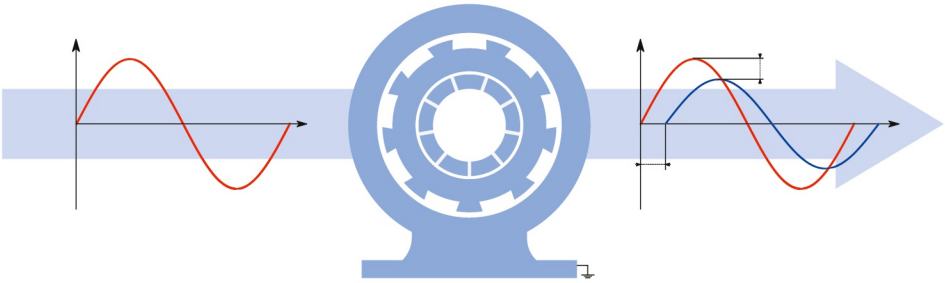
Universität Stuttgart

Fehlerdiagnose bei Drehstrommaschinen mittels Analyse der Frequenzantwort

Lukas Ranzinger



Schriftenreihe des Instituts für
Energieübertragung und Hochspannungstechnik

Band 51 – 2025

Komplexitätstheorie

Lucien Sina

1.9.2025

Inhaltsverzeichnis

Vorwort

Die Komplexitätstheorie ist eines der faszinierendsten und zugleich herausforderndsten Gebiete der theoretischen Informatik. Sie beschäftigt sich mit einer scheinbar einfachen, aber in ihrer Tiefe höchst bedeutenden Frage: Wie viel Aufwand ist nötig, um ein Problem zu lösen? Während die Berechenbarkeitstheorie untersucht, welche Probleme überhaupt algorithmisch lösbar sind, fragt die Komplexitätstheorie nach der Effizienz dieser Lösungen. Damit bildet sie die Brücke zwischen den Grundprinzipien der Berechnung und der praktischen Machbarkeit algorithmischer Verfahren.

Dieses Buch entstand aus dem Wunsch heraus, die Konzepte, Methoden und Ideen der Komplexitätstheorie in einer systematischen und zugleich verständlichen Weise darzustellen. Es führt von den grundlegenden Begriffen der Komplexitätsklassen und Reduktionen bis zu den modernen Themen wie der PCP-Theorie, interaktiven Beweissystemen, Speicherplatzkomplexität, Schaltkreis- und Kommunikationskomplexität. Dabei wird stets ver-

sucht, den Leser nicht nur mit Definitionen und Sätzen zu konfrontieren, sondern auch die dahinterliegenden Intuitionen und Zusammenhänge offenzulegen.

Ein zentrales Anliegen dieses Buches ist es, die innere Struktur der Komplexitätstheorie sichtbar zu machen: Wie bauen die einzelnen Konzepte aufeinander auf? Wie leiten sich heutige Forschungsthemen aus klassischen Fragestellungen ab? Und wie kann man aus theoretischen Grenzen wertvolle Einsichten für den praktischen Algorithmenentwurf gewinnen?

Die Kapitel sind so angelegt, dass sie sowohl einzeln als auch im Zusammenhang gelesen werden können. Jedes behandelt einen abgeschlossenen Themenbereich, von den Grundlagen über die NP-Vollständigkeit bis hin zu neueren Forschungsfeldern. Durch Beispiele, Beweisskizzen und ergänzende Übungen wird der Leser ermutigt, selbst aktiv mit den Begriffen zu arbeiten und ihre Tragweite zu erfassen.

Dieses Buch versteht sich nicht nur als Einführung, sondern als zusammenführendes Werk im Kontext anderer grundlegender Themen der Informatik. Meine begleitenden Bücher — *Algorithmen und Datenstrukturen*, *Berechenbarkeitstheorie*, *Logik: Grundlagen, das P vs. NP-Problem und informationstheoretische Perspektiven* sowie *Objektorientierte Programmierung in Java* — bilden zusammen mit dem vorliegenden Werk eine inhaltlich abgestimmte Einheit. Gemeinsam spannen sie den Bogen von der theoretischen Fundierung über algorithmisches Denken bis hin zur praktischen Umsetzung in Program-

men.

