Vorlesungsskript

Theoretische Informatik III

Sommersemester 2010

Prof. Dr. Johannes Köbler
Humboldt-Universität zu Berlin
Lehrstuhl Komplexität und Kryptografie

14. April 2010

Inhaltsverzeichnis

1 Einleitung **1**

2 Suchen und Sortieren **2**

2.1 Suchen von Mustern in Texten 2

2.1.1 String-Matching mit endlichen Automaten . . 3

1 Einleitung

In den Vorlesungen ThI 1 und ThI 2 standen folgende Themen im Vordergrund:

- Mathematische Grundlagen der Informatik, Beweise führen, Modellierung Aussagenlogik, Prädikatenlogik
- Welche Probleme sind lösbar? (Berechenbarkeitstheorie)
- Welche Rechenmodelle sind adäquat? (Automatentheorie)
- Welcher Aufwand ist nötig? (Komplexitätstheorie)

Dagegen geht es in der VL ThI 3 in erster Linie um folgende Frage:

- Wie lassen sich eine Reihe von praktisch relevanten Problemstellungen möglichst effizient lösen?
- Wie lässt sich die Korrektheit von Algorithmen beweisen und wie lässt sich ihre Laufzeit abschätzen?

Die Untersuchung dieser Fragen lässt sich unter dem Themengebiet Algorithmik zusammenfassen.

Der Begriff *Algorithmus* geht auf den persischen Gelehrten Muhammed Al Chwarizmi (8./9. Jhd.) zurück. Der älteste bekannte nicht-triviale Algorithmus ist der nach *Euklid* benannte Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen (300 v. Chr.). Von einem Algorithmus wird erwartet, dass er jede *Problemeingabe* nach endlich vielen Rechenschritten löst (etwa durch Produktion einer Ausgabe). Ein Algorithmus ist ein „Verfahren" zur Lösung eines Entscheidungs- oder Berechnungsproblems, das sich prinzipiell auf einer Turingmaschine implementieren lässt (Church-Turing-These).

Die Registermaschine

Bei Aussagen zur Laufzeit von Algorithmen beziehen wir uns auf die Registermaschine (engl. random access machine; RAM). Dieses Modell ist etwas flexibler als die Turingmaschine, da es den unmittelbaren Lese- und Schreibzugriff (random access) auf eine beliebige Speichereinheit (Register) erlaubt. Als Speicher stehen beliebig viele Register zur Verfügung, die jeweils eine beliebig große natürliche Zahl speichern können. Auf den Registerinhalten sind folgende arithmetische Operationen in einem Rechenschritt ausführbar: Addition, Subtraktion, abgerundetes Halbieren und Verdoppeln. Unabhängig davon geben wir die Algorithmen in Pseudocode an. Das RAM-Modell benutzen wir nur zur Komplexitätsabschätzung.

Die Laufzeit von RAM-Programmen wird wie bei TMs in der Länge der Eingabe gemessen. Man beachte, dass bei arithmetischen Problemen (wie etwa Multiplikation, Division, Primzahltests, etc.) die Länge einer Zahleingabe n durch die Anzahl $\lceil \log n \rceil$ der für die Binärkodierung von n benötigten Bits gemessen wird. Dagegen bestimmt bei nicht-arithmetischen Problemen (z.B. Graphalgorithmen oder Sortierproblemen) die Anzahl der gegebenen Zahlen die Länge der Eingabe.

Asymptotische Laufzeit und Landau-Notation

Definition 1. *Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{R}^+$. Wir schreiben $f(n) = \mathcal{O}(g(n))$, falls es Zahlen n_0 und c gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Die Bedeutung der Aussage $f(n) = \mathcal{O}(g(n))$ ist, dass f „nicht wesentlich schneller" als g wächst. Formal bezeichnet der Term $\mathcal{O}(g(n))$ die Klasse aller Funktionen f, die obige Bedingung erfüllen. Die Gleichung $f(n) = \mathcal{O}(g(n))$ drückt also in Wahrheit eine Element-Beziehung $f \in \mathcal{O}(g(n))$ aus. O-Terme können auch auf

der linken Seite vorkommen. In diesem Fall wird eine Inklusionsbeziehung ausgedrückt. So steht $n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$ für die Aussage $\{n^2 + f \mid f \in \mathcal{O}(n)\} \subseteq \mathcal{O}(n^2)$.

Beispiel 2.

- $7\log(n) + n^3 = \mathcal{O}(n^3)$ *ist richtig.*
- $7\log(n)n^3 = \mathcal{O}(n^3)$ *ist falsch.*
- $2^{n+\mathcal{O}(1)} = \mathcal{O}(2^n)$ *ist richtig.*
- $2^{\mathcal{O}(n)} = \mathcal{O}(2^n)$ *ist falsch (siehe Übungen).* ◁

Es gibt noch eine Reihe weiterer nützlicher Größenvergleiche von Funktionen.

