UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

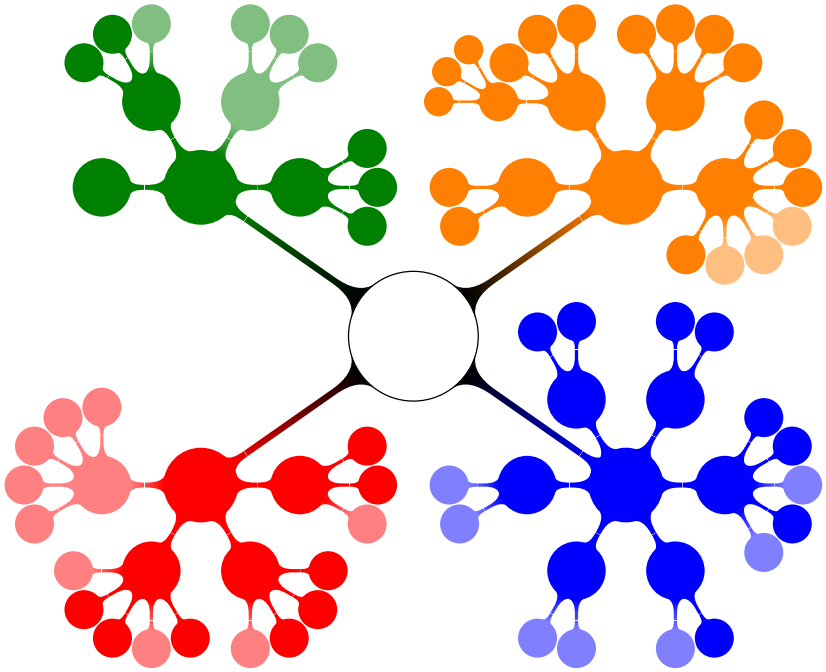
Vorlesungsskript

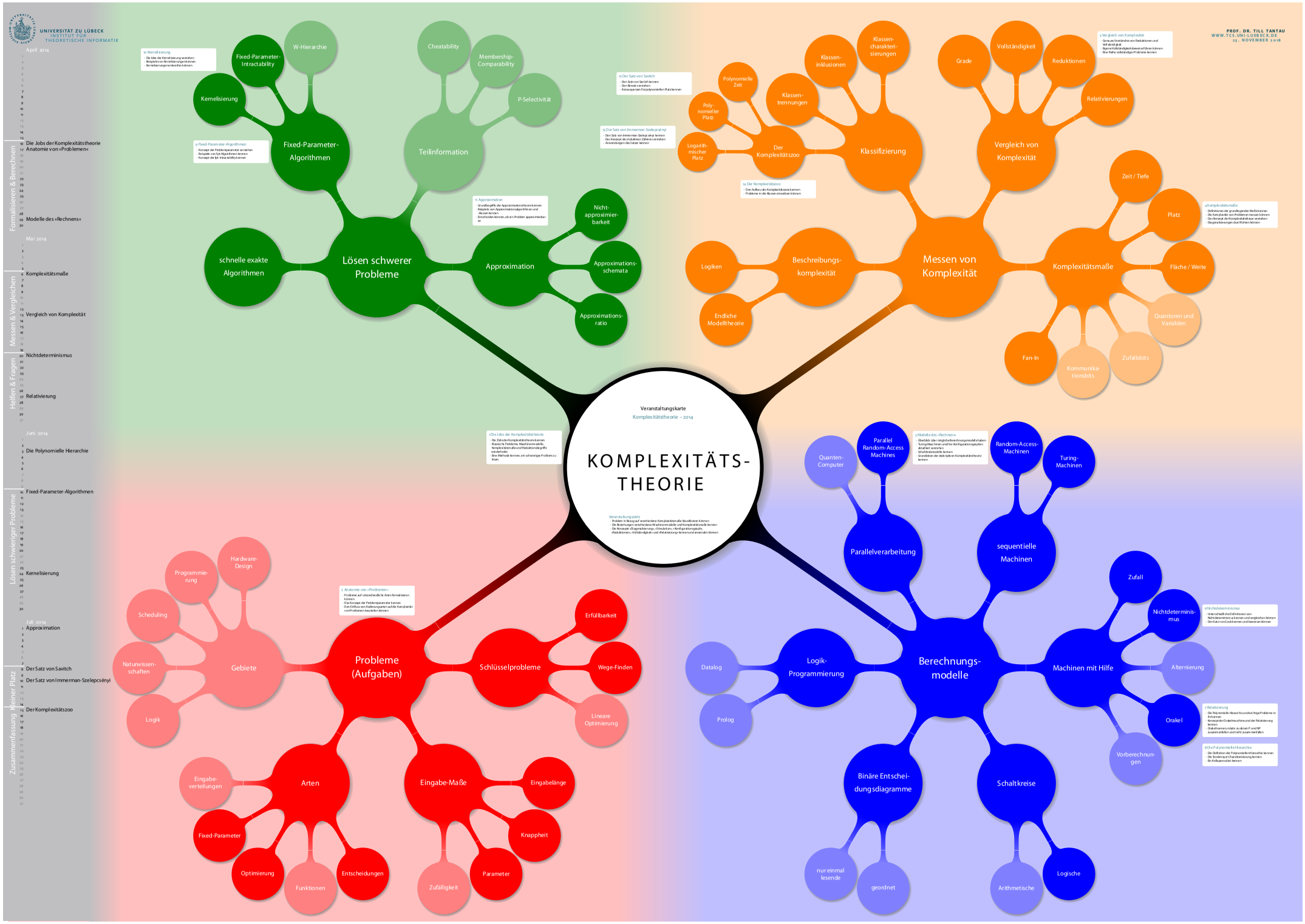
Komplexitätstheorie

CS4003, Sommersemester 2014

Fassung vom 25. November 2018

Till Tantau




UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR THEORETISCHE INFORMATIK
PROF. DR. TILL TANTAU
WWW.TCS.UNI-LUEBECK.DE
25. NOVEMBER 2014
April 2014
Formalisieren & Berechnen
Die Jobs der Komplexitätstheorie
Anatomie von »Problemen«
Modelle des »Rechnens«
Mai 2014
Messen & Vergleichen
Komplexitätsmaße
Vergleich von Komplexität
Helfen & Fragen
Nichtdeterminismus
Relativierung
Juni 2014
Die Polynomielle Hierarchie
Lösen schwieriger Probleme
Fixed-Parameter-Algorithmen
Kernelisierung
Juli 2014
Approximation
Zusammenfassung kleiner Platz
Der Satz von Savitch
Der Satz von Immerman-Szelepcsényi
Der Komplexitätszoo
Veranstaltungskarte
Komplexitätstheorie – 2014
KOMPLEXITÄTS-THEORIE
Veranstaltungsziele
Probleme in Bezug auf verschiedene Komplexitätsmaße klassifizieren können
Die Bedeutungen verschiedener Maschinenmodelle und Komplexitätsmaße kennen
Die Konzepte »Approximierung«, »Simulation«, »Konfigurationsgraph«, »Reduktionen«, »Vollständigkeit« und »Relativierung« kennen und anwenden können
1 Die Jobs der Komplexitätstheorie
Die Ziele der Komplexitätstheorie kennen
Klassische Probleme, Maschinenmodelle, Komplexitätsmaße und Reduktionsbegriffe wiederholen
Eine Methodik kennen, ein schwieriges Problem zu lösen
2 Anatomie von »Problemen«
Probleme auf unterschiedliche Arten formalisieren können
Das Konzept der Problemparameter kennen
Den Einfluss von Kodierungsarten auf die Komplexität von Problemen beurteilen können
3 Modelle des »Rechnens«
Überblick über mögliche Berechnungsmodelle haben
Turing-Maschinen und ihre Konfigurationsgraphen detailliert verstehen
Schaltkreismodelle kennen
Grundideen der diskreten Komplexitätstheorie kennen
4 Komplexitätsmaße
Definitionen der grundlegenden Maße kennen
Die Komplexität von Problemen messen können
Das Konzept der Komplexitätsklasse verstehen
Die grundlegenden Klassen P und NP kennen
5 Vergleich von Komplexität
Genaues Verständnis von Reduktionen und Vollständigkeit
Eigene Vollständigkeitsbeweise führen können
Eine Reihe vollständiger Probleme kennen
6 Nichtdeterminismus
Unterschiedliche Definitionen von Nichtdeterminismus kennen und vergleichen können
Den Satz von Cook kennen und beweisen können
7 Relativierung
Die Polynomielle Hierarchie und viele Höge Probleme in ihr kennen
Konzept der Orakelmaschine und der Relativierung kennen
Orakelklassen relativ zu denen P und NP zusammenfallen und nicht zusammenfallen
8 Die Polynomielle Hierarchie
Die Definition der Polynomiellen Hierarchie kennen
Die Stockmeyer Charakterisierung kennen
Ein Kollaps zu P kennen
9 Fixed-Parameter-Algorithmen
Konzept der Problemparameter verstehen
Beispiele von fpt-Algorithmen kennen
Konzept der fpt-Intraktabilität kennen
10 Kernelisierung
Die Idee der Kernelisierung verstehen
Beispiele von Kernelisierungen kennen
Kernelisierungen erarbeiten können
11 Approximation
Grundbegriffe der Approximationstheorie kennen
Beispiele von Approximationsalgorithmen und -klassen kennen
Entscheiden können, ob ein Problem approximierbar ist
12 Der Satz von Savitch
Den Satz von Savitch kennen
Den Beweis verstehen
Konsequenzen für polynomiellen Platz kennen
13 Der Satz von Immerman-Szelepcsényi
Den Satz von Immerman-Szelepcsényi kennen
Das Konzept der induktiven Zählens verstehen
Anwendungen des Satzes kennen
14 Der Komplexitätszoo
Den Aufbau des Komplexitätszoos kennen
Probleme in die Klassen einordnen können
Lösen schwerer Probleme
schnelle exakte Algorithmen
Modelle des »Rechnens«
Logiken
Endliche Modelltheorie
Beschreibungs-komplexität
Messen von Komplexität
Grade
Vollständigkeit
Reduktionen
Relativierungen
Vergleich von Komplexität
Klassen-charakteri-sierungen
Klassen-inklusionen
Klassen-trennungen
Der Komplexitätszoo
Klassifizierung
Polynomielle Zeit
Poly-nomieller Platz
Logarith-mischer Platz
Zeit / Tiefe
Platz
Fläche / Weite
Komplexitätsmaße
Quantoren und Variablen
Zufallsbits
Kommunika-tionsbits
Fan-In
Approximation
Nicht-approximier-barkeit
Approximations-schemata
Approximations-ratio
Teilinformation
Cheatability
Membership-Comparability
P-Selectivität
Fixed-Parameter-Intractability
W-Hierarchie
Kernelisierung
Fixed-Parameter-Algorithmen
Quanten-Computer
Parallel Random-Access Machines
Random-Access-Machinen
Turing-Machinen
Parallelverarbeitung
sequentielle Machinen
Berechnungs-modelle
Logik-Programmierung
Datalog
Prolog
Binäre Entschei-dungsdiagramme
nur einmal lesende
geordnet
Schaltkreise
Arithmetische
Logische
Machinen mit Hilfe
Zufall
Nichtdeterminis-mus
Alternierung
Orakel
Vorberechnun-gen
Probleme (Aufgaben)
Gebiete
Naturwissen-schaften
Logik
Scheduling
Programmie-rung
Hardware-Design
Schlüsselprobleme
Erfüllbarkeit
Wege-Finden
Lineare Optimierung
Arten
Eingabe-verteilungen
Fixed-Parameter
Optimierung
Funktionen
Entscheidungen
Eingabe-Maße
Eingabelänge
Knappheit
Parameter
Zufälligkeit

Inhaltsverzeichnis

Teil IV
Messen und Vergleichen von Komplexität

Teil V
Berechnungen mit Hilfe

Teil VI
Lösen schwerer Probleme

Teil VII
Die große Welt des kleinen Platzes

Teil VIII
Der Komplexitätszoo

Einleitung

Ich möchte zwei einfache Spiele mit Ihnen spielen. Beide beginnen damit, dass ich einige Zahlen auf ein Blatt Papier schreibe, die sich auf einen Wert größer als 1000 addieren. Danach streichen Sie möglichst viele der Zahlen durch. Die Spiele unterscheiden sich nur in Bezug darauf, wann Sie gewonnen haben: Beim ersten Spiel muss die Summe der verbleibenden Zahlen *mindestens* 1000 ergeben, beim zweien muss die Summe *genau* 1000 ergeben. Die zentrale Fragestellung der gesamten Komplexitätstheorie ist letztendlich, welches der beiden Spiele einfacher ist (und was »einfacher« eigentlich bedeutet). Es lohnt sich, über diese Frage ein wenig nachzudenken.

Die Komplexitätstheorie ist mein zentrales Forschungsthema seit vielen Jahren und ich möchte in dieser Vorlesung meine Faszination für sie mit Ihnen teilen. Da es sich um eine Master-Veranstaltung handelt, werden ich die Sachverhalte selbstverständlich etwas formaler formulieren: aus »Zahlen auf einem Blatt Papier« werden »Wörter über einem Eingabealphabet«, aus »Spielen« werden »formale Sprachen«, aus »einfacher zu lösen« wird »many-one-reduzierbar in logarithmischem Platz« – dennoch will ich aber neben der mathematischen Terminologie auch die zugrundeliegenden Ideen herausarbeiten.

Resultate der Komplexitätstheorie werden gerne als künstlerisch wertvoll und praktisch wertlos beschrieben. Und man muss es ja auch zugeben: Sowohl der Satz von Immerman–Szelepcsényi wie auch der Satz von Reingold haben wunderschöne Beweise, die ein tiefes Verständnis der Feinstruktur von logarithmischem Platz geschaffen haben – beide Sätze sind aber vom praktischen Standpunkt vollkommen nutzlos. Jedoch gibt es in der Komplexitätstheorie eine ganze Reihe von Resultaten, die sehr wohl praktische Bedeutung haben. Ein wichtiges Beispiel sind Fixed-Parameter-Algorithmen: Diese »aus der Theorie geborenen« Algorithmen erlauben es uns heutzutage, NP-vollständige Probleme wie das Knotenüberdeckungsproblem auf Graphen mit Tausenden Knoten in Minuten zu lösen.

