Algorithmen und Datenstrukturen

Zeichenkettensuche

Prof. Justus Piater, Ph.D.

4. Juni 2021

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

1 Problemstellung und *Brute-Force*-Algorithmus

Pattern Matching (Musterabgleich) [Slide 1]

Gegeben: Zwei Zeichenketten $T[0, \ldots, n-1]$ (***Text***) und $P[0, \ldots, m-1]$ (***Pattern***, ***Muster***).

Gesucht: Der Index des ersten Vorkommens von P in T (oder die Indizes aller Vorkommen), oder andernfalls -1.

Ein *Brute Force*-Algorithmus [Slide 2]

```
Algorithm findBrute(T, P):
  Require: a text T of n characters and a pattern P of m characters.
  Ensure: Return the lowest index at which P begins in T (or else −1).
  for j ← 0 to n − m do // try every starting index j within T
    k ← 0              // k is index into P
    while k < m and T[j + k] = P[k] do // kth char of P matches
      k ← k + 1
    if k = m then // if we reach the end of P,
      return j      // substring T[j, ..., j + m − 1] is a match
  return −1         // search failed
```

Laufzeit als Funktion von n und m?

$O(nm)$

Brute-Force: Beispiel [Slide 3]

Brute-Force: Beispiel [Slide 4]

a	b	r	a	k	a	d	a	b	r	a		a	b	e	r		a	b	r	a	k	a	d	a	b	r	e
a	b	r	a	k	a	d	a	b	r	e																	
	a	b	r	a	k	a	d	a	b	r	e																
		a	b	r	a	k	a	d	a	b	r	e															
			a	b	r	a	k	a	d	a	b	r	e														
				a	b	r	a	k	a	d	a	b	r	e													
					a	b	r	a	k	a	d	a	b	r	e												
						a	b	r	a	k	a	d	a	b	r	e											
							a	b	r	a	k	a	d	a	b	r	e										
								a	b	r	a	k	a	d	a	b	r	e									
									a	b	r	a	k	a	d	a	b	r	e								
										a	b	r	a	k	a	d	a	b	r	e							
											a	b	r	a	k	a	d	a	b	r	e						
												a	b	r	a	k	a	d	a	b	r	e					
													a	b	r	a	k	a	d	a	b	r	e				
														a	b	r	a	k	a	d	a	b	r	e			
															a	b	r	a	k	a	d	a	b	r	e		
																a	b	r	a	k	a	d	a	b	r	e	
																	a	b	r	a	k	a	d	a	b	r	e

Grüne Felder markieren Übereinstimmungen zwischen $T[j + k]$ und $P[k]$, orange Felder Nicht-Übereinstimmungen.

***Brute Force*: ein schlechtester Fall [Slide 5]**

a	a	a	b	a	a	a	b	a	a	a	b	a	a	a	b
a	a	a	a												
	a	a	a	a											
		a	a	a	a										
			a	a	a	a									
				a	a	a	a								
					a	a	a	a							
						a	a	a	a						
							a	a	a	a					
								a	a	a	a				
									a	a	a	a			
										a	a	a	a		
											a	a	a	a	
												a	a	a	a

Die hier gezeigte Matrix hat $n - m + 1$ Zeilen, also belegt das diagonale Band genau $(n - m + 1)m = nm + m - m^2$ Felder. Da n mindestens so groß sein muss wie m, sind dies $\Omega(nm)$ Felder.
In diesem Beispiel beinhalten mehr als die Hälfte der Felder dieses diagonalen Bands Vergleiche, also sind dies $\Omega(nm)$ Vergleiche.

2 Knuth-Morris-Pratt-Algorithmus

Der Knuth-Morris-Pratt-Algorithmus (KMP) [Slide 6]

Beobachtung: Wenn der Brute-Force-Algorithmus zum nächsten Zeichen n schreitet, verwirft er alle Informationen, die er bei den erfolgreichen Vergleichen über m gesammelt hat.

Idee: Bei Nicht-Übereinstimmung des aktuellen Zeichens,

- verschiebe das Muster *so weit wie möglich* nach rechts;
- vermeide *redundante* Vergleiche.

