Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 20 (13.7.2016)

String Matching (Textsuche)

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Textsuche / String Matching

UNI FREIBURG

Gegeben:

- Zwei Zeichenketten (Strings)
- Text T (typischerweise lang)
- Muster P (engl. pattern, typischerweise kurz)

Ziel:

- Finde alle Vorkommen von P in T

Notation:

- Länge Text T : $\boldsymbol{n}$, Länge Muster P : $\boldsymbol{m}$

Motivation

- Ist offensichtlich wichtig...
- Wird in jedem Texteditor gebraucht
  - jeder Editor hat eine find-Funktion
- Wird von Programmiersprachen unterstützt:
  - Java: String.indexOf(String pattern, int fromThisPosition)
  - C++: std::string.find(std::string str, size_t fromThisPosition)
  - Python: strong.find(pattern, from)

Naiver Algorithmus

- Gehe den Text von links nach rechts durch
- Das Muster kann an jeder der Stellen $s = 0, \dots, n - m$ vorkommen

- Prüfe an jeder dieser Stellen ob das Muster passt
  - indem das Muster Buchstabe für Buchstabe mit dem Text an der Stelle verglichen wird
  - Werden wir gleich noch etwas genauer anschauen...

Naiver Algorithmus

TestPosition(s): // tests if $T[s, \dots, s+m-1] == P$

$t := 0$

while $t < m$ **and** $T[s+t] = P[t]$ **do**

$t := t + 1$

return $(t = m)$

Naiver Algorithmus

```
TestPosition(s):          // tests if T[s, ..., s + m − 1] == P
   t := 0
   while t < m and T[s + t] = P[t] do
        t := t + 1
   return (t = m)

String-Matching:
   for s := 0 to n − m do
        if TestPosition(s) then
             report found match at position s
```

Naiver Algorithmus Analyse

Laufzeit von `TestPosition(`s`)`:

$$\textit{Laufzeit} = \begin{cases} x, & \text{falls erster Mismatch an Position } x \text{ ist} \\ m, & \text{falls Muster gefunden} \end{cases}$$

Laufzeit des Algorithmus:

- **Best case:**

  T=KESDJETNARKRETEJ **P=**AB

- **Worst case:**

  T=AAAAA….AAAA **P=**A…A

Rabin-Karp Algorithmus

UNI FREIBURG

Grundidee

- Wir schieben wieder ein Fenster der Grösse m über den Text und schauen an jeder Stelle, ob das Muster passt

- Zur Einfachheit nehmen wir an, dass der Text nur aus den Ziffern $0, \dots, 9$ besteht
  - dann können wir das Muster und das Fenster als Zahl verstehen
- Wenn wir das Fenster eins nach rechts schieben, kann die neue Zahl einfach aus der alten berechnet werden

Rabin-Karp Algorithmus

Beobachtungen:

- In jedem Schritt müssen wir einfach zwei Zahlen vergleichen
- Falls die Zahlen gleich sind, kommt das Muster an der Stelle vor
- Wenn man das Fenster um eins weiter schiebt, lässt sich die neue Zahl in $O(1)$ Zeit berechnen

- Falls wir zwei Zahlen in $O(1)$ vergleichen können, dann hat der Algorithmus Laufzeit $O(n)$

- **Problem:** Die Zahlen können sehr gross sein ($\Theta(m)$ bits)
  - Zwei $\Theta(m)$-bit Zahlen vergleichen benötigt Laufzeit $\Theta(m)$
  - Nicht besser als mit dem naiven Algorithmus

Rabin-Karp Algorithmus

- **Idee:** Benutze Hashing und vergleiche Hashwerte
  - Wenn man das Fenster eins weiter schiebt, sollte sich der neue Hashwert wieder in $O(1)$ Zeit aus dem alten Hashwert berechnen lassen

Rabin-Karp Algorithmus

Lösung von Rabin und Karp:

- Wir rechnen alles mit den Zahlen modulo M
  - M sollte möglichst gross sein, allerdings klein genug, damit die Zahlen $0, \dots, M-1$ in einer Speicherzelle (z.B. 32 Bit) Platz haben
- Hashwerte von Muster und Textfenster sind dann beides Zahlen aus dem Bereich

$$\{0, \dots, M-1\}$$

- Beim Schieben des Fensters um eine Stelle, lässt sich der neue Hashwert wieder in $O(1)$ Zeit berechnen
  - Falls das nicht klar ist, siehe spätere Folie...
- Falls das Muster vorhanden ist, sind die zwei Zahlen gleich, falls nicht, können sie trotzdem gleich sein
  - Falls die Hashwerte gleich sind, dann überprüfen wir nochmals wie beim naiven Algorithmus Buchstabe für Buchstabe

