Algorithmen und Datenstrukturen

Zeichenkettensuche

Prof. Justus Piater, Ph.D.

15. Juni 2022

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

> Wenn Sie Texte schreiben, nutzen Sie alle sicher immer wieder die Suchen/Ersetzen-Funktion. Und beim Lesen von Texten im Web-Browser oder in PDF-Datein suchen Sie sicher öfters nach bestimmten Wörtern. Die Zeichenkettensuche ist wohl eine der am häufigsten am Computer verwendeten Funktionen überhaupt. Sie ist allgegenwärtig und offensichtlich einfach zu implementieren, sollte man meinen. Einfach den Text sequenziell durchgehen und an jeder Stelle nachprüfen, ob er hier mit der gesuchten Zeichenkette übereinstimmt.
>
> Bei genauerem Hinsehen stellt sich heraus, dass es weit bessere Algorithmen gibt als dieses einfache Verfahren. Zeichenkettensuche ist ein gutes Beispiel dafür, wie man durch genaues Hinsehen und systematisches Schlussfolgern für scheinbar einfache Probleme neue, trickreiche, elegante, effiziente Algorithmen finden kann.

Pattern Matching (Musterabgleich) [Slide 1]

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

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

> Beginnen wir mit einer präzisen Formulierung des Problems. Gegeben sind zwei Zeichenketten T der Länge n und P der Länge m. Gesucht ist der Index des ersten Vorkommens von P in T, oder -1 falls P überhaupt nicht in T vorkommt.

Etwas allgemeiner können wir auch nach *allen* Vorkommen von P in T fragen. Diese Erweiterung ist jedoch trivial, und wir konzentrieren uns hier auf das Auffinden des ersten Auftretens von P in T.

P steht für das englische *Pattern*, also Muster, und T steht für Text. Wir suchen ein gegebenes Muster in einem Text. Zeichenkettensuche in Text ist jedoch lediglich ein Spezialfall des allgemeinen Problems, eine Mustersequenz P in einer Sequenz T zu finden. Beispielsweise kann es sich bei den Sequenzen auch um Zahlen handeln, oder um Basenpaare in einer DNA-Sequenz.

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?

$O(nm)$

Hier sehen wir unser naives Verfahren formalisiert als Algorithmus. In der äußeren Schleife gehen wir den Text T Zeichen für Zeichen von vorne bis hinten durch. Bei jedem Zeichen j prüfen wir nach, ob bei $T[j]$ das Muster P beginnt.

Dies tun wir mittels der inneren Schleife, deren Laufvariable k über die Zeichen von P iteriert. Treffen wir dabei auf zwei zwischen T und P *verschiedene* Zeichen, dann brechen wir die innere Schleife erfolglos ab und setzen die äußere Schleife fort.

Erreichen wir hingegen das Ende von P, dann haben wir eine Instanz von P in T gefunden. In diesem Fall kehrt die Funktion zurück, und liefert den Index j des ersten Zeichens von P in T zurück.

Erreicht die äußere Schleife das Ende von T, dann wissen wir, dass T das Muster P *nicht* enthält.

Brute-Force: Beispiel [Slide 3]

Beobachten wir diesen Algorithmus in Aktion. Wir suchen hier das Muster `abrakadabre` im grau hinterlegten Text. Jede Zeile entspricht einer Iteration j der äußeren Schleife. Der obere der beiden dunkelblauen Ring markiert $T[j + k]$ in der inneren Schleife, und der untere blaue Ring markiert $P[k]$. Sind diese beiden Zeichen identisch, dann wird das entsprechende Feld in Zeile j grün hinterlegt, ansonsten orange.

Bei jedem ungleichen Paar, orange markiert, bricht die innere Schleife ab, und die äußere beginnt ihre nächste Iteration, hier dargestellt in einer neue Zeile. Dabei rückt das Muster P um ein Zeichen vor, und die innere Schleife beginnt von vorne.

Beachten Sie, dass die beiden dunkelblauen Ringe bei jeder neuen Zeile wieder an den Anfang des Musters zurückspringen.

