ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Departement Informatik

Open Class – Sieben Wunder der Informatik

Prof. Dr. Juraj Hromkovič

Übungsaufgaben – Blatt 4

Zürich, 23. November 2006

Zusammenfassung

Wie messen wir die Komplexität eines Algorithmus?
Es sei A ein Algorithmus für ein Problem U. Wir sagen zunächst, was die Komplexität (als Menge der Arbeit) von A auf einer Eingabe X (für einen Problemfall X) ist. Die einfachste Art und Weise besteht darin, zu sagen, dass die Komplexität (genauer: die Zeitkomplexität) *die Anzahl der auszuführenden Rechnerbefehle* in der Berechnung von A auf X ist.
Zum Beispiel sind, wenn A den Wert des Polynoms

$$a \cdot x^2 + b \cdot x + c$$

für gegebene Zahlen

$$a = 3,\ b = 4,\ c = 5 \text{ und } x = 7$$

auf naive Weise berechnet, die auszuführenden Befehle

$b \cdot x$,	$x \cdot x$,	$a \cdot x^2$,	$bx + c$	und $ax^2 + (bx + c)$.
$4 \cdot 7 = 28$	$7 \cdot 7 = 49$	$3 \cdot 49 = 147$	$28 + 5 = 33$	$147 + 33 = 180$

Somit ist die Komplexität von A genau 5 für jedes quadratische Polynom. (Die Werte für a, b, c und x haben hier keinen Einfluss auf die Komplexität.) Die Vorgehensweise des Algorithmus kann man auch durch die folgende Darstellung des Polynoms veranschaulichen:

$$a \cdot x \cdot x \ + \ b \cdot x \ + \ c$$

Wir sehen direkt die drei Multiplikationen und zwei Additionen, die auszuführen sind. Mit Hilfe des bekannten Distributivgesetzes kann man das quadratische Polynom wie folgt umschreiben:

$$ax^2 + bx + c = (a \cdot x + b) \cdot x + c$$

Hier sind dann nur zwei Multiplikationen und zwei Additionen durchzuführen, also ist die Komplexität bei dem resultierenden verbesserten Algorithmus genau 4.

Aufgabe 11

Betrachten wir ein Polynom vierten Grades:

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Erklären Sie, wie man den Wert für gegebene a_4, a_3, a_2, a_1, a_0 und x mit nur vier Multiplikationen und vier Additionen berechnen kann. **10 Punkte**

Bonus-Aufgabe 4

Geben Sie einen Algorithmus an, der den Wert jedes Polynoms n-ten Grades

$$a_nx^n + a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x + a_0$$

für gegebene Werte von $a_0, \ldots, a_n$ und x mit höchstens n Multiplikationen und n Additionen ausrechnet. **10 Bonus-Punkte**

Wenn man die Komplexität des Algorithmus A auf einer Eingabe x definiert hat, kann man die Komplexität $\text{Zeit}_A(n)$ von A als eine Funktion definieren, die jeder Eingabegrösse n die maximale Anzahl der durchzuführenden Operationen über alle Problemfälle der Grösse höchstens n zuordnet.
Wie man die Eingabegrösse messen will, ist uns überlassen. Zum Beispiel kann man bei der Berechnung von Polynomwerten den *Grad* des Polynoms (also die maximale Anzahl erlaubter Koeffizienten minus eins) als Eingabegrösse heranziehen. Ein naiver Algorithmus A rechnet für die Eingabegrösse n nun wie folgt:

$$\underbrace{x \cdot x}_{\text{1. Multiplikation}} = x^2 \qquad \underbrace{x \cdot x^2}_{\text{2. Multiplikation}} = x^3 \qquad \underbrace{x \cdot x^3}_{\text{3. Multiplikation}} = x^4 \qquad \underbrace{x \cdot x^{n-1}}_{(n-1)\text{-te Multiplikation}} = x^n$$

$$\underbrace{a_1 \cdot x}_{n\text{-te Multiplikation}} = x^2 \qquad \underbrace{a_2 \cdot x^2}_{(n+1)\text{-te Multiplikation}} = x^3 \qquad \underbrace{a_3 \cdot x^3}_{(n+2)\text{-te Multiplikation}} = x^4 \qquad \underbrace{a_n \cdot x^{n-1}}_{(2n-2)\text{-te Multiplikation}} = x^n$$

und dann

$$a_0 \underset{\text{1. Add.}}{+} a_1x \underset{\text{2. Add.}}{+} a_2x^2 \underset{\text{3. Add.}}{+} \ldots \underset{n\text{-te Add.}}{+} a_nx^n$$

