Komplexitätstheorie

Klassifikation algorithmischer Probleme
($\longrightarrow$ formalisiert als Sprachen)
nach ihrem Bedarf an *Berechnungsressourcen:*

- Rechenzeit
- Speicherplatz

jeweils als Funktion der Eingabelänge n

Ziele:

- Entwicklung effizienter Verfahren
- Nachweis oberer und unterer Schranken für die Komplexität von Problemen

Die $\mathcal{O}$-Notation

Für Funktionen $g : \mathbb{N} \to \mathbb{N}$ definiert man die folgenden drei Mengen:

$$\mathcal{O}(g) := \{f \mid \exists\ c > 0\ \exists\ n_0 \in \mathbb{N}\ \forall\ n \geqslant n_0 : f(n) \leqslant c \cdot g(n)\}$$

$$\Omega(g) := \{f \mid \exists\ c > 0\ \exists\ n_0 \in \mathbb{N}\ \forall\ n \geqslant n_0 : f(n) \geqslant c \cdot g(n)\}$$

$$\Theta(g) := \{f \mid \exists\ c > 0\ \exists\ n_0 \in \mathbb{N}\ \forall\ n \geqslant n_0 : \tfrac{1}{c} \cdot g(n) \leqslant f(n) \leqslant c \cdot g(n)\}$$

Im Falle $h \in \mathcal{O}(g)$ sagt man

„h ist von der Ordnung g“

oder auch einfach „h ist $\mathcal{O}(g)$“.

Vereinfachende Schreibweisen:
$\mathcal{O}(f(n))$ statt $\mathcal{O}(f)$
$\mathcal{O}(f) = \mathcal{O}(g)$ statt $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ und
$\mathcal{O}(f) < \mathcal{O}(g)$ statt $\mathcal{O}(f) \subsetneq \mathcal{O}(g)$.

Rechenregeln der $\mathcal{O}$-Notation

Sind $f_1 \in \mathcal{O}(g_1)$ und $f_2 \in \mathcal{O}(g_2)$, so gilt:

1. $f_1 + f_2 \in \mathcal{O}(\underbrace{\max\{g_1, g_2\}}_{\text{argumentweise}})$

2. $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$

Satz: Ein Polynom $p(n) = \sum_{i=0}^{k} a_i n^i$ vom Grade k ist in $\mathcal{O}(n^k)$.

Wichtigste $\mathcal{O}$-Klassen

$$\mathcal{O}(1) \subsetneq \mathcal{O}(\log\ n) \subsetneq \mathcal{O}(n) \subsetneq \mathcal{O}(n \log\ n) \subsetneq$$

$$\mathcal{O}(n^2) \subsetneq \ldots \subsetneq \mathcal{O}(n^k) \subsetneq \ldots \subsetneq \mathcal{O}(2^n)$$

konstant — logarithmisch — linear — $n \log n$ —

— polynomiell — exponentiell

Komplexität von Algorithmen

Algorithmen als Turingprogramme für *Mehrband*-Turingmaschinen
$\longrightarrow$ realistischere Komplexität

Zeit	$\widehat{=}$	Anzahl Schritte der Turingmaschine
Platz	$\widehat{=}$	Anzahl besuchter Felder der Turingmaschine

Im folgenden nur Zeitkomplexität:

Zeitkomplexität

Definition:
Sei $\mathcal{A} \in TM_k(\Sigma)$ eine Mehrband-Turingmaschine.

Definiere $\underline{\texttt{time}}_{\mathcal{A}} : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ durch

$$\underline{\texttt{time}}_{\mathcal{A}}(w) := \begin{cases} min\{l \mid \kappa_0(w) \overset{l}{\vdash} (q, \ldots) \text{ mit Schlusskonfiguration } (q, \ldots)\} \\ \qquad \text{falls } \mathcal{A} \text{ bei Eingabe von } w \text{ stoppt} \\ \infty \quad \text{sonst} \end{cases}$$

$\kappa_0(w)$ bezeichne die Anfangskonfiguration von $\mathcal{A}$ bei Eingabe von w.

Bemerkungen:
Eine Mehrband-Turingmaschine mit Rechenzeitbeschränkung $f : \mathbb{N} \to \mathbb{N}$, d.h.

$$\forall w \in \Sigma^* : \underline{\texttt{time}}_{\mathcal{A}}(w) \leqslant f(|w|),$$

kann durch eine Einband-Turingmaschine mit Rechenzeitbeschränkung $n \mapsto f(n)^2$ simuliert werden.

