Theorie der Informatik

IV.1. Komplexitätstheorie: Motivation und Einführung

Malte Helmert Christian Tschudin

Universität Basel

29. April 2013

Überblick: Vorlesung

Vorlesungsteile

I. ~~Logik~~
II. ~~Automatentheorie und formale Sprachen~~
III. ~~Berechenbarkeitstheorie~~
IV. Komplexitätstheorie

Überblick: Vorlesung

Vorlesungsteile

I. ~~Logik~~

II. ~~Automatentheorie und formale Sprachen~~

III. ~~Berechenbarkeitstheorie~~

IV. Komplexitätstheorie $\longleftarrow$ Sie sind hier!

Motivation

Ein Szenario (1)

Beispielszenario

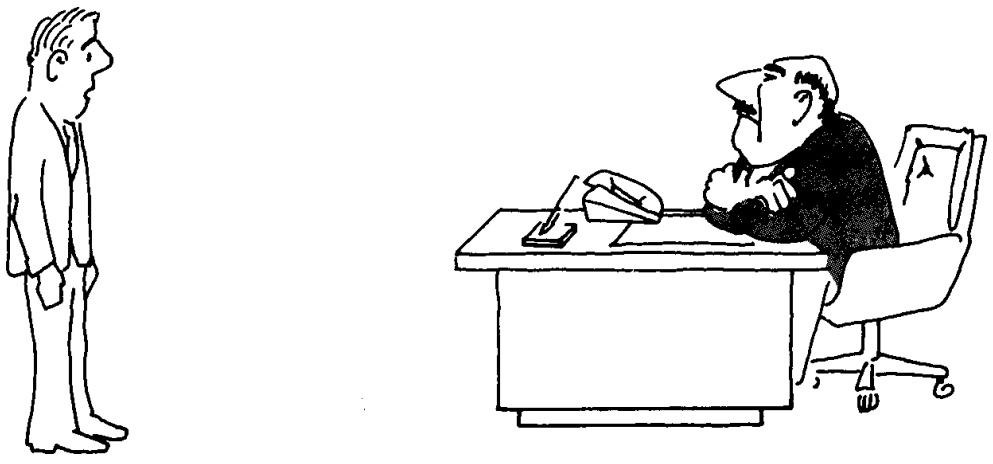
- Sie sind Programmierer(-in) bei einer Logistikfirma.
- Ihr Chef beauftragt Sie, ein Programm zu entwickeln, das die Route eines Lieferwagens Ihrer Firma optimiert:
  - Das Fahrzeug beginnt die Route am Lager Ihrer Firma.
  - Es muss auf seiner Route 50 Stationen anfahren.
  - Sie kennen die Entfernungen zwischen allen relevanten Orten (Stationen und Depot).
  - 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?

