Theoretische Informatik II

Einheit 8

Komplexitätstheorie

1. Komplexitätsmaße
2. Komplexität von Algorithmen (obere Schranken)
3. Komplexität von Problemen (untere Schranken)
4. NP-Vollständigkeit

Komplexitätstheorie
– Was kann mit vertretbarem Aufwand gelöst werden? –

- **Berechenbarkeit alleine reicht nicht**
  - Lösungen müssen effizient sein in praktischen Anwendungen
  - Berechenbarkeit/Entscheidbarkeit löst nur die Grundsatzfrage
- **Komplexität: Analyse benötigter Resourcen**
  - Zeitbedarf des Algorithmus — Time
  - Speicherbedarf des Verfahrens (RAM, Harddisk) — Space
  - Netzzugriffe, Zugriff auf andere Medien
- **Meßgröße muß unabhängig sein von**
  - Konkreter Hardware
  - Konkreter Programmiersprache
  - Optimierungsfähigkeiten des Compilers
  - Auswahl der Testdaten

$\Downarrow$

$\boxed{\textbf{Abstrakte Komplexitätsmaße erforderlich}}$

Fragestellungen der Komplexitätstheorie

- **Asymptotisches Verhalten von Algorithmen**
  - Komplexitätsfunktion: Bedarf abhängig von der Größe der Eingabe
  - Abschätzung der Komplexität großer Probleme
- **Analyse konkreter Verfahren**
  - Maximaler Verbrauch im Einzelfall — Worst case
    Wichtig bei sicherheitskritischen Anwendungen
  - Durchschnittlicher Bedarf im Langzeitverhalten — Average case
    Verlangt mathematisch schwierige statistische Analyse
- **Analyse von Problemen**
  - Wie effizient ist die bestmögliche Lösung? — Untere Schranken
  - Wieviel kann durch Hardwaresteigerungen erreicht werden?
  - Welche Verbesserung liefert Parallelität bzw. Nichtdeterminismus
  - Welche Probleme sind gleich schwierig? — Komplexitätsklassen
  - **Gibt es Probleme, die nicht effizient lösbar sind?**

Theoretische Informatik II

Einheit 8.1

Komplexitätsmaße

1. Zeit- und Platzkomplexität
2. Asymptotische Analyse
3. Praktische Konsequenzen

Zeit- und Platzkomplexität

- **Rechenzeit** $t_\tau(w)$ vgl. Kapitel 7.2
  – Anzahl der Elementaroperationen von τ bis Berechnung terminiert
  – Abhängig von konkreter Eingabe w
- **Zeitkomplexität** $time_\tau(n) = \max\{t_\tau(w) \mid |w|=n\}$
  – Maximale Rechenzeit relativ zur Größe n der Eingabe (worst-case)
- **Speicherbedarf** $s_\tau(w)$
  – Anzahl der Bandzellen, die τ während der Berechnung aufsucht
- **Platzkomplexität** $space_\tau(n) = \max\{s_\tau(w) \mid |w|=n\}$
  – Maximaler Speicherbedarf relativ zur Größe n der Eingabe (worst-case)

Analoge Maße für andere Berechnungsmodelle einschließlich nichtdeterministischer Maschinen

Komplexitätsanalyse einer Turing-Maschine

- $\tau_1 = (\{\mathtt{s}_0\}, \{\mathtt{1}\}, \{\mathtt{b},\mathtt{1}\}, \delta_1, \mathtt{s}_0, \mathtt{b})$ mit $\delta_1 =$

s	a	s'	a'	P
$\mathtt{s}_0$	$\mathtt{1}$	$\mathtt{s}_0$	$\mathtt{1}$	$\mathtt{r}$
$\mathtt{s}_0$	$\mathtt{b}$	$\mathtt{s}_0$	$\mathtt{1}$	$\mathtt{h}$

- **Mathematische Analyse**:
  - Anfangskonfiguration: $\alpha(\mathtt{1}^n) = (\mathtt{s}_0, f_n, 0)$, wobei $f_n(j) = \begin{cases} \mathtt{1} & \text{falls } j \in \{0,..,n-1\}, \\ \mathtt{b} & \text{sonst} \end{cases}$
  - Nachfolgekonfigurationen: $\hat{\delta}(\mathtt{s}_0, f_n, j) = \begin{cases} (\mathtt{s}_0, f_n, j+1) & \text{falls } j \in \{0,..,n-1\}, \\ (\mathtt{s}_0, f_{n+1}, n) & \text{falls } j=n \end{cases}$
  - Terminierung: $\min\{j \mid \hat{\delta}^j(\mathtt{s}_0, f_n, 0) = (\mathtt{s}_0, f_n, j) \wedge \delta(\mathtt{s}_0, f_n(j)) = (\mathtt{s}_0, \mathtt{b}, \mathtt{h})\} = n$

$$\Downarrow$$

$$\boxed{t_{\tau_1}(\mathtt{1}^n) = n+1 \quad \textbf{und} \quad time_{\tau_1}(n) = n+1 \quad \textbf{für alle } n}$$