Alternative Berechenbarkeitsmodelle als Basis möglich: WHILE / GOTO / LOOP

Kostenmaße

Uniformes Kostenmaß:

Der Zugriff und die Operation auf einer Zahl $n \in \mathbb{N}$ kostet eine Einheit.
$\longrightarrow$ nur realistisch bei beschränkten Speicherplatzanforderungen pro Datum

Logarithmisches Kostenmaß:

Beim logarithmischen Kostenmaß wird die Größe der Operanden berücksichtigt. Die Kosten sind hier proportional zur Länge der Zahl in der Binärdarstellung. Der Zugriff und die Operation auf einer Zahl $n \in \mathbb{N}$ kosten

$$\begin{cases} 1 & \text{falls n=0} \\ \log_2(n+1) & sonst \end{cases}$$

Beispiel Betrachte folgendes Loop-Programm:

$P =$ $\underline{\texttt{in}}(X_1)$; $\underline{\texttt{var}}(X_2)$;
$X_2 := 2$;
$\underline{\texttt{loop}}\ X_1(X_2 := X_2 \cdot X_2)$;
$X_1 := X_2$;
$\underline{\texttt{out}}(X_1)$

P berechnet die Funktion $n \mapsto 2^{(2^n)}$.
Nach dem uniformen Kostenmaß ist $[\![P]\!] \in \mathcal{O}(n)$.
Nach dem logarithm. Kostenmaß ist $[\![P]\!] \in \mathcal{O}(2^n)$.

Die Komplexitätsklasse P

Definition: Sei $f : \mathbb{N} \to \mathbb{N}$.

$$\texttt{TIME}(f(n)) := \{\ L \subseteq \Sigma^* \mid \exists\, \mathcal{A} \in \mathbf{D}\mathsf{TM}_k(\Sigma) : L = L(\mathcal{A}) \text{ und } \forall\ w \in \Sigma^* : \underline{\texttt{time}}_{\mathcal{A}}(w) \leqslant f(|w|)\ \}$$

$$P := \bigcup_{p \text{ Polynom}} \texttt{TIME}(p(n))$$

enthält die

von deterministischen Turingmaschinen
in **P**olynomialzeit

erkennbaren Sprachen.

Für P kann man 1-Band-Turingmaschinen betrachten, da Polynome unter Quadrieren abgeschlossen sind.
P umfasst Probleme, für die „effiziente“ Algorithmen existieren.
$P \subseteq$ Klasse der LOOP-berechenbaren Sprachen
($\Rightarrow$ charakteristische Fkt ist LOOP-ber.)

Beispiel: Kruskal-Algorithmus

Bestimmung eines minimal aufspannenden Baums (MST: minimal spanning tree) zu einem Graphen

Gegeben: gewichteter Graph $G = (V, E, w)$ mit
– Knotenmenge V (vertices)
– Kantenmenge $E \subseteq V \times V$ (edges)
– Gewichtsfunktion $w : E \to \mathbb{N}$

Gesucht: minimal aufspannender Baum von G, d.h. $T \subseteq E$, so dass

- T ist ein Baum, der alle Knoten von G enthält ($\to T$ ist aufspannender Baum von G)

- $\bigcup_{e \in T} w(e)$ ist minimal unter allen aufspannenden Bäumen von G

Kruskal-Algorithmus

Verwalte eine Menge S von disjunkten Teilbäumen von G, die zusammen alle Knoten umfassen

- Zu Beginn sei $S = V$.
- Solange S mehr als einen Baum enthält, wiederhole:
  - Wähle eine Kante e minimalen Gewichts, die zwei Bäume t_1 und t_2 aus S verbindet
  - Lösche t_1 und t_2 aus S und füge den Baum, der aus t_1, t_2 und der Kante e besteht, zu S hinzu

Aufwand: $\mathcal{O}(n^4)$, falls $n = |V|$

$\Longrightarrow$ MST $\in P$

MST $= \{code(G)code(n) \mid$ G besitzt MST mit Gewicht $\leqslant n$ $\}$

Die Komplexitätsklasse NP

Definition: Sei $f : \mathbb{N} \to \mathbb{N}$

$$\texttt{NTIME}(f(n)) := \{ L \subseteq \Sigma^* \mid \exists\, \mathcal{A} \in \mathsf{TM}_k(\Sigma) : L = L(\mathcal{A}) \text{ und } \forall\, w \in \Sigma^* : \underline{\texttt{time}}_{\mathcal{A}}(w) \leqslant f(|w|) \}$$

$$NP := \bigcup_{p \text{ Polynom}} \texttt{NTIME}(p(n))$$

enthält die

von **N**ichtdeterministischen Turingmaschinen
in **P**olynomialzeit

erkennbaren Sprachen.