Definition 3. *Wir schreiben $f(n) = o(g(n))$, falls es für jedes $c > 0$ eine Zahl n_0 gibt mit*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Damit wird ausgedrückt, dass f „wesentlich langsamer" als g wächst. Außerdem schreiben wir

- *$f(n) = \Omega(g(n))$ für $g(n) = \mathcal{O}(f(n))$, d.h. f wächst mindestens so schnell wie g)*
- *$f(n) = \omega(g(n))$ für $g(n) = o(f(n))$, d.h. f wächst wesentlich schneller als g, und*
- *$f(n) = \Theta(g(n))$ für $f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$, d.h. f und g wachsen ungefähr gleich schnell.*

2 Suchen und Sortieren

2.1 Suchen von Mustern in Texten

In diesem Abschnitt betrachten wir folgende algorithmische Problemstellung.

String-Matching (StringMatching):

Gegeben: Ein Text $x = x_1 \cdots x_n$ und ein Muster $y = y_1 \cdots y_m$ über einem Alphabet Σ.

Gesucht: Alle Vorkommen von y in x.

Wir sagen y *kommt in x an Stelle i vor*, falls $x_{i+1} \cdots x_{i+m} = y$ ist. Typische Anwendungen finden sich in Textverarbeitungssystemen (emacs, grep, etc.), sowie bei der DNS- bzw. DNA-Sequenzanalyse.

Beispiel 4. *Sei $\Sigma = \{A, C, G, U\}$.*

$$\begin{aligned} \textit{Text } x &= AUGACGAUGAUGUAGGUAGCGUAGAUGAUGUAG, \\ \textit{Muster } y &= AUGAUGUAG. \end{aligned}$$

Das Muster y kommt im Text x an den Stellen 6 und 24 vor. ◁

Bei naiver Herangehensweise kommt man sofort auf folgenden Algorithmus.

Algorithmus `naive-String-Matcher`(x, y)

```
1 Input: Text x = x_1 ··· x_n und Muster y = y_1 ··· y_m
2   V := ∅
3   for i := 0 to n − m do
```

4 **if** $x_{i+1} \cdots x_{i+m} = y_1 \cdots y_m$ **then**
5 $V := V \cup \{i\}$
6 **Output:** V

Die Korrektheit von `naive-String-Matcher` ergibt sich wie folgt:

- In der **for**-Schleife testet der Algorithmus alle potentiellen Stellen, an denen y in x vorkommen kann, und
- fügt in Zeile 4 genau die Stellen i zu V hinzu, für die $x_{i+1} \cdots x_{i+m} = y$ ist.

Die Laufzeit von `naive-String-Matcher` lässt sich nun durch folgende Überlegungen abschätzen:

- Die **for**-Schleife wird $(n - m + 1)$-mal durchlaufen.
- Der Test in Zeile 4 benötigt maximal m Vergleiche.

Dies führt auf eine Laufzeit von $\mathcal{O}(nm) = \mathcal{O}(n^2)$. Für Eingaben der Form $x = a^n$ und $y = a^{\lfloor n/2 \rfloor}$ ist die Laufzeit tatsächlich $\Theta(n^2)$.

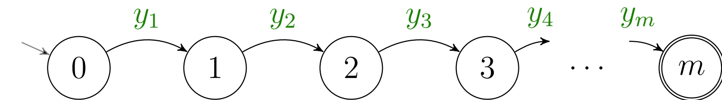
2.1.1 String-Matching mit endlichen Automaten

Durch die Verwendung eines endlichen Automaten lässt sich eine erhebliche Effizienzsteigerung erreichen. Hierzu konstruieren wir einen DFA M_y, der jedes Vorkommen von y in der Eingabe x durch Erreichen eines Endzustands anzeigt. M_y erkennt also die Sprache

$$L = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}.$$

Konkret konstruieren wir $M_y = (Z, \Sigma, \delta, 0, m)$ wie folgt:

- M_y hat $m + 1$ Zustände, die den $m + 1$ Präfixen $y_1 \cdots y_k$, $k = 0, \ldots, m$, von y entsprechen, d.h. $Z = \{0, \ldots, m\}$.
- Liest M_y im Zustand k das Zeichen y_{k+1}, so wechselt M_y in den Zustand $k + 1$, d.h. $\delta(k, y_{k+1}) = k + 1$ für $k = 0, \ldots, m - 1$:



- Falls das nächste Zeichen a nicht mit y_{k+1} übereinstimmt (engl. *mismatch*), wechselt M_y in den Zustand

$$\delta(k, a) = \max\{j \le m \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}.$$