Die Komplexitätstheorie bleibt ein offenes und lebendiges Forschungsgebiet. Viele ihrer zentralen Fragen — allen voran das P-vs.-NP-Problem — sind bis heute ungelöst. Dennoch hat sich in den letzten Jahrzehnten gezeigt, dass die Methoden und Einsichten der Komplexitätstheorie weit über die reine Theorie hinausreichen. Sie helfen uns, die Grenzen des Machbaren zu verstehen, effiziente Verfahren zu entwickeln und selbst dort, wo vollständige Lösungen unmöglich erscheinen, optimale Kompromisse zu finden.

Ich hoffe, dass dieses Buch den Leserinnen und Lesern ein klares, kohärentes und anregendes Bild dieses Fachgebiets vermittelt, sie zum Weiterdenken anregt und das theoretische Fundament legt, auf dem viele praktische Errungenschaften der Informatik ruhen.

Lucien Sina

Kapitel 1

Einführung

1.1 Was ist Komplexitätstheorie?

Die Komplexitätstheorie untersucht, welche Mindestressourcen zur Lösung algorithmischer Probleme erforderlich sind. Anders als der Entwurf konkreter Algorithmen, der obere Schranken für den Ressourcenbedarf liefert, versucht die Komplexitätstheorie untere Schranken zu beweisen — also Aussagen darüber, wie viel Zeit, Speicher oder andere knappe Mittel *jeder* Algorithmus benötigen muss, der ein gegebenes Problem korrekt löst. Solche Aussagen erlauben es, zwischen dem praktisch Lösbaren und dem prinzipiell Unmöglichen zu unterscheiden und so unnötige Suche nach unerreichbaren Lösungen zu vermeiden.

Ein algorithmisches Problem formuliert man typischerweise in allgemeiner Form, indem man Eingaben,

erwünschte Ausgaben und Qualitätskriterien präzisiert. Reicht die allgemein formulierte Variante nicht für effiziente Algorithmen aus, verfolgt man zwei klassische Strategien: Entweder man schränkt die Eingabeklasse oder die Anforderungen ein (Spezialisierung), oder man akzeptiert approximative/heuristische Lösungen (Schwächung der Anforderungen). Beide Wege können praktikable, oft sogar sehr nützliche Resultate liefern, weil sie den Kern der Schwierigkeit eines Problems offenerlegen.

Im Gegensatz zum reinen Algorithmenentwurf, der konkrete Verfahren liefert, liefert die Komplexitätstheorie ein formales Instrumentarium zur Einordnung von Problemen: Klassen wie P (polynomielle Laufzeit), NP (nichtdeterministisch polynomielle Verifizierbarkeit) oder Speicherklassen fassen Probleme nach ihrem asymptotischen Ressourcenbedarf zusammen. Reduktionen zwischen Problemen erlauben, ihre relative Schwierigkeit zu vergleichen: Lässt sich Problem A effizient auf Problem B reduzieren, so ist A nicht schwerer als B bezogen auf die betrachteten Ressourcen.

Das prominenteste noch offene Problem der Disziplin ist das P vs. NP-Problem: Ist jede Eigenschaft, deren Gültigkeit sich in polynomieller Zeit verifizieren lässt, auch in polynomieller Zeit entscheidbar? Viele wichtige kombinatorische Entscheidungsprobleme (z. B. SAT, Clique, Hamiltonkreis) sind NP-vollständig, d. h. sie gehören zur schwersten Schicht von NP. Fände man für eines dieser Probleme einen Algorithmus in Polynomzeit, ergäbe sich für alle NP-Probleme eine Polynomzeitschranke;

ein Beweis der Unmöglichkeit würde dagegen eine ganze Klasse effizienter Algorithmen ausschließen.

