J] := PATAlpha[0];
PATAlpha[0] := LPAT;       (*    Vertausche                        *)
                           (*++  ZF_I := ZF_I+3;  ++*)
                           (*++  Drucke_PATPath   ++*);

(*++ Begin: Nach Test oder Drucke_AA0 wegwerfen! ++*)
for J := 0 to PL + 1
 do begin UsedCol[J] := 0;
   for JB := FIRSTLETTER to LASTLETTER do AA0[JB,J] := -99;
 end;
(*++ End: Nach Test oder Drucke_AA0 wegwerfen!   ++*)

(*** Die Spalten S = 0, PL, PL + 1 belegen.                    ***)
for JB := FIRSTLETTER to LASTLETTER
 do begin S := A0[JB];     (*++  ZF_I := ZF_I+1;  ++*)
   AA0[JB,PLp1] := S;
   AA0[JB,PL] := S;        (*++ ZF2_I := ZF2_I+2; ++*)
```

```
 end (* od *);

AA0[ord(LPAT),PLp1] := -1;
AA0[ord(LPAT),PL] := -1;
AA0[ord(LPAT),0] := 0;    (*++ ZF2_I := ZF2_I+3; ++*)

(*** UsedCol mit den Verschiebungen von A0 belegen              ***)
for J := 1 to LL
 do begin S := A0[ord(PATAlpha[J])];
   IsInAA0[S] := true;
   UsedCol[J] := S;
 end (* od *);
SMax := LL;               (*++ ZF_I:= ZF_I+4*LL; ++*)

(*** Vorbelegung der Spalte -1                                  ***)
                          (*++  ZF_I := ZF_I+1;  ++*)
if PAT[1] = LPAT
 then for JB := FIRSTLETTER to LASTLETTER do AA0[JB,-1] := PL
 else for JB := FIRSTLETTER to LASTLETTER do AA0[JB,-1] := PLp1;
                          (*++ ZF2_I:= ZF2_I +
                                (-FIRSTLETTER + LASTLETTER +1); ++*)

AA0[ord(LPAT1),-1] := 1;  (*    (PAT[PL], PAT[PL-1])
                                => (PAT[PL-1], PAT[PL])          *)
                          (*++ ZF2_I := ZF2_I+1; ++*)

InvPAT[PL] := LPAT1;      (*    wegen InvPAT[DS] = PAT[PL-1]?    *)
                          (*++  ZF_I := ZF_I+1;  ++*)


(*** Korrektur von Spalte -1 und PATPath ***)
NextS := CSHIFT;
S := -1;
LB := 0;
while (NextS < PL) and (LB < LL)
do begin
    DS := NextS - S;      (*    Abstand zwischen Ausgangsposition
                                und aktueller Position           *)
                          (*    == Verschiebung des Musters      *)
   DB := InvPAT[DS];      (*    aufgedecktes Zeichen,
                                bestimmt die Verschiebung        *)
                          (*++  ZF_I:= ZF_I+1;  ++*)
                          (*++ ZF2_I:= ZF2_I+1; ++*)
   if AA0[ord(DB),S] >= PL
   then begin AA0[ord(DB),S] := DS;
                          (*    nur einmal belegen               *)
                          (*++ ZF2_I:= ZF2_I+1; ++*)
                          (*++  ZF_I:= ZF_I+1;  ++*)
     LB := LB + 1;
      if not IsInAA0[DS]
```

```
         then begin SMax := SMax + 1;
          UsedCol[SMax] := DS;
                              (*++  ZF_I:= ZF_I+2;  ++*)
           IsInAA0[DS] := true; (* Spalte DS ist zu belegen         *)
        end (* if not IsInAA0[DS] *);
    end (* if AA0[DB,S] >= PL *);
    NextS := PATPath[NextS];
                              (*++  ZF_I:= ZF_I+1;  ++*)
end; (* while do*)
                              (*++  Drucke_PATPath; ++*)
                              (*++  DRUCKE_AA0;     ++*)

