Komplexitätstheorie

Vorlesung im Wintersemester 2011

Organisatorisches

- Zeit und Ort:

  Di 10-12 MZH 1090 und Do 16-18 MZH 1100

- Vortragender:

  Prof. Carsten Lutz
  Raum 3090
  Tel. (218)-64431
  clu@uni-bremen.de

- Position im Curriculum:

  Wahlbereich Master-Ergänzung, Modulbereich Theorie,
  Vertiefungs VL

Organisatorisches

- Voraussetzungen:
  Grundvorlesung Theoretische Informatik

- Form: K4, voraussichtlich 7 Termine mit Übungen
  (aber Diskussion in VL jederzeit erwünscht!)

- Vorlesungsmaterial:

  Folien und Aufgabenblätter auf:

  http://www.informatik.uni-bremen.de/tdki/lehre/ws11/kt/

  Beispiele, Beweise, etc an der Tafel (mitschreiben!)

Literatur

- Sanjeev Arora, Boaz Barak. Computational Complexity: A Modern Approach. Cambridge University Press, 2009.
- Oded Goldreich. Computational Complexity: a Conceptual Perspective. Cambridge University Press, 2008.
- Christos H. Papadimitriou. Computational Complexity. Addison-Wesley, 1994.
- Ingo Wegener. Komplexitätstheorie - Grenzen der Effizienz von Algorithmen. Springer, 2003.
- Dexter Kozen. Theory of Computation. Springer, 2006.
- The Parallel Computation Project: http://cs.armstrong.edu/greenlaw/research/PARALLEL/index.html

Prüfungen

Übungen:

- Übungsaufgaben jede zweite Woche (mit Zusatzaufgaben)
- Werden in Gruppen (2-3 Personen) bearbeitet, abgegeben und korrigiert
- Jeder Teilnehmer muss mindestens eine Aufgabe in der Übungsgruppe vorrechnen

oder

Mündliche Prüfung

Komplexitätstheorie

Vorlesung im Sommersemester 2009

Ziel der Komplexitätstheorie

Analyse der inhärenten Komplexität von Problemen:

- Komplexität: hauptsächlich Laufzeit, aber auch andere Ressourcen (insbesondere Platzbedarf)
- Inhärent: Laufzeit / Ressourcenverbrauch des bestmöglichen Algorithmus für gegebenes Problem.
- Zwei Sichtweisen:
  - ⋆ Problemzentriert: welche Ressourcen brauche ich mindestens, um ein gegebenes Problem zu lösen?
  - ⋆ Ressourcenzentriert: welche Probleme kann ich mit gegebenen Ressourcen lösen

Also: was ist die Grenze der Berechenbarkeit unter beschränkten Ressourcen?

Beispiel: Erreichbarkeit in Graphen

Definition Pfad, Erreichbar

Sei $G = (V, E)$ gerichteter Graph. *Pfad* in G ist Folge von Knoten $v_0, \ldots, v_n$ so dass $(v_i, v_{i+1}) \in E$ für alle $i < n$.
Seien $v, v' \in V$. Dann ist v' *erreichbar* von v in G wenn es Pfad $v_0, \ldots, v_n$ gibt mit $v = v_0$ und $v_n = v'$.

Erreichbarkeitsproblem:

gegeben G, v, v', entscheide ob v' erreichbar von v in G.

Zentrales Problem der Informatik, z.B. Erreichbarkeit von Rechnern in Netzwerken.

Beispiel: Erreichbarkeit in Graphen

Algorithmus 1

Zähle alle Folgen von Knoten $v_0, \ldots, v_n$ auf mit $n \leq |V|$.
Überprüfe für jede Folge, ob

- $v = v_0$ und $v_n = v'$
- $(v_i, v_{i+1}) \in E$ für alle $i < n$.

Antworte “erreichbar”, wenn dies für mind. eine Folge zutrifft.
Sonst antworte “unerreichbar”.

Korrektheit:

- Wenn Algorithmus “erreichbar” antwortet, so ist dies offens. korrekt;
- Wenn v' von v erreichbar, so gibt es Pfad von v zu v', auf dem kein Knoten doppelt vorkommt
  Dieser Pfad hat max. Länge $|V|$ und wird vom Algorithmus gefunden.

Beispiel: Erreichbarkeit in Graphen

Algorithmus 1

Zähle alle Folgen von Knoten $v_0, \ldots, v_n$ auf mit $n \leq |V|$.
Überprüfe für jede Folge, ob

- $v = v_0$ und $v_n = v'$
- $(v_i, v_{i+1}) \in E$ für alle $i < n$.