In dieser Vorlesung über Komplexitätstheorie werden uns sowohl der künstlerisch wertvolle Teil wie auch der praktisch relevante der Komplexitätstheorie begegnen.

Selbst wenn diese Vorlesungstermine perfekt gestaltet wären (was sie nicht sein werden), so reicht der bloße Besuch dieser Termine nicht aus, um die Methoden und Techniken der Theorie zu erlernen. Sie werden sich mit Problemen im Rahmen des Übungsbetriebs auseinandersetzen müssen; wobei der Übungsbetrieb sehr eng mit den Vorlesungen verbunden sein wird.

Ich freue mich schon sehr auf diese Veranstaltung und hoffe, dass sich mein Vorfreude ein wenig auf Sie überträgt.

Till Tantau

Teil I
Einführung

Vorlesungen zur Komplexitätstheorie leiden daran, dass sie anfangs immer recht langweilig und oft fast ziellos wirken: Um eine Behauptung wie »das Finden von großen Cliquen in Graphen ist schwierig« überhaupt exakt zu *formulieren,* ist eine kleine Armada von Definitionen, Lemmas und Sätzen notwendig. Stunde und Stunde vergeht, in denen die Simulation von Mehrband-Turingmaschinen mit beidseitig unendlichen Bändern durch Einband-Turingmaschinen mit einseitig beschränkten Bändern im Detail untersucht wird, bis die Übersicht wofür man das alles macht endgültig verloren gegangen ist. Selbst wenn man all diese Details weglässt, so sind immernoch viele Definitionen notwendig, um über die »Schwere« von Problemen zu sprechen. Der Sinn und Zweck dieses ersten Teils der Vorlesung ist es, zunächst das »Big Picture« der Komplexitätstheorie einmal im Ganzen darzustellen, bevor wir uns auf die Details stürzen.

Die Komplexitätstheorie möchte herausfinden, wie schwierig unterschiedliche Berechnungsprobleme zu lösen sind. Nur, was bedeutet es eigentlich, ein Problem zu »lösen«? Wie sollte »schwierig« verstanden werden? Und was ist überhaupt ein »Problem«? Keine dieser Fragen hat eine einzelne Antwort; vielmehr bietet die Komplexitätstheorie für jede von ihnen eine ganze Reihe von Antworten, in Abhängigkeit vom »Ziel«, das man verfolgt:

- Die Theorie untersucht, wie man Probleme formal fassen kann. Ein Problem wie »finde einen Weg in einem Graphen« kann als Optimierungsproblem angesehen werden, als Entscheidungsproblem, man kann Instanzen »knapp« (englisch »succinct«) kodieren oder einfach als Adjazenzmatrix. Wie wir sehen werden, haben die so entstandenen Probleme SUCCINCT-REACH und REACH *nachweislich* unterschiedliche Komplexitäten, obwohl sie durchaus beide als (mehr oder minder) natürliche Formalisierungen der Problemstellung »finde einen Weg« angesehen werden können.
- Die Theorie untersucht der Einfluss der verwendeten »Hardware« auf die Schwierigkeit von Problemen, also den Einfluss des *Maschinenmodells,* das wir für unsere Berechnungen heranziehen. Bei manchen Modellen ist es eher Geschmackssache, welchem man den Vorzug gibt (wie bei Turing-Maschinen versus Random-Access-Maschinen). Berechnungen mit »Sondermodellen« wie Quantenrechner oder DNA-Computern haben hingegen wieder nachweislich andere Eigenschaften und beeinflussen, welche Probleme sich schnell lösen lassen.
- Die Theorie hat als Kern die verschiedenen Wege, wie sich Komplexität messen, klassifizieren und vergleichen lässt. Das wichtigste Komplexitätsmaß ist zweifelsohne die Zeit, aber exotische Maße wie die Anzahl an verschränkten Quantenbits in einer Mehrparteienkommunikation werden auch untersucht.

Sobald nun also in obigem Sinn Antworten auf die Frage gefunden sind, wie schwierig Probleme zu lösen sind, können wir anfangen, die Schwierigkeit von möglichst vielen und interessanten Problem zu messen. Oft stellt sich dabei heraus, dass zwei Probleme dieselbe Komplexität in Bezug auf ein bestimmtes Maß haben, aber gänzlich unterschiedliche in Bezug auf ein anderes. Nur weil zwei Problem beispielsweise NP-vollständig sind, müssen sie noch lange nicht gleich schwierig sein: So wie ein Elefant und eine Maus beide grau sind, sind auch das Cliquen-Problem und das Knotenüberdeckungsproblem beide NP-vollständig…

Die letzte und für die Anwendung wichtigstes Aufgabe der Komplexitätstheorie ist schließlich, Methoden zur Lösung von als »schwierig« erkannter Probleme zu finden. Wenn uns die Theorie aufzeigt, dass ein Problem in Bezug auf ein bestimmtes Maß besonders schwer ist, so schreit dies geradezu danach, einen Aspekte des Problems zu finden, in Bezug auf den es leicht ist. Diesen Schrei zu erhören, ist dann die Kunst.

1-1

Kapitel 1
Die Jobs der Komplexitätstheorie

Die Theorie hat viel zu tun. . .

1-2

Lernziele dieses Kapitels

1. Die Ziele der Komplexitätstheorie kennen
2. Klassische Probleme, Maschinenmodelle, Komplexitätsmaße und Reduktionsbegriffe wiederholen
3. Eine Methode kennen, ein schwieriges Problem zu lösen

Inhalte dieses Kapitels

Worum es heute geht

In Science-Fiction-Romanen werden gerne Welten beschrieben, in denen Quantencomputer keine Labor-Kuriositäten sind, sondern in Handys eingebaut Problem wie das Handlungsreisendenproblem in Sekundenbruchteilen lösen. Quantenrechner, die NP-vollständige Probleme schnell lösen, gehören jedoch eher in das Genre der Fantasy als zu Science-Fiction: Anders als viele Informatikerinnen und Informatiker glauben, sind Quantenrechner keine massiv parallele Rechner, die auf magische Weise gigantische Lösungsräume in Sekundenbruchteilen durchsuchen können. Es ist richtig, dass Quantenrechner *manche* schwierige Probleme schnell lösen können. Es ist auch richtige, dass Quantenrechner ein besonders wichtiges schwieriges Problem, nämlich das Faktorisieriungsproblem, besonders schnell lösen können. Jedoch sprechen gute Gründe dafür, dass die dort angewandten Tricks gerade *nicht* beim Handlungsreisendenproblem funktionieren werden.

Der Hauptjob der Komplexitätstheorie ist, die Fehler von Science-Fiction-Autoren zu vermeiden. Die Theorie will die Komplexität von Problemen exakt verstehen, wobei es eine ganze Reihen verschiedener Arten von »Komplexität« gibt. Für die beiden oben genannten Probleme »Faktorisieren« und »Handlungsreisender« wurde dies beispielsweise gemacht und Stand der Forschung ist: In Bezug auf ihre »klassische« Zeitkompleixtät sind die beiden Problem recht ähnlich (wir werden sagen, sie sind in der gleichen Komplexitätsklasse), in Bezug auf ihre Quantenkomplexität ist hingegen Faktorisieren leicht, das Handlungsreisendenproblem hingegen weiterhin schwierig.

Wer mehr wissen möchte, was in Europa passieren wird, wenn größere Quantencomputer eines Tages mal gebaut werden, dem sei das Buch *Halting State* [3] von Charles Stross wärmstens empfohlen.

1-4

Ein nützliches Bild.

Die Komplexitätstheorie hat vieles gemein mit der Biologie:
Die Biologie …

- untersucht Kreaturen.
- misst deren Eigenschaften: Anzahl Glieder, Anzahl Augen, Sozielverhalten…
- klassifiziert sie hiernach zu Spezies.

Die Komplexitätstheorie …
- untersucht Berechnungsprobleme.
- misst deren Eigenschaften: Zeit-, Platz-, Kommunikationskomplexität…
- klassifiziert sie hiernach zu Komplexitätsklassen.

1.1 Job 1: Formalisieren von Problemen

Job 1: Formalisieren von Problemen

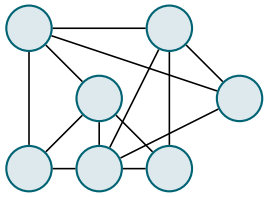
1-5

Die gesamte Theorie dreht sich um *(Berechnungs)probleme* (auch *Aufgaben* genannt).

Beispiele

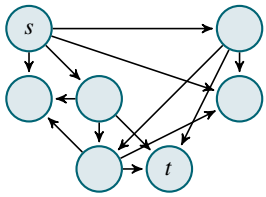
- Das Cliquen-Problem: Gegeben ein ungerichteter Graph und eine Größe k, finde eine Clique dieser Größe in dem Graphen.
- Das Wege-Finden-Problem: Gegeben ein Graph, finde einen Weg von Anfang zum Ende.

Typische Eingabe für das Cliquen-Problem:



Clique der Größe 4?

Typische Eingabe für das Wege-Finden-Problem:



Pfad von s nach t?

Wozu Probleme »formalisieren«?

1-6

Wie schwierig ein Problem zu lösen ist, kann stark von der »exakten Formulierung« abhängen. Es ist *leicht,* einen kürzesten Weg in einem Graphen zu finden, und *sehr schwer,* einen längsten Weg zu finden. Die exakte *Kodierung* eines Problems kann seine Komplexität ebenfalls stark beeinflussen. Schreibt man beispielsweise Zahlen unär statt binär auf, so hat man exponentiell viel mehr Zeit für Berechnungen in Bezug auf die Eingabelänge. Schließlich macht es einen Unterschied, *was man eigentlich wissen will.* Will man einfach irgendeinen Pfad finden? Einen kürzesten? Einen längsten? Einen besonders schönen? All dies beeinflusst die Komplexität.

Erstes Schlüsselproblem: Wege Finden

1-7

Problembeschreibung

Gegeben sind ein Graph und zwei Knoten in dem Graphen. Das Ziel ist, einen Weg zwischen den Knoten zu finden.

Zweites Schlüsselproblem: Constraint-Satisfaction

1-8

Problembeschreibung

Gegeben sind Variablen und Constraints (Bedingungen) für die Variablen. Ziel ist, eine Belegung für die Variablen zu finden, so dass alle Constraints erfüllt sind.

Das Problem 3-SAT als Constraint-Satisfaction-Problem

Belegungen Variablen müssen mit »wahr« oder »falsch« belegt werden.

Constraints Tripel von Variablen, für die jeweils eine Belegung verboten wird.

✎ Zur Diskussion

Formulieren Sie 3-Färbbarkeit als Constraint-Satisfaction-Problem.

1.2 Job 2: Formalisieren von Berechnungen

1-9

Job 2: Formalisierung von Berechnungen

Berechnungen sind unsere Methode zur Lösung von Problemen.

Beispiele von Berechnungsmodellen

- Turing-Maschine.
- Random-Access-Maschine.
- Quantum-Computer.

Wozu »Berechnungen« formalisieren?

Die exakte Formalisierung des Models hat einen großen Einfluss auf die *Effizienz*. Für Quantenrechner erlauben uns überhaupt nur Modelle, zu verstehen, was überhaupt passiert. Schließlich sind exakte Messungen von »Laufzeit« oder »Speicherbedarf« nicht möglich, wenn die Modelle zu vage sind.

1-10

Die Turing-Maschine.

Ein kurze Geschichte der Turing-Maschine

Alan Turing hat sein Maschinenmodell 1936 vorgeschlagen, lange bevor es überhaupt Computer gab. Er gelangte zu dem Modell durch eine philosophische Überlegung, in der er argumentiert, dass alles, was ein Mathematiker überhaupt berechnen kann, auch von seiner Maschine berechnet werden kann. Turing ging es dabei um Berechenbarkeit »als solche«, über Komplexität machte er sich keine Gedanken.

Wieso werden Turing-Maschinen immernoch verwendet?

- Sie sind extrem einfach aufgebaut und damit gut analysierbar.
- Sie sind extrem flexibel, da man leicht jede Menge Spezialbänder hinzufügen kann.
- Ihre natürlichen Eingaben sind Wörter (Strings), was gut zur Theorie der formalen Sprachen passt.

1-11

Ein realistischeres Modell von Computern.

Die Random-Access-Maschine

*RAM*s sind abstrakte Single-Core-Maschinen: Ihr Speicher ist in Zellen organisiert, die Zahlen speichern. Diese Speicherzellen können beliebig und in Zeit $O(1)$ adressiert werden. Programmiert werden die Maschinen in einer einfachen Maschinensprache.

Wieso ist das RAM-Modell nicht das Standardmodell?

- Eingaben, wie Wörter oder Graphen, müssen recht künstlich in Zahlen umgewandelt werden.
- Zellen können, anders als in der Realität, beliebig große Zahlen speichern.
- Außer betreffend Parallelismus ist das Modell schwer erweiterbar.
- Es liefert keine neue Einsichten, die man mit Turing-Maschinen nicht schon hatte.

1.3 Job 3: Messen von Komplexität

1-12

Job 3: Messen von Komplexität.

Die Komplexität von Problemen kann auf verschiedene Arten *gemessen* werden:

- Wie viel *Zeit* wird zur Lösung eines Problems benötigt?
- Wie viel *Speicherplatz* wird gebraucht?
- Wie gut kann das Problem parallelisiert werden?

Wozu Komplexität messen?