Neben Laufzeit und Speicher treten in praktischen und theoretischen Betrachtungen weitere Ressourcen in den Vordergrund: Ein- und Ausgabekapazität, Energieverbrauch, Kommunikationsbandbreite, Schaltkreisgröße und -tiefe, Quantenzugänge u.ä. Je nach Anwendung und Rechenmodell verändern sich die relevanten Klassen und die Techniken zu ihrer Analyse. Entsprechend vielfältig sind die Beweismethoden: Diagonalisierungen, Simulationen zwischen Modellen, Informationstheorie, Reduktionsbeweise, Adversary-Methoden und – in neueren Entwicklungen – Methoden aus der Algebra und der Statistik.

Die Komplexitätstheorie und der Algorithmenentwurf sind komplementär: Algorithmen zeigen, was erreichbar ist; untere Schranken zeigen, was nicht erreichbar ist. Beide Perspektiven zusammen führen zu einem tieferen Verständnis algorithmischer Probleme — sei es durch das Finden effizienter Verfahren, das Aufdecken von Hindernissen oder das Erkennen struktureller Ursachen von Härte.

1.2 Algorithmische Probleme

Definition 1. *Ein **algorithmisches Problem** ist eine Menge zulässiger Eingaben (Instanzen), die als endliche Wörter über einem endlichen Alphabet Σ dargestellt werden, zusammen mit einer Zuordnung, die jeder zu-*

lässigen Eingabe $x \in \Sigma^*$ eine nichtleere Menge korrekter Ausgaben $S(x)$ (Antworten, Zertifikate, Werte) zuweist. Die Ausgaben sind ebenfalls endliche Wörter über einem endlichen Alphabet (eventuell über einem anderen Alphabet). Wir nennen eine konkrete Eingabe auch eine Instanz *des Problems; das Problem selbst ist die Menge aller Instanzen zusammen mit der Spezifikation der zulässigen Ausgaben.*

Durch diese Formalisierung passen wir Probleme an die Verarbeitungsmöglichkeiten von Rechnern an: Eingaben und Ausgaben werden kodiert als bit- oder zeichenbasierte endliche Wörter, so dass die Frage nach Berechenbarkeit und Ressourcennutzung präzise gestellt werden kann.

Man unterscheidet nach der Art der zu liefernden Ausgabe mehrere gebräuchliche Problemtypen:

- **Entscheidungsprobleme:** Die Ausgabe ist eine von zwei Antworten (z. B. `Ja`/`Nein`). Entscheidungsprobleme werden häufig als Sprachen $L \subseteq \Sigma^*$ formuliert: eine Instanz x gehört zu L genau dann, wenn die richtige Antwort `Ja` ist.

- **Suchprobleme:** Es genügt, für eine Instanz x irgendein $y \in S(x)$ zu finden (z. B. ein Beweis oder ein Zertifikat).

- **Optimierungsprobleme:** Zu jeder Instanz gehört eine Menge von zulässigen Lösungen mit einer Bewertungsfunktion; gesucht wird eine Lösung mit

maximaler (oder minimaler) Qualität (z. B. kürzester Hamiltonkreis).

- **Auswertungsprobleme:** Statt einer konkreten Lösung interessiert nur der zugehörige Wert (z. B. die Länge des kürzesten Weges zwischen zwei Knoten).

Oft lassen sich Probleme ineinander überführen: Aus einem Optimierungsproblem kann man durch Parametrisierung eines Schwellwerts ein Entscheidungsproblem machen (z. B. „Existiert ein Zyklus der Länge $\leq k$?¨). Umgekehrt kann ein Entscheidungsproblem als Spezialfall eines Such- oder Auswertungsproblems gesehen werden.

Für exakt lösbare Probleme gelten alle korrekten Ausgaben als gleichwertig; bei schwereren Problemen wird hingegen häufig eine Qualitätsmetrik eingeführt und Approximation zugelassen. Eine übliche Formulierung für Approximationsgüte ist ein Faktor $\alpha \geq 1$ (für Minimierungsprobleme gilt eine Lösung als α-approximativ, wenn ihr Wert höchstens α mal so groß ist wie der optimale Wert). Solche Abschwächungen der Forderung nach Optimalität sind sowohl praktisch als auch theoretisch bedeutsam.

