

Komplexitätstheorie

SoSe 2019

Dr. Jean Christoph Jung, Prof. Dr. Thomas Schneider
AG Theorie der künstlichen Intelligenz
{jeanjung,thomas.schneider}[ÄT]uni-bremen.de

Homepage der Vorlesung: <http://tinyurl.com/ss19-kt>

Organisatorisches

Zeit und Ort:

Di 12–14 MZH 6190
Do 14–16 MZH 6340

Vortragende:

Thomas Schneider
Cartesium Raum 1.56
Tel. (218)-64432
thomas.schneider@uni-bremen.de

Jean Christoph Jung
Cartesium Raum 1.55
Tel. (218)-64433
jeanjung@uni-bremen.de

Position im Curriculum:

Wahlbereich Master-Ergänzung
Modulbereich Theorie
Master-Profil SQ und KIKR

Organisatorisches

Voraussetzungen

Grundkenntnisse aus Vorlesung „Theoretische Informatik 1+2“

Form

K4, gemischte Lehrform:

- 6 Vorlesungen & 2 Übungen zu Grundlagen
- dann selbstständige Bearbeitung eines weiterführenden Themas
- in Semestermitte 2 weitere Vorlesungen durch uns
- am Semesterende Präsentation eurer Themen

Diskussion in Vorlesung **jederzeit erwünscht!**

Vorlesungsmaterial

- Folien und Übungsblätter in [Stud.IP](#)
- Beispiele, Beweise etc. an der Tafel (**mitschreiben!**)

Literatur (allgemein)

- 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

...sind semesterbegleitend, mit Hauptaugenmerk auf Eurem eigenen Beitrag

Prüfungsanforderungen:

- Übungsserien 1+2 (Abgabe Mi. 10.4. bzw. 1.5., mit Zusatzaufgaben)
Bearbeitung in Gruppen à 2–3 Personen
- Bearbeitung eines eigenen Themas mit
 - schriftlicher Ausarbeitung
 - Halten einer Vorlesungseinheit inkl. kurzer Übung (insg. 90 Min.)
- Fachgespräch am Ende des Semesters
- Note für den gesamten Kurs setzt sich ungefähr so zusammen:
40% Vortrag; je 20% Ausarbeitung, Übungsserien, Fachgespräch

Terminübersicht

Vorlesung und Übung

- voraussichtlich 4 Wochen, also bis einschl. 30.4.
- Mehr zu Themen später
- Termine der Übungsblätter:

Blatt	Erscheinen (geplant)	Abgabe	Besprechung, Übungstermin
1	Di. 2.4.	Mi. 10.4.	Do. 11.4.
2	Fr. 12.4.	Mi. 1.5.	Do. 2.5.

- Blätter erscheinen in [Stud.IP](#)
- Abgabe per PDF in [Stud.IP](#) (separater Ordner, bis 23:59 Uhr)

Terminübersicht

Weiterführende Themen

- Themenvorstellung am Di. 23.4. direkt nach Vorlesung
- Themenwahl Di. 30.4. direkt nach Vorlesung
- 2 Vorlesungen durch uns, voraussichtlich 4.6. und 6.6.
- Eure Vorträge voraussichtlich in letzten 1–2 Semesterwochen,
also z. B. 2.7., 4.7., 9.7., 11.7.
- Kurzer Themenüberblick nachher

Komplexitätstheorie

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:**
Wie ressourcenaufwändig ist gegebenes Problem?
- **Ressourcenzentriert:**
Welche Probleme kann ich mit gegebenen Ressourcen lösen?
Also: was ist Grenze der Berechenbarkeit
unter beschränkten Ressourcen?

Beispiel: Erreichbarkeit in Graphen

Definition 1.1 (Pfad, Erreichbarkeit)

Sei $G = (V, E)$ ein gerichteter Graph.

Ein **Pfad** in G ist Folge von Knoten v_0, \dots, v_k ,
so dass $(v_i, v_{i+1}) \in E$ für alle $i < k$.