Somit ist $\text{Zeit}_A(n) = \underbrace{2n-2}_{\text{Multipl.}} + \underbrace{n}_{\text{Addit.}} = 3n - 2$.
Die Komplexitätsfunktion kann man graphisch darstellen (vgl. Abb. 1).
Was uns meistens interessiert, ist folgendes: Wenn man etwa ein Zeit-Limit von 10^{16} Operationen hat, wie gross sind die Eingabegrössen, für die unser Algorithmus noch einsetzbar wäre. Für $\text{Zeit}_A(n) = 3n - 2$ erhalten wir

$$\begin{aligned} 3n - 2 &\leq 10^{16} && \mid +2 \text{ auf beiden Seiten} \\ 3n &\leq 10^{16} + 2 && \mid \div 3 \text{ auf beiden Seiten} \\ n &\leq \frac{1}{3}(10^{16} + 2) \quad . \end{aligned}$$

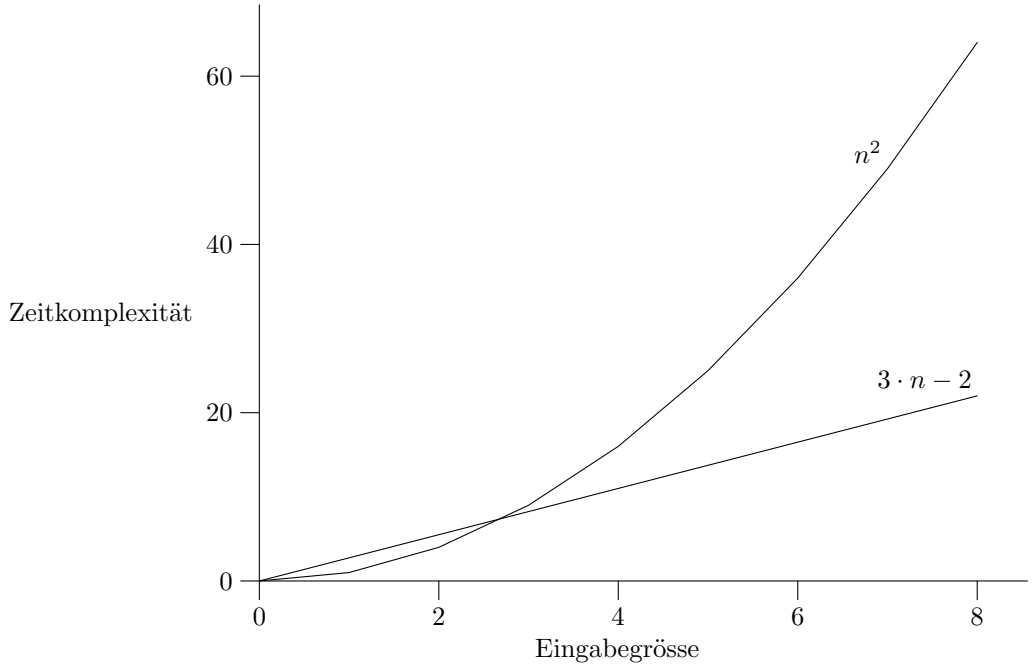


Abbildung 1: Zwei Komplexitätsfunktionen

Wir sehen, dass solch riesige Eingabegrössen wie $\frac{1}{3} \cdot 10^{16}$ sowieso nie vorkommen werden, also ist unser Algorithmus in der Praxis immer durchführbar. Für einen Algorithmus B mit $\text{Zeit}_B(n) = n^4$ erhalten wir

$$\begin{aligned} n^4 &\leq 10^{16} && | \sqrt[4]{\ } \text{ von beiden Seiten} \\ n &\leq (10^{16})^{\frac{1}{4}} \\ n &\leq 10^4 = 10\text{'}000 \quad . \end{aligned}$$

Also ist B für Eingabegrössen bis zu 10'000 praktikabel. Viel schlimmer steht es für $\text{Zeit}_C(n) = 10^n$.

$$\begin{aligned} 10^n &\leq 10^{16} && | \log_{10} \text{ von beiden Seiten} \\ n &\leq \log_{10}(10^{16}) = 16 \end{aligned}$$

Algorithmus C ist nur für sehr kleine Eingaben anwendbar, nämlich solchen bis zu einer Grösse von 16.

