

Theorie der Informatik

19. Komplexitätstheorie: Motivation und Einführung

Malte Helmert Gabriele Röger

Universität Basel

11. Mai 2015

Überblick: Vorlesung

Vorlesungsteile

- I. Logik ✓
- II. Automatentheorie und formale Sprachen ✓
- III. Berechenbarkeitstheorie ✓
- IV. **Komplexitätstheorie**

Überblick: Komplexitätstheorie

IV. Komplexitätstheorie

- 19. **Motivation und Einführung**
- 20. P, NP und polynomielle Reduktionen
- 21. Satz von Cook und Levin
- 22. einige NP-vollständige Probleme

Nachlesen

Literatur zu diesem Vorlesungskapitel

Theoretische Informatik - kurz gefasst
von Uwe Schöning (5. Auflage)

- Kapitel 3.1



Motivation

Ein Szenario (1)

Beispielszenario

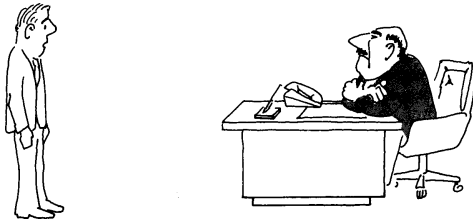
- Sie sind Programmierer(-in) bei einer Logistikfirma.
- Ihr Chef beauftragt Sie, ein Programm zu entwickeln, das die Route eines Lieferwagens Ihrer Firma optimiert:
 - Das Fahrzeug beginnt die Route am Lager Ihrer Firma.
 - Es muss auf seiner Route 50 Stationen anfahren.
 - Sie kennen die Entfernungen zwischen allen relevanten Orten (Stationen und Lager).
 - Ihr Programm soll eine Route vom Lager über alle Stationen zurück zum Lager berechnen, die **so kurz wie möglich ist**.

Ein Szenario (2)

Beispielszenario (Fortsetzung)

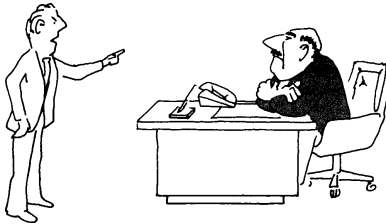
- Sie arbeiten wochenlang an dem Problem, schaffen es aber nicht, ein solches Programm zu entwickeln. Alle ihre Versuche
 - berechnen Routen, die möglicherweise suboptimal sind, oder
 - terminieren nicht in vernünftiger Zeit (sagen wir: innerhalb eines Monats)
- Was sagen Sie Ihrem Chef?

Was Sie nicht sagen wollen



„Ich kann keinen effizienten Algorithmus finden,
weil ich zu dumm dafür bin.“

Was Sie am liebsten sagen würden



„Ich kann keinen effizienten Algorithmus finden,
weil es so einen Algorithmus nicht gibt!“

Was Komplexitätstheorie Ihnen zu sagen erlaubt



„Ich kann keinen effizienten Algorithmus finden,
aber all diese berühmten Informatiker können das auch nicht.“

Warum Komplexitätstheorie?

Komplexitätstheorie

Komplexitätstheorie sagt uns, welche Probleme **schnell** gelöst werden können („einfache Problem“) und welche **nicht** („schwere Probleme“).

- Das ist praktisch nützlich, weil einfache und schwere Probleme **unterschiedliche Techniken** zur Lösung erfordern.
- Wenn wir zeigen können, dass ein Problem schwer ist, brauchen wir nicht unsere Zeit mit der (vergeblichen) Suche nach „einfachen“ Algorithmen zu verschwenden.

Warum Reduktionen?

Reduktionen

Ein wichtiger Teil der Komplexitätstheorie sind (polynomielle) **Reduktionen**, die zeigen, wie ein neues Problem P auf ein bekanntes Problem Q zurückgeführt werden kann.

- nützlich für **theoretische Analyse** von P , weil es uns erlaubt, unser Wissen über Q auf P zu übertragen
- oft auch nützlich für **praktische Algorithmen** für P :
führe P auf Q zurück und verwende dann den besten bekannten Algorithmus für Q

Testen Sie Ihre Intuition! (1)

- Die folgende Folie benennt einige **Graphenprobleme**.
- Die Eingabe ist immer ein **gerichteter Graph** $G = \langle V, E \rangle$.
- **Wie schwierig** sind die Probleme Ihrer Einschätzung nach?
- Sortieren Sie die Probleme von
am einfachsten (= benötigt die geringste Zeit zur Lösung)
zu **am schwierigsten** (benötigt die meiste Zeit zur Lösung)
- **keine Begründungen nötig**, folgen Sie nur Ihrer Intuition!
- **anonym** und **unbewertet**