Vereinfachte Komplexitätsabschätzungen

- **Genaue Betrachtungen sind unpraktikabel**
  - Zu mühsam bei nichttrivialen Algorithmen
  - Zu abhängig von Programmierdetails und Maschinenmodell
  - Welches Maschinenmodell sollte der Standard sein?
- **Abschätzung der Komplexität**
  - Nur asymptotisches Verhalten auf großen Problemen ist interessant
  - $\mapsto$ Einheitskostenmodell: Vereinfachte Zählung von Elementaroperationen
  - $\mapsto$ Additive Konstanten werden nicht berücksichtigt
  - $\mapsto$ Konstante Faktoren werden nicht berücksichtigt

$$\Downarrow$$

Analyse des wesentlichen Laufzeitverhaltens/Speicherbedarfs

Asymptotische Analyse

- **Asymptotischer Vergleich von Funktionen**
  - f_2 wächst schneller als f_1, falls $f_1(n) \leq f_2(n)$ für alle $n \in \mathbb{N}$
  - f_2 wächst asymptotisch schneller als f_1, falls es ein $n_0 \in \mathbb{N}$ gibt mit
    $$f_1(n) \leq f_2(n) \text{ für alle } n \geq n_0$$

- **Ordnung $\mathcal{O}(f)$ einer Funktion**
  - $\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists n_0, c.\ \forall n \geq n_0.\ g(n) \leq c * f(n)\}$
  - Alternativ: $\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists k, c.\ \forall n.\ g(n) \leq k + c * f(n)\}$
  - **Gängige Schreibweisen**
    - $g = \mathcal{O}(f)$ bedeutet $g \in \mathcal{O}(f)$, $\mathcal{O}(f_1) = \mathcal{O}(f_2)$ bedeutet $\mathcal{O}(f_1) \subseteq \mathcal{O}(f_2)$
    - $\mathcal{O}(1) \equiv \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \equiv \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \equiv \mathcal{O}(\lambda n.n^2)$, ...

- **Ordnung konkreter Funktionen**
  - Konstante Funktion: $g_1(n) = k$ für alle n — $g_1 \in \mathcal{O}(1)$
  - Polynome: $g_2(n) = c_0 + c_1 * n + .. + c_m * n^m$ — $g_2 \in \mathcal{O}(n^m)$
  - Logarithmenfunktionen: $g_3(n) = log_b n$ — $g_3 \in \mathcal{O}(log_2 n)$
  - Fakultätsfunktion: $g_4(n) = n! = 1 * 2 * .. * n$ — $g_4 \in \mathcal{O}(n^n)$

Komplexität von Algorithmen

- **Asymptotischer Effizienzvergleich**
  - τ_1 ist schneller als τ_2, falls $time_{\tau_1}(n) \leq time_{\tau_2}(n)$ für alle $n \in \mathbb{N}$
  - τ_1 ist asymptotisch schneller als τ_2, falls es ein $n_0 \in \mathbb{N}$ gibt mit $time_{\tau_1}(n) \leq time_{\tau_2}(n)$ für alle $n \geq n_0$
- **Komplexität $\mathcal{O}(f)$**
  - τ hat Zeitkomplexität $\mathcal{O}(f)$, falls $time_\tau \in \mathcal{O}(f)$
  - τ hat Platzkomplexität $\mathcal{O}(f)$, falls $space_\tau \in \mathcal{O}(f)$
- **Komplexitätklassen**
  - τ hat konstante (Zeit-)komplexität, falls $time_\tau \in \mathcal{O}(1)$
  - τ hat logarithmische Komplexität, falls $time_\tau \in \mathcal{O}(log_2 n)$
  - τ hat lineare Komplexität, falls $time_\tau \in \mathcal{O}(n)$
  - τ hat quadratische Komplexität, falls $time_\tau \in \mathcal{O}(n^2)$
  - τ hat kubische Komplexität, falls $time_\tau \in \mathcal{O}(n^3)$
  - τ hat **polynomielle** Komplexität, falls $time_\tau \in \mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$
  - τ hat **exponentielle** Komplexität, falls $time_\tau \in \mathcal{O}(2^{n^k})$ für ein $k \in \mathbb{N}$
  - τ hat superexponentielle Komplexität, falls $time_\tau \in \mathcal{O}(2^{2^{n^k}})$ für ein $k \in \mathbb{N}$

$\vdots$

Analoge Klassen für Platzkomplexität

Wie schnell wächst Rechenzeit mit der Grösse der Eingabe?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	…	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller sind?

	$\log_2$n	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach	31-fach	10-fach	plus 10	plus 6

Komplexitätklassen – Praktische Auswirkungen

- **Große Probleme benötigen polynomielle Lösungen**
  – Exponentielle Algorithmen sind für die Praxis unakzeptabel
  – Auch innerhalb der polynomiellen Komplexität gibt es große Unterschiede
- **Bessere Hardware ist selten eine Lösung**
  – Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig
  – Es lohnt sich, in die Verbesserung von Algorithmen zu investieren
- **Es gibt noch ungeklärte Fragen**
  – Kann Parallelismus signifikante Effizienzsteigerung bewirken?
    · z.B. von exponentieller auf polynomielle Zeit?
  – Was ist der Zusammenhang zwischen Platzbedarf und Laufzeitverhalten
    · Bisher nur grobe Abschätzungen bekannt