String-Matching mit endlichem Automat

Wir erinnern uns an eine alte Aufgabe:

Finde DEA für Menge der Wörter *uababcv* mit $u, v \in \{a, b, c\}^*$.

Wenn nach dem zweiten b wieder ein a kommt, springt man nicht an den Anfang zurück, sondern nur um 2 Buchstaben.

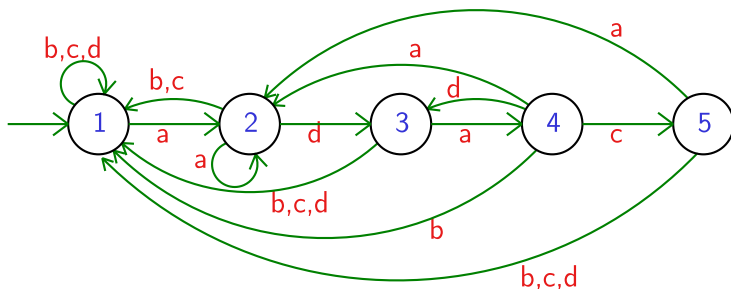
Diesen Gedanken verwenden wir auch jetzt wieder.

Der zugehörige Algorithmus ist denkbar einfach:

```
Berechne Übergangsfunktion δ;
z := 0;
FOR i := 1 TO n DO
   z := δ(z, T[i]);
   IF z = m THEN OUT(i − m) ENDIF
ENDFOR
```

String-Matching mit endlichem Automat (2)

Wie sieht der Automat für unser Beispiel adac aus?



Falls es noch weitere Buchstaben, z.B. σ, im Alphabet Σ gibt (außer a,b,c,d), ist $\delta(q, \sigma) = 1$ für alle $q \in \{1, 2, 3, 4, 5\}$.

Bitte schreiben Sie sich selbst die Funktion δ vollständig in Tabellenform auf!

String-Matching mit endlichem Automat (3)

Wie sieht der Automat für das Standardbeispiel `ananas` aus?

Das *Skelett* des Automaten ist das folgende:

Bitte ergänzen Sie die fehlenden Übergänge und schreiben Sie sich auch hier die Funktion δ vollständig in Tabellenform auf.

Knuth-Morris-Pratt Algorithmus

Der KMP-Algorithmus arbeitet ähnlich wie der vorherige, aber statt der zweidimensionalen Tabelle für δ kommt er mit einer eindimensionalen *Verschiebetabelle* aus.

Die Verschiebetabelle wird in linearer Zeit in der Patternlänge erstellt, so dass der Gesamtalgorithmus nur Linearzeit benötigt.

Die Definition der Tabelle ist gegeben durch

$$Shft[q] = \max\{k < q \mid P[1 \dots k] = P[q - k + 1 \dots q]\}$$

für alle $i \in \{1, \dots, m\}$.

Man überlegt sich leicht, dass bei einem Mismatch in der $(q+1)$-ten Stelle des Patterns die nächste Suche erst $q - Shft[q]$ Stellen weiter hinten anfangen muss, und dass dabei die ersten $Shft[q]$ Buchstaben nicht nochmal zu prüfen sind.

Knuth-Morris-Pratt (2)

Zunächst schreiben wir den Algorithmus auf, der mit einer Prozedur `BerechneTabelle` die Verschiebetabelle schon komplett berechnet bekommt:

```
BerechneTabelle(); q := 0;
FOR i := 1 TO n DO
   WHILE (q > 0) ∧ (P[q+1] ≠ T[i]) DO
                                        q := Shft[q]
   ENDWHILE;
   IF P[q+1] = T[i] THEN  q := q+1 ENDIF;
   IF q = m THEN  OUTP(i − m); q := Shft[q] ENDIF
ENDFOR
```

Mit der Berechnung der Verschiebetabelle und der Komplexität des Verfahrens beschäftigen wir uns jetzt.