Testen Sie Ihre Intuition! (2)

- ① Finden Sie einen **einfachen Pfad** (= ohne Zyklus) von $u \in V$ zu $v \in V$ mit **minimaler Länge**.
- ② Finden Sie einen **einfachen Pfad** (= ohne Zyklus) von $u \in V$ zu $v \in V$ mit **maximaler Länge**.
- ③ Bestimmen Sie, ob G **stark zusammenhängend ist** (jeder Knoten ist von jedem erreichbar).
- ④ Finden Sie einen **Zyklus** (nichtleerer Pfad von u nach u für irgendein $u \in V$; mehrfache Besuche von Knoten erlaubt).
- ⑤ Finden Sie einen **Zyklus**, der **alle** Knoten besucht.
- ⑥ Finden Sie einen **Zyklus**, der einen **gegebenen Knoten u** besucht.
- ⑦ Finden Sie einen Pfad, der **alle Knoten besucht**, ohne einen Knoten zu wiederholen.
- ⑧ Finden Sie einen Pfad, der **alle Kanten benutzt**, ohne eine Kante zu wiederholen.

Wie misst man Laufzeit?

Wie misst man Laufzeit?

- **Zeitkomplexität** ist ein Mass, das uns sagt, **wie viel Zeit** die Lösung eines Problems benötigt.
- Wie können wir so ein Mass angemessen **definieren**?

Wie misst man Laufzeit?

- **Zeitkomplexität** ist ein Mass, das uns sagt, **wie viel Zeit** die Lösung eines Problems benötigt.
- Wie können wir so ein Mass angemessen **definieren**?

Beispielhafte Aussagen über Laufzeit:

- „Der Aufruf `sort /usr/share/dict/words` auf dem Computer `kibo` benötigt 0.105 Sekunden.“
- „Bei einer Eingabedatei der Grösse 1 MB benötigt `sort` auf einem modernen Computer höchstens 1 Sekunde.“
- „Quicksort ist schneller als Sortieren durch Einfügen.“
- „Sortieren durch Einfügen ist langsam.“

↪ sehr unterschiedliche Aussagen
mit verschiedenen **Vor- und Nachteilen**.

Präzise Aussagen vs. allgemeine Aussagen

Beispielaussage über Laufzeit

„Der Aufruf `sort /usr/share/dict/words`
auf dem Computer `kibo` benötigt 0.105 Sekunden.“

Vorteil: sehr **präzise**

Nachteil: nicht **allgemein**

- **eingabespezifisch:**
Was ist, wenn wir andere Dateien sortieren wollen?
- **maschinenspezifisch:**
Was passiert auf einem anderen Computer?
- sogar **situationsspezifisch:**
Erhalten wir morgen dasselbe Ergebnis wie heute?

Allgemeine Aussagen über Laufzeit

In dieser Vorlesung wollen wir **allgemeine** Aussagen über Laufzeit treffen. Dies erreichen wir auf drei Weisen:

Allgemeine Aussagen über Laufzeit

In dieser Vorlesung wollen wir **allgemeine** Aussagen über Laufzeit treffen. Dies erreichen wir auf drei Weisen:

1. Allgemeine Eingaben

Statt **konkreter** Eingaben sprechen wir über **allgemeine Arten** von Eingaben:

- **Beispiel:** Laufzeit zum Sortieren einer Eingabe der Grösse n im **schlechtesten Fall**
- **Beispiel:** Laufzeit zum Sortieren einer Eingabe der Grösse n im **durchschnittlichen Fall**

hier: Laufzeit für Eingabegrösse n im **schlechtesten Fall**

Allgemeine Aussagen über Laufzeit

In dieser Vorlesung wollen wir **allgemeine** Aussagen über Laufzeit treffen. Dies erreichen wir auf drei Weisen:

2. Ignorieren von Details

Statt **exakter Formeln** für die Laufzeit geben wir die **Größenordnung** an:

- **Beispiel:** statt zu sagen, dass wir Zeit $\lceil 1.2n \log n \rceil - 4n + 100$ benötigen, sagen wir, dass wir Zeit **$O(n \log n)$** benötigen.
- **Beispiel:** statt zu sagen, dass wir Zeit $O(n \log n)$, $O(n^2)$ oder $O(n^4)$ benötigen, sagen wir, dass wir **polynomiell viel** Zeit benötigen.

hier: Was kann in **polynomieller Zeit** berechnet werden?