- Wir wollen *vorhersagen*, welche Arten von Eingaben wir werden verarbeiten können.
- Wir wollen *verstehen*, wie sich Algorithmen verbessern lassen können.
- Wir wollen *verstehen*, warum wir keine besseren Algorithmen finden.

Das wichtigste Komplexitätsmaß: Zeit.

1-13

Unknown author, public domain

»Zeit« kommt in unterschiedlichen Gewändern

- Anzahl Rechenschritte
- Schaltkreistiefe
- Tiefe von Entscheidungsdiagrammen
- Rundenanzahl eines Protokolls

Warum ist Zeit so wichtig?

- Time is money.
- Ein Problem in 10^{10000} Jahren lösen zu können ist dasselbe, wie es nicht lösen zu können.

Das zweitwichtigste Maß: Platz

1-14

Platz miss, wie viel *Speicherplatz* eine Berechnung benötigt.

Warum ist Platz weniger wichtig?

- Uns geht *nie* der Platz aus, bevor uns die Zeit ausgeht.
- Speicherplatz ist billig.
- Sublinearer Platz ist unrealistisch: Wie haben *immer* mindestens noch mal so viel Speicherplatz wie unsere Eingabe lang ist.

Warum untersuchen wir Platz dann trotzdem?

- *Manche* Algorithmen nutzen doch riesige Mengen an Speicher, oft linear viel relativ zur Laufzeit.
- Kleine Platzklassen sind eng mit Parallelismus verbunden.
- Algorithmen, die beispielsweise auf dem Graph des Internet arbeiten oder die für RFID-Chips implementiert werden, haben eben doch nur sublinear viel Platz.

1.4 Job 4: Vergleichen von Problemen

Wie man die Komplexität von Problemen vergleicht.

1-15

Selbst wenn man die Komplexität eines Problems nicht genau kennt, so kann man doch zumindest oft feststellen, dass das Problem *nicht schwerer als ein anders ist*:

- Die genaue Komplexität des MATCHING Problems ist nicht bekannt (»Hat ein Graph ein Matching einer bestimmten Größe?«)
- Betrachten wir nun folgende Probleme:

$$\text{MATCHMATCH} = \{xx \mid x \in \text{MATCHING}\},$$
$$\text{HCTAMHCTAM} = \{x^{\text{reversed}} \mid x \in \text{MATCHMATCH}\}.$$

- Offenbar sind diese Problem *genauso schwierig wie* MATCHING, sie sind lediglich andere »Kodierungen« desselben Problems.

Viele andere Problem sind auch nur »andere Kodierungen« von MATCHING, man sieht es nur nicht sofort, beispielsweise beim »incomplete perfect path phylogeny non-continuous partitioning« Problem.

1-16

Reduktionen: Eine Methode zum Vergleich zweier Probleme A und B.

- Wir nehmen an, eine Firma / Aliens / Geister können die Frage "Ist $y \in B$?" beliebig schnell beantworten.
- Falls uns dies massiv hilft, die Frage "Ist $x \in A$?" zu beantworten, so sagen wir *A reduziert sich auf B*.
- Die »Übersetzung« der Frage "Ist $x \in A$?" in "Ist $y \in B$?" nennen wir *die Reduktion.*

Beispiel

- Wir reduzieren MATCHING auf MATCHMATCH durch »Verdopplung«, also durch $y = xx$.
- Wir reduzieren MATCHMATCH auf MATCHING durch »Halbierung«.

Wenn A sich auf B reduziert und B sich auch auf A reduziert, nennen wir A und B *äquivalent*.

1-17

Für Anfänger

Beschreiben Sie eine Reduktion von CLIQUE auf INDEPENDENT-SET.

For Fortgeschrittene

Beschreiben Sie eine Reduktion von 3-COLORABLE auf 4-COLORABLE.

For Experten

Beschreiben Sie eine Reduktion von 4-COLORABLE auf 3-COLORABLE.

1.5 Job 5: Klassifizieren von Problemen

1-18

Job 5: Klassifikation von Problemen.

Probleme können *gruppiert* beziehungsweise *klassifiziert* werden in Abhängigkeit ihrer Komplexität.

Komplexitätsklasse

Eine *Komplexitätsklasse* ist eine Menge von Problemen, für die ein *Algorithmus für ein bestimmtes Modell* existiert, der für alle Eingaben eine bestimmte *Ressourcenschranke* einhält.

Wozu Probleme klassifizieren?

- Resultate für eine ganze Klasse treffen eben auf alle ihre Element zu. Insbesondere *Inklusionen* zwischen Klassen zeigen auf, wie sich Algorithmen allgemein »umwandeln« lassen.
- Komplexitätsklassen stellen einen nützlichen Rahmen dar, um Komplexitäten zu vergleichen.

1-19

Drei Beispiele von Komplexitätsklassen.

Beispiel

P enthält alle Sprachen L, für die eine deterministische Turing-Maschine M existiert, die L entscheidet und maximal $n^{O(1)}$ Schritte bei Eingaben der Länge n macht.

Beispiel

E enthält alle Sprachen L, für die eine deterministische Turing-Maschine M existiert, die L entscheidet und maximal $2^{O(n)}$ Schritte bei Eingaben der Länge n macht.

Beispiel

ApxPO enthält alle Optimierungsprobleme P, für die eine deterministische Turing-Maschine M existiert, die Lösungen für P produziert und maximal $n^{O(1)}$ Schritte bei Eingaben der Länge n macht und für die die produzierten Lösungen eine konstante Approximationsrate haben.

1.6 Job 6: Neue algorithmische Ansätze finden

Job 6: Neue algorithmische Ansätze finden.

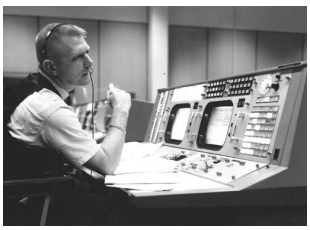
1-20

Die Ergebnisse der Theorie sollen im Endeffekt helfen, *neue algorithmische Ansätze* zu finden.

Beispiele neuer Ansätze

- Fixed-Parameter-Algorithmen
- Quanten-Algorithmen
- Randomisierte Algorithmen
- Approximative Algoirthmen

Gene Kranz: "Failure is not an option."



Copyright by NASA.

Wieso hilft die Theorie?

Die Theorie macht Vorhersagen, dass bestimmte Ansätze scheitern müssen. Damit erlaubt sie es, sich auf neue Ansätze zu konzentrieren. Sie kann auch aufzeigen, wie ähnliche Probleme gelöst wurden.

Wie schwierig ist das 3-SAT Problem?

1-21

✎ Zur Übung

Wir wollen 3-SAT lösen. Eingaben sind Formeln φ mit n Variablen und m Klauseln. Sicherlich können wir $\varphi \in$ 3-SAT entscheiden, indem wir einfach alle 2^n möglichen Belegungen durchgehen. Wenn es 1μ s dauert, eine Lösung zu überprüfen, was ist die maximale Anzahl n an Variablen, so dass wir die Frage in 1000 Sekunden beantworten können?
Wie groß ist n, wenn wir ein Jahr zeit haben (ein Jahr hat etwa 32 Millionen Sekunden)?

Die Komplexitätstheorie sagt uns, dass 3-SAT ein NP-vollständiges Problem ist. Daher zeigt sie auf, dass wir *neue Ansätze suchen müssen.*

Schönings Algorithmus für 3-SAT.

1-22

```
1  input Variablen x_1, ..., x_n
2  input Klausen C_1, ..., C_m mit je drei Literalen
3  do X times
4    wähle eine zufällige Belegung β : {x_1,...,x_n} → {true,false}
5    do Y times
6      U ← {C_i | C_i ist nicht erfüllt von β}
7      if U = ∅ then
8        output ''β ist eine erfüllende Belegung''; stop
9      else
10       wähle zufällig C_i ∈ U
11       wähle zufällig v in C_i
12       β ← β mit dem Wert von v ''geflippt''
13 output ''wahrscheinlich keine erfüllende Belegung vorhanden''
```

Ein großer Fortschritt.

1-23

▶ Satz

Schönings Algorithmus findet für erfüllbare φ eine erfüllende Belegung mit Wahrscheinlich $1/2$ für $Y = 3n$ und $X = (4/3)^n$.

Photo taken at STACS 2005 in Stuttgart

Folgerungen:

- Betrachten wir noch einmal die Maschine, die eine Belegung in einer Mikrosekunde Überprüfen kann. Dann können wir nun in 1000 Sekunden den Algorithmus für Formeln mit bis zu $n = 54$ Variablen nutzen.
- Die Fehlerwahrscheinlichkeit können wir auf 1:1.000.000 drücken, wenn wir den Algorithmus 20 Mal länger laufen lassen.

Zusammenfassung dieses Kapitels

1-24

1. Die Komplexitätstheorie *formalisiert* Probleme und Berechnungsmodelle,
2. um dann die Komplexität von Problemen zu messen, zu klassifizieren und zu vergleichen.
3. Ziel ist letztendlich, *bessere Algorithmen* zu finden, insbesondere für Probleme, wo man solche noch nicht kennt.

Zum Weiterlesen

[1] Alan M. Turing, On Computable Numbers With an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society,* 42(2):230–265, 1936

Dies ist die Originalarbeit, in der Turing die Turing-Maschine einführt (auch wenn sie dort natürlich noch nicht so heißt) und beweist, dass das Halteproblem unentscheidbar ist. Den Artikel gibt es online bei »The History of Computing Project«. Es lohnt sich auch sehr, dort Turings zweiten berühmten Artikel *Computing Machinery and Intelligence* zu lesen, wo er den Turing-Test einführt.

[2] Uwe Schöning, A probabilistic algorithm for k-SAT and constraint satisfaction problems, *Proceedings of the 40th Symposium on Foundations of Computer Science,* IEEE, October 1999, pages 410–414.

Die FOCS-Konferenz ist eine der renommiertesten Konferenzen in Theoretischer Informatik. Es ist selten, dass dort Artikel zu »einfachen« Algorithmen angenommen werden; noch viel seltener ist es, dass ein Artikel mit nur sechs Seiten angenommen wird – normalerweise versuchen Autoren jede Zeile des Zwölf-Seiten-Limits voll auszuschöpfen. Damit dies geschieht, muss das Ergebnis wahrlich brillant sein.

[3] Charles Stross, *Halting State,* Ace, 2007

Meiner Meinung sollte so wie Gibsons *Neuromancer* die Bibel des echten Hackers ist, Stross' *Halting State* die Bibel des echten Komplexitätstheoretikers sein. Sie werden kaum irgendwo unterhaltsameres über die Zukunft der Verschlüsselung lesen.

Teil II
Probleme

Wohingegen die meisten Menschen verständlicher Weise Probleme eher aus dem Weg gehen, die Komplexitätstheoretiker Hobbysammler von Problemen. Sie werden liebevoll katalogisiert, benannt, analysiert und schließlich weggeschlossen. Die frage, was ein Problem nun genau sei, wird in der Regel recht lapidar mit »eine Menge von Wörtern« beantwortet. Diese Antwort ist jedoch wenig adäquat, um das »Wesen« beispielsweise eines Optimierungsproblems zu treffen – hier ist doch wesentlich mehr definitorischer Überbau nötig.

2-1

Kapitel 2
Anatomie von »Problemen«
Probleme als Skulpturen

2-2

Lernziele dieses Kapitels

1. Probleme auf unterschiedliche Arten formalisieren können
2. Das Konzept der Problemparameter kennen
3. Den Einfluss von Kodierungsarten auf die Komplexität von Problemen beurteilen können

Inhalte dieses Kapitels

Worum es heute geht

Als Sie das erste Mal die Definition von »formalen Problemen« in der Theoretischen Informatik gesehen haben, kam Ihnen diese Definition hoffentlich ebenso einfach (»Eine Menge von Wörtern«) wie unpassend vor. Die verschiedenen Arten von Problemen »da draußen« sind irgendwie nicht nur einfach Mengen von Wörtern: Man wird der Sache einfach nicht gerecht, wenn man das Finden eines möglichst guten Flugplanes für eine Fluggesellschaft als eine »Menge von Wörtern« formalisiert.

In der Theoretischen Informatik versucht man trotzdem, »möglichst weit« zu kommen mit der Sicht »Problem = Menge von Wörtern«. Dieser Zugang hat sich als äußerst erfolgreich herausgestellt, da sich so auch scheinbar sehr unterschiedliche Probleme überhaupt erstmal vergleichen lassen. Jedoch hat diese Sicht der Dinge auch ihre Grenzen und es gibt Bereiche in der Theorie, wo komplexere Sichten auf »Probleme« nötig sind. Sehr häufig will man beispielsweise nicht nur einfach »Ja« oder »Nein« als Antwort auf ein Problem, sondern eben eine komplexere »Lösung«, die dann auch noch Gütekriterien entsprechen sollte – also ein Maß, ob die Lösung perfekt, sehr gut, gut, so lala, schrecklich oder schlicht beleidigend ist.

Neben der Arten der »Lösungen« der Probleme, können auch die »Eingaben« (»Instanzen« genannt) komplexer sein als einfach nur Wörter. Besonders bizarre Eingaben haben sich die netten Leute von der Abteilung *desktriptive Komplexitätstheorie* ausgedacht: Hier sind Eingaben logische Strukturen im Sinne der Prädikatenlogik.

2.1 Einführung

Was sind Berechnungsprobleme?

2-4

✎ Zur Diskussion

Welche der folgenden Fragen sind »Berechnungsprobleme«?

- Wie viele perfekte Matchings hat ein gegebener Graph?
- Ist 2.325.630.272.114.329.980.342.342.347 prim?
- Hält eine gegebene Turing-Maschine nach endliche vielen Schritten an?
- Ist π^π eine algebraische Zahl?
- Wie lautet eine gegebene Stelle von π?
- Gilt $\mathrm{P} = \mathrm{NP}$?

▶ Definition: Computational Problem

Ein *Berechungsproblem* besteht aus

- einem *Instanzenbereich*, dessen Elemente *Instanzen* heißen, und
- einer *Abbildung* von Instanzen auf *Antworten*.