Der DFA M_y speichert also in seinem Zustand die maximale Länge k eines Präfixes $y_1 \cdots y_k$ von y, das zugleich ein Suffix der gelesenen Eingabe ist:

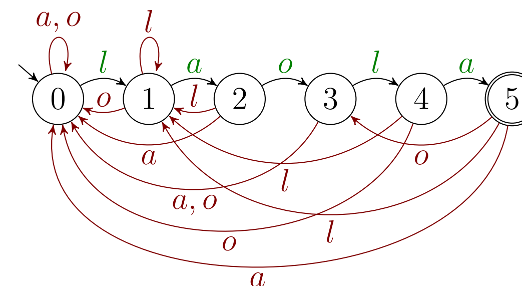
$$\hat{\delta}(0, x) = \max\{k \le m \mid y_1 \cdots y_k \text{ ist Suffix von } x\}.$$

Die Korrektheit von M_y folgt aus der Beobachtung, dass M_y isomorph zum *Äquivalenzklassenautomaten* M_{R_L} für L ist. M_{R_L} hat die Zustände $[y_1 \cdots y_k]$, $k = 0, \ldots, m$, von denen nur $[y_1 \cdots y_m]$ ein Endzustand ist. Die Überführungsfunktion ist definiert durch

$$\delta([y_1 \cdots y_k], a) = [y_1 \cdots y_j],$$

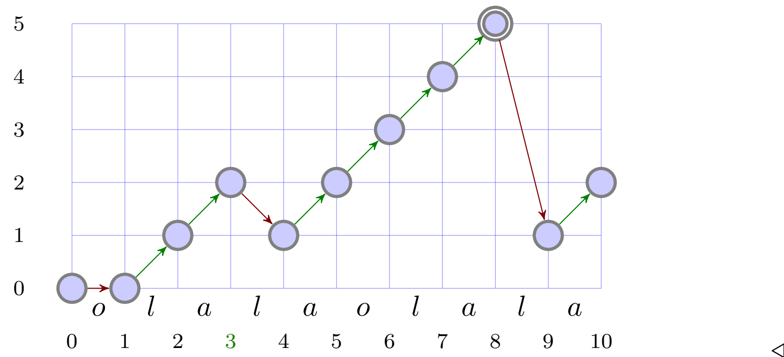
wobei $y_1 \cdots y_j$ das längste Präfix von $y = y_1 \cdots y_m$ ist, welches Suffix von $y_1 \cdots y_j a$ ist (siehe Übungen).

Beispiel 5. *Für das Muster $y = laola$ hat M_y folgende Gestalt:*



δ	0	1	2	3	4	5
a	0	2	0	0	5	0
l	1	1	1	4	1	1
o	0	0	3	0	0	3

M_y macht bei der Suche nach dem Muster $y = laola$ im Text $x = olalaolala$ folgende Übergänge:



◁

Insgesamt erhalten wir somit folgenden Algorithmus.

Algorithmus `DFA-String-Matcher`(x, y)

```
1 Input: Text x = x_1 ··· x_n und Muster y = y_1 ··· y_m
2   konstruiere den DFA M_y = (Z, Σ, δ, 0, m)
3   V := ∅
4   k := 0
5   for i := 1 to n do
6     k := δ(k, x_i)
7     if k = m then V := V ∪ {i − m}
8 Output: V
```

Die Korrektheit von `DFA-String-Matcher` ergibt sich unmittelbar aus der Tatsache, dass M_y die Sprache

$$L(M_y) = \{x \in \Sigma^* \mid y \text{ ist Suffix von } x\}$$

erkennt. Folglich fügt der Algorithmus genau die Stellen $j = i - m$ zu V hinzu, für die y ein Suffix von $x_1 \cdots x_i$ (also $x_{j+1} \cdots x_{j+m} = y$) ist.

Die Laufzeit von `DFA-String-Matcher` ist die Summe der Laufzeiten für die Konstruktion von M_y und für die Simulation von M_y bei Eingabe x, wobei letztere durch $\mathcal{O}(n)$ beschränkt ist. Für δ ist eine Tabelle mit $(m+1)\|\Sigma\|$ Einträgen

$$\delta(k, a) = \max\{j \leq k+1 \mid y_1 \cdots y_j \text{ ist Suffix von } y_1 \cdots y_k a\}$$

zu berechnen. Jeder Eintrag $\delta(k, a)$ ist in Zeit $\mathcal{O}(k^2) = \mathcal{O}(m^2)$ berechenbar. Dies führt auf eine Laufzeit von $\mathcal{O}(\|\Sigma\| m^3)$ für die Konstruktion von M_y und somit auf eine Gesamtlaufzeit von $\mathcal{O}(\|\Sigma\| m^3 + n)$. Tatsächlich lässt sich M_y sogar in Zeit $\mathcal{O}(\|\Sigma\| m)$ konstruieren.