Motivierendes Beispiel:

a	b	r	a	k	a	d	a	b	r	a							
a	b	r	a	k	a	d	a	b	r	e							
							a	b	r	a	k	a	d	a	b	r	e

- Beim der Nicht-Übereinstimmung zwischen `a` und `e` (orange) wissen wir aufgrund der vorhergehenden Vergleiche und der Struktur von P, dass die nächstmögliche Übereinstimmung des ***Präfix*** `abr` des Musters P beim ***Suffix*** des soeben erfolgreich verglichenen Textabschnitts `abrakadabr` liegt. Wir können das Muster also bis dorthin verschieben und dabei geschlagene 6 Felder überspringen.
- Aufgrund dieser bereits bekannten Übereinstimmung zwischen diesem Präfix von P mit dem Suffix des letzen Textabschnitts müssen diese nicht mehr miteinander verglichen werden (blau).

Wichtig

Die Information, wie weit wir das Muster P vorrücken können, können wir *im Voraus* berechnen. Sie hängt nicht vom Text T ab, da hier beide identisch sind (grün).

KMP: *Failure*-Funktion [Slide 7]

Failure-Funktion $f(k) =$

- die Länge des längsten ***Präfix*** des Musters P, das ebenfalls ein ***Suffix*** von $P[1, \ldots, k]$ ist, also

  $P[0]$ wird ausgelassen, da das Muster immer um mindestens 1 Zeichen weitergeschoben wird.

- wie viele der unmittelbar vorhergehenden Zeichen *ohne wiederholten Vergleich* für den nächsten Musterabgleich-Versuch wiederverwendet werden können:

k	0	1	2	3	4	5	6	7	8	9	10
$P[k]$	`a`	`b`	`r`	`a`	`k`	`a`	`d`	`a`	`b`	`r`	`e`
$f(k)$	0	0	0	1	0	1	0	1	2	3	0

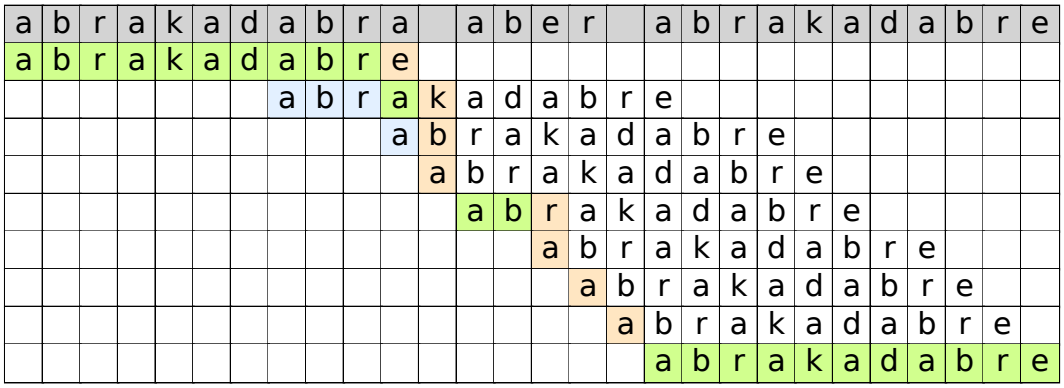
KMP: Algorithmus [Slide 8]

```
Algorithm findKMP(T, P):
  Require: a text T of n characters and a pattern P of m characters.
  Ensure: Return the lowest index at which P begins in T (or else −1).
  if m = 0 then return 0 // trivial search for empty string
  f ← computeFailKMP(P) // computed by private utility
  j ← 0                 // index into T
  k ← 0                 // index into P
  while j < n do
    if T[j] = P[k] then // P[0, ..., k] matched thus far
      if k = m − 1 then return j − k // match is complete
      j ← j + 1 // otherwise, try to extend match
      k ← k + 1
    else if k > 0 then
      k ← f[k − 1] // reuse suffix of P[0, ..., k − 1]
    else
      j ← j + 1 // nothing to reuse; start over at next position
  return −1     // reached end without match
```

`findKMP()`: Beispiel [Slide 9]

`findKMP()`: Beispiel [Slide 10]

k	0	1	2	3	4	5	6	7	8	9	10
$P[k]$	a	b	r	a	k	a	d	a	b	r	e
$f(k)$	0	0	0	1	0	1	0	1	2	3	0



Grüne Felder markieren Übereinstimmungen zwischen $T[j]$ und $P[k]$, orange Felder Nicht-Übereinstimmungen. Blaue Felder wurden überhaupt nicht verglichen, da feststeht, dass der Vergleich Übereinstimmung ergeben würde.