Ein Knoten v' ist **erreichbar** von einem Knoten v in G ,
wenn es einen Pfad v_0, \dots, v_k gibt mit $v = v_0$ und $v_k = v'$.

T1.1

Erreichbarkeitsproblem:

Gegeben G, v, v' , entscheide ob v' erreichbar von v in G ist.

- **Zentrales Problem** der Informatik, z. B. Erreichbarkeit von Knoten
in Netzwerken, Analyse von Automaten, vieles andere mehr.

Beispiel: Erreichbarkeit in Graphen

Algorithmus 1

Zähle alle Folgen von Knoten v_0, \dots, v_k auf mit $k \leq |V|$.

Überprüfe für jede Folge, ob

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

Antworte „erreichbar“, wenn dies für mindestens eine Folge zutrifft.
Sonst antworte „unerreichbar“.

T1.2

Korrektheit:

- Wenn der Algorithmus „erreichbar“ antwortet,
so ist dies offensichtlich korrekt.
- Wenn v' von v erreichbar ist,
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, \dots, v_k auf mit $k \leq |V|$.

Überprüfe für jede Folge, ob

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

Antworte „erreichbar“, wenn dies für mindestens eine Folge zutrifft.
Sonst antworte „unerreichbar“.

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

Wenn $|V| = n > 1$, so untersucht der Algorithmus **mehr als $n^n \geq 2^n$** Pfade.

Insgesamt braucht der Algorithmus also **exponentiell** viel Zeit

Beispiel: Erreichbarkeit in Graphen

Natürliche Frage:

Ist das Problem inhärent schwer oder der Algorithmus schlecht?

Algorithmus 2

```
S := {v}
markiere v
while S ≠ ∅ do
  wähle u ∈ S
  S := S \ {u}
  forall (u, u') ∈ E do
    if u' nicht markiert then
      S := S ∪ {u'}
      markiere u'
if v' markiert then antworte „erreichbar“ else antworte „unerreichbar“
```

T1.3

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 forall-Schleife höchstens n Schritte.
- Insgesamt $\mathcal{O}(n^2)$ Schritte \rightsquigarrow **polynomiell!**

Algorithmus 1 war also tatsächlich **suboptimal!**

Weiter optimierter Algorithmus hat sogar lineare Laufzeit ($\mathcal{O}(n)$ Schritte)

Beispiel: Cliquesproblem

Definition 1.2 (Clique)

Sei $G = (V, E)$ ein ungerichteter Graph.

Eine **Clique** in G ist eine nicht-leere Knotenmenge $C \subseteq V$, so dass $\{v, v'\} \in E$ für alle $v, v' \in C$.

Die **Größe** einer Clique ist die Anzahl der darin enthaltenen Knoten.

T1.4

Cliquesproblem:

Gegeben Graph G und Zahl k , entscheide ob G Clique der Größe k hat.

Beispiel: Cliquesproblem

Algorithmus 3

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“.

T1.5

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. $k = \frac{n}{2}$, dann $\binom{n}{k} \geq 2^n$, also **exponentieller Zeitbedarf!**

Beispiel: Cliquenproblem

Dieselbe 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 Cliquenproblem als „**schwierig**“ zu identifizieren. (**NP-Vollständigkeit**)

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

Beispiel: Cliquenproblem

Diese Antwort ist typisch für die Komplexitätstheorie:

- Es gibt in diesem Bereich viele schwierige und ungelöste Probleme, darunter die bekanntesten in der Informatik.
- Nur wenig Resultate über die **nicht**-Machbarkeit (z. B. „das Cliquenproblem kann **nicht** in polynomieller Zeit gelöst werden“)
- Stattdessen **relative Aussagen**: „Problem X ist mindestens so schwer wie all diese anderen Probleme hier; für keines ist ein effizienter Algorithmus bekannt“
- Auf diese Weise ergibt sich eine reiche Struktur im Raum aller Probleme
- Dies liefert ein sehr gutes Werkzeug, um die Schwierigkeit eines gegebenen Problems einzuschätzen. (sehr wichtig auch in der Praxis!)