Statt dies genauer allgemein zu formalisieren, werden wir lieber konkrete Ausprägungen der Definition betrachten.

2.2 Instanzen

2.2.1 Kodierungen

Wörter als Instanzen

2-5

Wörter sind gute Instanzen, da die meisten Maschinenmodelle, wie beispielsweise Turing-Maschinen, Schaltkreise oder so genannte binäre Entscheidungsdiagramme, Wörter verarbeiten und Computer ja auch tatsächlich alle Daten als Dateien speichern, was letztendlich auch nur lange Wörter sind.

▶ Definition: Alphabet, Wort

Ein *Alphabet* ist eine nichtleere, endliche Menge. Die Elemente eines Alphabets heißen *Buchstaben* oder *Symbole*. Ein *Wort* ist eine endliche Folge von Buchstaben aus einem Alphabet.

Notationen

- Das leere Wort bezeichnen wir mit ε.
- Die Länge eines Wortes w bezeichnen wir mit $|w|$.
- Den i-ten Buchstaben von w bezeichnen wir mit $w[i]$ oder w_i, die Zählung beginnt bei 1.

Zahlen als Instanzen

2-6

Zahlen sind gute Instanzen, da Modelle wie RAMs und PRAMs oder der Kalkül der primitivrekursiven Funktionen Zahlen als Eingaben erwarten. Weiter gibt es so genannte *algebraische* Berechnungsmodelle, bei denen die Eingaben und Ausgaben Zahlen sind. Schließlich sind Zahlen für viele Probleme der Mathematik die natürlichen Eingaben (zum Beispiel Matrix-Matrix-Mulitplikation).

Welche Arten von Zahlen sollte man nutzen?

- Für das RAM- und das PRAM-Modell sind endliche Vektoren von natürlichen (nichtnegativen) Zahlen sinnvoll.
- In der Algebraischen Komplexitätstheorie sind Vektoren von reellen Zahlen die Eingaben.

Kodierung von Wörtern als Zahlen

Um ein Wort $w \in \Sigma^*$ als Zahl zu kodieren, fixieren wir eine Bijektion $e\colon \Sigma \to \{1,\dots,|\Sigma|\}$. Dann können wir w kodieren als

- den Vektor $\big(e(w[1]),\dots,e(w[|w|])\big)$ oder
- die Zahl $n_w := \sum_{i=1}^{|w|} |\Sigma|^i \cdot e(w[i])$ oder
- die Zahl $\prod_{i=1}^{|w|} p_i^{e(w[i])}$, wobei p_i die i-te Primzahl ist. Diese Idee der Kodierung geht auf Gödels berühmten Beweis des Unvollständigkeitssatzes zurück.

2-7

Logische Strukturen als Instanzen

Logische Strukturen sind gute Instanzen, da sie die natürlichen Eingaben in der »Deskriptiven Komplexitätstheorie« sind und bei vielen Problemen, beispielsweise allen Graphproblemen und allen Datenbankproblemen, sich Eingaben am natürlichsten als logische Strukturen darstellen lassen.

Skript-Referenz

▶ **Definition:** Signatur
Eine *Signatur* ist eine Menge von *Relationssymbolen* zusammen mit einer Abbildung, die jedem Symbol eine positive Zahl, genannt die *Stelligkeit* zuordnet. Stelligkeiten werden als hochgestellte Indizes notiert.

▶ **Definition:** Universum
Ein *Universum* ist eine nichtleere Menge.

▶ **Definition:** (Relationale) logische Struktur
Eine *(relationale) logische Struktur $\mathcal{S}$ über einer Signatur τ* besteht aus einem Universum U zusammen mit Relationen $R^{\mathcal{S}} \subseteq U^n$ für jedes n-stellige Relationssymbol $R \in \tau$.

2-8

Bespiele von logischen Strukturen als Eingaben

Beispiel: Graphen als logische Strukturen
Ein Graph G mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$ ist bereits eine logische Struktur über der Signatur $\tau_{\text{graphs}} = (E^2)$.

Beispiel: Wörter als logische Strukturen
Ein Wort $w \in \Sigma^*$ wird wie folgt als Struktur $\mathcal{W}$ kodiert:

- Die Signatur enthält ein Prädikat P^1_σ für jeden Buchstaben $\sigma \in \Sigma$.
- Das Universum U ist die Menge $\{1, \dots, |w|\}$.
- Es gilt $i \in P^{\mathcal{W}}_\sigma$ für eine Element $i \in U$ falls $w[i] = \sigma$.

Beispielsweise würde $w = aabcca \in \{a,b,c\}^*$ kodiert als

$$(\underbrace{\{1,\dots,6\}}_{\text{Universum}}, \underbrace{\{1,2,6\}}_{P_a^{\mathcal{W}}}, \underbrace{\{3\}}_{P_b^{\mathcal{W}}}, \underbrace{\{4,5\}}_{P_c^{\mathcal{W}}}).$$

2.2.2 Parameter

2-9

Es gibt noch mehr »Intanzenparametern« als die Eingabelänge.

Um später die Komplexität von Problemen zu messen, werden wir diese relativ zu *Instanzenparametern* messen wollen: Je größer der Parameter, desto mehr Zeit / Platz / Prozessoren werden wir bereit sein einzusetzen. Der zentrale Parameter *Wortlänge* ist allerdings *weder der natürlichste noch der nützlichste*:

Beispiel: Eine Aussage über Sortieren
»Sortieren benötigt $\Theta(n \log n)$ Vergleiche, wobei n die Anzahl der zu sortierenden Element ist.«

Diese Aussage in Abhängigkeit der Eingabelänge neu zu formulieren, ist schwierig. Allgemein sind *Instanzenparameter* Funktionen, die Instanzen auf natürliche Zahlen abbilden.

2-10

Beispielse von Instanzenparametern.

Beispiel: Parameter für Wörter

- Länge des Wortes.
- Anzahl der unterschiedlichen Buchstaben im Wort.

Beispiel: Parameter für Graphen

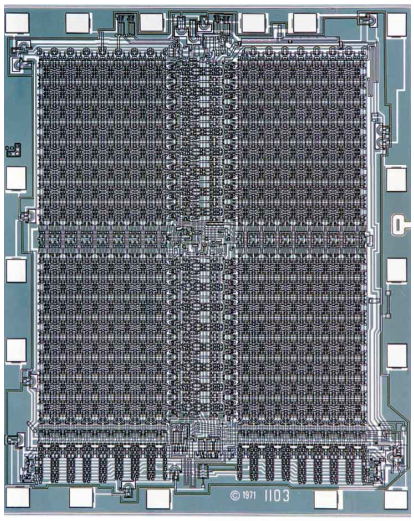
- Anzahl Knoten.
- Anzahl Kanten.
- Maximaler oder minimaler Knotengrad.

Beispiel: Parameter für Zahlen

- Die Zahl selbst.
- Die Länge ihrer Binärkodierung, also grob ihr Logarithmus.

2.2.3 Knappe Darstellungen

Wirklich große Graphen . . . 2-11
. . . können knappe Darstellungen haben



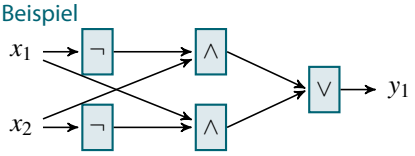

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.std_arith.all;

entity COUNT is
  port( CLK, ENA : in std_logic;
        Q        : buffer std_logic_vector(31 downto 0);
        CARRY    : out std_logic);
end COUNT;