Ein algorithmisches Problem gilt als *gelöst*, wenn es einen effektiven, wohldefinierten Mechanismus (Algorithmus, Turingmaschine o. ä.) gibt, der für jede zulässige Instanz eine korrekte Ausgabe aus $S(x)$ liefert. In der Komplexitätstheorie interessiert uns darüber hinaus, mit welchem Aufwand—etwa in Zeit, Speicher, Kommunikationsschritten, Energie oder Schaltkreistiefe—dies ge-

schieht.

Schließlich ist die Repräsentation von Eingaben wichtig: Verschiedene Kodierungen derselben mathematischen Eingabe können zu unterschiedlichen algorithmischen Problemen führen. In der Praxis betrachtet man Kodierungen als äquivalent, wenn sie sich effizient ineinander transformieren lassen (z. B. in polynomieller Zeit). Diese Idee der effizienten Kodierung erlaubt es, die Schwierigkeit von Problemen unabhängig von willkürlichen Darstellungsdetails zu vergleichen.

Beispiele. Typische Beispiele, die die obigen Unterscheidungen illustrieren:

- **SAT** (entscheidend): Gibt es eine befüllende Variablenzuweisung für eine gegebene boolesche Formel?
- **Kürzester Weg** (auswertend/optimierend): Bestimme die Länge bzw. den Pfad mit minimaler Distanz zwischen zwei Knoten in einem gewichteten Graphen.
- **TSP** (optimierend / approximativ relevant): Finde einen kürzesten Hamiltonkreis in einem vollständigen, gewichteten Graphen.

Die präzise Formulierung algorithmischer Probleme und ihrer Kodierungen bildet die Grundlage für alle weiteren Definitionen von Komplexitätsmaßen und -klassen, die im folgenden Kapitel behandelt werden.

1.3 Algorithmus

Was ist ein Algorithmus?

Definition 2. *Ein* ***Algorithmus*** *ist eine wohldefinierte, eindeutige Regelvorschrift, die für jede zulässige Eingabe des betrachteten algorithmischen Problems eine endliche, wohlgeordnete Folge von Rechenschritten beschreibt, die — bei Terminierung — eine korrekte Ausgabe erzeugt.*

Für die Komplexitätstheorie sind neben dem Begriff „Algorithmus“ verschiedene Rechenmodelle und Ausführungsmodi von Bedeutung. Wir geben hier prägnante Definitionen der üblichen Typen:

Definition 3. *Ein Algorithmus heißt* ***deterministisch****, wenn bei jeder Konfiguration (jeweils bestehend aus Eingabe, Arbeitsinhalt und Zustand) der nächste Rechenschritt eindeutig festgelegt ist. Deterministische Algorithmen entsprechen dem intuitiven Ablaufprogramm, in dem jede Entscheidung durch die bisher gemachten Angaben bestimmt wird.*

Definition 4. *Ein Algorithmus heißt* ***nichtdeterministisch*** *(im theoretischen Sinn), wenn er an manchen Stellen mehrere mögliche Folgeschritte zulässt und eine Eingabe als* akzeptiert *gilt, sobald* mindestens ein *Ausführungszweig zu einer akzeptierenden Endkonfiguration führt. Formal modelliert man nichtdeterministische Algorithmen z. B. durch nichtdeterministische Turingmaschinen, die simultan alle Wahlmöglichkeiten „erraten“ (oder äquivalent: durch Existenz eines passenden Zertifikats).*

Definition 5. *Ein Algorithmus ist **randomisiert** (oder **probabilistisch**), wenn seine Ausführung zusätzlich Zugang zu Zufallsbits hat und einzelne Entscheidungen während der Ausführung vom Ergebnis dieser Zufallsbits abhängen. Randomisierte Algorithmen werden nach ihrem Fehlerverhalten weiter klassifiziert (z. B. Monte-Carlo mit schlechtester Fehlerrate $< \frac{1}{2}$ oder Las-Vegas, die stets korrekte Ergebnisse liefern, dafür aber mit zufälliger Laufzeit).*

Ein wichtiger Punkt: Nichtdeterminismus und Randomisierung sind verschiedene Konzepte. Nichtdeterminismus ist ein theoretisches Modell, das das Vorhandensein eines „richtigen“ Wahlzweigs verlangt (Existenz einer Lösung), während Randomisierung echte Zufallsentscheidungen mit quantifizierbaren Fehlern bzw. Erwartungswerten für Laufzeit oder Ausgabe erlaubt. Es ist nicht korrekt, Nichtdeterminismus pauschal als Spezialfall von Randomisierung zu betrachten — die genaue Beziehung der entsprechenden Komplexitätsklassen (etwa P, NP, RP, BPP, ZPP) ist Gegenstand zentraler Fragestellungen der Theorie und teilweise offen.