Was ist die Laufzeit dieses Algorithmus? Wir haben zwei verschachtelte Schleifen. Die innere Schleife iteriert über die Länge m des Musters P und enthält ausschließlich Anweisungen mit konstanter Laufzeit. Damit hat die innere Schleife eine asymptotische Laufzeit von $O(m)$.

Die äußere Schleife iteriert über die Länge n von T, minus der Länge m des Musters P. Das Muster kann beliebig kurz sein; also iteriert die äußere Schleife $O(n)$ Mal.

Die komplexeste Anweisung innerhalb der äußeren Schleife ist die innere Schleife mit Laufzeit $O(m)$. Weiters existiert dort neben der äußeren Schleife lediglich die `return`-Anweisung mit konstanter Laufzeit. Daher ist die Gesamtaufzeit von `findBrute()` $O(nm)$.

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.

Eine engere Charakterisierung der asymptotischen Laufzeit mittels Groß-O ist nicht möglich.

Hier sehen wir ein Beispiel eines schlechtesten Falls. Wir suchen das Muster `aaaa` im Text `aaabaaabaaabaaab`. Bei jeder Iteration der äußeren Schleife findet eine maximale Anzahl Vergleiche in der inneren Schleife statt. Zur Erinnnerung: Jedes farbige Feld entspricht einem Vergleich. Grüne Felder markieren gleiche Paare, und orange Felder ungleiche Paare.

Wir sehen, dass das Muster P sich als diagonales Band der Breite m durch die Matrix zieht. Diese Matrix hat $n - m + 1$ Zeilen, also belegt dieses diagonale Band genau $(n - m + 1)m$ Felder. Da n mindestens so groß sein muss wie m, sind dies $\Omega(nm)$ Felder.

Anhand der Dreiecksstruktur der farbigen Felder sehen wir anschaulich, dass mehr als die Hälfte der Felder dieses diagonalen Bands tatsächlich Vergleiche beinhalten. Wir haben in diesem Beispiel also $\Omega(nm)$ Vergleiche.

Wenn wir uns diese Vergleiche anschauen, dann sehen wir, dass wir viele Zeichenpaare wiederholt miteinander vergleichen. Beispielsweise vergleichen wir das zweite `a` des Textes mit dem zweiten `a` des Musters, und nach dem Vorrücken des Musters um ein Zeichen vergleichen wir das zweite `a` des Textes mit dem *ersten* `a` des Musters.

Dabei ist dies eigentlich völlig unnötig! Wenn wir irgendwie ausnutzen können, dass die ersten drei Zeichen des Musters identisch sind, dann können wir uns diese redundanten Vergleiche sparen.

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	

- Bei 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).

> Wir haben gesehen, dass der Brute-Force-Algorithmus viele redundante Vergleiche ausführt, weil er nicht ausnutzt, wo das Muster sich selbst ähnelt.
>
> Das Vermeiden solcher redundanter Vergleiche ist die Grundidee des von James H. Morris entwickelten Musterabgleich-Algorithmus. Er wurde unabhängig und fast zeitgleich von Donald Knuth entdeckt und einige Jahre später, 1977, von Knuth, Morris und Vaughan Pratt veröffentlicht.
>
> Um ihren Ansatz zu motivieren, schauen wir uns dieses Beispiel an. Wir suchen die Zeichenkette `abrakadabre`, finden jedoch `abrakadabra`. Erst der letzte Buchstabe des Musters stimmt mit dem Text nicht überein.
>
> An dieser Stelle würde der *Brute-Force*-Algorithmus das Muster um ein Zeichen vorrücken, wieder von vorne beginnen, und damit alles wieder verwerfen, was wir durch die vorhergehenden Vergleiche über den Text erfahren haben.
>
> Wir haben nämlich etwas sehr Wichtiges über den Text erfahren: Wir wissen nun, dass die ersten 10 Zeichen des Texts mit den ersten 10 Zeichen des Musters identisch sind (hier grün hinterlegt). Aufgrund der Struktur des Musters können wir nun erschließen, dass das Muster bei Vorrückung um 1 bis 6 Zeichen unmöglich dem Text gleichen kann.
>
> Erst bei Vorrückung um 7 Zeichen müssen wir wieder aufpassen. Hier nämlich stimmen die ersten 3 Zeichen des Musters mit den letzten 3 bereits verglichenen Zeichen des

Texts überein. Mit anderen Worten: An dieser Stelle ist das 3 Zeichen lange *Suffix* `abr` des zuletzt erfolgreich verglichenen Texts mit dem *Präfix* `abr` des Musters identisch.