architecture A of COUNT is
begin
   P1: process( CLK )
   begin
      if( CLK'event and CLK = '1' ) then
         if( ENA = '1' ) then
            Q <= Q + 1;
         end if;
      end if;
   end process;
   CARRY <= '1' when Q = "1111" else '0';
end A;
```

2-12

Das Galperin-Wigderson-Model von knapp repräsentierten Wörtern

▶ Definition
Sei C ein Schaltkreis mit n Eingabegattern und einem Ausgabegatter. Wir sagen, *C repräsentiert das Wort* $w \in \{0,1\}^{2^n}$ mit $w[i] = 1$ für all i, für die C zu 1 auswertet, wenn die Eingabegatter mit der Binärdarstellung von i belegt werden.

Beispiel



repräsentiert 0110.

Mit knappen Darstellungen lassen sich *riesige* Wörter mit *winzigen* Wörtern beschreiben.

2.3 Problemarten

2.3.1 Entscheidungsprobleme

2-13

Die einfachste Art einer »Problemstellung«: Ja oder Nein?

Für ein *Entscheidungsproblem* müssen wir für jede mögliche Instanz eine feste *Frage* mit »Ja« oder »Nein« beantworten. Dies kann man formalisieren als *charakteristische Funktion,* die Instanzen mit Antwort »Ja« auf 1 abbildet und die anderen auf 0, oder als Paritionierung der Instanzenmenge in Ja-Instanzen und Nein-Instanzen.

▶ Definition: Entscheidungsproblem
Sei $\mathfrak{D}$ eine Instanzenmenge (also typischerweise $\mathfrak{D} = \Sigma^*$). Ein *Entscheidungsproblem auf* $\mathfrak{D}$ ist eine Teilmenge von $\mathfrak{D}$ (die Menge der Ja-Instanzen).

- Für Instanzenmengen der Form Σ^* heißen Entscheidungsprobleme *Sprachen*. Für unäre Alphabete heißen sie *Tally-Sprachen* (von *to tally* = zählen).
- Sind die Instanzen logische Strukturen, so nennt man Entscheidungsproblem *boolesche Anfragen* (Boolean queries).

2-14

Beispielse interessanter Sprachen

▶ Problem: Das Paritätsproblem
$\text{PARITY} = \{w \in \{0,1\}^* \mid \text{die Anzahl 1en in } w \text{ ist ungerade}\}$.

▶ Problem: Das Primzahlproblem
$\text{PRIMES} = \{w \in \{0,1,\ldots,9\}^* \mid w \text{ ist die Dezimaldarstellung einer Primzahl}\}$.

▶ Problem: Das Erreichbarkeitsproblem
$\text{REACH} = \{\text{code}(G,s,t) \mid G \text{ ist ein gerichteter Graph, in dem es einen Weg von } s \text{ nach } t \text{ gibt}\}$.

2-15

Knappe Versionen von Entscheidungsproblemen.

▶ Definition: Knappe (succinct) Version einer Sprache
Sei A eine Sprache. Dann ist SUCCINCT-A die Sprache aller Codes von Schaltkreisen, die Elemente von A repräsentieren.

- Die Idee ist, dass manche Instanzen natürlicherweise in einer knappen Darstellung gegeben sind.
  Beispiel: Die Beschreibung eines Chips in der einer Hardware-Beschreibungssprache.
- Die Problemstellung bezieht sich aber eigentlich nicht auf die knappe Darstellung, sondern auf das, was von ihr beschrieben wird.
  Beispiel: Hat der Chip einen Kurzschluss?

2.3.2 Funktionsprobleme

Die meisten Probleme erfordern mehr als eine einsilbige Anwort. . . 2-16

Nur wenige Probleme (wie PRIMES) sind wirklich Entscheidungsprobleme. Bei Problemen wie der *Addition* geht es vielmehr darum, für jede Instanz eine »ausführliche Antwort« zu bekommen – formal sind die Probleme also *Funktionen*.

▶ **Definition:** Funktionsproblem
Sei $\mathfrak{D}$ eine Instanzenmenge. Ein *Funktionsproblem* ist eine Abbildung $f\colon \mathfrak{D} \to \mathfrak{D}$.

Funktionsprobleme im Kontext der logischen Strukturen (wo also $\mathfrak{D}$ die Menge aller logischer Strukturen für eine Signatur ist) heißen *Anfragen* (queries).

Beispiele von Funktionsproblemen 2-17

▶ **Problem:** Die arithmetischen Probleme
f-ADDITION ist die Funktion, die ein Wort der Form $\text{code}(u)\#\text{code}(v)$ auf $u+v$ abbildet für $u,v \in \mathbb{N}$. »Falsche« Eingaben wie 00##1# werden beispielsweise auf 0 abgebildet.
Die Funktionen f-MULTIPLICATION und f-DIVISION werden analog definiert.

▶ **Problem:** Das Faktorisierungsproblem
f-FACTORS bildet Codes von Zahlen auf die sortierten Listen ihrer Primteiler ab.

2.3.3 Optimierungsprobleme

Die wirkliche Frage: Was ist die beste Lösung? 2-18

Viele Probleme in der Praxis haben nicht »die eine richtige Lösung«. Vielmehr gibt es viele mögliche »Lösungen«, manche besser, manche schlechter.

▶ **Definition:** Optimierungsproblem
Sei $\mathfrak{D}$ eine Instanzenmenge. Ein *Optimierungsproblem auf* $\mathfrak{D}$ ist ein Tupel bestehend aus

1. einer *Lösungsrelation* $S \subseteq \mathfrak{D} \times \mathfrak{D}$,
2. einer *Maßfunktion* $m\colon S \to \mathbb{N}$ und
3. einem *Typ* $t \in \{\min, \max\}$.

Für eine Instanz $x \in \mathfrak{D}$ und ein Paar $(x,s) \in S$ sagen wir, *s sei eine Lösung für x*.

Das Maß misst, wie »gut« eine Lösung ist. Der Typ beschreibt, ob große Maßzahlen oder kleine Maßzahln »gut« sind.

Beispiel eines Optimierungsproblems 2-19

▶ **Problem:** Das Cliquen-Problem
MAX-CLIQUE ist das Optimierungsproblem, bei dem

1. die Lösungsrelation S die Menge alle (kodierten) Paare (G,C) ist, wobei C eine Clique in dem Graphen G ist,
2. $m(G,C)$ die Anzahl der Knoten in C ist und
3. $t = \max$.

✎ **Zur Übung** 2-20

Das Erfüllbarkeitsproblem SAT wird klassischerweise als Entscheidungsproblem definiert. Wie sieht eine sinnvolle Variante dieses Problems aus als Funktionsproblem und als Optimierungsproblem?
Es gibt viele richtige Lösungen.

2.4 Problemreferenz

Skript-Referenz

Im Folgenden sind Probleme aufgelistet, die in späteren Kapiteln relevant werden.

2.4.1 Arithmetische Probleme

▶ Problem: Funktion f-ADDITION
- Instanzen: Ein (kodiertes) Paar (u,v) von Zahlen.
- Ausgabe: Die (kodierte) Summe $u+v$.

▶ Problem: Sprache BIT
- Instanzen: Ein (kodiertes) Paar (x,i) von Zahlen.
- Frage: Ist das i-te Bit von x eine 1? (Mit $i=0$ wird das Least-Significant-Bit referenziert.)

▶ Problem: Funktion f-DIVISION
- Instanzen: Ein (kodiertes) Paar (u,v) von Zahlen.
- Ausgabe: Der (kodierte) abgeschnitte Quotient $\lfloor u/v \rfloor$ oder 0, falls $v=0$.

▶ Problem: Funktion f-MULTIPLICATION
- Instanzen: Ein (kodiertes) Paar (u,v) von Zahlen.
- Ausgabe: Das (kodierte) Produkt uv.

▶ Problem: Sprache PARITY
- Instanzen: Ein Wort $x \in \{0,1\}^*$.
- Frage: Hat x eine ungerade Anzahl Einsen?

2.4.2 Probleme aus der Zahlentheorie

▶ Problem: Funktion f-FACTORS
- Instanzen: Eine (kodierte) Zahl n.
- Ausgabe: Die (kodierte) Sequenz der Primfaktoren in aufsteigender Reihenfolge.

2.4.3 Graphprobleme

Bei den folgenden Graphproblemen seien die Graphen beispielsweise als Adjazenzmatrizen kodiert.

▶ Problem: Sprache CLIQUE
- Instanzen: Ein ungerichteter Graph $G=(V,E)$ und eine Zahl k.
- Frage: Hat G eine Clique der Größe k?

(Eine *Clique* eines Graphen ist eine Knotenmenge $C \subseteq V$, so dass je zwei Knoten in C mit einer Kante verbunden sind.)

▶ Problem: Optimierungsproblem MAX-CLIQUE
- Instanzen: Ein ungerichteter Graph $G=(V,E)$.
- Lösungen: Cliquen C in G.
- Maß: Größe von C.
- Typ: Maximierung.

▶ Problem: Sprache DISTANCE
- Instanzen: Ein gerichteter Graph $G=(V,E)$, zwei Knoten $s,t \in V$ und eine Zahl d.
- Frage: Gibt es einen Pfad von s nach t der Länge maximal d?

▶ Problem: Funktion f-DISTANCE
- Instanzen: Ein gerichteter Graph $G=(V,E)$ und zwei Knoten $s,t \in V$.
- Ausgabe: Die Distanz von s nach t in G.

▶ Problem: Sprache DOMINATING-SET

Instanzen Ein ungerichteter Graph $G = (V, E)$ und eine Zahl k.

Frage Hat G eine dominierende Menge der Größe k?

(Eine *dominierende Menge* eines Graphen ist eine Knotenmenge $D \subseteq V$, so dass jeder Knoten $v \in V$ entweder in D liegt oder wenigstens mit einem Knoten in D verbunden ist.)

▶ Problem: Optimierungsproblem MIN-EUCLIDIAN-TSP = MIN-2D-EUCLIDIAN-TSP

Instanzen Eine (kodierte) Menge von Punkten in der Ebene.

Lösungen Permutationen der Punktemenge.

Maß Summe der Distanzen von einem Punkt der Permutation zum nächsten plus die Distanz zwischen dem ersten und letzten Punkt.

Typ Minimierung.

▶ Problem: Sprache INDEPENDENT-SET

Instanzen Ein ungerichteter Graph $G = (V, E)$ und eine Zahl k.

Frage Hat G eine unabhängige Menge der Größe k?

(Eine *unabhängige Menge* eines Graphen ist eine Knotenmenge $U \subseteq V$, so dass es keine Kanten zwischen Knoten in U gibt.)

▶ Problem: Sprache LONGEST-PATH

Instanzen Ein ungerichteter Graph $G = (V, E)$, zwei Knoten $s, t \in V$ und eine Zahl d.

Frage Gibt es einen Pfad von s nach t der Länge mindestens d? (Der Pfad darf keinen Knoten zweimal enthalten.)

▶ Problem: Optimierungsproblem MAX-LONGEST-PATH

Instanzen Ein ungerichteter Graph $G = (V, E)$, zwei Knoten $s, t \in V$.

Lösungen Pfade von s nach t. (Kein Knoten doppelt auf solch einem Pfad.)

Maß Länge des Pfades.

Typ Maximierung.

▶ Problem: Sprache REACH = GAP = s-t-CONN

Instanzen Ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$.

Frage Gibt es einen Pfad von s nach t?

(»GAP« steht für »graph accessibility problem«.)

▶ Problem: Optimierungsproblem MIN-REACH = MIN-SHORTEST-PATH = MIN-DISTANCE

Instanzen Ein gerichteter Graph $G = (V, E)$, zwei Knoten $s, t \in V$.

Lösungen Pfade von s nach t. (Kein Knoten doppelt auf solch einem Pfad.)

Maß Länge des Pfades.

Typ Minimierung.

▶ Problem: Optimierungsproblem MIN-TRIANGLE-TSP = MIN-Δ-TSP

Instanzen Ein kantengewichteter vollständiger Graph G, wobei die Kantengewichte die Dreiecksungleichung erfüllen.

Lösungen Ein hamiltonischer Pfad durch den Graphen (=Permutationen der Knotenmenge).

Maß Summe der Gewichte entlang des Pfades.

Typ Minimierung.

▶ Problem: Optimierungsproblem MIN-TSP

Instanzen Ein kantengewichteter Graph G.

Lösungen Ein hamiltonischer Pfad durch den Graphen.

Maß Summe der Gewichte entlang des Pfades.

Typ Minimierung.

▶ Problem: Sprache UDISTANCE
Instanzen Ein ungerichteter Graph $G = (V,E)$, zwei Knoten $s,t \in V$ und eine Zahl d.
Frage Gibt es einen Pfad von s nach t der Länge maximal d?

▶ Problem: Funktion f-UDISTANCE
Instanzen Ein ungerichteter Graph $G = (V,E)$ und zwei Knoten $s,t \in V$.
Ausgabe Die Distanz von s nach t in G.

▶ Problem: Sprache UREACH = UGAP = s-t-UCONN
Instanzen Ein ungerichteter Graph $G = (V,E)$ und zwei Knoten $s,t \in V$.
Frage Gibt es einen Pfad von s nach t?

▶ Problem: Sprache VC = VERTEX-COVER
Instanzen Ein ungerichteter Graph $G = (V,E)$ und eine Zahl k.
Frage Hat G eine Knotenüberdeckung der Größe k?
(Eine *Knotenüberdeckung* eines Graphen ist eine Knotenmenge $K \subseteq V$, so dass jede Kanten mindestens einen Endpunkt in K hat.)

2.4.4 Probleme zu Formeln und Schaltkreisen

Formeln und Schaltkreise seien sinnvoll als Wörter kodiert.

▶ Problem: Sprache BF = BOOLEAN-FORMULA
Instanzen Ein aussagenlogische Formel φ ohne Variablen.
Frage Wertet φ zu wahr aus?

▶ Problem: Sprache CVP = CIRCUIT-VALUE-PROBLEM
Instanzen Ein Schaltkreis C ohne Eingabegatter und mit einem Ausgabegatter.
Frage Wertet der Schaltkreis zu 1 aus?

▶ Problem: Sprache CIRCUIT-SAT
Instanzen Ein Schaltkreis C mit einem Ausgabegatter.
Frage Gibt es einen Belegung für die Eingabegatter, so dass der Schaltkreis zu 1 auswertet?

▶ Problem: Optimierungsproblem MAX-WEIGHTED-CIRCUIT-SAT
Instanzen Ein Schaltkreis C mit einem Ausgabegatter.
Lösungen Belegungen, die C zu 1 auswerten lassen.
Maß Anzahl der 1 in der Belegung.
Typ Maximierung.

▶ Problem: Optimierungsproblem MAX-SAT = MAX-3-SAT
Instanzen Ein aussagenlogische Formel φ in konjuktiver Normalform mit 3 Literalen pro Klausel.
Lösungen Belegungen.
Maß Anzahl der erfüllten Klauseln
Typ Maximierung.

▶ Problem: Optimierungsproblem MAX-WEIGHTED-SAT
Instanzen Ein aussagenlogische Formel φ.
Lösungen Belegungen, die φ wahr machen.
Maß Anzahl der mit wahr belegten Variablen in der Belegung.
Typ Maximierung.

▶ Problem: Sprache NAE-k-SAT für festes $k \geq 3$

Instanzen Eine aussagenlogische Formel φ in konjuktiver Normalform mit genau k Literalen pro Klausel.

Frage Gibt es einen Belegung, so dass in jeder Klausel nicht alle Literale falsch sind (erfüllbar) und auch nicht alle wahr sind (»not all equal«)?

▶ Problem: Sprache SAT

Instanzen Eine aussagenlogische Formel φ.

Frage Ist φ erfüllbar?

▶ Problem: Sprache k-SAT für festes $k \geq 1$

Instanzen Eine aussagenlogische Formel φ in konjuktiver Normalform mit genau k Literalen pro Klausel.

Frage Ist φ erfüllbar?

Zusammenfassung dieses Kapitels

▶ Arten von Instanzen

- Wörter.
- Zahlen und Vektoren.
- Logische Strukturen.

▶ Instanzenparameter

Ein *Instanzenparameter* ist eine Funktion, die Instanzen Zahlen zuordnet.

▶ Knappe Darstellung von Instanzen

Eine *knappe Darstellung* eines Wortes ist ein Schaltkreis, der bei Eingabe einer Bitposition ausrechnet, ob dort eine 1 oder eine 0 in dem Wort steht.

▶ Arten von Problemen

- Entscheidungsprobleme (formalisiert als Sprachen).
- Funktionsprobleme (formalisiert als Funktionen).
- Optimierungsprobleme (formalisiert als Tupel von Lösungsrelation, Maßfunktion und Typ).

Zum Weiterlesen

[1] Hana Galperin, Avi Widgerson, Succinct representations of graphs, *Information and Control*, 56(3):183–198, 1984

In diesem Artikel werden die »knappen Beschreibungen« (succinct representations) eingeführt. Zur gleichen Zeit haben andere Autoren, insbesondere Klaus Wagner, ähnliche Modelle vorgestellt, jedoch stellte sich schnell heraus, dass die verschiedenen Modelle im Wesentlichen gleichmächig sind, das Galperin-Wigderson-Model jedoch formell am einfachsten zu handhaben ist.

Teil III
Modelle

Die Frage, was genau eine »Berechnung« sei, hat eine lange Geschichte. Im Rahmen seiner philosophischen Überlegungen zur Logik und zu Argumentationen hat Blaise Pascal bereits die Hoffnung geäußert, dass eines Tages Dispute nicht mehr von demjenigen »gewonnen« würden, der die spitzere Zunge hat und beeindruckender formuliert. Seine Hoffnung war, eines Tages werden er sagen können »Sire, let us calculate.« Heute würden wir sagen, Pascal strebte die Arithmetisierung logischer Kalküle an. (Etwas traurig ist an dieser Stelle anzumerken, dass wir heutzutage logische Kalküle ganz wunderbar arithmetisieren können, der Weltfrieden hierdurch jedoch auch nicht erreicht wurde.)

