Konzepte der Komplexitätstheorie illustriert am Beispiel des Sortierens

von

Kurt Mehlhorn
Fachbereich 1o
Universität des Saarlandes
D-6600 Saarbrücken

Die grundlegende Fragestellung der Komplexitätstheorie lautet: Wie effizient kann ich mein Lieblingsproblem auf meiner Lieblingsmaschine lösen?

Ein Problem P existiert meist in unendlich vielen Fragestellungen. Eine Fragestellung des Maximumproblems ist: Bestimme die größte der 5 Zahlen 17, 3, 7, 5, 22. Jeder Fragestellung $p \in P$ messen wir eine natürliche Zahl $g(p)$ als Größe zu. Diese Zuordnung ist willkürlich, ergibt sich aber meist in natürlicher Weise. Beim Maximumproblem könnte man etwa die Kardinalität der Menge nehmen, deren Maximum zu bestimmen ist.

Zur Lösung von Problemen dienen Algorithmen. Die Ausführung eines Algorithmus in einer Rechenanlage benötigt Ressourcen, z.B. Rechenzeit und Speicherplatz. Für einen Algorithmus A zur Lösung des Problems P sei $T_A(p)$ der Rechenzeitverbrauch an der Fragestellung p. [Bemerkung: Diese Notation setzt deterministische Algorithmen voraus. Aus theoretischen und praktischen Erwägungen werden wir später auch nichtdeterministische und probabilistische Algorithmen betrachten].

Interessanter als der Rechenzeitverbrauch für jede einzelne Eingabe ist eine globale Aussage über den Verbrauch an einer beliebigen Eingabe der Größe n. Zwei Abstraktionen bieten sich an: Verhalten im Mittel und Verhalten im schlechtesten Fall.

Das Verhalten im schlechtesten Fall $T_A(n)$ definieren wir als maximale Laufzeit an einer Eingabe der Größe n,

$$T_A(n) = \sup\{T_A(p);\ p \in P \text{ und } g(p) = n\}\ ,$$

die mittlere Laufzeit definieren wir als Erwartungswert der Laufzeit bei Eingaben der Größe n. Dabei wird eine Wahrscheinlichkeitsverteilung auf der Menge der Fragestellungen vorausgesetzt. Solche Verteilungen sind

$$T_A^{av}(n) = E\{T_A(p) \; ; \; p \in P \text{ und } g(p) = n\}$$

oft nur schwer in sinnvoller Weise zu definieren.

Auf welcher Maschine sollen wir nun den Rechenzeitverbrauch messen? Zum Glück hängt das Ergebnis nicht allzu sehr vom gewählten Maschinentyp ab. Die Laufzeit von Algorithmen auf Registermaschinen mit verschiedenen Arten der Adressenrechnung (Indexregister, allgemeine Adresssensubstitution, direkte Adressenrechnung) differiert zum Beispiel nur um einen konstanten Faktor [vgl. Mehlhorn]. Für theoretische Untersuchungen (vgl. die Diskussion von P und NP weiter unten) benutzt man gern die Turingmaschine. Hier ist der Zusammenhang zur Laufzeit auf einer Registermaschine durch ein Polynom gegeben und das genügt für die meisten theoretischen Untersuchungen [vgl. Mehlhorn oder Paul]. In unseren Beispielen werden wir eine Algolähnliche Sprache wählen und die Zahl der ausgeführten Algolbefehle zählen.

Beispiel 1: Bestimme das Maximum der Elemente des Feldes A[1:N].

```
        i ← 1, max ← A[1];
M:      i ← i+1;
        if i > N then goto Ende;
        if A[i] < max then goto M;
        max ← A[i];
        goto M;
Ende:
```

Offensichtlich ist die maximale Laufzeit an einer Eingabe der Größe n gleich 5n-1. Die mittlere Laufzeit ist dagegen viel schwieriger zu bestimmen, da die Befehle max ← A[i]; goto M nur ausgeführt werden, wenn ein neues Maximum gefunden wird; also ist die Anzahl der Ausführungen gleich der Anzahl der "Links-Rechts-Maxima" der Folge A[1:N]. Unter der Annahme, daß alle Permutationen der Eingabe gleichwahrscheinlich sind, ist diese Anzahl ln n + O(1) [vgl. Mehlhorn] und die mittlere Laufzeit daher 3n + 2ln n + O(1). Die Laufzeit im Mittel und im schlechtesten Fall ist also O(n). [f = O(g) für Funktionen f,g:N → N bedeutet $f(n) \leq cg(n)$ für eine Konstante c und alle $n \geq n_0$].