Beispiele.

- Ein **deterministischer** Algorithmus: Mergesort — für jede Eingabe ist der Ablauf eindeutig und liefert in $O(n \log n)$ Zeit eine sortierte Folge.
- Ein **nichtdeterministischer** Entscheidungsverfahren für SAT: „Errate“ eine Belegung der Variablen

und verifiziere deterministisch, dass die Formel erfüllt ist. Formal zeigt dies, dass SAT in NP liegt.

- Ein **randomisierter** Algorithmus: Randomized Quicksort (erwartete Laufzeit $O(n \log n)$); Miller–Rabin als Monte-Carlo-Test für Komposität (kleine Fehlerrate, sehr schnelle Ausführung).

In der Komplexitätstheorie formalisieren diese Konzepte unterschiedliche Rechenmodelle und führen zu verschiedenen Klassen, die im Folgenden systematisch eingeführt und verglichen werden.

1.4 Laufzeit eines Algorithmus

Die Laufzeit eines Algorithmus ist ein zentrales Maß für dessen Effizienz. Eine naive Definition — die absolute, in Sekunden gemessene Rechenzeit auf einem konkreten Rechner — ist für theoretische Vergleiche ungeeignet, weil sie von vielen externen Faktoren abhängt (Hardware, Programmiersprache, Implementierungsdetails, Betriebssystem, Compiler-Optimierungen u. ä.). Ziel einer theoretisch sauberen Betrachtung ist es, ein Laufzeitmaß zu wählen, das nur vom Algorithmus selbst und von seiner Eingabe abhängt und das verschiedene sinnvolle Rechenmodelle in verträglicher Weise vergleicht.

Dazu zählen zwei grundlegende Schritte:

1. Wahl eines abstrakten Rechenmodells, das die elementaren Rechenoperationen und deren Kosten festlegt.

2. Formulierung eines asymptotischen Laufzeitmaßes, das die Abhängigkeit der benötigten Ressourcen von der Eingabelänge beschreibt.

1.4.1 Elementare Rechenschritte und Kostenmaße

Als Laufzeit messen wir üblicherweise die Anzahl elementarer Rechenschritte, wobei als elementare Operationen solche Operationen angesehen werden, die auf einem realistischen Rechenmodell als „konstant“ gelten: arithmetische Grundoperationen auf Wörtern fester Länge, Lesen/Schreiben einer Speicherzelle, Zustandstransitionen, Verschieben eines Lesekopfs, etc. Zwei verbreitete Kostenmaße sind:

Einheitskostenmodell (unit cost): Jede elementare Operation kostet 1. Dies ist einfach zu handhaben, kann aber bei Operationen auf sehr großen Zahlen unrealistische Laufzeitangaben liefern (z. B. Addition beliebig großer Ganzzahlen wird konstant gezählt).

Logarithmisches Kostenmaß (bit cost): Die Kosten einer Operation hängen proportional an der Bitlänge der Operanden; das Modell zählt sozusagen Bitoperationen. Damit reflektiert das Maß die tatsächliche Schwierigkeit von Rechenoperationen auf großen Zahlen besser.

Für viele theoretische Aussagen (insbesondere im Bereich der Komplexitätsklassen wie P und NP) ist es vorteilhaft, ein Kostenmaß zu wählen, das robust gegenüber der Wahl eines konkreten, „vernünftigen“ Rechenmodells ist. Robustheit bedeutet hier, dass zwei sinnvolle Modelle die Laufzeiten nur um einen polynomiellen Faktor unterscheiden — Polynom-Äquivalenz ist das übliche Äquivalenzkriterium in der Komplexitätstheorie.