Bei kürzerer Verschiebung kann keine Übereinstimmung möglich sein, denn dann wäre das mit dem Text übereinstimmende Präfix des Musters entsprechend länger.

Da also bei kürzerer Verschiebung keine Übereinstimmung möglich ist, können wir für unseren nächsten Vergleichsversuch unser Muster um ganze 7 Zeichen vorrücken. Damit haben wir schon einmal jede Menge Zeit gespart.

Zweitens wissen wir bereits, dass die Zeichen `abr` hier zwischen Muster und Text übereinstimmen. Hier sparen wir also noch mehr Zeit, indem wir diese nicht noch einmal vergleichen. Darum sind diese Zeichen hier blau hinterlegt.

Woher genau wissen wir, dass wir aufgrund der Struktur des Musters erst nach 7 Zeichen Vorrückung wieder mit einer Übereinstimmung mit dem Text rechnen müssen? Und woher genau wissen wir, dass wir die ersten drei Zeichen des Musters diesmal überspringen können?

Diese Information können wir unabhängig vom Text vorausberechnen, denn der Text ist ja bis hier mit dem Muster identisch! Genau diese Information wurde vom *Brute-Force*-Algorithmus verworfen.

Um es noch einmal zu betonen: Die entscheidende Information ist die Länge der längsten Zeichenkette, die sowohl ein Präfix des Musters ist, als auch ein Suffix des zuletzt erfolgreich verglichenen Textabschnitts. Da dieser Textabschnitt mit dem Muster identisch ist, finden wir ihn an entsprechender Stelle im Muster.

KMP: *Failure*-Funktion [Slide 7]

Failure-Funktion $f(k) =$

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

  $P[0]$ wird ausgelassen, da das Muster immer um mindestens 1 Zeichen weitergeschoben wird.
  Übung: Was passiert, wenn wir hier $P[1, \dots, k]$ durch $P[0, \dots, k]$ ersetzen?

- 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

Was wir vorausberechnen müssen, ist also für jeden beginnenden Teilabschnitt von P die Länge seines längsten Präfix, das ebenfalls sein Suffix ist. Dies ist genau die Anzahl Zeichen am Anfang des Musters, die wir nach seinem Vorrücken nicht noch einmal vergleichen müssen.

Diese Präfix-Längen formalisieren wir hier mit der sogenannten *Failure*-Funktion $f(k)$. Dies ist ein Array mit m Zellen, deren Wert $f(k)$ die Länge des längsten Präfix des Musters P enthält, das ebenfalls ein Suffix des nach dem Index k abgeschnittenen Musters ist. Auch den Index 0 von P berücksichtigen wir beim Suffix nicht, da das Muster immer um mindestens ein Zeichen vorrücken muss.

Sehen wir unten die *Failure*-Funktion $f(k)$ für unser Muster $P =$`abrakadabre`. Wir betrachten immer Präfixe des gesamten P, und $f(k)$ ist die Länge des längsten Präfix von P, das auch ein Suffix von $P[1, ..., k]$ ist.

Bei $k = 0$ ist $P[1, ..., k]$ leer, also ist $f(0)$ immer 0.

Bei $k = 1$ ist hier das Präfix `a` kein Suffix von `b`, also ist $f(1) = 0$. Ebenso ist bei $k = 2$ weder das Präfix `a` noch das Präfix `ab` ein Suffix von `br`; daher ist $f(2) = 0$.