Allgemeine Aussagen über Laufzeit

In dieser Vorlesung wollen wir **allgemeine** Aussagen über Laufzeit treffen. Dies erreichen wir auf drei Weisen:

3. Abstrakte Kostenmasse

Statt der **Laufzeit auf einem konkreten Computer** betrachten wir **abstraktere** Kostenmasse:

- **Beispiel:** zähle die ausgeführten **Maschinencode-Anweisungen**
- **Beispiel:** zähle die ausgeführten **Java-Bytecode-Anweisungen**
- **Beispiel:** zähle bei Sortialgorithmen die **Elementvergleiche**

hier: zähle Berechnungsschritte einer **Turingmaschine**
(**polynomiell äquivalent** zu anderen Massen)

Entscheidungsprobleme

Entscheidungsprobleme

- Wie zuvor vereinfachen wir unsere Untersuchung, indem wir uns auf **Entscheidungsprobleme** beschränken.
- Komplexere Berechnungsprobleme können auf geeignet definierte Entscheidungsprobleme zurückgeführt werden.
- Formal sind Entscheidungsprobleme (wie zuvor) als **Sprachen** definiert, aber wir verwenden ab jetzt wo möglich eine informelle „**Gegeben**“-„**Gefragt**“-Notation.

Beispiel: Entscheidungs- vs. Berechnungsproblem (1)

Definition (Hamiltonkreis)

Sei $G = \langle V, E \rangle$ ein (gerichteter oder ungerichteter) Graph.

Ein **Hamiltonkreis** in G ist eine Folge von Knoten

$\pi = v_0, \dots, v_n \in V$ mit folgenden Eigenschaften:

- π ist ein **Pfad**: für alle $0 \leq i < n$ führt eine Kante von v_i nach v_{i+1}
- π ist ein **Kreis**: $v_0 = v_n$
- π ist **einfach**: $v_i \neq v_j$ für alle $i \neq j$ mit $i, j < n$
- π ist **hamiltonsch**: alle Knoten von V sind in π enthalten

Beispiel: Entscheidungs- vs. Berechnungsproblem (2)

Beispiel (Hamiltonkreise in gerichteten Graphen)

P: Berechnungsproblem DIRHAMILTONCYCLEGEN

- **Eingabe:** gerichteter Graph $G = \langle V, E \rangle$
- **Ausgabe:** ein Hamiltonkreis in G
bzw. Meldung, dass keiner existiert

E: Entscheidungsproblem DIRHAMILTONCYCLE

- **Gegeben:** gerichteter Graph $G = \langle V, E \rangle$
- **Frage:** Enthält G einen Hamiltonkreis?

Diese Probleme sind **polynomiell äquivalent**:

aus einem polynomiellen Algorithmus für das eine Problem kann man einen polynomiellen Algorithmus für das jeweils andere Problem konstruieren. (**Ohne Beweis.**)

Algorithmen für Entscheidungsprobleme

Algorithmen für Entscheidungsprobleme:

- Wo möglich, geben wir Algorithmen für Entscheidungsprobleme in **Pseudo-Code** an (analog zu WHILE-Programmen).
- Da es nur um Ja-/Nein-Fragen geht, müssen wir keinen Ergebniswert berechnen.
- Stattdessen verwenden wir Anweisungen **ACCEPT**, um die gegebene Eingabe zu **akzeptieren** („Ja“-Antwort) und **REJECT**, um sie **abzulehnen** („Nein“-Antwort).
- Wo wir formaler sein müssen, verwenden wir **Turingmaschinen** und den Akzeptanzbegriff aus Kapitel 12.

Nichtdeterminismus

Nichtdeterminismus

- Um die Komplexitätstheorie zu entwickeln, benötigen wir das algorithmische Konzept des **Nichtdeterminismus**.
- Schon bekannt für **Turingmaschinen** ($\rightsquigarrow$ **Kapitel 12**):
 - Eine NTM kann **mehrere mögliche Nachfolgekongfigurationen** für eine gegebene Konfiguration haben.
 - Eingabe x wird akzeptiert, wenn es **mindestens einen möglichen Berechnungsweg** gibt, der in einen Endzustand führt.
- Hier führen wir analog dazu Nichtdeterminismus für Pseudo-Code (bzw. WHILE-Programmen) ein.

Nichtdeterministische Algorithmen

Nichtdeterministische Algorithmen:

- Alle Bestandteile deterministischer Algorithmen sind auch in nichtdeterministischen Algorithmen erlaubt: **IF**, **WHILE**, etc.
- Zusätzlich gibt es eine **nichtdeterministische Zuweisung**:

GUESS $x_i \in \{0, 1\}$

Hierbei ist x_i eine Programmvariable.