1.4.2 Turingmaschine: formales Rechenmodell

Das formale Modell, das dem intuitiven Begriff des Algorithmus am nächsten kommt und sich in der Komplexitätstheorie als Standard durchgesetzt hat, ist die Turingmaschine. Wir geben hier die übliche (deterministische) Definition in komprimierter Form.

Definition 6 (Deterministische Turingmaschine)**.** *Eine (deterministische) Turingmaschine ist ein Tupel*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}),$$

wobei Q eine endliche Menge von Zuständen, Σ das Eingabealphabet (ohne das Blank-Symbol $\sqcup$), Γ das Arbeitsalphabet mit $\Sigma \subseteq \Gamma$, $q_0 \in Q$ der Startzustand sowie $q_{acc}, q_{rej} \in Q$ die akzeptierenden bzw. ablehnenden Endzustände sind. Die Übergangsfunktion

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

bestimmt in jedem Schritt den nächsten Zustand, das zu schreibende Symbol und die Kopfbewegung (links, rechts, Stillstand).

Eine Konfiguration einer Turingmaschine beschreibt die aktuelle Maschine, den Bandinhalt und die Kopfposition. Ein Schritt entspricht einer Anwendung von δ. Die *Laufzeit* von M auf Eingabe x ist die Anzahl der Schritte bis zur ersten Erreichung eines Endzustands (sofern M terminiert). Die worst-case-Laufzeit einer Maschine für Eingaben der Länge n schreibt man als $T_M(n) = \max\{\,\text{Schritte}(M, x) \mid |x| = n\,\}$.

In der Praxis arbeitet man oft mit Mehrband-Turingmaschinen (mehrere Arbeitsbänder), da sie viele Algorithmen natürlicher modellieren; die verschiedenen Varianten sind jedoch polynomiell äquivalent (Mehrband-Maschinen simulieren Einband-Maschinen mit nur quadratischem bzw. polynomiellem Overhead und umgekehrt).

1.4.3 Polynomielle Robustheit und die erweiterte Churchsche These

Die (erweiterte) Church–Turing-These in ihrer polynomiellen Form (häufig als *polynomial Church–Turing thesis* oder *extended Church–Turing thesis*) behauptet im Wesentlichen Folgendes:

Aussage (polynomielle Simulation). *Jedes „vernünftige“ Rechenmodell R (z. B. Random-Access-Maschine,*

Registermaschine, realistische Hardwaremodelle ohne unbeschränkte, magische Operationen) lässt sich durch eine Turingmaschine so simulieren, dass eine Ausführung mit t elementaren Schritten (unter einem angemessenen Kostenmaß, z. B. dem logarithmischen/bitweisen Kostenmaß) auf R in höchstens $p(t)$ Schritten auf einer (Mehrband-)Turingmaschine ausgeführt werden kann, wobei p ein Polynom ist, das von der gewählten Simulationsmethode abhängt.

Diese polynomielle Robustheit rechtfertigt, Komplexitätsklassen (z. B. P, NP) und asymptotische Laufzeitaussagen auf dem Turingmodell zu formulieren — Aussagen sind dann modellunabhängig bis auf polynomielle Faktoren und damit für die meisten theoretischen Fragestellungen stabil.

Wir halten fest

Sei R ein realistisches Rechenmodell und sei die Laufzeit eines Algorithmus auf R unter dem logarithmischen Kostenmaß durch $t(n)$ für Eingaben der Länge n gegeben. Dann existiert eine (Mehrband-)Turingmaschine M und ein Polynom p mit

$$T_M(n) \leq p(t(n))$$

für hinreichend große n. Damit ist die zeitliche Komplexität in polynomieller Hinsicht modellunabhängig — ein zentraler Pfeiler der theoretischen Komplexitätsanalyse.