Die Notwendigkeit, den Begriff des »Berechnens« genauer zu fassen, wurde Anfang des 20. Jahrhunderts mit David Hilberts Programm der kompletten Formalisierung der Mathematik plötzlich sehr dringend. Hilberts großes Ziel war, die Wahrheit eines beliebigen mathematischen Satzes wie »Körper sind nullteilerfrei« einfach »auszurechnen«. Dabei stellte sich alsbald heraus, dass es ziemlich kniffelig war, den Begriff des »Rechnens« mathematisch exakt zu fasse – er ist ähnlich schwer wie beispielsweise der Begriff des »Denkens«. (Wieder ist traurig anzumerken, dass auch Hilberts Traum nicht in Erfüllung gegangen ist. Gödels Unvollständidkeitssatz zeigt nämlich, dass man nicht für *jeden* mathematischen Satz einfach ausrechnen kann, ob er wahr oder falsch ist. Dies hat Gödel in dem treffend *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I* betitelten Aufsatz gezeigt.)

Die große Wende kam 1936 mit Alan Turings epochalem Aufsatz *On Computable Numbers With an Application to the Entscheidungsproblem* (man beachten das deutsche Wort in dem Titel und überlege, was dies impliziert). Die Bedeutung dieser Arbeit kann gar nicht überbetont werden, denn seit ihr wissen wir, was »berechenbar« bedeutet.

Obwohl die Turingmaschine der Goldstandard in Bezug auf die Frage ist, was man überhaupt berechnen kann, heißt dies nicht, dass wir mit Turing-Maschinen tatsächlich etwas berechnen. Schaltkreise, Random-Access-Maschinen, ja selbst die spukhaften (laut Einstein) Quantenschaltkreise sind realistischere Modelle von »Berechnen«.

Von theoretischen Standpunkt ist es letztendlich besonders interessant zu wissen, wie die praktischen Modelle nun zur Turing-Maschine stehen. Kurz zusammengefasst: Die Modelle können alle ähnlich viel, nur die Schaltkreise, insbesondere die Quantenschaltkreise fallen etwas aus dem Rahmen.

3-1

Kapitel 3
Modelle des »Rechnens«

Lernziele dieses Kapitels

1. Überblick über mögliche Berechnungsmodelle haben
2. Turing-Maschinen und ihre Konfigurationsgraphen detailliert verstehen
3. Schaltkreismodelle kennen
4. Grundideen der deskriptiven Komplexitätstheorie kennen

3-2

Inhalte dieses Kapitels

Worum es heute geht

Wenn Sie an Verschwörungstheorien glauben, so werden Sie am Ende dieses Kapitels eventuell die These vertreten, die Komplexitätstheorie hätte sich mit dem Software-Engineering verschworen, alle Programmierinnen und Programmierer arbeitslos zu machen. Das ultimative Ziel sowohl der deskriptiven Komplexitätstheorie wie auch des Software-Engineerings ist ja, das Programmieren abzuschaffen und sich nur noch auf eine exakte Beschreibung des Problems zu konzentrieren. Wo wir schon bei Verschwörungstheorien sind: Wer macht dann eigentlich noch das Programmieren? Die Computer? Dies könnte man als Skynet-Ansatz bezeichnen. Es gibt übrigens ein Gebiet, bei dem sich der Ansatz »Beschreibe das Problem, nicht die Lösung« durchgesetzt hat: Datenbanken. Wenn Sie von einer Datenbank etwas wissen wollen, geben Sie keinen Code an, wie man in den Indizes suchen soll; vielmehr formulieren Sie eine SQL-Anfrage und es ist Aufgabe der Datenbank, hieraus eine Strategie zur Beantwortung Ihrer Anfrage abzuleiten.

In der Komplexitätstheorie gibt es in Form der deskriptiven Komplexitätstheorie ein »Berechnungsmodell«, das dem Traum der Software-Ingenieure recht nahe kommt: Wir beschreiben Probleme sehr exakt durch Formeln und machen uns keine Gedanken, wie die Probleme algorithmisch gelöst werden sollen. Das erledigen dann Sätze und Lemmata für uns, die aufzeigen, wie sich solche Formeln automatisch in Programme überführen lassen, die bestimmten Zeit- und Platzschranken unterliegen.

3.1 Turing-Maschinen

3-4

Wiederholung: Alan Turings Maschinenmodell

Ein kurze Geschichte der Turing-Maschine

Alan Turing hat sein Maschinenmodell 1936 vorgeschlagen, lange bevor es überhaupt Computer gab. Er gelangte zu dem Modell durch eine philosophische Überlegung, in der er argumentiert, dass alles, was ein Mathematiker überhaupt berechnen kann, auch von seiner Maschine berechnet werden kann. Turing ging es dabei um Berechenbarkeit »als solche«, über Komplexität machte er sich keine Gedanken.

Wieso werden Turing-Maschinen immernoch verwendet?

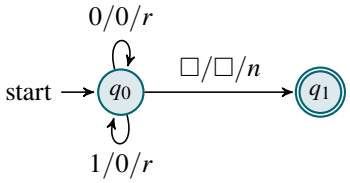
- Sie sind extrem einfach aufgebaut und damit gut analysierbar.
- Sie sind extrem flexibel, da man leicht jede Menge Spezialbänder hinzufügen kann.
- Ihre natürlichen Eingaben sind Wörter (Strings), was gut zur Theorie der formalen Sprachen passt.

3-5

Beispiel einer Turing-Maschine.

Beispiel: Eine simple Turing-Maschine

- Die Zustandsmenge ist $Q = \{q_0, q_1\}$.
- Der Anfangszustand ist q_0.
- Das Bandalphabet ist $\Gamma = \{0, 1, \square\}$.
- Das Eingabealphabet ist $\Sigma = \{0, 1\}$.
- Die Bandanzahl ist 1.
- Das Programm ist



Zur Standard-Definition von Turing-Maschinen siehe das Skript zu »Theoretische Informatik«.

Skript-Referenz

▶ **Definition: Maschinen-Alphabete**
Ein *Bandalphabet* Γ ist ein Alphabet, das das Blanksymbol $\square$ enthält und wenigstens ein weiteres Symbol. Ein *Eingabealphabet* Σ ist eine Teilmenge von Γ, die $\square$ *nicht* enthält.

▶ **Definition: Bänder, Schränke**
Ein *Band* ist eine Sequenz von Symbolen, die beidseitig unendlich ist. Ein *Schrank* enthält eine endliche Anzahl von Bändern. Formal ist ein Band eine Funktion $t : \mathbb{Z} \to \Gamma$ und ein Schrank ist ein Tupel $(t_1, \ldots, t_k)$ von Bändern.

▶ **Definition: Kopf**
Ein *Kopf* ist eine Position auf einem Band, also eine ganze Zahl.

▶ **Definition: Zustände**
Ein *Zustand* ist im Prinzip ein Programmzähler. Formal ist ein Zustand ein Element einer *endlichen Menge Q von Zuständen*. Der *Anfangszustand* q_0 ist ein Element von Q. Die Menge der *akzeptierenden Zuständen* Q_a ist eine Teilmenge von Q.

▶ **Definition: Kopf-Bewegungen**
Während einer Berechnung kann sich der Kopf bewegen. Es gibt *drei* erlaubte Bewegungen: links, rechts und stehenbleiben. Diese werden durch drei Symbole angedeutet: $\{\text{links}, \text{rechts}, \text{neutral}\}$ oder $\{l, r, n\}$ oder $\{-1, 1, 0\}$.

▶ Definition: Programm
Ein *Programm für h Köpfe* ist eine *partielle* Funktion, die Paare von
- einem Zustand und
- einem h-Tupel von gelesenen Bandsymbolen

auf Tripel abbildet, bestehend aus
- einem neuen Zustand,
- einem h-Tupel von geschriebenen Bandsymbolen und
- einem h-Tupel von Kopf-Bewegungen.

Formal ist dies eine partielle Funktion

$$\delta\colon Q \times \Gamma^h \dashrightarrow Q \times \Gamma^h \times \{l,r,n\}^h.$$

▶ Notation
Wir stellen Programme als Graphen dar mit den Zuständen als Knoten. Es gibt eine Kanten von q nach q' mit der Markierung $a/b/c$, falls $\delta(q,a) = (q',b,c)$.

▶ Definition: Deterministische Mehrband-Turing-Maschine
Eine *k-Band-Turing-Maschine* (k-DTM) besteht aus:
- Einer endlichen Menge Q von Zuständen.
- Einem Anfangszustand $q_0 \in Q$.
- Einer Menge $Q_a \subseteq Q$ von akzeptierenden Zuständen.
- Einem Bandalphabet Γ mit $\square \in \Gamma$.
- Einem Eingabealphabet $\Sigma \subseteq \Gamma$ mit $\square \notin \Gamma$.
- Einer Anzahl k an Bändern.
- Einem Programm δ für k Köpfe.

▶ Notation
Wir stellen Turing-Maschinen durch ihre Programme dar. Der Initialzustand ist durch einen Pfeil angedeutet, akzeptierende Zustände durch Doppelkreise.

▶ Definition: Konfigurationen
Eine *Konfiguration* einer Turingmaschine M ist ein Tupel, bestehend aus
- dem *aktuellen Zustand* $q \in Q$,
- dem *aktuellen Schrank an Bändern* und
- den *aktuellen Kopfpositionen*.

Formal ist dies also ein Element von $Q \times [\mathbb{Z} \to \Gamma]^k \times \mathbb{Z}^k$.
Hierbei ist $[\mathbb{Z} \to \Gamma]$ die Menge aller Funktionen $t\colon \mathbb{Z} \to \Gamma$.

▶ Notation
Ein Band mit einem Kopf schreiben wir wie folgt auf:
1. Wir beginnen mit Pünktchen.
2. Dann schreiben wir alle Symbole vor der Kopfposition auf, beginnend mit der ersten Stelle, die nicht $\square$ ist, oder einer früheren Stelle.
3. Dann schreiben wir das spezielle Dreiecks-Symbol hin, *das kein Element des Bandalphabets ist, sondern nur eine Markierung*.
4. Dann kommen alle folgenden Symbole, mindestens bis zum letzten, das nicht $\square$ ist.
5. Wir enden mit Pünktchen.

Beispiel: Ein Band mit dem Kopf auf der ersten 1
$\cdots\square\square 0022002 \triangleright 12000020\square 1101\square\square\cdots$

▶ Definition: Berechnungsschritt
Eine Konfiguration $C = (q, t_1, \ldots, t_k, h_1, \ldots, h_k)$ hat die *Nachfolgekonfiguration* $C' = (q', t_1', \ldots, t_k', h_1', \ldots, h_k')$, geschrieben $C \vdash_M C'$, falls:
- Sei σ_i das Symbol an Position h_i des Bandes t_i, also $\sigma_i = t_i(h_i)$.
- Sei $\delta(q, \sigma_1, \ldots, \sigma_k) = (q', \sigma_1', \ldots, \sigma_k', m_1, \ldots, m_k)$.
- Dann ist

$$t_i'(j) = \begin{cases} t_i(j), & \text{falls } j \neq h_i, \\ \sigma_i', & \text{sonst.} \end{cases}$$

$$h_i' = h_i + \begin{cases} -1, & \text{falls } m_i = l, \\ 1, & \text{falls } m_i = r, \\ 0, & \text{falls } m_i = n. \end{cases}$$

▶ Definition: Anfangskonfiguration
Die *Anfangskonfiguration* $C_{\text{init}}(w)$ einer Maschine M für ein Wort $w \in \Sigma^*$ lautet:

- Der Zustand ist q_0.
- $t_1(i) = w[i]$ für $i \in \{1, \ldots, |w|\}$.
- Alle anderen Zellen sind mit $\square$ gefüllt.
- Alle Köpfe sind auf Position 1.

▶ Definition: Endkonfigurationen und akzeptierende Konfigurationen
Eine *Endkonfiguration* ist eine Konfiguration ohne Nachfolgekonfiguration (dies ist der Fall, wenn die partielle Funktion δ für die gelesenen Symbole undefiniert ist).
Eine Endkonfiguration heißt *akzeptierend*, wenn ihr Zustand ein Element von Q_a ist.

▶ Definition: Akzeptierte Sprache
Sei M eine Turingmaschine. Wir sagen, M *akzeptiert ein Wort* $w \in \Sigma^*$, falls eine akzeptierende Endkonfiguration C existiert mit $C_{\text{init}}(w) \vdash_M^* C$. Die von M *akzeptierte Sprache* $L(M)$ ist die Menge aller von M akzeptierten Wörter.

▶ Definition: Berechnete Funktion
Für eine Turingmaschine M mit einem *Ausgabeband* sagen wir, dass M die Funktion $f\colon \Sigma^* \to \Sigma^*$ berechnet, wenn für alle $w \in \Sigma^*$ gilt: Es gibt eine akzeptierende Endkonfiguration C mit $C_{\text{init}}(w) \vdash C$ und $f(w)$ ist der Inhalt des Ausgabebandes in C, wenn man alle Blanksymbole vor dem ersten Nicht-Blanksymbol und dem letzten Nicht-Blanksymbol auf dem Band ignoriert. Wir schreiben dann auch $M(x)$ für $f(x)$.

3-6

Der Konfigurationsgraph.

In der Komplexitätstheorie können Turing-Maschinen in aller Regel nur *endlich viele Konfigurationen* bei einer Eingabe erreichen – bevor ihnen die Zeit oder der Platz ausgehen:

▶ Definition: Konfigurationsgraph
Für eine Eingabe w sei $\mathcal{C}$ die Menge alle *potentiell erreichbaren Konfigurationen* (aufgrund der Zeit- oder Platzschranken) für eine Maschine M. Dann ist der *Konfigurationsgraph* der Maschine M der gerichtete Graph mit der Knotenmenge $\mathcal{C}$ und Kanten von einer Konfiguration C zu einer anderen C' genau dann, wenn $C \vdash_M C'$.

3-7

✎ Zur Übung
Geben Sie obere Schranken für die Größe der Menge an potentiell erreichbaren Konfigurationen (und damit für die Größe des Konfigurationsgraphen) an in Abhängigkeit von der Länge n der Eingabe:

1. Eine Maschine M_1, die niemals irgendwelche Bandsymbole modifiziert (also ein Zwei-Wege-DFA).
2. Eine Maschine M_2, die maximal polynomiell viele Schritte macht, bevor sie anhält.
3. Eine Maschine M_3, die ausschließlich die Bandpositionen 1 bis $\log_2 n$ auf den Bändern ändert.

3-8

Nur-Lese- und Nur-Schreib-Bänder

▶ Definition: Nur-Lese-Bänder (read only)
Auf einem *Nur-Lese-Band* muss das geschriebene Symbol immer gleich dem gelesenen Symbol sein.

▶ Definition: Nur-Schreibe-Bänger (write only)
Bei einem *Nur-Schreibe-Band* muss der Wert der Zustandüberführungsfunktion δ unabhängig sein vom gelesenen Symbol.

Man beachte, dass diese Eingschränkungen *syntaktischer* Natur sind. Wir werden für unsere Maschinen immer annehmen, dass das Eingabeband ein Nur-Lesen-Band ist und das Ausgabeband (wenn vorhanden) ein Nur-Schreiben-Band ist.

Eine Übersicht von möglichen Spezialbändern. 3-9

In späteren Kapiteln werden wir noch jede Menge Spezialbänder nutzen:

Zufallsband Ein Nur-Lese-Band, das mit einem String von zufälligen *Bits* initialisiert ist.

Auswahlband Ein Nur-Lese-Band, das mit einer Folge von Bits initialisiert ist, die nichtdeterministische Entscheidungen steuern.

Orakelband Ein Band, das gelesen und geschrieben werden kann. Wenn die Maschine in einen bestimmten Zustand übergeht, wird der Inhalt des Bandes augenblicklich durch einen anderen ersetzt.

Komminikationsband Ein Band, über das die Maschine Nachrichten mit anderen Maschinen austauscht.

3.2 Die Random-Access-Maschine

Wiederholung: Ein realistischeres Modell von Computern. 3-10

Die Random-Access-Maschine

RAMs sind abstrakte Single-Core-Maschinen: Ihr Speicher ist in Zellen organisiert, die Zahlen speichern. Diese Speicherzellen können beliebig und in Zeit $O(1)$ adressiert werden. Programmiert werden die Maschinen in einer einfachen Maschinensprache.

Wieso ist das RAM-Modell nicht das Standardmodell?

- Eingaben, wie Wörter oder Graphen, müssen recht künstlich in Zahlen umgewandelt werden.
- Zellen können, anders als in der Realität, beliebig große Zahlen speichern.
- Außer betreffend Parallelismus ist das Modell schwer erweiterbar.
- Es liefert keine neue Einsichten, die man mit Turing-Maschinen nicht schon hatte.

Beispiel einer RAM. 3-11

Für eine detaillierte Definition von Random-Access-Maschinen, siehe das Skript zur Theoretischen Informatik.

Beispiel