Natürlich ist die Laufzeit eines jeden Algorithmus zur Maximumsuche mindestens n; jedes Element der Eingabe muß betrachtet werden. Damit ist obiger Algorithmus also optimal bis auf einen konstanten Faktor und wir können sagen, daß die Komplexität des Maximumproblems linear in n ist. Kann man eine ähnliche Aussage für jedes Problem machen, d.h. gibt es stets bis auf konstante Faktoren optimale Algorithmen?

Eine Antwort auf diese Frage gibt die abstrakte Komplexitätstheorie. Sie macht Aussagen über den Betriebsmittelverbrauch von Algorithmen unter zwei sehr allgemeinen Annahmen:

1) der Betriebsmittelverbrauch einer Rechnung ist endlich genau dann, wenn die Berechnung terminiert,

2) es ist entscheidbar, ob ein Algorithmus an einer bestimmten Eingabe in einer vorgegebenen Betriebsmittelschranke läuft.

Aus diesen Axiomen folgt das Speed-Up-Theorem: Es gibt Probleme, die keinen optimalen Algorithmus besitzen; genauer, für jede Funktion $g: N \to N$ gibt es ein Problem P, so daß es für jeden Algorithmus A für P einen Algorithmus B für P gibt mit $g(T_B(n)) \leq T_A(n)$ für unendlich viele n. Algorithmus B ist also um den "Faktor" g schneller als A. [vgl. Schnorr oder Paul für eine eingehende Diskussion]. Der Beweis des Speed-Up-Theorems benutzt Diagonalisierung und liefert kein natürliches Problem mit Speed-Up. Es ist auch kein natürliches Problem mit Speed-Up bekannt. Allerdings hat man z.B. für die Matrizenmultiplikation immer schnellere Algorithmen gefunden: klassisch $O(n^3)$, Strassen $O(n^{2.81})$, Pan $O(n^{2.79})$ und diese Reihe ist damit sicher nicht zu Ende.

Im allgemeinen kann man also die Komplexität eines Problems nur durch die Angabe von Schranken eingrenzen, obere Schranken durch die Angabe und Analyse eines Algorithmus und (schwieriger) untere Schranken durch den Beweis,daß kein Algorithmus einer bestimmten Laufzeit existieren kann.

Beispiel 2: Sortieren: Gegeben ist eine Folge A[1:n]. Man sortiere diese Folge in aufsteigender Reihenfolge. Unter Verwendung unserer Lösung des Maximumproblems können wir in quadratischer Zeit sortieren. Effizienter ist es, Quicksort zu benutzen, dessen Laufzeit im Mittel (wieder unter der Annahme, daß alle Permutationen der Eingabe gleichwahrscheinlich sind) nur O(n log n) beträgt. Quicksort beruht auf dem Prinzip "Teile und Beherrsche", das sich als sehr fruchtbar bei der Entwicklung effizienter Algorithmen erwiesen hat. [vgl. Mehlhorn, Borodin/Munro]. Man teilt dabei das Problem in Teilprobleme [bei

Quicksort, indem man die Ausgangsmenge in die Menge der Elemente $\leq$ A[1] und die Menge der Elemente > A[1] partitioniert] auf, löst die Teilprobleme rekursiv und setzt die Lösungen zu einer Gesamtlösung zusammen. [hier schreibt man die sortierten Folgen hintereinander] .