Knuth-Morris-Pratt (3)

Wie wird die Verschiebetabelle berechnet?

```
Shft[1] := 0;  k := 0;
FOR q := 2 TO m DO
   WHILE (k > 0) ∧ (P[k+1] ≠ P[q]) DO
                                        k := Shft[k]
   ENDWHILE;
   IF P[k+1] = P[q] THEN k := k+1 ENDIF;
   Shft[q] := k;
ENDFOR
```

Und wie sieht es mit der Komplexität des KMP-Algorithmus aus?

Auf den ersten Blick könnte man meinen, dass die Laufzeit bei zwei geschachtelten Schleifen (FOR, WHILE) im wesentlichen auf die Größenordnung $m \cdot n$ käme...

Bei genauerem Hinsehen stellt sich aber heraus, dass die WHILE-Schleife *insgesamt* nur $O(n)$ mal ausgeführt wird. Damit liegt die Gesamtlaufzeit auch in $O(n)$.

Beispiele für Verschiebetabellen

Wie sehen die Verschiebetabellen für unsere beiden bisherigen Beispiele aus?

Wir berechnen zuerst die Tabelle für das Wort `ananas`:

Aus der Definition lesen wir die Shft-Werte ab. Es ergibt sich die Folge 001230.

Der Ablauf von BerechneTabelle ist in diesem Fall so:

$Shft[1] = 0,\ k = 0$

$q = 2:$ WHILE $0 > 0$... F
IF $P[1] = P[2]$... F
$Shft[2] = 0$

$q = 3:$ WHILE $0 > 0$... F
IF $P[1] = P[3]$ THEN $k = 1$
$Shft[3] = 1$

$q = 4:$ WHILE $1 > 0 \wedge P[2] \neq P[4]$... F
IF $P[2] = P[4]$ THEN $k = 2$
$Shft[4] = 2$

$q = 5:$ WHILE $2 > 0 \wedge P[3] \neq P[5]$... F
IF $P[3] = P[5]$ THEN $k = 3$
$Shft[5] = 3$

$q = 6:$ WHILE $3 > 0 \wedge P[4] \neq P[6]$ DO $k = 1$
WHILE $1 > 0 \wedge P[2] \neq P[6]$ DO $k = 0$
WHILE $0 > 0$... F
IF $P[1] = P[6]$... F
$Shft[6] = 0$

Beispiel: adac

Beim Wort adac ist der Ablauf zu langweilig, weil das Muster so kurz ist. Daher nehmen wir stattdessen adacadac:

Der Ablauf von BerechneTabelle ist in diesem Fall so:

$Shft[1] = 0,\ k = 0$

$q = 2:$ $0 > 0?$ $P[1] = P[2]?$ $Shft[2] = 0$

$q = 3:$ $0 > 0?$ $P[1] = P[3]$, also $k = 1$, $Shft[3] = 1$

$q = 4:$ $1 > 0$, $P[2] \neq P[4]$, also $k = 0$

$0 > 0?$ $P[1] = P[4]?$ $Shft[4] = 0$

$q = 5:$ $0 > 0?$ $P[1] = P[5]$, also $k = 1$, $Shft[5] = 1$

$q = 6:$ $1 > 0$, $P[2] \neq P[6]?$ $P[2] = P[6]$, also $k = 2$, $Shft[6] = 2$

$q = 7:$ $2 > 0$, $P[3] \neq P[7]?$ $P[3] = P[7]$, also $k = 3$, $Shft[7] = 3$

$q = 8:$ $3 > 0$, $P[4] \neq P[8]?$ $P[4] = P[8]$, also $k = 4$, $Shft[8] = 4$

Die Shift-Werte ergeben hier also die Folge 00101234.

Boyer-Moore

Auch der Algorithmus von Boyer und Moore löst das String Matching Problem sehr effizient. In der Vorlesung wird die Grundidee nur sehr kurz erläutert. Interessierte Teilnehmer können sich in der Literatur informieren.

Stichpunkte: bad character-Strategie und good suffix-Strategie

Anmerkung:

Unter anderem stellt Wikipedia im Eintrag über den Boyer-Moore-Algorithmus (Stand 25.11.2019) eine sehr kurze, aber verständliche Einführung mit mehreren Beispielen zur Verfügung.