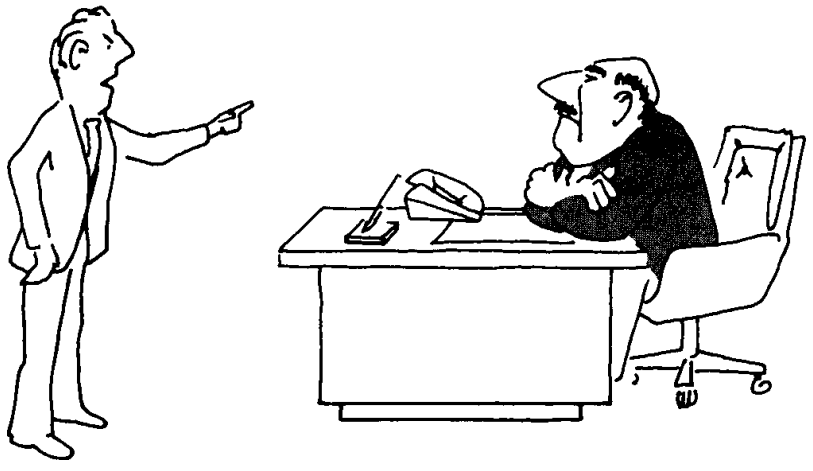
Was Sie nicht sagen wollen



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

Quelle: M. Garey & D. Johnson, Computers and Intractability, Freeman 1979, S. 2

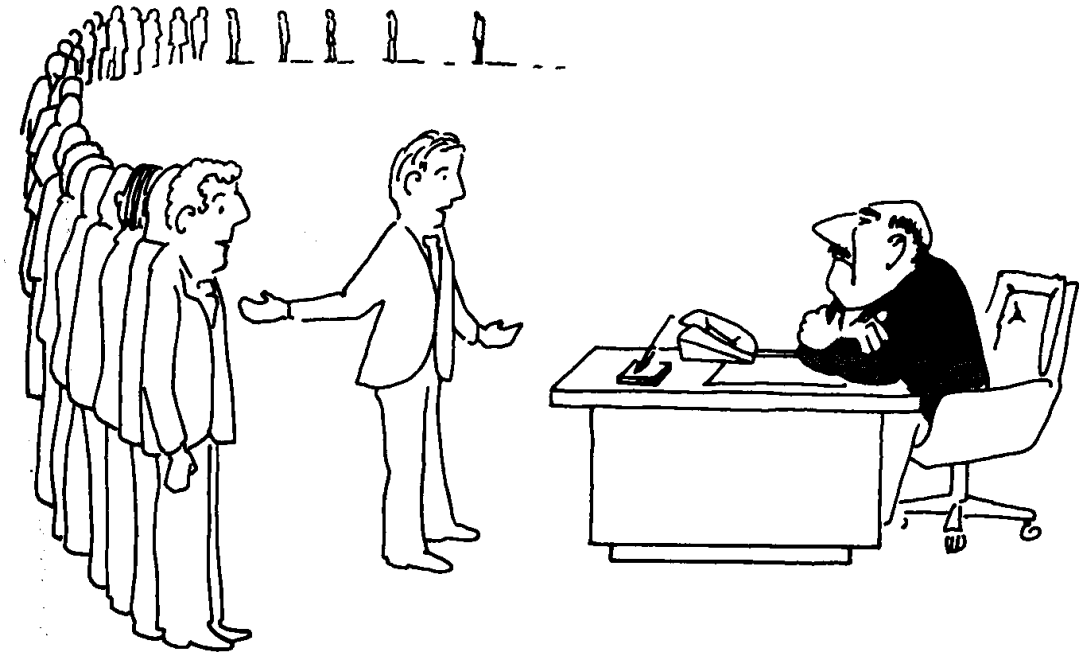
Was Sie am liebsten sagen würden



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

Quelle: M. Garey & D. Johnson, Computers and Intractability, Freeman 1979, S. 2

Was Komplexitätstheorie Ihnen zu sagen erlaubt



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

Quelle: M. Garey & D. Johnson, Computers and Intractability, Freeman 1979, S. 3

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!

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 gegeben 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, dass 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, dass 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.“

$\rightsquigarrow$ 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

Allgemeine Aussagen über Laufzeit

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

2. 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 Sortieralgorithmen die Elementvergleiche

Allgemeine Aussagen über Laufzeit

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

3. Ignorieren von Details

Statt exakter Formeln für die Laufzeit geben wir die Grössenordnung an:

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

Welches Berechnungsmodell ist angemessen?

Welches Berechnungsmodell verwenden wir?

Wir kennen viele Berechnungsmodelle:

- Programme in einer Programmiersprache
  - z. B. Java, C++, Python, OCaml, Haskell, ...
- endliche Automaten
  - Varianten: deterministisch oder nichtdeterministisch
- Kellerautomaten
- Turingmaschinen
  - Varianten: unterschiedliche Bandalphabete
  - Varianten: ein Band oder mehrere Bänder
  - Varianten (neu): deterministisch oder nichtdeterministisch
- WHILE-, LOOP- und GOTO-Programme

Berechnungsmodell: Turingmaschinen

Unsere Wahl: Turingmaschinen

Wir verwenden Turingmaschinen als Berechnungsmodell, weil sie

- zu den mächtigsten Berechnungsmodellen in unserer Liste gehören und
- einfach genug strukturiert sind, um sie theoretisch untersuchen zu können.

Programmiersprachen sind genauso mächtig, aber oft

- nicht vollständig präzise definiert und
- für viele theoretische Untersuchungen zu kompliziert.

Sind Turingmaschinen ein adäquates Modell?

- Nach der Church-Turing-These kann alles, was berechnet werden kann, mit einer Turingmaschine berechnet werden.
- Aber: viele auf echten Computern einfache Operationen benötigen auf Turingmaschinen viele Einzelschritte.

$\rightsquigarrow$ Laufzeit auf Turingmaschinen ist nicht unbedingt aussagekräftig für Laufzeit auf einem Computer!

Sind Turingmaschinen ein adäquates Modell?