Für eine eingeschränkte Klasse von Sortieralgorithmen kann man n log n auch als untere Schranke beweisen: Algorithmen, die ihre Information über die Eingabe nur aus Vergleichen ziehen dürfen [sog. Entscheidungsbaumalgorithmen, vgl. Mehlhorn]. Das schließt insbesondere Algorithmen aus, die den Aufbau der Elemente von A als Zeichenreihen ausnutzen [z.B. Sortieren durch Fachverteilung]. Für festes n kann man die möglichen Ausführungen eines solchen Algorithmus als binären Baum darstellen. Beachten Sie, daß nur Vergleiche zwischen Elementen von A zu Verzweigungen führen können. Da es n! mögliche Permutationen der Eingabe gibt, muß ein solcher Baum n! Blätter haben und daher (mittlere) Tiefe log n! = O(n log n).

In den letzten Jahren sind für viele Probleme nichttriviale obere Schranken angegeben worden, z.B. für Such- und Sortierprobleme (vgl. Mehlhorn), Graphenprobleme (vgl. Mehlhorn, Aho/Hopcroft/Ullman, Lawler), algebraische Probleme (Borodin/Munro).

Allerdings hat auch eine große Klasse von Problemen allen Versuchen wiederstanden, effiziente Algorithmen für sie zu finden: die NP-vollständigen Probleme: z.B. Erfüllbarkeitsproblem der Aussagenlogik, Problem des Handlungsreisenden, Rucksackproblem, und im Augenblick etwa 2ooo andere [vgl. Mehlhorn]. Jedoch sind diese Probleme vom komplexitätstheoretischen Standpunkt sämtlich equivalent. Eines dieser Probleme besitzt nämlich einen Algorithmus, dessen Laufzeit durch ein Polynom beschränkt ist, genau dann, wenn alle diese Probleme einen solchen besitzen. Das technische Hilfsmittel für einen Äquivalenzbeweis von Problemen ist die Reduktion. Reduktionen bringen Struktur in die Menge der Probleme!

Beispiel 3: Reduktion: Gegeben sei eine Folge A[1:n]. Besitzt die Folge zwei gleiche Elemente? Offensichtlich kann man dieses Problem lösen, indem man die Folge sortiert und dann überprüft, ob ein Paar von benachbarten Elementen gleich ist. Wir haben damit das "zwei-gleich-Problem" auf das Sortierproblem reduziert. Im Kontext der Entscheidungsbaumalgorithmen gilt auch die Umkehrung.

Beispiel 4: Hamilton'scher Kreis: Sei G = (V,E) ein gerichteter Graph. Gibt es einen geschlossenen Pfad durch G, der jeden Knoten genau einmal berührt?

Alle bekannten Algorithmen für das Hamilton'sche Kreis-Problem probieren im wesentlichen sämtliche |V|! Möglichkeiten eines Kreises durch und haben daher nichtpolynomielle Laufzeit. Ändern wir dagegen das Berechnungsmodell, so wird das Problem fast trivial: nichtdeterministische Algorithmen. Nichtdeterministische Algorithmen erhält man durch Einführung eines "Auswahlbefehls": goto M_1 oder M_2 für Marken M_1 und M_2. Wir verlangen nur noch, daß eine der möglichen Rechnungen (bestimmt durch die Auswahl in den Auswahlbefehlen) zum richtigen Ergebnis führt und definieren die Länge der kürzesten solchen Rechnung als (nichtdeterministische) Zeitkomplexität.

Beispiel 4 Fortsetzung: Ein nichtdeterministischer Algorithmus für das Hamilton'sche Kreisproblem schreibt nichtdeterministisch eine Folge $v_o, v_1, \ldots, v_n$ von Knoten nieder und überprüft dann deterministisch, ob $n=|V|$, $v_i \neq v_j$ für $1 \leq i < j \leq n$, $v_o = v_n$ und $(v_i, v_{i+1}) \in E$ für $o \leq i \leq n$. Die Laufzeit ist sicher polynomiell.
Nimmt man V={o,...,n} an, so muß man nichtdeterministisch eine Folge von n+1 Zahlen aus {o,...,n} wählen. Das folgende Programmstück wählt nichtdeterministisch eine solche Zahl.

```
      i ← 0;
M     goto M1 or M2;
M1:   i ← i+1;
      if  i < n  then goto  M;
M2:
```

Beispiel 5: Sortieren: Ein nichtdeterministischer Algorithmus kann in Linearzeit sortieren, indem er nichtdeterministisch eine Permutation niederschreibt und dann überprüft, ob diese Permutation sortiert.

