Textsuche

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de , Raum 149B

Version: 1.1.6.1

Quelle: Die Folien sind teilweise inspiriert von oder basierend auf Lehrmaterial von Prof. Dr. Ritterbusch.

Folien: https://delors.github.io/theo-algo-text_suche/folien.de.md.html
https://delors.github.io/theo-algo-text_suche/folien.de.md.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

KI Verwendung: Bei der Erstellung der Unterlagen wurden KI Assistenten (insbesondere Claude aber ggf. auch ChatGPT, Ollama mit Gemma/LLama/Qwen oder OpenCode mit Kimi/Qwen/Deepseek...) unterstützend eingesetzt. Dies erfolgte insbesondere zur Unterstützung bei der Generierung von Grafiken (d. h. SVG Dateien), oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert und angepasst.

Arrays und Textsuche

Texte können als unsortierte Arrays von Zeichen verstanden werden. Eine typische Frage ist hier das Finden von Textsequenzen im Text.

Beispiel: Gesucht ist "labore"

> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.
>
> —loremipsum.de

Einfache Textsuche

```
function NaiveTextSearch(text,needle)
    n := length(text)
    m := length(needle)
    for i := 1,...,n-m + 1 do
        j := 0
        while text[i + j] == needle[j + 1] do
            j := j + 1
            if j == m then
                return i // Found at i
    return nil
```

> ◎ Bemerkung
>
> Die Laufzeit der einfachen Textsuche kann asymptotisch durch O(n·m) abgeschätzt werden.

Beispiel bei einfacher Suche nach `aaab` in `aaaaaaaab`:

```
a a a a a a a a b
_________________

a a a b
  a a a b
    a a a b
      a a a b
        a a a b
          a a a b
```

Sind so viele Vergleiche notwendig?

1. Knuth-Morris-Pratt (KMP) Verfahren

Grundlagen

Das Verfahren von Knuth-Morris-Pratt vermeidet unnötige Vergleiche, da es zunächst die Suchwortteile auf den größten Rand, also das größte Präfix, das auch Postfix ist, untersucht.

Definition: Präfix, Postfix und Rand

Für ein Wort $w = (w_1, ..., w_n)$ mit $0 \leq k \leq n$. sind:

- die Präfixe $p^{(k)} = (w_1, ..., w_k)$ und
- die Postfixe $q^{(k)} = (w_{n-k+1}, ..., w_n)$

Ist $p^{(k)} = q^{(k)} = r^{(k)}$ für ein $0 \leq k < n$, so ist $r(k)$ ein Rand von w.

Für $k < n$ werden $p^{(k)}$ und $q^{(k)}$ auch echte Prä- und Postfixe genannt.

Beispiel/Idee

Text	010110101
Gesucht/Muster	010101
Übereinstimmung	✓✓✓✓✗

Beobachtungen:

1. Wir haben an Stelle 5 ein Mismatch.
2. Wenn wir im Text das Muster um eine Stelle nach rechts verschoben suchen, so haben wir wieder ein Mismatch.

Frage

Wie weit kann man also das Muster im Allgemeinen verschoben werden ohne ein Vorkommen zu übersehen?

Beispiel/Idee

	1.	2.
Text	01101100	0102111
Gesucht/Muster	01100	010201
Übereinstimmungen	✓✓✓✓✗	✓✓✓✓✗

Beobachtungen bzgl.:

1. Beim Mismatch an Stelle 5 kann das Muster „nur“ um 3 Stellen nach rechts verschoben werden.

   (Wir haben 4 Übereinstimmungen und einen Rand von 1 für den Teil mit den Übereinstimmungen. Daraus ergibt sich eine Verschiebung von 4 - 1 = 3!)
2. Beim Mismatch an Stelle 5 kann das Muster um 4 Stellen nach rechts verschoben werden.

Wie weit wir das Muster verschieben können, hängt also vom Rand des Teils des Musters ab, der bereits übereinstimmt.

Bei einem Mismatch betrachtet man die Anzahl j der bereits übereinstimmenden Zeichen (das gematchte Präfix des Musters) und bestimmt dessen längsten echten Rand der Länge b. Das Muster wird dann um j – b Stellen nach rechts verschoben; der Textzeiger bleibt auf dem Mismatch-Zeichen, und der Vergleich wird ab Position b im Muster fortgesetzt, da die b Randzeichen bereits übereinstimmen. Gibt es keinen Rand (b = 0), verschiebt man entsprechend um j Stellen und vergleicht das Mismatch-Zeichen erneut mit dem Musteranfang.