Klassiker



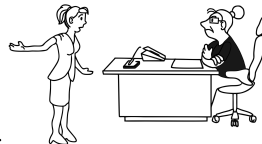
Ich konnte keinen effizienten Algorithmus finden; ich bin wohl zu dumm.



Ich konnte keinen effizienten Algorithmus finden, weil es keinen gibt (glaube ich).

Gäbe es einen effizienten Algorithmus, dann auch für Tausende andere Probleme, an denen sich Generationen von Forscher*innen die Zähne abgebissen haben.

.....

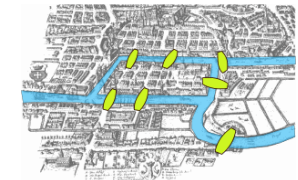


Hätten Sie's gewusst?

Manchmal entscheiden Kleinigkeiten über die Schwierigkeit eines Problems:

Einfach (polynomielle Zeit):

Euler-Kreis: gegeben ungerichteter Graph, entscheide ob es einen Pfad gibt, der jede **Kante** genau einmal besucht und wieder am Anfang ankommt.



Schwer (NP-vollständig):

Hamilton-Kreis: gegeben ungerichteter Graph, entscheide ob es einen Pfad gibt, der jeden **Knoten** genau einmal besucht und wieder am Anfang ankommt.

Hätten Sie's gewusst?

Manchmal entscheiden Kleinigkeiten über die Schwierigkeit eines Problems:

Einfach (polynomielle Zeit):

Minimaler Schnitt: gegeben ungerichteter Graph $G = (V, E)$ und Zahl k , kann man V in Mengen V_1, V_2 partitionieren, so dass es **höchstens** k Kanten zwischen Elementen von V_1 und V_2 gibt?

T1.6

Schwer (NP-vollständig):

Maximaler Schnitt: gegeben ungerichteter Graph $G = (V, E)$ und Zahl k , kann man V in Mengen V_1, V_2 partitionieren, so dass es **mindestens** k Kanten zwischen Elementen von V_1 und V_2 gibt?

Übersicht Vorlesung

Erste 4 Wochen (bis 2.5.):

NOW



§1 Einführung

§2 Turingmaschinen

§3 P vs. NP (Grundlagen)

§4 Platzkomplexität (Grundlagen)

§5 Orakel (Grundlagen)

Später (ca. 4.6. und 6.6.):

§6 Schaltkreise (Grundlagen)

§7 logarithmischer Platz (Grundlagen)

Übersicht Vorlesung

Mögliche weiterführende Themen

- mehr zu NP (Isomorphie, spärliche Mengen, NP-intermediate Probleme)
- Constraint-Satisfaction-Probleme
- Schaltkreiskomplexität, untere Schranken
- logarithmischer Platz, Satz von Immerman-Szelepcényi
- Alternierung
- Orakel; Satz von Baker-Gill-Solovay
- Zählkomplexität
- Selbstreduktion
- Lernkomplexität
- Kolmogorov-Komplexität
- Kommunikationskomplexität
- Natürliche Beweise
- das PCP-Theorem

Kapitel 1

Vier Klarstellungen:

- Laufzeitanalyse
- Worst-Case-Komplexität
- Repräsentation der Eingabe
- Entscheidungsprobleme vs. Berechnungsprobleme vs. Optimierungsprobleme

Laufzeitanalyse

Laufzeit ist Funktion:

Anzahl Schritte in Abhängigkeit von **Größe** der Eingabe

Beim Beschreiben der Funktion abstrahieren wir häufig stark:

- Konstanten werden ignoriert (\mathcal{O} -Notation)
- Meist 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	Autsch!

„**Autsch!**“: Moderner Prozessor braucht mehr Zeit als vom Anfang des Universums bis heute!

Worst-Case-Komplexität

Welche Art von Komplexität studieren wir?