Bei $k = 3$ ist in der Tat das Präfix `a` ein Suffix von `bra`. Die längeren Präfixe `ab` und `abr` sind jedoch keine Suffixe von `bra`, also ist $f(3) = 1$. Auf dieselbe Weise sind auch $f(5)$ und $f(7) = 1$.

Bei $k = 8$ ist `ab` das längste Präfix von `brakadab`, also ist $f(8) = 2$. Das nächste Zeichen stimmt ebenfalls überein: `abr` ist ein Suffix von `brakadabr`, also ist $f(9) = 3$.

Wie genau diese Tabelle $f(k)$ effizient berechnet wird, stellen wir hier für einen Augenblick zurück. Zunächst schauen wir uns an, wie der Knuth-Morris-Pratt-Algorithmus $f(k)$ *verwendet*.

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
```

Hier sehen wir den kompletten Algorithmus `findKMP()`. Er berechnet als erstes die *Failure*-Funktion $f(k)$ mittels des Algorithmus `computeFailKMP()`, auf den wir in Kürze zurückkommen werden.

`findKMP()` verwendet zwei laufende Indizes, den Index j in T und den Index k in P. Beide Indizes stehen zunächst am Anfang von T bzw. P, und werden dann innerhalb einer einzigen Schleife verwaltet. Diese Schleife wird spätestens beendet, wenn j am Ende von T angekommen ist.

Innerhalb der Schleife vergleichen wir zunächst die beiden aktuellen Zeichen $T[j]$ und $P[k]$ miteinander. Sind sie identisch, dann können wir Erfolg vermelden falls k bereits am Ende von P angelangt ist; andernfalls fahren wir mit dem nächsten Zeichenpaar in T und P fort.

Sind die beiden aktuellen Zeichen verschieden, dann unterscheiden wir, ob wir irgendwo in der Mitte des Musters P stehen, oder ob bereits gleich das erste Zeichen von P nicht gepasst hat. Im letzteren Fall bleibt uns nichts anderes übrig, als im Text um ein Zeichen vorzurücken und im Muster wieder beim ersten Zeichen zu beginnen. Dies ist der 3. Zweig der `if-else if-else`-Anweisung.

Der interessante Fall tritt ein, wenn das ungleiche Paar irgendwo in der Mitte von P auftrat. In diesem Fall ist $k > 0$, und alle vorhergehenden Vergleiche für kleinere Werte von k waren erfolgreich. Dies nutzen wir nun aus, indem wir das Muster so weit wie möglich vorrücken, bis es ggf. wieder passen kann. Wir suchen also die Länge seines längsten Präfix, für das alle unmittelbar vorhergehenden Vergleiche erfolgreich waren.

Diesen Wert finden wir definitionsgemäß in $f(k-1)$. Auf diesen Wert setzen wir also unseren Index k und setzen unsere zeichenweise Vergleiche fort.

`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

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

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

Sehen wir nun `findKMP()` in Aktion. Zu Beginn sind alle Vergleiche erfolgreich, und beide Indizes j und k werden gemeinsam hochgezählt, bis zum ungleichen Paar `a` und `e` am Index $k = 10$. Das letzte gleiche Paar befand sich also am Index 9. Wir setzen also nun k auf $f(9) = 3$, und lassen j unberührt. Damit wird das Präfix `abr` der Länge 3 übersprungen und als nächstes $T[10]$ mit $P[3]$ verglichen, welche in der Animation durch entsprechende Einrückung des Musters P untereinander ausgerichtet werden.