Aufgabe 12

Nehmen wir an, wir verfügen über einen Rechner, der 10^9 Operationen in der Sekunde durchführen kann. Die Anzahl der seit dem Urknall vergangenen Sekunden ist kleiner als 10^{18}. Wir sind bereit, während dieser Zahl von 10^{18} Sekunden zu warten. Welche Eingabegrössen können bei einem Algorithmus A bearbeitet werden, wenn

(a) $\text{Zeit}_A(n) = 10 \cdot n^2$

(b) $\text{Zeit}_A(n) = 50 \cdot n^3$

(c) $\text{Zeit}_A(n) = 2^n$

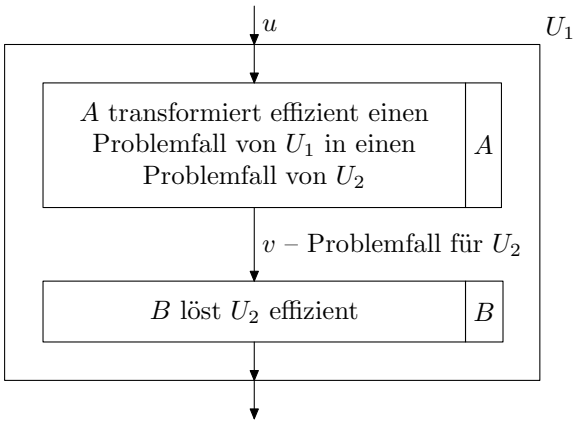
(d)* $\text{Zeit}_A(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$ (Bonus-Aufgabe)

10 Punkte + 10 Bonus-Punkte

Leider ist es so, dass es bezüglich der Zeitkomplexität, wie man mittels Diagonalisierung zeigen kann, beliebig schwere Probleme gibt. Zum Beispiel gibt es Aufgaben, die man in der Zeit 2^n lösen kann, aber nicht schneller. Gleichfalls gibt es Aufgaben, zu deren Lösung bei Eingabegrösse n wirklich $2^{2^{2^n}}$ Operationen benötigt werden.
Uns fehlen aber mathematische Methoden, um zu beweisen, dass konkrete Probleme in bezug auf ihre Zeitkomplexität schwer sind, d. h. um die Nichtexistenz effizienter Algorithmen für diese Probleme zu zeigen. Wir können über 4'000 praktisch relevante Aufgaben (in der Informatik sogenannte *NP-vollständige* Aufgaben) zitieren, für die nur Algorithmen mit exponentieller Komplexität gefunden wurden. Dennoch gelingt es uns nicht, zu beweisen, dass es keine besseren (d. h. schnelleren) Algorithmen für diese Aufgaben gibt. Trotzdem glauben wir dies, weil wir beweisen können, dass die Existenz eines einzigen effizienten (d. h. in polynomieller Zeit arbeitenden) Algorithmus für eine beliebige dieser 4'000 Aufgaben die Existenz effizienter Algorithmen für *alle* dieser Aufgaben bedeuten würde.
Diesen gleichen Schwierigkeitsgrad aller NP-vollständigen Aufgaben beweist man mittels effizienter (d. h. in polynomieller Zeit arbeitenden) Reduktionen. Es seien U_1 und U_2 zwei Probleme. Wir schreiben

$$U_1 \leq_{\text{eff}} U_2$$

und lesen: „U_1 ist auf U_2 effizient (d. h. mit polynomiellem Aufwand) reduzierbar“ oder „U_1 ist nicht schwerer als U_2 bezüglich der Existenz effizienter Algorithmen“, wenn ein effizienter Algorithmus A existiert, der jeden Problemfall u von U_1 so in einen Problemfall v von U_2 transformieren kann, dass die korrekten Lösungen für u und v gleich sind.



Beispiel: Das Aufsichtsproblem Aufs besteht darin, für ein gegebenes Strassennetz mit n Kreuzungen und m Strassen zwischen Kreuzungen und einer natürlichen Zahl k zu entscheiden, ob k Beobachter zusammen alle Strassen beobachten können. Ein Beobachter darf sich nur in einem Kreuzungspunkt befinden, und von dort überblickt er von allen in die Kreuzung mündenden Strassen den Teil bis zur jeweils nächsten Kreuzung.
Die Frage ist, ob k Beobachter für das gesamte Strassennetz ausreichen.