Bemerkungen

- Die genaue Form von p hängt von den erlaubten Operationen auf R ab; Modelle, die „unfaire“ Operationen erlauben (z. B. direkte Multiplikation beliebig großer Zahlen in konstanter Zeit oder leistungsfähige Orakel), fallen nicht unter diese Aussage.
- Für praktische Feinheiten (z. B. Kosten von großen integer-Operationen) ist das logarithmische Kostenmaß realistischer als das Einheitskostenmodell.
- Die polynomielle Äquivalenz begründet die Untersuchung von Komplexitätsklassen unter der Äquivalenzrelation „Gleichheit bis auf polynomielle Faktoren“, insbesondere wenn es um Fragen wie P vs. NP geht.

1.5 Komplexität algorithmischer Probleme

Sei $t_A(x)$ die Rechenzeit eines Algorithmus A für eine Eingabe x im unitären Kostenmaß auf einem Referenzmodell (z. B. einer Turingmaschine). Um Laufzeiten verschiedener Algorithmen für dasselbe algorithmische Problem vergleichbar zu machen, sind einige Vorsichtsmaßnahmen nötig; eine naive Punkt-für-Punkt-Vergleichsregel

$$A \text{ ist mindestens so schnell wie } B$$
$$\text{falls} \quad \forall x : \; t_A(x) \leq t_B(x)$$

führt auf mehrere praktische und theoretische Schwierigkeiten:

1. Die absoluten Rechenzeiten hängen vom gewählten Referenzmodell ab (Einband- vs. Mehrband-Turingmaschine, RAM-Modell usw.).

2. Die Forderung, alle Eingaben x zu betrachten, ist oft zu streng und nicht ergonomisch für asymptotische Aussagen.

3. Für kleine Eingaben kann ein „einfacherer“ Algorithmus schneller sein, während spezialisierte Algorithmen erst für große Eingaben ihre asymptotischen Vorteile zeigen.

Zur Überwindung dieser Probleme verwendet man in der Komplexitätstheorie folgende Standardkonzepte.

1.5.1 Worst-Case-Laufzeit

Für eine Maschine A definiert man die (worst-case-)Laufzeitfunktion

$$t_A(n) := \sup\{\, t_A(x) \mid |x| \leq n \,\},$$

bzw. oft äquivalent

$$t_A(n) = \max\{\, t_A(x) \mid |x| = n \,\},$$

sofern das Maximum existiert. Für die meisten sinnvollen Modelle ist $t_A(n)$ monoton wachsend in n. Der Vergleich

von Algorithmen erfolgt dann in asymptotischer Form: A sei *asymptotisch mindestens so schnell* wie B, falls

$$t_A(n) = O\big(t_B(n)\big).$$

1.5.2 Average-Case und Verteilungen

Falls es „Ausreißerinstanzen“ gibt, die die Worst-Case-Laufzeit dominieren, kann es sinnvoll sein, ein Wahrscheinlichkeitsmaß ν_n auf den Eingaben der Länge n zu betrachten. Die durchschnittliche Laufzeit von A unter ν ist dann

$$t_A^{\nu}(n) := \sum_{|x|=n} \nu_n(x)\, t_A(x),$$

wobei ν_n für jedes n eine Wahrscheinlichkeitsverteilung auf $\{x \mid |x| = n\}$ ist. Average-case-Analysen sind nur so aussagekräftig wie die Annahmen über die Eingabeverteilung.

Asymptotische Komplexitätsmaße

Die üblichen asymptotischen Notationen $O(\cdot)$, $\Omega(\cdot)$ und $\Theta(\cdot)$ fassen die Wachstumsordnung von Laufzeitfunktionen zusammen und machen Vergleiche modellrobust bis auf polynomielle Faktoren (vgl. die polynomielle Fassung der Church–Turing–These).

Definition 7. *Ein Algorithmus A löst ein Problem in Zeit $O(f(n))$, falls $t_A(n) = O(f(n))$. Man sagt, die (zeitliche)* ***algorithmische Komplexität*** *des Problems sei $f(n)$, falls*

1. *es einen Algorithmus A mit $t_A(n) = O(f(n))$ gibt (obere Schranke), und*

2. *jede Maschine, die das Problem löst, genügt mindestens einer Laufzeit $\Omega(f(n))$ (untere Schranke).*

Ist sowohl eine obere als auch eine untere Schranke durch dasselbe asymptotische $f(n)$ gegeben, so schreibt man die Komplexität als $\Theta(f(n))$ und hat damit die Laufzeitordnung des Problems bestimmt.