Dieser Vergleich ergibt Übereinstimmung, und beide Indizes werden inkrementiert.

Der nächste Vergleich schlägt allerdings fehl. Hier haben wir nun bereits $k = 4$ erfolgreiche Vergleiche, einen expliziten, grün markiert, und drei implizite, blau markiert. Damit konsultieren wir wiederum $f(k-1)$, um ggf. ein übereinstimmendes Präfix zu überspringen. Dies können wir in der Tat tun; unser längstes Präfix hat immerhin die Länge $f(3) = 1$. Besser als nichts.

Nun schlägt bereits der erste explizite Vergleich fehl; das `b` des Musters stimmt nicht mit dem Leerzeichen des Texts überein. Wir haben jedoch bereits den einen impliziten erfolgreichen Vergleich des `a` und konsultieren deshalb $f(0)$, was jedoch zwangsläufig 0 ergibt.

Der nächste interssante Moment ereignet sich nach dem übereinstimmenden Musterpräfix `ab` beim ungleichen Paar `e` und `r`. Hier liefert uns $f(1) = 0$; wir können also an dieser Stelle kein übereinstimmendes Präfix überspringen.

Von nun an fällt der Vergleich bereits des ersten Zeichens des Musters jeweils negativ aus, bis am Ende das Muster tatsächlich im Text gefunden wird.

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.

Wenden wir uns nun, wie versprochen, der Berechnung der *Failure*-Funktion zu. Sie erfordert, dass wir Präfixe von P weiter hinten in P suchen, wo sie als Suffixe von $P[1, ..., k]$ enden. Für diese Suche können wir genialerweise – den KMP-Algorithmus nutzen!

Natürlich rufen wir nicht buchstäblich den KMP-Algorithmus mit dem Muster P auf sich selbst auf; das wäre sinnlos. Aber wir können die Idee des KMP-Algorithmus nutzen, um Präfixe von P in P zu suchen. KMP benötigt die *Failure*-Funktion, und diese wollen wir ja gerade erst berechnen! Dies hindert uns jedoch nicht, denn immer, wenn wir bei der Suche die *Failure*-Funktion konsultieren, ist der Teil, den wir benötigen, bereits bekannt.

Genau wie im `findKMP()`-Algorithmus verwalten wir also wieder zwei Indizes j und k. j geht durch den Text, der hier identisch ist mit P, und k geht durch das Muster P und zählt die Anzahl der erfolgreich verglichenen Zeichen, genau wie im `findKMP()`-Algorithmus.

Statt mit $T[j]$ vergleichen wir in der Schleife $P[j]$ mit $P[k]$. Bei Übereinstimmung wird die Länge $f[j]$ des aktuellen Suffix, das bei $P[j]$ endet, auf die Anzahl $k+1$ der bisher übereinstimmenden Zeichen gesetzt. Statt wie bei `findKMP()` zu prüfen, ob das Muster komplett übereinstimmt, setzen wir hier die Anzahl der übereinstimmenden Zeichen des aktuellen Präfix.

Anschließend inkrementieren wir j und k.

Dies ist auch schon der einzige Unterschied zu `findKMP()`. Wie bei `findKMP()` konsultieren wir auch hier $f(k-1)$. Dieser Wert ist zu diesem Zeitpunkt bereits fertig berechnet, denn j ist immer größer als k, und wir weisen nur an $f[j]$ zu und lesen nur $f[k-1]$ aus.

j muss immer größer sein als k, denn j beginnt größer als k, anschließend werden immer entweder sowohl j als auch k oder nur j inkrementiert, und die einzige weitere Zuweisung an k ist der Wert $f[k-1]$. Auch dieser muss jedoch kleiner sein als j, denn bei der Zuweisung von $k+1$ an $f[j]$ kann $k+1$ maximal gleich j sein.