- Nach der Church-Turing-These kann alles, was berechnet werden kann, mit einer Turingmaschine berechnet werden.
- Aber: viele auf echten Computern einfache Operationen benötigen auf Turingmaschinen viele Einzelschritte.

$\rightsquigarrow$ Laufzeit auf Turingmaschinen ist nicht unbedingt aussagekräftig für Laufzeit auf einem Computer!

- Die Haupteinschränkung von Turingmaschinen ist, dass sie keinen wahlfreien Zugriff (random access) erlauben.
- Alternative formale Berechnungsmodelle existieren:
  - Beispiele: Lambda-Kalkül, Registermaschinen, Random-Access-Maschinen (RAMs), ...
- Manche davon sind näher an heutigen Computern (insbesondere RAMs).

Turingmaschinen sind ausreichend adäquat

- TMs sind also nicht das genaueste Modell für echte Computer.
- Aber: alles, was ein „realistisches“ Berechnungsmodell in n Schritten berechnen kann, kann eine TM in polynomieller Zeit in n (z. B. in $O(n^2)$ Schritten) berechnen.
- Für die zentrale Fragestellung, die wir untersuchen wollen, die P-vs.-NP-Frage, ist uns polynomieller Zusatzaufwand egal.
- Daher sind für unsere Zwecke TMs ein geeignetes Modell, und sie haben den Vorteil, einfach analysierbar zu sein.
- Daher verwenden wir im Folgenden TMs als Berechnungsmodell.

Für detailliertere Fragestellungen (z. B. lineare vs. quadratische Zeit) sollte man ein anderes Berechnungsmodell wählen.

Welche Variante von Turingmaschinen verwenden wir? (1)

Es gibt viele Varianten von Turingmaschinen:

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Welche Variante sollen wir verwenden?

Welche Variante von Turingmaschinen verwenden wir? (2)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Welche Variante von Turingmaschinen verwenden wir? (2)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

TMs mit mehreren Bändern können in Zeit $O(n^2)$
durch Ein-Band-TMs simuliert werden.
$\rightsquigarrow$ Wir verwenden die einfacheren Ein-Band-TMs.

Welche Variante von Turingmaschinen verwenden wir? (3)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Welche Variante von Turingmaschinen verwenden wir? (3)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Für die volle Mächtigkeit ist ein unendliches Band nötig.

Ob einseitig oder beidseitig unendlich, ist dabei egal.
(Eine TM mit einseitig unendlichem Band kann n Schritte einer TM mit beidseitig unendlichem Band in $O(n)$ Schritten simulieren.)

Wir wählen ein beidseitig unendliches Band,
um den Bandrand nicht als Sonderfall behandeln zu müssen.

Welche Variante von Turingmaschinen verwenden wir? (4)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Welche Variante von Turingmaschinen verwenden wir? (4)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Für kompakte (= effiziente) Kodierung ist mindestens ein zweielementiges Eingabealphabet nötig.

Mehr als zwei Elemente sind nicht nötig: ein Alphabet der Grösse k kann effizient durch $\lceil \log_2 k \rceil$ binäre Symbole kodiert werden.

$\rightsquigarrow$ Wähle immer Eingabealphabet $\Sigma = \{0, 1\}$

$\rightsquigarrow$ Bandalphabet muss Blank beinhalten: $\Gamma = \{0, 1, \square\}$ reicht aus und kann grössere Bandalphabete effizient simulieren.

Welche Variante von Turingmaschinen verwenden wir? (5)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Welche Variante von Turingmaschinen verwenden wir? (5)

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Wir haben bisher nur deterministische Turingmaschinen eingeführt.

Für die Entwicklung unserer Ergebnisse sind auch nichtdeterministische Turingmaschinen wichtig.

$\rightsquigarrow$ Wir werden beide Arten von TMs (deterministisch und nichtdeterministisch) verwenden.

Welche Variante von Turingmaschinen verwenden wir?

- ein Band oder mehrere Bänder
- Band endlich, einseitig unendlich oder beidseitig unendlich
- Bandalphabet der Grösse 2, 3, 4, ...
- deterministisch oder nichtdeterministisch

Ausblick

Überblick über diesen Vorlesungsteil

Überblick über diesen Vorlesungsteil:

IV. Komplexitätstheorie

IV.1. Motivation und Einführung

IV.2. Nichtdeterminismus

IV.3. P, NP und polynomielle Reduktionen

IV.4. Satz von Cook und Levin

IV.5. einige NP-vollständige Probleme