1.5.3 Modellunabhängigkeit

Die polynomielle Formulierung der Church–Turing–These besagt, dass für „vernünftige“ Rechenmodelle R_1, R_2 ein Polynom p existiert, so dass eine Ausführung von R_1 mit t elementaren Schritten in höchstens $p(t)$ Schritten auf R_2 simuliert werden kann. Dadurch sind Zeitkomplexitätsaussagen in polynomieller Hinsicht modellunabhängig — ein Grundpfeiler der Verwendung von Turingmaschinen als Standardmodell in der Komplexitätstheorie.

Bemerkungen

- Für feinere Analysen (z. B. Kosten großer Integer-Operationen) ist das gewählte Kostenmaß (Einheits- vs. Bitkosten) entscheidend.

- Neben Zeit betrachtet man selbstverständlich auch andere Ressourcen (Speicher, Kommunikation, Zufallsbits, Schaltkreistiefe), wobei für jede Res-

source analoge Worst-Case- bzw. Average-Case-Funktionen definiert werden.

- Untere Schranken sind oft schwer zu beweisen und bilden das Kernproblem vieler Resultate der Komplexitätstheorie.

Kapitel 2

Komplexitätsklassen

2.1 Die Komplexitätsklasse P

Polynomielle Laufzeiten nehmen in der Komplexitätstheorie eine besondere Stellung ein. Unter der (polynomiellen) Fassung der Church–Turing–These sind verschiedene „vernünftige“ Rechenmodelle polynomiell äquivalent, sodass Laufzeitfunktionen, die sich nur um polynomielle Faktoren unterscheiden, für die meisten theoretischen Fragen als gleichwertig gelten. Deshalb werden polynomielle Laufzeiten als eine robuste und modellunabhängige Formalisierung des Begriffs *effizient berechenbar* verwendet.

Ein weiterer praktischer Grund für die Bedeutung polynomialer Laufzeiten ist, dass viele Algorithmen die gesamte Eingabe lesen müssen; dadurch entsteht häufig bereits eine lineare Untergrenze $\Omega(n)$ für die

Laufzeit, und polynomiale Laufzeiten sind im Verhältnis zur Eingabelänge vergleichsweise moderat. Polynomielles Wachstum skaliert ferner gut mit konstanten Geschwindigkeitssteigerungen der Hardware: Vergrößert sich die Rechengeschwindigkeit um einen konstanten Faktor $a > 1$ und ist die Laufzeit eines Algorithmus $t(n) = n^k$, so lässt sich die verarbeitbare Eingabelänge von n auf etwa $\lfloor a^{1/k} n \rfloor$ erhöhen — ein Vorteil, der bei superpolynomiellen Laufzeitfunktionen verschwindet.

Definition 8. *Ein algorithmisches Problem A gehört zur Komplexitätsklasse* P *(polynomiell lösbare Probleme), geschrieben $A \in \mathrm{P}$, falls es eine deterministische Turingmaschine M und ein Polynom $p(n)$ gibt, so dass für alle Eingaben x gilt:*

$$\mathit{Laufzeit}_M(x) \leq p(|x|).$$

Äquivalent: die worst-case-Laufzeit $T_M(n) = \max_{|x|=n} \mathit{Laufzeit}_M(x)$ liegt in $O(p(n))$.

Anschaulich fasst P diejenigen Probleme zusammen, für die es einen Algorithmus gibt, dessen Laufzeit durch ein Polynom in der Eingabelänge beschränkt ist. In der theoretischen Informatik gilt P weithin als die Klasse der „praktisch lösbaren“ oder „effizient lösbaren“ Entscheidungsprobleme, wobei man sich bewusst ist, dass auch Algorithmen mit hohem Polynomgrad oder großen Konstanten in der Praxis unpraktisch sein können.

Abgrenzung zu anderen Wachstumsordnungen. Probleme mit Laufzeiten, die über alle Polynome hinaus