Sei NP die Menge der in Polynomzeit auf nichtdeterministischen Maschinen lösbaren Probleme. Die oben erwähnten 2000 Probleme sind sämtlich in NP und sie sind schwierigste (im Sinn von Reduktionen) Probleme in NP; sie sind NP-vollständig. Die Frage, ob eines dieser Probleme einen polynomialen Algorithmus besitzt, ist das P=NP-Problem, eine der zentralen Fragestellungen der konkreten Komplexitätstheorie. Dabei ist P die Klasse der in Polynomzeit auf deterministischen Maschinen lösbaren Probleme. Andere typische Fragestellungen der konkreten Komplexitätstheorie sind die Frage nach dem Unterschied zwischen Platz (reusable

resource) und Zeit (non-reusable resource) und die Frage nach der Berechnungskraft von verschiedenen Maschinenmodellen wie Turingmaschinen, Registermaschinen, Multiprozessormaschinen. (vgl. Paul).

Zum Schluß des Vortrages möchten wir wieder auf die Ausgangsfrage zurückkommen:schnelle Algorithmen. Nichtdeterministische Algorithmen scheinen sehr schnell zu sein, sind aber leider nicht praktikabel. Die implicite Annahme ist ja, daß sie in Auswahlbefehlen stets die richtige Wahl treffen. Aber münzwerfende (probabilistische, Monte Carlo) Algorithmen sind realistisch. Solche Algorithmen treffen bei Auswahlbefehlen die Wahl durch Werfen einer Münze(mechanischer Arm mit Fernsehauge oder Zufallsgenerator oder ...)

Beispiel 6: probabilistisches Quicksort: Die Laufzeit von Quicksort ist im schlechtesten Fall quadratisch. Wählt man etwa stets das erste Element als Partitionselement, so ist die sortierte Eingabefolge 1,2,...,n ein schlechtester Fall. Wählt man dagegen das Partitionselement zufällig (oder ordnet man die Folge vor dem Sortieren zufällig um), so ist die Laufzeit an jeder einzelnen Eingabe im Mittel O(nlogn).

Die Laufzeit und im allgemeinen auch die Ausgabe eines probabilistischen Algorithmus hängt im allgemeinen von der Folge der Münzwürfe ab. Beispiel 6 zeigt, wie man mittleres Verhalten auf jede einzelne Eingabe übertragen kann. Wesentlich bemerkenswerter; ist man bereit bei der Antwort einen kleinen Fehler zuzulassen, so kann man in manchen Fällen einen deutlichen Effizienzgewinn erzielen. So haben Solovay/ Strassen gezeigt, daß man Primzahlen bei einem beliebig kleinen Fehler in Polynomzeit in der Länge der Dezimaldarstellung erkennen kann. Genauer: Primzahlen werden immmer als solche erkannt, aber Nichtprimzahlen werden mit kleiner Wahrscheinlichkeit zu Primzahlen erklärt. Alle bekannten determininistischen Algorithmen für dieses Problem haben nicht-polynomielle Laufzeit. Dieser gewaltige Unterschied in der Laufzeit wirft eine philosophische Frage auf. Besteht ein qualitativer Unterschied zwischen der "eingebauten" Möglichkeit einer falschen Antwort bei probabilistischen Algorithmen und der Möglichkeit einer falschen Antwort aufgrund eines Hardwarefehlers bei deterministischen Algorithmen, auch wenn die zweite Wahrscheinlichkeit wegen der größeren Laufzeit wesentlich höher ist?

Referenzen

K. Mehlhorn : Effiziente Algorithmen, Teubner Studienbücher Informatik

C.P. Schnorr : Rekursive Funktionen und ihre Komplexität, Teubner Studienbücher Informatik

W. Paul : Komplexitätstheorie, Teubner Studienbücher Informatik

Borodin/Munro: The Computational Complexity of Algebraic and Numeric Problems

D.E. Knuth : The Art of Computer Programming

Aho, Hopcroft, Ullmann : The Design and Analysis of Computer Algorithms.

Lawler E. : Discrete Optimization, Graphs and Matroids

Solovay/Strassen: A fast Monte-Carlo Test for Primality, SIAM J. of Computing, 1978