Beispiel

Das Wort $aufkauf$ hat die *echten* Präfixe und Postfixe:

$$\{p^{(k)} : 0 \leq k < n\} = \{\varepsilon, a, au, auf, aufk, aufka, aufkau\}$$
$$\{q^{(k)} : 0 \leq k < n\} = \{\varepsilon, f, uf, auf, kauf, fkauf, ufkauf\}$$

und die Ränder:

$$\{r^{(k)} : 0 \leq k < n\} = \{\varepsilon, auf\}.$$

Das bedeutet, dass wenn $aufkauf$ erkannt wurde, die letzten drei Buchstaben schon den nächsten Treffer einleiten können, wie beispielsweise in $aufkaufkauf$.

Das KMP-Verfahren fängt nicht immer von vorne an, sondern prüft, ob *ein Rand eines Präfixes* (ohne das leere Wort ε) ausgenutzt werden kann. Dazu werden die entsprechenden größten Ränder bestimmt.

Beispiel: ananas

Präfixe $\setminus\{\varepsilon\}$	Größter Rand	Länge des Randes
a	ε	0
an	ε	0
a̲nā	a	1
a̲n̲ā n̄	an	2
a̲n̲ā n̄ā	ana	3
ananas	ε	0

Beispiel: axaaxax

Präfixe $\setminus\{\varepsilon\}$	Größter Rand	Länge des Randes
a	ε	0
ax	ε	0
a̲xā	a	1
a̲xaā	a	1
a̲xa̲ā x̄	ax	2
a̲xa̲ā x̄ā	axa	3
a̲xaaxā x̄	ax	2

Die Idee ist also, dass wir beim Musterabgleich nach einem Mismatch, wenn der übereinstimmende Teil einen Rand hat, beim Abgleich des Musters an einer späteren Stelle - basierend auf der Größe des Randes - weitermachen können. Wir müssen also nicht immer das ganze Muster von vorne anfangen zu vergleichen.

✎ Übung

1.1. Ränder und Randlängen bestimmen

Bestimmen Sie die Ränder und die Längen der $Präfixe - \varepsilon$ für die Worte:

1. $tultatul$
2. $eikleike$
3. $okokorok$
4. $trattrad$

Beispiel für eine KMP-Textsuche

Gesucht wird `ananas` in `saansanananas`

```
   s a a n s a n a n a n a s
i  ———————————————————————
1  a
...
3    a n
...
5      a n a
...                          Hinweise zum Algorithmus:
11           a n a n a s     Beim Auftreten des Mismatch (Zeile 7) ist
...                          q=5 und wird auf p[5]=3 (Zeile 8) gesetzt
13               a n a n a s
```

◆ Bemerkung

Dargestellt sind die Fälle, in denen ein Mismatch auftritt. `i` ist der Index des aktuellen Zeichen im Text, das mit dem Muster verglichen wird.

Algorithmus

```
function ComputePrefixFunction(needle)
    m := length(needle)
    sei B[1...m] ein Array  // Array für die Längen der Ränder der Teilworte
    B[1] := 0
    j := 0                  // j ist die Länge des echten Randes
    for i := 2,...,m do
        j := j + 1
        while j > 0 and needle[j] ≠ needle[i] do
            if j > 1 then
                j := B[j-1] + 1
            else
                j := 0
        B[i] := j
    return B
```

Die Funktion $ComputePrefixFunction$ berechnet die größten Werte der Präfixe für das Suchwort $needle$ der Länge m und gibt diese als Array B zurück. Das Array B enthält somit die größten Ränder der Präfixe $needle[1, ..., i]$.

(Der Wert von $B[1]$ ist immer 0, da es keinen Rand gibt.)