Die Funktion `findKMP()` ruft also `computeFailKMP()` auf, um die *Failure*-Funktion zu

berechnen, und `computeFailKMP()` ist ihrerseits eine einfache Variante von `findKMP()`, die diese *Failure*-Funktion selbst verwendet, während sie diese überhaupt erst berechnet. Genial!

`computeFailKMP()`: Beispiel [Slide 12]

Sehen wir nun `computeFailKMP()` in Aktion. Der orange Ring zeigt den Text-Index j, und der dunkelblaue Ring den Muster-Index k.

Zu Beginn ist $k = 0$, und j wird so lange inkrementiert, bis $P[j] = P[k]$ ist. Dies ist der Fall bei $j = 3$, mit $P[3] = P[0] =$`a`.

Nun haben wir also eine Übereinstimmung des Präfix der Länge 1 von P mit dem Suffix der Länge 1 von $P[1, ..., 3]$. Mit anderen Worten, $f[3] = k + 1 = 1$, was wir am Index $j = 3$ in der Tabelle vermerken. Nun inkrementieren wir j auf 4 und k auf 1, und fahren fort.

Bei der nächsten Iteration stimmen $P[j]$ und $P[k]$ nicht überein, aber dank der vorherigen Übereinstimmung ist $k > 0$. Daher setzen wir k nun auf $f[k-1] = f[0] = 0$.

Bei der nächsten Iteration stimmen $P[k]$ und $P[j]$ (mit $j = 4$) nicht überein. Da $k = 0$ ist, wird nun j auf 5 inkrementiert.

Nun haben wir wieder eine Übereinstimmung von $P[j]$ mit $P[k]$, und setzen $f[5]$ entsprechend auf 1.

Die nächste Übereinstimmung erfolgt bei $j = 7$ und $k = 0$. Wir setzen entsprechend $f[7] = 1$ und inkrementieren j und k. Hier haben wir wiederum eine Übereinstimmung zwischen $P[j]$ und $P[k]$, die beide ein `b` sind. Nun ist $k = 1$, also setzen wir $f[8] = k+1 = 2$.

Genauso setzen wir bei der nächsten Iteration $f[9] = 3$.

Bei der folgenden Iteration haben wir keine Übereinstimmung mehr, aber $k = 3$ ist größer als 0. Also setzen wir k auf $f[2] = 0$.

Nun folgt die letzte Iteration, in der der Vergleich zwischen $P[j]$ und $P[k]$ negativ ausfällt und $k = 0$ ist. Daher wird lediglich j inkrementiert. Nun ist $j = m$, und die Schleife endet.

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:

Die *Eleganz* des KMP-Algorithmus ist hiermit nachgewiesen. Uns interessiert aber natürlich auch sein asymptotisches Laufzeitverhalten. Ist es wirklich besser als der *Brute-Force*-Algorithmus?

Die entscheidende Beobachtung ist, dass es bei jeder Iteration der Schleife entweder im Text oder im Muster vorwärts geht. Im Text geht es niemals zurück, im Gegensatz zum *Brute-Force*-Algorithmus.

Um dies zu zeigen, definieren wir uns eine Hilfvariable s, die die aktuelle Verschiebung des Musters im Verhältnis zum Text angibt. Diese Verschiebung beträgt im Algorithmus genau $j - k$ Zeichen und ist genau der Wert, den `findKMP()` als Fundstelle des Musters im Text zurückliefert.

Nun analysieren wir, wie sich s in jedem der drei Zweige der `if-else if-else`-Anweisung innerhalb der Schleife entwickelt:

Im ersten Zweig wachsen j und k jeweils um 1; s bleibt also unverändert.

Im zweiten Zweig bleibt j unverändert, und s wächst von $j - k$ auf $j - f(k-1)$, also um $k - f(k-1)$. Dieser Wert ist ≥ 1, da k immer größer ist als $f(k-1)$, wie wir bereits gesehen haben.