Offensichtlich gilt:

$$P \subseteq NP \subseteq \text{LOOP-ber. Sprachen}$$

Berühmtes offenes P-NP-Problem:

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

Beispiel: TSP

Traveling Salesman Problem
(Problem des Handlungsreisenden)

Gegeben: gewichteter Graph $G = (V, E, w)$ mit
– Knotenmenge V (vertices)
– Kantenmenge $E \subseteq V \times V$ (edges)
– Gewichtsfunktion $w : E \to \mathbb{N}$

Gesucht: Kreis in G, der alle Knoten umfasst und dessen Kantengewicht minimal ist

Jeder Knoten soll exakt einmal enthalten sein.
$\to$ Hamiltonscher Kreis

$\to$ Optimierungsproblem

Variante Entscheidungsproblem:
Gesucht ist ein Kreis mit Kantengewicht $\leqslant k$ für ein gegebenes k.

Für das TSP ist kein deterministischer Algorithmus bekannt, der wesentlich besser ist als der Folgende:

- Zähle systematisch alle Folgen $v_1, \ldots, v_n$ von Knoten von G auf, die jeden Knoten genau einmal enthalten.

  $\rightarrow$ Aufwand $\mathcal{O}(n!)$

- Teste, ob es sich um einen Kreis handelt, und wähle den Kreis mit minimalem Kantengewicht aus

  $\rightarrow$ Aufwand $\mathcal{O}(n)$

Es gilt $n! \sim (\frac{n}{e})^n \sqrt{2\pi n}$.

Dies übersteigt für jede Konstante c schließlich 2^{c*n}.

Eine nichtdeterministische Turingmaschine kann Folgen $v_1, \ldots, v_n$ *erraten* und dann in Polynomialzeit feststellen, ob es sich um eine Lösung handelt.

Eine Lösung kann also **nichtdeterministisch in Polynomialzeit** bestimmt werden, d.h.

$$TSP \in NP$$

Das P-NP-Problem

wichtige Frage der Theoretischen Informatik

Viele für die Praxis wichtige Probleme liegen in NP, z.B.
– Erfüllbarkeitsproblem der Aussagenlogik SAT
– Traveling Salesman Problem TSP
– viele weitere Graphenprobleme
Andererseits ist bisher kein polynomialer Algorithmus bekannt.

Charakteristisch für NP-Probleme:
– exponentiell großer Suchraum für Lösungen
$\rightarrow$ nichtdeterministische Auswahl
– polynomialer Aufwand für Feststellung, ob eine Lösung gefunden wird

Strukturtheorie für P-NP-Problem
(Cook 1971, Karp 1972)
Bis auf wenige Ausnahmen sind NP-Probleme, für die kein polynomialer Algorithmus bekannt ist ($\rightarrow$ Kandidaten für $NP \setminus P$),
so miteinander verknüpft, dass
– entweder alle diese Probleme polynomiale Algorithmen besitzen ($\rightarrow$ Fall $P = NP$)
– oder keines ($\rightarrow$ Fall $P \neq NP$)

NP-Vollständigkeit

Definition: Seien $A, B \subseteq \Sigma^*$.
A heißt *auf B polynomial reduzierbar*,
Bez.: $A \leqslant_p B$, falls es eine totale, **in Polynomzeit** berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass für alle $w \in \Sigma^*$ gilt:

$$w \in A \qquad \curvearrowright \qquad f(w) \in B$$

Lemma:
– $A \leqslant_p B \wedge B \in P \curvearrowright A \in P$
– $A \leqslant_p B \wedge B \in NP \curvearrowright A \in NP$
– $\leqslant_p$ ist transitiv.

Definition: Sei $A \subseteq \Sigma^*$.

(i) A heißt *NP-hart*, falls $\forall\ L \in NP : L \leqslant_p A$.

(ii) A heißt *NP-vollständig*, falls A NP-hart und $A \in NP$.

Satz:
Sei A NP-vollständig. Dann gilt:

$$A \in P \curvearrowright P = NP$$

.

Der Satz von Cook

Das Erfüllbarkeitsproblem der Aussagenlogik
SAT $= \{code(F) \mid$ F ist erfüllbare Formel der Aussagenlogik $\}$
ist NP-vollständig.

Beweis: $\mathbf{SAT} \in \mathbf{NP}$ ist einfach zu zeigen (guess and check):
Rate Belegung der AL-Variablen und teste, ob die Belegung F erfüllt

Hauptteil des Beweises (Nachweis der NP-Härte):
Zeige für beliebiges $L \in NP$: $L \leqslant_p SAT$.
Konstruiere Formel der AL F, so dass

$$w \in L \curvearrowright F \text{ ist erfüllbar.}$$

$\curvearrowright$ erstes NP-vollständiges Problem

Nachweis der Existenz weiterer solcher Probleme durch Reduktion

Beispiele für NP-harte Probleme:
– Wortproblem für Typ-1-Sprachen
– Äquivalenzproblem für Typ-3-Sprachen