Komplexität: O(m)

```
function KMP(text,needle)
    n := length(text), m := length(needle)
    B := ComputePrefixFunction(needle)
    q := 0              // Anzahl der übereinstimmenden Zeichen
    R := []             // Ergebnisliste der Indizes der Übereinstimmungen
    for i := 1,...,n do
        while q > 0 and needle[q + 1] ≠ text[i] do
            q := B[q]   // ... die nächsten Zeichen stimmen nicht überein
        if needle[q + 1] == text[i] then
            q := q + 1  // Übereinstimmung
        if q == m then
            R append (i - m + 1)
            q := B[q]   // Suche nach nächster Übereinstimmung
    return R
```

Komplexität: O(n+m)

Implementierung in Java (0-basiert)

```
static int[] computePrefixFunction(String needle) {
    int m = needle.length();
    int[] b = new int[m];
    b[0] = 0;
    int j = 0;

    for (int i = 1; i < m; i++) {
        j++;
        while (j > 0 && needle.charAt(j - 1) != needle.charAt(i)) {
            if (j > 1) {
                j = b[j - 2] + 1;
            } else {
                j = 0;
```

```
            }
        }
        b[i] = j;
    }
    return b;
}
```

✎ Übung

1.2. KMP-Algorithmus

Bestimmen Sie die Randlängen der Muster und stellen Sie die Teilschritte bei der Durchführung des KMP-Algorithmus zur Suche des Wortes/Muster im Text dar.

Stellen Sie insbesondere die Fälle dar in denen ein Mismatch auftritt.

Muster	Text
`aaab`	`aaaaaaaab`
`barbara`	`abbabarabarbarbara`

2. Boyer-Moore-Algorithmus (vereinfacht)

Grundlagen

Beobachtung

Häufig ist das Alphabet des Textes größer als das des Musters.

- Der Algorithmus vergleicht das Muster (Pattern) von rechts nach links mit dem Text.

  Bemerkung

  Viele andere Algorithmen führen die Vergleiche von links nach rechts durch.

- Der Boyer-Moore-Algorithmus nutzt dies aus, indem er die Verschiebung des Musters anhand des letzten Zeichens des Musters und des Textes vornimmt.

Wird beispielsweise das Wort

`Banane`

im Text

`Orangen,␣Ananas␣und␣Bananen`

gesucht, so wird zunächst die Sprungtabelle für das verwendete Alphabet in Bezug auf das Suchwort (`Banane` mit Länge 6) bestimmt:

Zeichen im Text	␣	,	A	B	O	a	d	e	g	n	r	s	u
Sprung	6	6	6	5	6	2	6	0	6	1	6	6	6

```
O r a n g e n , ␣ A n a n a s ␣ u n d ␣ B a n a n e n
B a n a n e
        B a n a n e
          B a n a n e
             B a n a n e
                B a n a n e
                          B a n a n e
                             B a n a n e
                                B a n a n e
                                  B a n a n e
```

Bemerkung

Unterschrichen sind die durchgeführten Vergleiche. Die Verschiebung des Musters erfolgt anhand des letzten Zeichens des Musters und des Textes, dass nicht übereinstimmt. Dabei ist die Verschiebung durch das Zeichen des Textes gegeben, das nicht mit dem Muster übereinstimmt.

Komplexität

Im guten und häufigen Fall erreicht das Verfahren $O(\frac{n}{m})$, aber in speziellen Fällen ist auch $O(n \cdot m)$ möglich.

Bei der Sprungtabelle handelt es sich um eine Tabelle, die *für jedes Zeichen des Alphabets des Textes* die Verschiebung des Musters angibt, wenn das Zeichen im Text mit dem Muster nicht übereinstimmt. Die Zeichen A, O, d, g, r, s, u, , und das Leerzeichen haben die größte Verschiebung, da sie nicht im Muster vorkommen. Das Zeichen e hat die kleinste Verschiebung, da es das letzte Zeichen des Musters ist. Das Zeichen n hat eine Verschiebung von 1, da es im Muster als vorletztes Zeichen vorkommt, das Zeichen a hat eine Verschiebung von 2, da das späteste Vorkommen an drittletzer

Stelle ist, und das Zeichen `B` hat eine Verschiebung von 5, da es nur einmal vorkommt und das erste Zeichen des Musters ist.

✎ Übung - Boyer-Moore-Algorithmus

2.1. „belli“

Suchen Sie das Wort `belli` im Text `It is a dark time for the Rebellion.`

Bauen Sie erst die Sprungtabelle auf:

Zeichen	␣	.	I	R	a	b	d	e	f	h	i	k	l	m	n	o	r	s	t
Häufigkeit																			

2.2. "barbara"

Suchen Sie das Wort `barbara` im Text `abbabarabarbarbara`

Bauen Sie zuerst die Sprungtabelle auf.