Im dritten Zweig bleibt k unverändert und j wächst um 1, also wächst auch s um 1.

In jeder Iteration wird also entweder das Muster weitergeschoben, indem s erhöht wird, oder es geht im Text vorwärts, indem j inkrementiert wird. Da beides jeweils nicht häufiger passieren kann als der Text lang ist, ist die Gesamtzahl der Iterationen also maximal $2n$.

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

Da `findKMP()` zuerst `computeFailKMP()` aufruft und anschließend bis zu $2n$ Iterationen ausführt, wobei jede andere Anweisung konstante Laufzeit hat, ist die Gesamtlaufzeit von `findKMP()` also $O(m) + O(n)$. Dies lässt sich auf $O(n)$ begrenzen, indem man `findKMP()` vor dem Aufruf von `computeFailKMP()` abbricht, falls $m > n$ ist.

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.

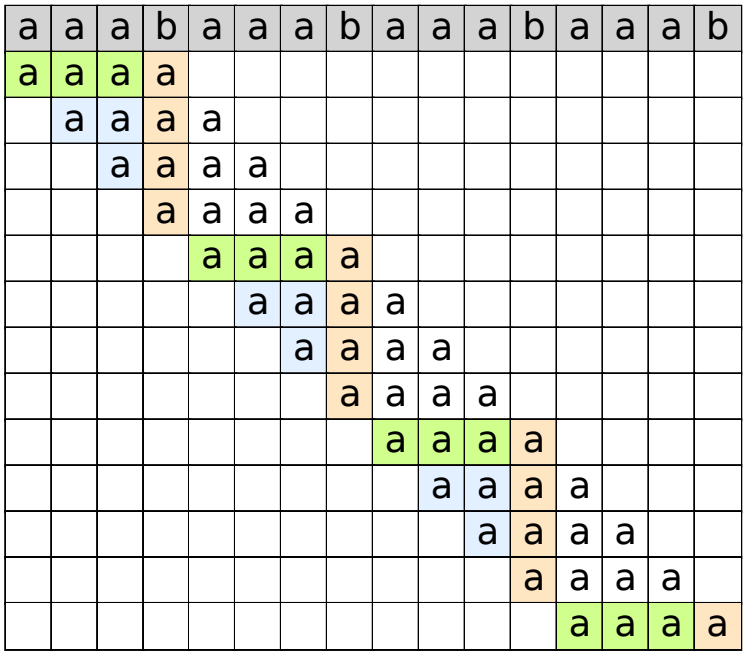
Jeder Suchalgorithmus muss schlimmstenfalls alle Zeichen des Texts und alle Zeichen des Musters mindestens einmal überprüfen.

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

Diese lineare Laufzeit ist deutlich besser als die des *Brute-Force*-Algorithmus, da der Faktor m verschwindet. Diese Laufzeit ist auch asymptotisch optimal, da schlimmstenfalls jedes einzelne Zeichen des Texts verglichen werden muss.

Die Korrektheit des KMP-Algorithmus ergibt sich aus der Definition der *Failure*-Funktion, die so konstruiert ist, dass vor dem gegebenen Zeichen im Muster keine Übereinstimmungen mit dem Text möglich sind.

KMP: Es geht immer vorwärts. [Slide 15]



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) $O(n)$.

> Wie gesehen ergibt sich die lineare Laufzeit des KMP-Algorithmus aus der Tatsache, dass bei jeder Iteration seiner einzigen Schleife entweder das Muster vorgeschoben oder der Index im Text inkrementiert wird. Insbesondere springt der Textindex niemals zurück, im Gegensatz zum *Brute-Force*-Algorithmus. Dies sehen wir hier sehr schön an diesem Beispiel, das einen schlechtesten Fall für den *Brute-Force*-Algorithmus darstellt. Anstatt der grün ausgefüllten Dreiecke sehen wir, dass hier die Vergleiche lediglich eine Linie ziehen, mit horizontalen Abschnitten bei Übereinstimmungen und vertikalen Abschnitten bei Nicht-Übereinstimmungen. Da bei Nicht-Übereinstimmungen das Muster vorwärtsgeschoben wird, ist die Summe der vertikalen Abschnitte auf $n - m$ begrenzt. Damit ist die maximale Gesamtzahl der Vergleiche die Summe der horizontalen plus der vertikalen Abschnitte, also $O(n)$.

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.

> Dies war die wunderbare Erzählung vom Musterabgleich, uns allen alltäglich bekannt durch die Suchfunktion in Web-Browsern und Textverarbeitungen. Diese simple Aufgabe birgt erstaunliches Potenzial für algorithmische Kreativität. Unter den vielen Suchalgorithmen ist der Knuth-Morris-Pratt-Algorithmus ein besonders elegantes Prachtstück, und erzielt die optimale asymptotische Laufzeit von $\Theta(n)$ im schlechtesten Fall.

Bibliographie [Slide 17]

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