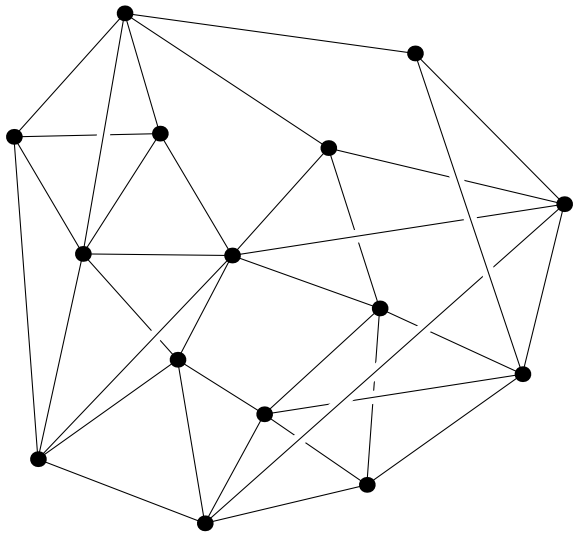


Abbildung 2: Ein Strassennetz

Dieses Problem ist NP-vollständig. Wir zeigen, dass man es auf das Entscheidungsproblem *LIN* reduzieren kann. *LIN* zu entscheiden, bedeutet, für ein gegebenes System linearer Ungleichungen mit Unbekannten x_1, x_2, ..., x_n festzustellen, ob eine Lösung existiert, in der jede Unbekannte x_1, x_2, ..., x_n nur einen der Werte 0 oder 1 annimmt.
Es sei G ein Strassennetz mit n Kreuzungen K_1, K_2, ..., K_n, und es sei k die Maximalzahl zugelassener Beobachter. Wir sollen die Probleminstanz (G, k) in ein lineares Gleichungssystem als eine Instanz von *LIN* umwandeln, und zwar auf solche Art und Weise, dass diese *LIN*-Instanz genau dann eine 0-1-Lösung hat, wenn k Beobachter ausreichen.

Es seien K_1, K_2, ..., K_n die Kreuzungen von G. Wir wählen n Variablen x_1, x_2, ..., x_n mit der Bedeutung

$x_i = 1$,	wenn ein Beobachter in der Kreuzung K_i steht;
$x_i = 0$	sonst.

Dann besagt die Ungleichung

$$x_1 + x_2 + x_3 + \ldots + x_n \leq k \quad ,$$

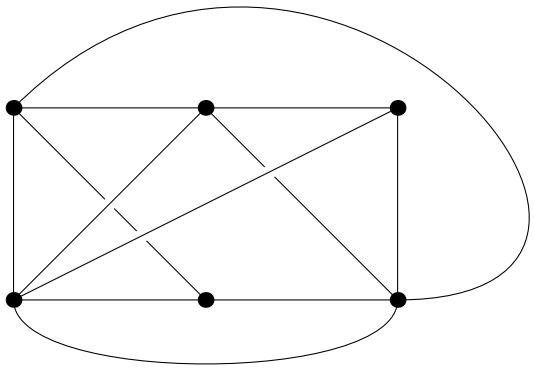
dass es höchstens k Beobachter geben darf. Für jede Strasse $\{K_i, K_j\}$ zwischen zwei (unterschiedlichen) Kreuzungen K_i und K_j nehmen wir die Ungleichung

$$x_i + x_j \geq 1$$

hinzu, die besagt, dass zur Beobachtung der Strasse $\{K_i, K_j\}$ ein Beobachter in K_i oder in K_j stehen muss.

Aufgabe 13

Gegeben sei folgendes Strassennetz mit 6 Kreuzungen und 12 Strassen



sowie als Maximalzahl von Beobachtern die Zahl 3. Formulieren Sie diesen Problemfall von Aufs in einen äquivalenten Problemfall von *LIN* um. **10 Punkte**

Ihre Lösungen zu den Aufgaben können Sie entweder persönlich bei der Open-Class-Veranstaltung am 30. November 2005 abgeben oder bis zum 30. November per E-Mail (möglichst als PDF-Datei) an `hjb@inf.ethz.ch` oder per Post an folgende Adresse schicken:

Dr. Hans-Joachim Böckenhauer
Informationstechnologie und Ausbildung
ETH Zentrum CAB F 11.1
Universitätsstrasse 6
8092 Zürich

Bitte vergessen Sie nicht, Ihre Lösung mit Ihrem Namen und Ihrer E-Mail-Adresse zu versehen.
Falls Ihre Lösung uns bis zum 28. November 2005 erreicht, können Sie die korrigierte Lösung bereits in der Veranstaltung am 30. November abholen (sonst eine Woche später).