(*** Initialisierung des Kerns von AA0, d.h. der Spalten
                                    1 bis PL - 2.                  ***)
(*** if CSHIFT = 1 then SC := 0 else                               ***)
SC := 0;
while SC < SMax
 do begin SC := SC + 1;
    S := UsedCol[SC];         (*    ansteuerbare Spalte             *)
    B := InvPAT[S];           (*    zur Verschiebung S
                                    gehoerendes Zeichen             *)
                              (*++  ZF_I:= ZF_I+2;  ++*)
    for JB := FIRSTLETTER to LASTLETTER do AA0[JB,S] := PL;
    AA0[ord(LPAT),S] := -1; (*   Text[Ke] = PAT[PL]               *)
                              (*++ ZF2_I := ZF2_I +
                                    (LASTLETTER-FIRSTLETTER+2);    ++*)


    (*** Zunaechst dem an der Stelle S beginnenden Weg
                                    durch PAT folgen.              ***)
    NextS := PATPath[S];      (*    naechste Station auf dem Weg    *)
    LB := 0;
                              (*++  ZF_I:= ZF_I+1;  ++*)
    while (NextS <= PL) and (LB < LL)
     do begin DS := NextS - S;
                             (*++  Drucke_PATPath; ++*)
                              (*    Abstand zwischen Ausgangsposition
                                    und  aktueller Position         *)
                              (*    == Verschiebung des Musters     *)
       DB := InvPAT[DS];      (*    aufgedecktes Zeichen,
                                    bestimmt die Verschiebung       *)
                              (*++  ZF_I:= ZF_I+1;  ++*)
                              (*++ ZF2_I:= ZF2_I+1; ++*)
       if AA0[ord(DB),S] >= PL
        then begin AA0[ord(DB),S] := DS;  (* nur einmal belegen    *)
                              (*++  ZF_I:= ZF_I+1;  ++*)
                              (*++ ZF2_I:= ZF2_I+1; ++*)
          LB := LB+1;
          if not IsInAA0[DS]
           then begin SMax := SMax + 1;
```

```
              UsedCol[SMax] := DS;
               IsInAA0[DS] := true; (* Spalte DS ist zu belegen      *)
               ZF_I:= ZF_I+2;
             end (* if not IsInAA0[DS] *);
          end (* if AA0[DB,S] >= PL *);
          NextS := PATPath[NextS];
                              (*++  ZF_I:= ZF_I+1;  ++*)
       end (* while NextS <= PL *);
         (* AA0[.,S] belegt fuer alle von S erreichbaren Positionen. *)
                              (*++  Drucke_UsedCol; ++*)

      (*** AA0[.,S] belegen fuer alle nicht auf dem Weg S
                                   angesprochenen Zeichen.         ***)
      DS := PLp1 - S;
      while (LB < LL) and (DS <= PL)
       do begin
          DB := InvPAT [DS]; (*    nicht zu jedem DB ein S gefunden  *)
                              (*++  ZF_I:= ZF_I+1;  ++*)
                              (*++ ZF2_I:= ZF2_I+1; ++*)
            if AA0[ord(DB),S] >= PL
             then begin       (*    Zeichen gilt als
                                   noch nicht angesprochen           *)
               AA0[ord(DB),S] := DS;
                              (*++ ZF2_I:= ZF2_I+1; ++*)
               LB := LB + 1;
                              (*++  ZF_I:= ZF_I+1;  ++*)
               if not IsInAA0[DS]

                then begin SMax := SMax + 1;
                  UsedCol[SMax] := DS;
                  IsInAA0[DS] := true; (* Spalte DS ist zu belegen   *)
                              (*++  ZF_I:= ZF_I+2;  ++*)
                end (* if not IsInAA0[DS] *);
             end (* if AA0[DB,S] >= PL *);
          DS := DS + 1;
       end (* while LB <  LL *);
    end (* while SC < SMax *);
   AA0[ord(LPAT),1] := 0;     (*   PAT[PL-1...PL] passt,
                                   Textzeiger = Ke!                  *)
                              (*++ ZF2_I:= ZF2_I+1; ++*)
                              (*++ Drucke_UsedCol;  ++*)
                              (*++ DRUCKE_ASAA0;    ++*)
                              (*++ DRUCKE_AA0;      ++*)
 end (* INIT_AA0_nackt *);
```