```
1 R3 ← R1
2 R4 ← 0
3 R4 ← R4 + R2
4 R3 ← R3 − 1
5 if R3 > 0 goto 3
6 stop
```

✎ Zur Diskussion

Was macht das Programm?

3.3 Schaltkreise

3-12

Computer so modellieren, wie sie wirklich arbeiten

Das Schaltkreismodell

Praktisch alle Computer sind intern letztendlich hochintegrierte Schaltkreise. Die in der Theoretischen Informatik benutzten Schaltkreismodelle stellen eine ganz gute Modellierung solcher Schaltungen dar. Tatsächlich stellen sie die bei weitem »realtistischste« Modellierung von Rechnern dar innerhalb der Theoretischen Informatik.

Warum ist das Schaltkreismodell nicht das Standardmodell?

- Schaltkreise sind sehr anstrengend zu »programmieren«.
- Schaltkreise können nur Eingaben einer fixen Länge verarbeiten, was »unschöne« Definition nach sich zieht.
- Mit Schaltkreisen kann man schlecht Schleifen modellieren.

3.3.1 Syntax

3-13

Überblick zu Gattern.

Schaltkreise (in Sinne der Theoretischen Informatik) bestehen aus *Gattern*, die durch Kanten (Leitungen) verbunden sind. Sie unterscheiden sich in Bezug darauf, welche Arten von Gatter erlaubt sind und wie die Gatter verbunden werden dürfen.

▶ Definition: Gatter
Ein *Gatter* ist ein Knoten in dem Graphen, aus dem der Schaltkreis besteht. Der Eingrad des Gatters ist der *Fan-in*, der Ausgrad der *Fan-Out*.

▶ Definition: Kleine und große Gatter
Ein Gatter mit Fan-in maximal 2 heißt *klein*. Bei größerem Fan-In heißt es *groß* oder *weit*.

▶ Definition: Ein- und Ausgabegatter
Ein *Eingabegatter* hat Fan-in 0, *Ausgabegatter* haben Fan-In 1 und Fan-Out 0. Sie sind mit *Positionen* des Eingabe- beziehungsweise des Ausgabestrings markiert.

3-14

Schaltkreise sind kreisfrei.

▶ Definition: Allgemeiner Schaltkreis
Ein *n-Eingabe-m-Ausgabe-Schaltkreis* ist ein endlicher DAG, dessen Knoten Gatter sind und in dem es genau n Eingabegatter und genau m Ausgabegatter gibt.

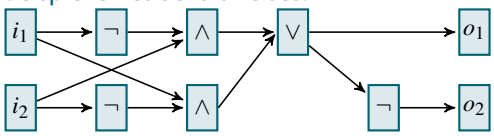
▶ Definition: Boolesche Schaltkreise
Ein *boolescher Schaltkreis* ist ein Schaltkreis, der folgende Gatter hat:

- Ein- und Ausgabegatter,
- Konstantengatter (Eingrad 0),
- Negationsgattter,
- Und-Gatter und
- Oder-Gatter.

Wenn keine Negationsgatter vorhanden sind, heißt der Schaltkreis *monoton*.

3-15

Beispiel eines Schaltkreises.



3.3.2 Semantik

Semantik von Schaltkreisen: Effekte der Gatter. 3-16

▶ Definition
Eine *Gatter-Belegung* für einen booleschen Schaltkreis ist eine Funktion γ, die Gatter auf $\{0,1\}$ abbildet.

Diese Funktionen geben an, welche *Ausgaben* die Gatter (vermutlich) produzieren. Man muss nun natürlich definieren, wenn einen solche Belegung *korrekt* ist.

▶ Definition
Für einen n-Eingabe-m-Ausgabe-Schaltkreis C und eine Eingabe $x \in \{0,1\}^n$ und eine Ausgabe $y \in \{0,1\}^m$ heißt eine Gatter-Belegung γ *korrekt*, wenn:

1. Für jedes *Konstantengatter* g mit Label $i \in \{0,1\}$ gilt $\gamma(g) = i$.
2. Für jedes *Eingabegatter* g mit Label i gilt $\gamma(g) = x[i]$.
3. Für jedes *Ausgabegatter* g mit Label i und Vorgänge g' gilt $\gamma(g) = \gamma(g') = y[i]$.
4. Für jedes *Negationsgatter* g mit Vorgänger g' gilt $\gamma(g) = 1 - \gamma(g')$.
5. Für jedes *Und-Gatter* g mit Vorgängern $g_1, \ldots, g_k$ gilt $\gamma(g) = \min\{\gamma(g_1), \ldots, \gamma(g_k)\}$.
6. Für jedes *Oder-Gatter* g mit Vorgängern $g_1, \ldots, g_k$ gilt $\gamma(g) = \max\{\gamma(g_1), \ldots, \gamma(g_k)\}$.

▶ Lemma
Für jeden n-Eingabe-m-Ausgabe-Schaltkreis C und Eingabe $x \in \{0,1\}^n$ existieren genau eine Ausgabe $y \in \{0,1\}^m$ und genau eine Gatter-Belegung γ, die für den Schaltkreis, die Eingabe und die Ausgabe korrekt ist.

Beweis. Da C ein DAG ist, können wir C topologisch sortieren. Nun zeigt man durch Induktion über die Knotennummer entlang dieser Sortierung, dass der Wert des jeweils nächsten Gatters determiniert wird durch die vorherigen. □

▶ Definition: Ausgabe eines Schaltkreises
Für einen n-Eingabe-m-Ausgabe-Schaltkreis C und eine Eingabe $x \in \{0,1\}^n$ wird die aufgrund der Lemmas eindeutig bestimmte Ausgabe mit $C(x) \in \{0,1\}^m$ bezeichnet.

3.3.3 Varianten: Spezialgatter

Zusatzgatter für boolesche Schaltkreise. 3-17
Wir können unsere Schaltkreise kleiner machen, wenn wir mächtigere Gatter zulassen (alle sind weite Gatter).

Spezialgatter

Majority	Gibt 1 aus, wenn die *Mehrheit* seiner Eingänge 1 sind.
Threshold	Gibt 1 aus, wenn die Anzahl der Eingänge mit einer 1 mindestens gleich einer *auf den Gatter angegebene Zahl ist.*
Parity	Gibt 1 aus, wenn die Anzahl der Eingänge mit einer 1 ungerade ist.
Modulo	Gibt 1 aus, wenn die Anzahl der Eingänge mit einer 1 *modulo* einer auf dem Gatter stehenden Zahl 0 ist.

✎ Zur Diskussion
Argumentieren Sie, dass Majority-Gatter Threshold-Gatter simulieren können und umgekehrt, ohne die Tiefe des Schaltkreises zu verändern.

3.3.4 Varianten: Arithmetische Schaltkreise

3-18

Schaltkreise auf Zahlen.
Bei *arithmetische Schaltkreisen* werden statt booleschen Werten *Zahlen* zwischen den Gattern weitergereicht. Die Gatter verarbeiten die Zahlen und liefern neue Zahlen.

▶ **Definition:** Arithmetische Schaltkreise
Ein *arithmetischer Schaltkreise* ist ein Schaltkreis der folgende Gatter hat:

- Ein- und Ausgabegatter,
- Konstantengatter (Eingrad 0),
- Additionsgatter und
- Multiplikationsgatter.

Man kann als Spezialgatter auch Negationsgatter, Divisionsgatter und weitere Gatterarten zulassen.

Die Semantik ist ähnlich wie bei booleschen Schaltkreisen definiert.

3-19

Schaltkreise auf Mengen von Zahlen.
Statt booleschen Werten oder Zahlen kann man sogar *ganze Mengen von Zahlen* über die Leitungen reichen. Die Gatter führen dann *Operationen auf Mengen von Zahlen durch* wie die Vereinigung.

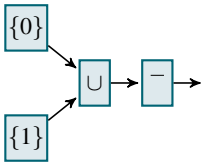
▶ **Definition:** McKenzie-Wagner-Schaltkreise, Syntax und Semantik
Ein MC$(\cup, \cap, \overline{\ }, +, \times)$-Schaltkreis verarbeitet Mengen von ganzen Zahlen. Die Gatter haben folgende Effekte:

- Die Eingabegatter und die Konstantengatter liefern immer einelementige Mengen.
- Die *Vereinigungsgatter* und *Schnittgatter* geben die Vereinigung beziehungsweise den Schnitt ihrer Eingaben aus.
- *Komplementgatter* liefern $\gamma(g) = \mathbb{N} \setminus \gamma(g')$, wobei g' das Vorgängergatter von g ist.
- Bei Werten $A \subseteq \mathbb{N}$ und $B \subseteq \mathbb{N}$ an ihren Eingängen liefern *Additionsgatter* und *Multiplikationsgatter* die Werte $\{a+b \mid a \in A, b \in B\}$ und $\{a \cdot b \mid a \in A, b \in B\}$.

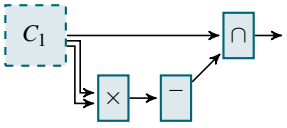
3-20

✎ **Zur Übung**
Welche Mengen geben die folgenden Schaltkreise aus?

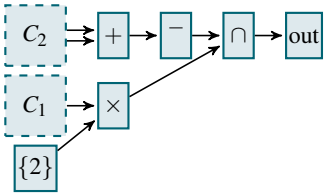
1. C_1:



2. C_2:



3. C_3:



3.4 Deskriptive Modelle

3.4.1 Idee

Ein Modell von »Berechnung«, bei dem nicht gerechnet wird. 3-21

Ziel der deskriptiven Komplexitätstheorie

Statt *konkrete Berechnungen* zu beschreiben, beschreiben wir lieber *das Problem* sehr formell und exakt. Die Hoffnung ist nun, *allein aufgrund der Komplexität der Problembeschreibung* auf die Komplexität des Problems zu schließen.

Wie beschreibt man nun Probleme »genau«?

Wir haben schon gesehen, dass bei manchen Problemen die *Instanzen* gerade *endliche logische Strukturen sind*. Für eine *prädikatenlogische Formel* ist die Menge aller ihrer *endlichen Modelle* genau eine solche Menge von logischen Strukturen. Wir benutzen also *Formeln,* um *Probleme zu beschreiben*.

Beispiele, wie sich Probleme mit Formeln beschreiben lassen. 3-22

Beispiel: Das Tournamentproblem

Frage Ist ein Graph ein Graph ein Tournament (je zwei unterschiedliche Knoten sind durch genau eine Kante verbunden)?

Beschreibung Die Menge aller endlichen logischen Strukturen $\mathcal{G} = (V,E)$, für die gilt

$$\forall x \forall y \big(x \neq y \to (E(x,y) \leftrightarrow \neg E(y,x)) \big).$$

Beispiel: 3-Färbbarkeit

Frage Ist ein Graph 3-färbbar?

Beschreibung Die Menge aller endlichen logischen Strukturen $\mathcal{G} = (V,E)$, für die gilt

$$\begin{aligned} \exists R \exists G \exists B \big(& \forall x (B(x) \vee G(x) \vee B(x)) \wedge \\ & \forall x \forall y \big(E(x,y) \to \\ & \quad \neg \big((R(x) \wedge R(y)) \vee \\ & \quad\quad (G(x) \wedge G(y)) \vee \\ & \quad\quad (B(x) \wedge B(y)) \big) \big) \big). \end{aligned}$$

✎ Zur Übung 3-23

Finden Sie prädikatenlogische Formeln, die folgende Probleme beschreiben (nach Schwierigkeit sortiert):

1. Das Graph ist 3-regulär (alle Knoten haben Grad 3).
2. Der Graph hat Durchmesser höchstens 4 (der Durchmesser eines Graphen ist die maximale Entfernung von Knoten im Graphen).
3. Der Graph ist stark zusammenhängend. (Hierfür benötigen Sie Logik zweiter Stufe.)
4. Der Graph ist ein stark zusammenhängendes Tournament.

3.4.2 Klassen

3-24

Logiken liefern Problemklassen.

▶ Definition: Deskriptive Komplexitätsklassen
Fixiert man eine bestimmte Logik, so liefert die Menge an Problemen, die sich mit Formeln dieser Logik beschreiben lassen, eine *Komplexitätsklasse*.

Beispiele
Die *Klasse* $\mathrm{FO}_{\mathrm{wo}<}$ enthält alle Probleme, die sich in Prädikatenlogik erster Stufe beschreiben lassen. Ähnlich enthält SO alle Probleme, die sich mit Prädikatenlogik zweiter Stufe (second order) beschreiben lassen.

- TOURNAMENT $\in \mathrm{FO}_{\mathrm{wo}<}$, da sich Tournaments mit einer erststufigen Formel beschreiben lassen (siehe oben).
- 3-COLORABLE $\in$ SO, da sich 3-Färbbarkeit in zweitstufiger Logik beschreiben lässt (siehe oben).
- REACH $\notin \mathrm{FO}_{\mathrm{wo}<}$ aufgrund einer einfachen Anwendung des Kompaktheitssatzes (nachdem, wenn jede endliche Teilmenge einer Formelmenge ein Modell hat, so auch die gesamte Formelmenge).

3-25

Logiken entsprechen Klassen.

- Die Klasse $\mathrm{FO}_{\mathrm{wo}<}$ entspricht keiner Standard-Komplexitätsklasse. Varianten aber schon:
  - $\mathrm{FO} = \mathrm{AC}^0$.
  - $\mathrm{FO}(\mathrm{DTC}) = \mathrm{L}$.
  - $\mathrm{FO}(\mathrm{TC}) = \mathrm{NL}$.
  - $\mathrm{FO}(\mathrm{LFP}) = \mathrm{P}$.
- Varianten von zweitstufiger Logik entsprechen größeren Klassen:
  - $\mathrm{SO}\exists = \mathrm{NP}$.
  - $\mathrm{SO}\forall = \mathrm{coNP}$.
  - $\mathrm{SO} = \mathrm{PH}$.

Zusammenfassung dieses Kapitels

3-26

▶ Turing-Maschinen

Vorteile

+ (Vergleichsweise) einfach und sehr bewährt.