Antworte “erreichbar”, wenn dies für mind. eine Folge zutrifft.
Sonst antworte “unerreichbar”.

Wenn $|V| = n$, so untersucht der Algorithmus $n^n \leq 2^{n^2}$ Pfade

Für jeden Pfad kann die angegebene Bedingung in Zeit $p(n)$ überprüft werden, mit $p(\cdot)$ Polynom

Insgesamt braucht der Algorithmus also $p(n) \cdot 2^{n^2}$ Zeit: exponentiell!

Beispiel: Erreichbarkeit in Graphen

Natürliche Frage:
Ist das Problem inhärent schwer oder der Algorithmus schlecht?

Algorithmus 2

$S := \{v\}$
markiere v
while $S \neq \emptyset$ do
    wähle $u \in S$
    $S := S \setminus \{u\}$
    for all $(u, u') \in E$ do
        if u' nicht markiert then
            markiere u'
            $S := S \cup \{u'\}$
        endif
    endfor
endwhile
wenn v' markiert antworte “erreichbar”, sonst “unerreichbar”

Beispiel: Erreichbarkeit in Graphen

Es ist nicht schwer, zu zeigen, dass Algorithmus 2 korrekt ist (Übung)

Laufzeitanalyse:

- Jeder Knoten wird höchstens einmal zu S hinzugefügt (beim Markieren)
- Die while Schleife macht also höchstens $n = |V|$ Schritte.
- Da jeder Knoten höchstens n Nachbarn hat, macht die “for all” Schleife höchstens n Schritte
- Insgesamt $c \cdot (n^2)$ Schritte: polynomiell!

Algorithmus 1 war also tatsächlich suboptimal!!

Weiter optimierter Algorithmus hat sogar lineare Laufzeit ($c \cdot n$ Schritte)

Beispiel: Cliquenproblem

Definition Clique

Sei $G = (V, E)$ ein ungerichteter Graph. *Clique* in G ist nicht-leere Knotenmenge $C \subseteq V$ so dass $\{v, v'\} \in E$ für all $v, v' \in C$.
Die *Größe* einer Clique ist die Anzahl der darin enthaltenen Knoten.

Cliquenproblem:

gegeben G, Zahl k, entscheide ob G Clique der Größe k hat.

Beispiel: Cliquenproblem

Algorithmus

Zähle alle Teilmengen $C \subseteq V$ der Kardinalität k auf.
Für jede solche Teilmenge prüfe, ob sie Clique ist.
Wenn eine Clique gefunden wurde, antworte “ja”
Sonst antworte “nein”.

Der Algorithmus ist offensichtlich korrekt.

Laufzeit:

- Es kann in polynomieller Zeit geprüft werden, ob gegebenes $C \subseteq V$ eine k-Clique ist.
- Wenn $|V| = n$, dann gibt es $\binom{n}{k}$ Teilmengen der Größe k;
- Wenn z.B. $k = n/2$, dann $\binom{n}{k} \geq 2^n$, also exponentieller Zeitbedarf!

Beispiel: Cliquenproblem

Natürliche Frage:
Ist das Problem inhärent schwer oder der Algorithmus schlecht?

Beachte: der Algorithmus hat viel Ähnlichkeit mit Algorithmus 1 für Erreichbarkeit!

Antwort:

1. Das ist unbekannt (wie für viele natürliche Probleme)

2. Die Komplexitätstheorie erlaubt es uns trotzdem, das Cliquen-problem als "schwierig" zu identifizieren (später genaueres!)

Wir brauchen also keine Zeit damit zu verschwenden, nach einem besseren Algorithmus zu suchen!

Beispiel: Cliquenproblem

Diese Antwort ist typisch für die Komplexitätstheorie:

- es gibt in diesem Bereich sehr viele schwierige und ungelöste Probleme, darunter die bekanntesten in der Informatik
- insbesondere gibt es nur wenig Aussagen über die nicht-Machbarkeit (z.B. "das Cliquenproblem kann nicht in polynomieller Zeit gelöst werden")
- Stattdessen konzentriert man sich auf relative Aussagen: "Problem X ist genauso schwer wie Problem Y"
- Auf diese Weise ergibt sich eine reiche Struktur im Raum aller Probleme
- Dies gibt einem ein sehr gutes Werkzeug in die Hand, um die Schwierigkeit eines gegebenen Problems einzuschätzen (sehr wichtig auch in der Praxis!)

