Suche in Zeichenketten

Wolfgang Mulzer

Seien $s = s_1 s_2 \dots s_k$ und $t = t_1 t_2 \dots t_\ell$ zwei Zeichenketten mit $\ell \leq k$. Wir wollen entscheiden, ob t in s enthalten ist, und falls ja, an welcher Stelle t zum ersten Mal vorkommt. Der naive Algorithmus ist wie folgt:

```
for i := 1 to k - l + 1
  // Does s contain t at position i?
  j <- 1
  while j <= k and s[i+j-1] = t[j] do
    j++
  // If yes, return postion i
  if j = k+1 then
    return i
// Not found, return -1
return -1
```

Die Laufzeit ist $O(k\ell)$. Das ist nur akzeptabel, wenn ℓ klein ist. Rabin und Karp haben folgenden Vorschlag gemacht, um die Laufzeit zu verbessern: Der Flaschenhals besteht darin, dass wir innerhalb der `for`-Schleife jedesmal den kompletten Substring $s_i \dots s_{i+\ell-1}$ mit t vergleichen. Wir könnten die Schleife beschleunigen, wenn wir vorher einen schnellen Test hätten, ob es wahrscheinlich ist, dass $s_i \dots s_{i+\ell-1} = t$ ist. Hier hilft eine *Hashfunktion*. Eine solche Hashfunktion weist $s_i \dots s_{i+\ell-1}$ und t jeweils eine Zahl zu. Diese Zahlen lassen sich schnell vergleichen. Außerdem hat eine gute Hashfunktion wenig Kollisionen, so dass es selten vorkommt, dass unterschiedliche Strings $s_i \dots s_{i+\ell-1}$ und t denselben Hashwert erhalten. Dies führt zu folgendem Ansatz (Rabin-Karp-Algorithmus). Sei h eine Hashfunktion.

```
for i := 1 to k - l + 1
  if h(s[i...i+l-1]) = h(t) then
    if s[i...i+l-1] = t then
      return i
return -1
```

Wenn Kollisionen selten sind, sollte der Aufwand für den Vergleich $s_i \dots s_{i+\ell-1} \stackrel{?}{=} t$ vernachlässigbar sein. Aber es gibt ein neues Problem: Wie kann man h schnell berechnen? (Ansonsten wäre die ganze Idee natürlich witzlos.) Zur Erinnerung: Vor ein paar Monaten hatten wir eine Hashfunktion h' für einen String $a = a_0 a_1 \dots a_{\ell-1}$ folgendermaßen definiert. Interpretiere die Zeichen a_i als Zahlen zwischen 0 und $S-1$ ($S = |\Sigma|$, die Alphabetgröße) und schreibe

$$h'(a) = \left(\sum_{j=0}^{\ell-1} a_j S^j \right) \bmod p$$

Für Rabin-Karp ist es besser, die Hashfunktion etwas anders zu definieren:

$$h(s_i \dots s_{i+\ell-1}) = \left(\sum_{j=0}^{\ell-1} s_{i+j} S^{\ell-1-j} \right) \bmod p$$

h unterscheidet sich von h' in zwei Punkten: (a) wir addieren zum Index im String immer i (weil es sich um einen Teilstring handelt) und (b) die Potenzen von S sind fallend statt steigend (das macht die Formel unten einfacher). Der entscheidende Punkt ist nun, dass

$$h(s_{i+1} \dots s_{i+\ell}) = \left(S \cdot h(s_i ... s_{i+\ell-1}) - S^l \cdot s_i + s_{i+\ell}\right) \bmod p$$

ist. Das heißt: *kennen wir $h(s_i \dots s_{i+\ell-1})$, so können wir $h(s_{i+1} \dots s_{i+\ell})$ mit $O(1)$ Operationen berechnen.* (Beachte: Wir müssen S^l nur einmal ausrechnen und speichern). Es folgt: wir können $h(s_1 \dots s_\ell)$, $h(s_2 \dots s_{\ell+1})$, …, $h(s_{k-\ell} \dots s_\ell)$ sowie $h(t)$ in Gesamtzeit $O(k+\ell)$ berechnen. Heuristisch gesehen sollte der Algorithmus von Rabin-Karp nun $O(k+\ell)$ Zeit benötigen, da wir hoffen, dass Kollisionen selten sind. (Genauer ist die Idee der Laufzeitanalyse, $p = \Theta(\ell)$ zu wählen. Dann sollte die Wahrscheinlichkeit einer Kollision etwa $1/\ell$ betragen, so dass man nur in jedem ℓ. Schleifendurchlauf den Stringvergleich durchführen muss. Dies gibt konstante amortisierte Zeit pro Schleifendurchlauf.)

Bemerkungen:

1. Bei der Implementierung sollte man bei der Berechnung von h nach jeder Multiplikation das Ergebnis mod$p$ nehmen und sicher stellen, dass p^2 nicht zu gross ist, um Überläufe zu vermeiden.
2. Wenn p zufällig gewählt ist, benötigt der Algorithmus von Rabin-Karp $O(k+\ell)$ Zeit im Erwartungswert.
3. Es gibt Algorithmen zur Suche in Zeichenketten mit $O(k+\ell)$ worst-case-Zeit (Knuth-Morris-Pratt, Boyer-Moore, Suffix-Bäume). Diese sind aber zum Teil bedeutend komplizierter.
4. Die Idee, Strings durch Hashfunktionen darzustellen/zu approximieren hat viele weitere Anwendungen (sichere Speicherung von Passwörtern, digitale Unterschriften, Verifikation von Downloads, …)