3 KMP *Failure*-Funktion

KMP *Failure*-Funktion: Algorithmus [Slide 11]

```
Algorithm computeFailKMP(P):
  Require: a pattern P of m characters.
  Ensure: f[k] is the length of the longest prefix of P
                that is also a suffix of P[1, ..., k].
  f ← [0, ..., 0] // array of m zeros
  j ← 1
  k ← 0
  while j < m // compute f[j] during this pass, if nonzero
    if P[j] = P[k] then // k + 1 characters match thus far
      f[j] ← k + 1
      j ← j + 1
      k ← k + 1
    else if k > 0 // k follows a matching prefix
      k ← f[k − 1] // reuse suffix of P[0, ..., k − 1]
    else // no match found starting at j
      j ← j + 1 // nothing to reuse; start over at next position
  return f
```

- Diese Funktion wendet das KMP-Prinzip an, um das Muster mit sich selbst zu vergleichen.
- Da immer $j > k$ ist, ist $f(k-1)$ immer definiert, wenn es benötigt wird.

`computeFailKMP()`: Beispiel [Slide 12]

KMP: Laufzeitanalyse [Slide 13]

Sei $s = j - k \leq n$ die aktuelle Verschiebung des Musters im Verhältnis zum Text.

Bei jeder Iteration der `while`-Schleife tritt genau einer der folgenden Fälle auf:

1. Ist $T[j] = P[k]$, dann wachsen j und k jeweils um 1; s bleibt also unverändert.
2. Ist $T[j] \neq P[k]$ und $k > 0$, dann bleibt j unverändert, und s wächst von $j - k$ auf $j - f(k-1)$, also um $k - f(k-1) \geq 1$.
3. Ist $T[j] \neq P[k]$ und $k = 0$, dann wächst j um 1 und s wächst um 1, da k unverändert bleibt.

Bei jeder Iteration wachsen daher j oder s (oder beide) um mindestens 1. Die Gesamtzahl der Iterationen ist daher maximal $2n$.

Entweder geht es im Text weiter, oder das Muster wird weitergeschoben.

Die Analyse von `computeFailKMP()` verläuft analog, mit einem Muster der Länge m, das mit sich selbst verglichen wird; ihre Laufzeit ist also $O(m)$.

Es folgt die folgende Proposition:

KMP: Laufzeit und Korrektheit [Slide 14]

Proposition: Der KMP-Algorithmus sucht ein Muster der Länge m in einem Text der Länge n in einer Laufzeit von $O(m+n)$. Für $m \leq n$ ist dies gleich $O(n)$.

Anmerkung

Dies ist asymptotisch optimal.

Jeder Suchalgorithmus muss schlimmstenfalls alle Zeichen des Texts und alle Zeichen des Musters mindestens einmal überprüfen.

Korrektheit: Die *Failure*-Funktion garantiert, dass die übersprungenen Vergleiche überflüssig sind. (Dies folgt aus ihrer Definition.)

KMP: Es geht immer vorwärts. [Slide 15]

a	a	a	b	a	a	a	b	a	a	a	b	a	a	a	b
a	a	a	a												
	a	a	a	a											
		a	a	a	a										
			a	a	a	a									
				a	a	a	a								
					a	a	a	a							
						a	a	a	a						
							a	a	a	a					
								a	a	a	a				
									a	a	a	a			
										a	a	a	a		
											a	a	a	a	
												a	a	a	a

Im Vergleich zum *Brute-Force*-Algorithmus entfallen die Vergleiche in den blauen Dreiecken. Da die Matrix nicht mehr als $O(n)$ Zeilen haben kann, ist die Gesamtanzahl der Vergleiche (grüne + rote Felder) $O(n)$.

4 Epilog

Epilog [Slide 16]

Selbst so ein einfaches Problem wie die String-Suche lässt sich in seiner asymptotischen Laufzeit deutlich verbessern –

- dank einer durch sorgfältiges Hinschauen gewonnenen Einsicht, und
- auf Kosten eines etwas komplizierteren Algorithmus.

Das Ergebnis ist in seiner Eleganz ein Schmuckstück der Algorithmik.

Bibliographie [Slide 17]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.