Kapitel 1

Vier Klarstellungen:

- Was ist eine Laufzeitanalyse?
- Welche Rolle spielt die Repräsentation der Eingabe?
- Welche Art von Komplexität werden wir studieren?
- Was für Probleme werden wir studieren?

Laufzeitanalyse

Laufzeit ist Funktion:
Anzahl Schritte in Abhängigkeit von Größe der Eingabe

Beispiel Cliquenproblem mit k binär kodiert:

- Eingabe mit $|V| = 105$, $k = 1$ hat Größe 106:
  Algorithmus untersucht 105 Teilmengen
- Eingabe mit $|V| = 100$, $k = 50$ hat Größe 106:
  Algorithmus untersucht $>$100.000.000.000.000.000.000.000.000.000 Teilmengen
- Eingabe mit $|V| = 100$, $k = 100$ hat Größe 106:
  Algorithmus untersucht 1 Teilmenge

Wir betrachten die maximale Anzahl Schritte für alle Eingaben derselben Größe.

Laufzeitanalyse

- Beim Beschreiben der Funktion abstrahieren wir häufig stark:
  - ⋆ Konstanten werden ignoriert ($\mathcal{O}$-notation)
  - ⋆ Oft ignorieren wir sogar den Grad von Polynomen
- Insbesondere interessieren wir uns für polynomielle vs. exponentielle Laufzeit

Eingabegröße	1	2	3	4	5	6	7	8		20	128
n^2	1	4	9	16	25	36	49	64		400	16.384
2^n	2	4	8	16	32	64	128	256		1.048.576	RIESIG!

"riesig": Moderner Prozessor braucht mehr Zeit als vom Anfang des Universums bis heute

Kapitel 1

Repräsentation der Eingabe

Repräsentation

Eingabe kann i.d.R. verschieden repräsentiert werden:

- Graphen: z.B. Liste von Knoten +
  entweder Adjazenzmatrix oder Liste von Kanten
- Zahlen: z.B. unär/binär/dezimal kodiert.

Größe kann von Repräsentation abhängen:

- Zahl n: unär Größe n, binär Größe $\log(n)$, jede andere Basis $\frac{1}{c} \cdot \log(n)$
- Kantenloser Graph mit n Knoten: Adjazenzmatrix hat Größe n^2,
  Kantenliste hat Größe 0

Repräsentation

Also hängt auch die Laufzeit von Repräsentation ab, z.B.:

- Man kann einfach in polynomieller Zeit entscheiden, ob Zahl prim ist, wenn diese unär kodiert ist.
- Für binäre Kodierung konnte dies erst 2002 von Agrawal, Kayal und Saxena gezeigt werden.

Meist unterscheidet sich Größe versch. Repräsentationen nur polynomiell, z.B. Adjazenzmatrix vs. Kantenliste

Dann werden wir die Repräsentation meist nicht im Detail fixieren

Ausnahme: Zahlen (wir nehmen stets binäre Kodierung an!)

Graphen: wir betrachten Anzahl Knoten als Eingabegröße

Kapitel 1

Welche Art von Komplexität studieren wir?

Worst Case Komplexität

Laufzeit

- ist obere Schranke für alle Eingaben derselben Größe
- bezieht sich also auf den schlimmstmöglichen Fall

Man spricht von Worst Case Komplexität

Worst Case Komplexität

"Wir brauchen also keine Zeit damit zu verschwenden, nach einem besseren Algorithmus zu suchen"

heißt also:

- wir finden (wahrscheinlich) keinen Algorithmus, der auf jeder Eingabe besser ist.
- wir können aber durchaus Algorithmen finden, die auf vielen Eingaben besser sind

Dazu verwendet man Heuristiken, z.B.:

vor dem Suchen einer Clique der Größe k, eliminiere alle Knoten mit weniger als k Nachbarn

verbessert Laufzeit auf Graphen, in denen die meisten Knoten wenig Nachfolger haben (viel weniger Teilmengen zu untersuchen!).

Worst Case Komplexität

Auch möglich ist die Untersuchung der Laufzeit auf typischen Eingaben (average case Komplexität).

Dies ist jedoch i.d.R. schwieriger:

- man braucht eine gute Charakterisierung von typischen Eingaben (oft schwer zu bestimmen)
- typische Eingaben werden oft über Wahrscheinlichkeitsverteilungen beschrieben (technisch anspruchsvoll)

In der klassischen Komplexitätstheorie und dieser VL geht es:

- um worst case Komplexität (vorsichtiger Ansatz!)
- nicht um Heuristiken / average case Komplexität

Kapitel 1

Was für Probleme werden wir studieren?