Rabin-Karp Algorithmus: Beispiel

Text: 572830354826 **Muster: 283** **Modulus $M = 5$**

Rabin-Karp Algorithmus: Pseudo-Code

Text $T[0 \ldots n-1]$, Muster $P[0 \ldots m-1]$, Basis b, Modulus M

$h := b^{m-1} \bmod M$

$p := 0$; $t := 0$;

for $i := 0$ **to** $m-1$ **do**

  $p := (p \cdot b + P[i]) \bmod M$

  $t := (t \cdot b + T[i]) \bmod M$

$s := 0$;

while $s \leq n-m$ **do**

  if $p = t$ **then**

    `TestPosition(`s`)`

  $t := \big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$

Rabin-Karp Algorithmus: Laufzeit

Vorberechnung:

Im schlechtesten Fall:

- Der schlechteste Fall tritt ein, falls die Hashwerte in jedem Schritt übereinstimmen. Dann muss man in jedem Schritt Buchstabe für Buchstabe überprüfen, ob man das Muster wirklich gefunden hat.
  - Sollte bei guter Wahl von M nicht allzu oft geschehen...
  - ausser, wenn das Muster tatsächlich sehr oft ($\Theta(n)$ mal) vorkommt...

Im besten Fall:

- Im besten Fall sind die Hashwerte nur gleich, falls das Muster auch wirklich gefunden wird. Die Kosten sind dann $O(n + k \cdot m)$, falls das Muster im Text k Mal vorkommt.

Wahl der Parameter...

UNI FREIBURG

Zahlendarstellung und Wahl von M

- Wir hätten gerne, dass wenn $x \neq y$, dann ist $h(x) = h(y)$ "unwahrscheinlich" (für $h(x) := x \bmod M$)
- Nehmen wir an, dass die Buchstaben in Muster und Text als Ziffern zur Basis b dargestellt werden
  - in unserem Fall, haben wir $b = 10$
- Falls b und M einen gemeinsamen Teiler haben, ist $h(x) = h(y)$ trotz $x \neq y$ nicht so unwahrscheinlich

Wahl der Parameter...

UNI FREIBURG

Zahlendarstellung und Wahl von M

- Wir hätten gerne, dass wenn $x \neq y$, dann ist $h(x) = h(y)$ "unwahrscheinlich" (für $h(x) := x \bmod M$)
- Nehmen wir an, dass die Buchstaben in Muster und Text als Ziffern zur Basis b dargestellt werden
  - in unserem Fall, haben wir $b = 10$
- Falls b und M einen gemeinsamen Teiler haben, ist $h(x) = h(y)$ trotz $x \neq y$ nicht so unwahrscheinlich

Wir wählen deshalb

- Die Basis b als genug grosse Primzahl
  - bei ASCII-Zeichen muss $b > 256$ sein
- M kann dann beliebig gewählt werden, am besten als Zweierpotenz
  - Zwischenresultate sind $< M \cdot b$, das sollte also in 32 (64) Bit Platz haben

Rechnen Modulo m

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot M \ \wedge\ y \in \{0, \dots, M-1\}$$

- $x \bmod M$: addiere/subtrahiere M von x bis die Zahl im Bereich $\{0, \dots, M-1\}$ ist

Rechenregeln:

$$(a \cdot b) \mod \mathrm{M} = \big((a \bmod M) \cdot (b \bmod M)\big) \bmod \mathrm{M}$$
$$(a + b) \bmod M = \big((a \bmod M) + (b \bmod M)\big) \bmod \mathrm{M}$$

Rechnen Modulo m

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot M \ \wedge\ y \in \{0, \dots, M-1\}$$

- $x \bmod M$: addiere/subtrahiere M von x bis die Zahl im Bereich $\{0, \dots, M-1\}$ ist

Rechenregeln:

$$(a \cdot b) \bmod M = \big((a \bmod M) \cdot (b \bmod M)\big) \bmod M$$
$$(a + b) \bmod M = \big((a \bmod M) + (b \bmod M)\big) \bmod M$$

Schieben des Fensters:

- Fenster von Stelle s nach Stelle $s+1$ schieben

$$\text{t=}\big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$$

Rechnen Modulo m

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot M \ \wedge\ y \in \{0, \dots, M-1\}$$

Negative Zahlen

- Damit ist $x \bmod M$ immer im Bereich $\{0, \dots, M-1\}$

  Beispiele:

  $$24 \bmod 10 = 4, \qquad 4 \bmod 10 = 4, \qquad -4 \bmod 10 = 6$$

- **Aber:** In Java / C++ / Python (?) ist $-x \,\%\, m = -(x \,\%\, m)$

  Beispiele:

  $$24 \,\%\, 10 = 4, \qquad 4 \,\%\, 10 = 4, \qquad -4 \,\%\, 10 = -4$$

- **Workaround:** Falls das Resultat von $x \,\%\, M$ negativ ist, einfach M dazuaddieren, dann kommt man in den richtigen Bereich

Algorithmus von Knuth, Morris, Pratt

- Kann wir das Problem immer in Zeit $O(n)$ lösen?
  - im schlechtesten Fall...