Nichtdeterministische Algorithmen: Akzeptanz

- Bedeutung von **GUESS** $x_i \in \{0, 1\}$:
 x_i wird **entweder** der Wert **0** **oder** der Wert **1** zugewiesen.
- Damit ist nicht mehr eindeutig festgelegt, wie sich das Programm für eine gegebene Eingabe verhält: es gibt mehrere Möglichkeiten, was passieren kann.
- Das Programm akzeptiert eine gegebene Eingabe, wenn **mindestens ein möglicher Ausführungspfad** existiert, der zu einer **ACCEPT**-Anweisung führt.
- Ansonsten wird die Eingabe abgelehnt.

Anmerkung: **Asymmetrie** zwischen Akzeptieren und Ablehnen!
(vgl. Semi-Entscheidbarkeit)

Komplexere GUESS-Anweisungen

- Wir verwenden im Folgenden im Pseudo-Code auch Anweisungen wie

GUESS $x \in \{1, 2, \dots, n\}$

oder allgemeiner

GUESS $x \in S$

für eine Menge S .

- Solche Anweisungen sind Kurzschreibweisen und lassen sich in $\lceil \log_2 n \rceil$ bzw. $\lceil \log_2 |S| \rceil$ „atomare“ **GUESS**-Anweisungen zerlegen.

Beispiel: nichtdeterministischer Algorithmus (1)

Beispiel (DIRHAMILTONCYCLE)

Eingabe: gerichteter Graph $G = \langle V, E \rangle$

$start :=$ ein beliebiger Knoten aus V

$current := start$

$remaining := V \setminus \{start\}$

WHILE $remaining \neq \emptyset$:

GUESS $next \in remaining$

IF $\langle current, next \rangle \notin E$:

REJECT

$remaining := remaining \setminus \{next\}$

$current := next$

IF $\langle current, start \rangle \in E$:

ACCEPT

Beispiel: nichtdeterministischer Algorithmus (2)

- bei geeigneten Datenstrukturen löst der Algorithmus das Problem in $O(n \log n)$ Programmschritten, wobei $n = |V| + |E|$ die Eingabegrösse ist.
- Wie viele Schritte würde ein **deterministischer** Algorithmus benötigen?

Raten und Prüfen

- Das DIRHAMILTONCYCLE-Beispiel illustriert ein allgemeines Design-Prinzip für nichtdeterministische Algorithmen:
Raten und Prüfen („guess and check“)
- Im Allgemeinen können nichtdeterministische Algorithmen ein Problem lösen, indem sie zuerst eine „Lösung“ raten und dann überprüfen, dass diese tatsächlich eine Lösung ist.
- Wenn Lösungen zu einem Problem **effizient überprüfbar** sind, dann kann das Problem auch **effizient gelöst** werden, sofern Nichtdeterminismus verwendet werden darf.

Die Macht des Nichtdeterminismus

- Nichtdeterministische Algorithmen sind sehr mächtig, da sie den „richtigen“ Berechnungsschritt „raten“ können.
- Oder anders interpretiert: sie durchlaufen viele mögliche Berechnungen „parallel“, und es reicht, wenn **eine** davon erfolgreich ist.
- Können sie Probleme effizient (in polynomieller Zeit) lösen, die deterministische Algorithmen **nicht** effizient lösen können?
- **Das ist die grosse Frage!**

Zusammenfassung

Zusammenfassung (1)

- **Komplexitätstheorie** befasst sich (unter anderem) damit, welche Probleme **schnell** gelöst werden können und welche nicht.
- **hier**: Fokus auf der Frage, was **in polynomieller Zeit** berechnet werden kann
- Wir verwenden zur Formalisierung Turingmaschinen, aber andere Berechnungsformalismen sind **polynomiell äquivalent**.
- Wir betrachten **Entscheidungsprobleme**, aber die Ergebnisse sind direkt auf allgemeine Berechnungsprobleme übertragbar.

Zusammenfassung (2)

wichtiges Konzept: **Nichtdeterminismus**:

- **Nichtdeterministische Algorithmen** können „raten“, d.h. mehrere Berechnungen „gleichzeitig“ durchführen.
- Eine Eingabe erhält eine „Ja“-Antwort, wenn **mindestens ein möglicher Berechnungsweg** existiert, der zur Akzeptanz führt.
- in NTMs: durch **nichtdeterministische Übergänge** ($\delta(z, a)$ enthält mehrere Elemente)
- in Pseudo-Code: durch **GUESS**-Anweisungen