Varianten algorithmischer Problem

Algorithmische Probleme haben meist (mindestens) drei Varianten.

Am Beispiel des Cliquenproblems:

(a) Entscheidungsproblem (wie vorher)
Gegeben G und k, entscheide ob G eine k-Clique hat

(b) Berechnungsproblem
Gegeben G und k, berechne eine k-Clique in G
(gib $\perp$ aus, wenn keine existiert)

(c) Optimierungsproblem
Gegeben G, berechne eine Clique in G von maximaler Größe

Welche Variante die natürlichste ist, hängt vom Problem ab.

Beim Cliquenproblem scheinen (b) und (c) etwas befriedigender als (a)

Varianten algorithmischer Probleme

Definition (Single-Prozessor) Sequenzierung

Sei A eine Menge von *Aufgaben*, die durch $\prec$ partiell geordnet ist und $d : A \to \mathbb{N}$ Funktion, die Deadlines beschreibt.

Sequenzierung für A mit $k \in \mathbb{N}$ Verspätungen ist injektive Abbildung $\tau : A \to \{0, \ldots, |A| - 1\}$ so dass

1. $a \prec a'$ impliziert $\tau(a) < \tau(a')$;
2. für höchstens k Aufgaben $a \in A$ gilt $\tau(a) > d(a)$.

Intuition:

- wir nehmen vereinfachend an, dass jede Aufgabe dieselbe Verarbeitungszeit braucht
- $a \prec a'$ bedeutet, dass a' das Ergebnis von a verwendet

Varianten algorithmischer Probleme

(a) Entscheidungsproblem
Gegeben A (inkl. $\prec$ und d) und k, entscheide ob es Sequenzierung für A mit k Verspätungen gibt.

(b) Berechnungsproblem
Gegeben A (inkl. $\prec$ und d) und k, berechne Sequenzierung für A mit k Verspätungen (gib $\perp$ aus, wenn keine existiert).

(c) Optimierungsproblem
Gegeben A (inkl. $\prec$ und d), berechne Sequenzierung für A mit k Verspätungen, k minimal.

Sequenzierung ist typisches Optimierungsproblem:

(c) ist natürlichste Variante

Varianten algorithmischer Probleme

Komplexitätstheorie konzentriert sich auf Entscheidungsprobleme:

- einfacher zu handhaben (z.B. keine Optimierungsfunktion)
- für "natürliche" Probleme sind die Ressourcen, die für Entscheidungs- / Berechnungs- / Optimierungsproblem benötigt werden, sehr ähnlich.

Beispiel: polynomielle Algorithmen für das Cliquenproblem.

1. Optimierungsproblem polynomiell => Entscheidungsproblem polynomiell

   trivial: berechne größten Clique C, überprüfe ob $|C| \geq k$

Varianten algorithmischer Probleme

2. Entscheidungsproblem polynomiell => Berechnungsproblem polynomiell

Wir können o.B.d.A. annehmen, dass Eingabegraph k-Clique enthält und dass $k \leq |V|$

procedure b-clique(G, k)
  $G' = G$
  while G' enthält k-Clique do
    wähle Knoten $v \in V$
    entferne v aus G'
  done
  gib v aus (ist Teil der berechneten Clique)
  Entferne alle Knoten aus G', die in G nicht adjazent zu v sind
  if $k = 1$ then stop
  b-clique$(G', k-1)$

Varianten algorithmischer Probleme

3. Berechnungsproblem polynomiell => Optimierungsproblem polynomiell

Trivial: berechne Cliquen für alle $k = 1, \ldots, |V|$.
Sobald $\perp$ zurückgegeben wird, wurde max. Clique gefunden

Polynomiell, da nur polynomiell viele Aufrufe des polynomiellen Berechnungsproblems notwendig sind (wegen $k \leq |V|$) .

Für andere Probleme ist hier oft binäre Suche erforderlich.
(wenn eine binär kodierte Zahl die Eingabe ist, deren Wert nicht von einer anderen Eingabe dominiert wird)

Übersicht Vorlesung

- Kapitel 1: Einführung
- Kapitel 2: Turingmaschinen
- Kapitel 3: P vs. NP
- Kapitel 4: Mehr Ressourcen, mehr Möglichkeiten?
- Kapitel 5: Platzkomplexität
- Kapitel 6: Zeitkomplexität Reloaded
- Kapitel 7: Probabilistische Komplexitätsklassen