Schauen wir uns nochmals ein Beispiel an:

- Pattern: dubadubi
  Text: dubadubadudadubidubadubidubiduda

Knuth-Morris-Pratt Algorithmus

UNI FREIBURG

Idee:

- Falls wir beim Testen des Musters P an Stelle t feststellen, dass $P[t]$ nicht mit dem Text an der entsprechenden Stelle übereinstimmt, dann wissen wir, dass die Stellen $P[0 \ldots t-1]$ übereingestimmt haben.
- Das können wir bei der weiteren Suche ausnutzen

Beispiel: $P = \mathrm{ABDABLABDABD}$

Knuth-Morris-Pratt Alg.: Initialisierung

- Wir merken uns an jeder Stelle des Musters, wie weit wir das Suchfenster bei einem “Mismatch” weiterschieben können.

  (dabei ignorieren wir den “Typ” des Mismatches)

- Aequivalent dazu: Stelle im Muster an der wir weiter suchen muessen

$$P = \mathrm{A\ B\ D\ A\ B\ L\ A\ B\ D\ A\ B\ D}$$

Knuth-Morris-Pratt Algorithmus

Vorberechnung: Array S der Länge $m+1$

- $S[i]$: Stelle in P, an welcher man die neue Suche beginnt, falls beim Testen der Stelle i im Pattern ein Mismatch auftritt
- $S[0] = -1, \quad S[1] = 0$
- $S[m]$: Stelle in P, an welcher man weitersucht, nachdem P erfolgreich gefunden wurde

Beispiel:

$P =$ `[ A, B, D, A, B, L, A, B, D, A, B, D]`

$S =$ `[-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]`

Knuth-Morris-Pratt Algorithmus

UNI FREIBURG

$t \coloneqq 0;\ \ p \coloneqq 0$ *// t: Position in Text, p: Position im Pattern*

while $t < n$ **do**

  if $T[t] = P[p]$ **then** *// characters match*

    if $p = m - 1$ **then** *// pattern found*

      `pattern found at position` $t - m + 1$

      $p \coloneqq S[m];\ \ t \coloneqq t + 1$

    else

      $p \coloneqq p + 1;\ \ t \coloneqq t + 1$

  else *// characters don't match*

    if $p = 0$ **then** *// mismatch at first character*

      $t \coloneqq t + 1$

    else

      $p \coloneqq S[p]$

Knuth-Morris-Pratt Alg.: Beispiel

Pattern: ABCABC $S = [-1,0,0,0,1,2,3]$

Text:

A	D	A	B	C	D	A	B	C	A	A	A	B	C	A	B	C	A	B	C					

Knuth-Morris-Pratt Alg.: Laufzeit

UNI FREIBURG

Laufzeit ohne Initialisierung des Arrays S:

$t \coloneqq 0;\ \ p \coloneqq 0$

while $t < n$ **do**

    if $T[t] = P[p]$ **then**

        if $p = m - 1$ **then**

            pattern found

            $p \coloneqq S[m];\ \ t \coloneqq t + 1$

        else

            $p \coloneqq p + 1;\ \ t \coloneqq t + 1$

    else

        if $p = 0$ **then**

            $t \coloneqq t + 1$

        else

            $p \coloneqq S[p]$

Initialisierung

Vorberechnung von Array S:

- $P =$ [A, B, D, A, B, L, A, B, D, A, B, D]

  $S =$ [-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]

- An Position in $S[i]$ (für $i \in \{2, \dots, m\}$) steht

$$\boldsymbol{S[i] := \underset{k<i}{argmax}\{P[i-k \dots i-1] = P[0 \dots k-1]\}}$$

- $S[i]$: Länge des längsten **echten** Teilstückes von $P[0 \dots i-1]$, welches an Stelle $i-1$ endet, und welches auch Anfangsstück von P ist

Initialisierung

Vorberechnung von Array S:

- $S[i]$: Länge des längsten *passenden* Teilstückes von $P[1 \ldots i-1]$,

- $P =$ `[ A, B, D, A, B, L, A, B, D, A, B, D]`

  $S =$ `[-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]`

Berechnung von $S[i]$: (von links nach rechts)

- Falls $P\big[S[i-1]\big] = P[i-1]$, dann ist $S[i] = S[i-1] + 1$
- Sonst testen, ob es einen kürzeres, passendes Anfangsstück gibt
  - Wir werden gleich anschauen, wie man das macht...

Berechnung von $S[i]$: Beispiel

$h \coloneqq S[i-1]$

`while` $h \geq 0$ `do`

    `if` $P[i-1] = P[h]$ `then`

        $S[i] \coloneqq h+1$; $h \coloneqq -1$

    `else`

        $h \coloneqq S[h]$

`if` $h = -1$ `then` $S[i] = 0$

Beispiel: $P = [\texttt{A},\texttt{B},\texttt{D},\texttt{A},\texttt{B},\texttt{L},\texttt{A},\texttt{B},\texttt{D},\texttt{A},\texttt{B},\texttt{D},\texttt{X}]$