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

Auf **bestimmten Eingaben/Eingabeklassen** kann der Algorithmus aber viel besser sein als im Worst Case.

Beispiel Cliquenproblem:

Betrachte Klasse der spärlichen (sparse) Graphen: viele Knoten haben nur sehr wenige Nachbarn.

Dann ist folgende **Heuristik** sehr effektiv:

Vor dem Suchen einer Clique der Größe k eliminiere alle Knoten mit weniger als k Nachbarn.

Die Worst-Case-Komplexität des naiven Algorithmus ändert sich dadurch jedoch überhaupt nicht!

Worst-Case-Komplexität

Auch möglich ist die Untersuchung der Laufzeit auf **typischen Eingaben** (Average-Case-Komplexität).

Dies ist jedoch in der Regel 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

Repräsentation der Eingabe

Eingabe kann in der Regel verschieden repräsentiert werden:

- **Graphen:** Adjazenzmatrix oder Liste von Kanten **T1.7**
- **Zahlen:** z. B. unär/binär/dezimal kodiert

Größe kann von Repräsentation abhängen:

- **Zahl n :**
 - unär Größe n
 - binär Größe $\log(n)$
 - jede andere Basis $\frac{1}{c} \cdot \log(n)$

→ „log“ ohne Index steht ab jetzt immer für „ \log_2 “, also „Logarithmus zur Basis 2“.

T1.8
- **Graph mit n Knoten:**
 - Adjazenzmatrix hat immer Größe n^2
 - Größe der Kantenliste hängt von Anzahl Kanten ab ($\leq n^2$)

Repräsentation der Eingabe

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

- Man kann einfach in polynomieller Zeit entscheiden, ob Zahl prim ist, wenn diese unär kodiert ist. **T1.9**
- Für binäre Kodierung konnte dies erst 2002 von Agrawal, Kayal und Saxena gezeigt werden („AKS Primality Test“).

Meist unterscheiden sich die Größen verschiedener Repräsentationen nur **polynomiell**, z. B. Adjazenzmatrix vs. Kantenliste

Daher werden wir die **Repräsentation meist nicht im Detail fixieren**.

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

Kapitel 1

Entscheidungs- / Berechnungs- / Optimierungsprobleme

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 von Problem und Anwendung ab.

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 Cliquesproblem (hypothetisch!)

1. Optimierungsproblem polynomiell \Rightarrow Entscheidungsproblem polynomiell

Eingabe Entscheidungsproblem: Graph G , Cliquesgröße k

Algorithmus für Optimierungsproblem liefert in Polyzeit größte Clique C

G hat k -Clique gdw. $|C| \geq k$

Varianten algorithmischer Probleme

2. Entscheidungsproblem polynomiell \Rightarrow Berechnungsproblem polynomiell

Eingabe Berechnungsproblem: Graph G , Cliquesgröße k

Zunächst entscheiden wir in Polyzeit, ob G eine k -Clique enthält und geben \perp aus, wenn das nicht der Fall ist.

Algorithmus 4

Function b-clique(G, k):

$G' := G$

while G' enthält k -Clique **do**

 wähle Knoten $v \in V$

 entferne v aus G'

Ausgabe v % letzter entfernter Knoten 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($G', k - 1$)

T1.10

Varianten algorithmischer Probleme

3. Berechnungsproblem polynomiell \Rightarrow Optimierungsproblem polynomiell

Eingabe Optimierungsproblem: Graph G

Berechne Cliques für alle $k = 1, \dots, |V|$.

Sobald \perp zurückgegeben wird, wurde max. Clique gefunden

Polynomiell, da nur polynomiell viele Aufrufe des polynomiellen Berechnungsproblems notwendig sind.

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

NEXT 

Kapitel 1: Einführung

Kapitel 2: Turingmaschinen

Kapitel 3: P vs. NP

Kapitel 4: Mehr Ressourcen, mehr Möglichkeiten?

Kapitel 5: Platzkomplexität

Kapitel 6: Schaltkreise

Kapitel 7: Orakel