+ Flexibel, da neue Bänder hinzugefügt werden können.
+ Konzept des Konfigurationsgraphen ist sehr nützlich.

Nachteile

– Kein realistisches Modell.

▶ Random-Access-Maschinen

Vorteile

+ (Vergleichsweise) realistisches Modell.
+ Sehr gut erweiterbar zur Modellierung von Parallelismus.

Nachteile

– Arbeiten nicht auf Strings.
– Schlecht erweiterbar.

▶ Boolesche Schaltkreise

Vorteile

+ Realistisches Modell.
+ Sehr genaue Aussagen möglich.
+ Wichtige untere Schranken bekannt.
+ Vielseitig einsetzbar.

Nachteile

– Definition von Klassen schwierig.
– Schwierig zu entwerfen und zu »programmieren«.

▶ Deskriptive Methoden

Vorteile

+ Abstraktion von konkreter Berechnung, Konzentration auf das Problem.
+ Formeln oft leichter zu finden als konkrete Algorithmen.
+ Die mächtigen Methoden der Logik können auf Berechnungsprobleme angewandt werden.

Nachteile

– Bei kleinen Klassen sehr technische Definitionen.
– Wissen über Logik und endliche Modelltheorie nötig.
– Liefern selten hocheffiziente Algorithmen

Zum Weiterlesen

[1] Alan M. Turing, On Computable Numbers With an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society,* 42(2):230–265, 1936

Noch ein Blick in die Arbeit lohnt sich, um Turings Definition von Maschinen mit der »modernen« Definition zu vergleichen.

[2] J. C. Shepherdsom and H. E. Sturgis, Computability of Recursive Functions, *Journal of the ACM*, 10(2):217–255, 1963

Die Arbeit, in der Random-Access-Maschinen eingeführt wurden.

[3] Pierre McKenzie and Klaus Wagner, The complexity of membership problems for circuits over sets of natural numbers, *Proceedings of* STACS *2003*, LNCS, Springer, 2607:571–582, 2003.

Diese Arbeit untersucht das Schaltkreisauswertungsproblem für verschiedene Arten von Gattern und Schaltkreise. Das »Standard«-Circuit-Value-Problem ist vollständig für P, aber Varianten sind sehr leicht, andere zu vermutlich unentscheidbar.
Die Abkürzung STACS steht für »Symposium on Theoretical Aspects of Computer Science,« eine der wichtigsten Konferenzen der Theoretischen Informatik. Die Abkürzung LNCS steht für »Lecture Notes in Computer Science.« In dieser Reihe werden die meisten Tagungsbände der Theoretischen Informatik veröffentlicht.

Teil IV

Messen und Vergleichen von Komplexität

Der gesunde Menschenverstand sagt uns, dass manche Probleme einfacher zu lösen sind als andere. Jedes Schulkind wird sofort bestätigen, dass es einfacher ist, zwei Zahlen zu addieren, als sie zu dividieren. Jeder Computer wird bestätigen, dass es einfacher ist, ein Gleichungssystem mit 10 Variablen zu lösen als eines mit 10 Billionen Variablen. Um zu verstehen, *warum* machen Problem einfacher sind als andere, brauchen wir Hilfsmittel, um die Komplexität von Problemen zu messen und zu vergleichen.

Wenn man etwas darüber nachdenkt, so sieht man schnell, dass es einen grundsätzlichen Unterschied bei den beiden obigen Beispielen gibt: Bei »10 Variablen versus 10 Billionen Variablen« handelt es sich zweimal um dasselbe Problem, nur bei unterschiedlichen Eingabegrößen. Da verwundert es wenig, dass man bei größeren Eingaben auch mehr Zeit braucht. Bei »Addieren versus Dividieren« handelt es sich um zwei unterschiedliche Probleme. Hier ist es schon viel schwieriger, sauber zu definieren, was es heißen soll, dass das eine Problem »prinzipiell« schwieriger sei als das andere – beispielsweise ist es sicherlich einfacher durch 1 zu dividieren als 1 zu addieren, trotzdem ist Division »an sich« schwieriger. Tatsächlich zu *beweisen* dass ein bestimmtes Problem einfacher ist als ein anderes ist bis jetzt leider nur in sehr wenigen Fällen gelungen.

4-1

Kapitel 4
Komplexitätsmaße

Die drei wichtigsten Maße: Zeit, Zeit und Zeit

Lernziele dieses Kapitels

1. Definitionen der grundlegenden Maße kennen
2. Die Komplexität von Problemen messen können
3. Das Konzept der Komplexitätsklasse verstehen
4. Diagonalisierungen durchführen können

4-2

Inhalte dieses Kapitels

Worum es heute geht

In der Einleitung zu diesem Skript habe ich Ihnen bereits die beiden Zahlen-auf-einer-Tafel-Probleme vorgestellt: Ich schreibe eine Zahlen auf eine Tafel und Sie müssen möglichst viel durchstreichen, so dass die Summe noch mindestens Tausend ist; oder beim zweiten Problem genau ein Tausend. Mittlerweile können wir diese Probleme etwas formaler fassen:

▶ Problem: Subset-Sum, Version A

Instanzen (Codes von) Mengen $N \subseteq \mathbb{N}$ und ein Wert t.

Lösungen Teilmengen $S \subseteq N$ mit $\sum_{s \in S} s \geq t$.

Maß $|S|$

Typ Minimierung

▶ Problem: Subset-Sum, Version A

Instanzen (Codes von) Mengen $N \subseteq \mathbb{N}$ und ein Wert t.

Lösungen Teilmengen $S \subseteq N$ mit $\sum_{s \in S} s = t$.

Maß $|S|$

Typ Minimierung

Es ist nicht sonderlich schwer zu sehen, dass Version A schnell gelöst werden kann, indem man die Zahlen sortiert. Deutlich schwieriger ist es schon, die *Platzkomplexität* der Version A zu bestimmen; wenn man die Tricks nicht kennt, so erscheint es reichlich erstaunlich, dass logarithmisch viel Platz ausreicht. Über Version B wissen wir hingegen sehr wenig; tatsächlich kennt niemand ein Polynomialzeitverfahren seiner Lösung. Das Erstaunliche hieran

ist weniger, dass wie kein solches Verfahren kennen (unsere Unwissenheit mag peinlich sein, wichtig ist sie nicht). Wichtig ist, dass die beiden Versionen des Problems *fast identisch* sind, aber sehr unterschiedliche Komplexitäten haben.

4-4

Viele Sichten auf ein Problem.

Die *Komplexität* eines Problems beschreibt, wie viele Ressourcen zu seiner Lösung nötig sind. Zwei unterschiedliche Problem können in Bezug auf ein Maß gleich viele Ressourcen benötigen, in Bezug auf ein anderes sehr unterschiedliche. *Es ist daher wichtig, nicht nur Zeit als Maß zu betrachten.*

Beispiel: Die besten Resultate für zwei NP-vollständige Probleme

Maß	VERTEX-COVER	CLIQUE
Zeit	exponentiell	exponentiell
Platz	polynomiell	polynomiell
Approximierbarkeit	gut (konstant)	sehr schlecht
Kernelgröße	sehr gut (linear)	sehr schlecht (keiner)

Beispiel: Die besten Resultate für zwei NL-vollständige Probleme

Maß	DISTANCE	TOURNAMENT-DISTANCE
Zeit	linear	linear
Platz	$\log^2$	$\log^2$
Approximierbarkeit	sehr schlecht	sehr gut

4.1 Zeit

4.1.1 Definition

4-5

Das wichtigste Maß: Zeit.

▶ Definition: Das Maß »Zeit«

Sei M eine Turing-Maschine mit Eingabealphabet Σ. Wir definieren $t_M : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ wie folgt:

$$t_M(x) = \begin{cases} s, & \text{falls eine Folge } C_{\text{init}}(x) \vdash C_1 \vdash \cdots \vdash C_s \\ & \text{existiert, in der } C_s \text{ Endkonfiguration ist,} \\ \infty, & \text{sonst.} \end{cases}$$

Wir definieren $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ durch

$$T_M(n) = \max_{x \in \Sigma^n} t_M(x).$$

4.1.2 Klassen

4-6

Eine »Zeitklasse« enthält Probleme, die sich in einer vorgegebenen Zeit lösen lassen.

▶ Definition: Zeitklasse

Sei $\mathcal{F} \subseteq \{f \mid f : \mathbb{N} \to \mathbb{N}\}$ eine Klasse von Zeitschranken. Die Klasse $\text{TIME}(\mathcal{F})$ enthält alle Sprachen L, so dass eine Turingmaschine M und ein $f \in \mathcal{F}$ existieren mit

1. $L(M) = L$ und
2. für alle $n \in \mathbb{N}$ gilt $T_M(n) \leq f(n)$.

Die zentralen Zeitklassen

4-7

▶ Definition

Sei poly(n) eine Abkürzung für $n^{O(1)}$, also für ein beliebiges Polynom.

$$\begin{aligned}
\text{LINTIME} &= \text{TIME}\big(O(n)\big),\\
\text{P} &= \text{TIME}\big(\text{poly}(n)\big),\\
\text{quasi-P} &= \text{TIME}\big(n^{O(\log n)}\big),\\
\text{E} &= \text{TIME}\big(2^{O(n)}\big),\\
\text{EXP} &= \text{TIME}\big(2^{\text{poly}(n)}\big),\\
\text{EE} &= \text{TIME}\big(2^{2^{O(n)}}\big),\\
\text{EEXP} &= \text{TIME}\big(2^{2^{\text{poly}(n)}}\big),\\
&\vdots\\
\text{ELEMENTARY} &= \text{TIME}(2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}).
\end{aligned}$$

Klassifikation einiger Probleme

4-8

- PARITY $\in$ LINTIME.
- DISTANCE $\in$ P.
- CVP $\in$ P.
- SAT $\in$ E.
- CLIQUE $\in$ E.
- SUCCINCT-PARITY $\in$ E.

✎ Zur Übung

Klassifizieren Sie:

- REACH,
- SUCCINCT-REACH,
- SUCCINCT-SUCCINCT-REACH.

Klassen für Funktionsprobleme

4-9

▶ Definition: Zeitklassen für Funktionen

Sei $\mathcal{F} \subseteq \{f \mid f\colon \mathbb{N} \to \mathbb{N}\}$ eine Klasse von Zeitschranken. Die Klasse FTIME($\mathcal{F}$) enthält alle Funktionen $g\colon \Sigma^* \to \Sigma^*$, so dass eine Turingmaschine M und ein $f \in \mathcal{F}$ existieren mit

1. M berechnet g und
2. für alle $n \in \mathbb{N}$ gilt $T_M(n) \leq f(n)$.

▶ Definition: Wichtige Zeitklassen für Funktionen

$$\begin{aligned}
\text{FLINTIME} &= \text{FTIME}\big(O(n)\big),\\
\text{FP} &= \text{FTIME}\big(\text{poly}(n)\big),\\
\text{FEXP} &= \text{FTIME}\big(2^{\text{poly}(n)}\big).
\end{aligned}$$

4.2 Platz

4.2.1 Definition

4-10 Die zweitwichtigste Ressource: Platz.

▶ Definition: Platzverbrauch einer Berechnung
Sei $C_1 \vdash_M C_2 \vdash_M \dots$ eine Berechnung einer DTM M. Sei h_i^t jeweils die Kopfposition auf Band t in Konfiguration C_i.

1. Der *Platzverbrauch der Berechnung auf Band t* ist
$$1 + \max\{h_1^t, h_2^t, \dots\} - \min\{h_1^t, h_2^t, \dots\} \in \mathbb{N} \cup \{\infty\}.$$
2. Der *Platzverbrauch der Berechnung* ist die Summe des Platzverbrauchs über alle *Arbeitsbänder*.

Merke
Die Eingabe- und Ausgabebänder zählen *nicht* mit, wenn der Platzverbrauch bestimmt wird.

▶ Definition: Platzverbrauch einer DTM
Sei M eine DTM mit Eingabealphabet Σ^*. Wir definieren $s_M \colon \Sigma^* \to \mathbb{N} \cup \{\infty\}$ durch
$$s_M(x) = \text{Platzverbrauch der mit } C_{\text{init}}(x) \text{ beginnenden Berechnung.}$$
Wir definieren $S_M \colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ als
$$S_M(n) = \max_{x \in \Sigma^n} s_M(x).$$

4.2.2 Klassen

4-11 Klassifikation von Problemen bezüglich des Platzverbrauchs.

▶ Definition: Platzklassen für Sprachen und Funktionen
Sei $\mathcal{F} \subseteq \{f \mid f \colon \mathbb{N} \to \mathbb{N}\}$ eine Klasse von Platzschranken.
Die Klasse $\text{SPACE}(\mathcal{F})$ ist die Menge aller Sprachen L, so dass eine DTM M und ein $f \in \mathcal{F}$ existieren mit

1. $L(M) = L$ und
2. für alle $n \in \mathbb{N}$ gilt $S_M(n) \leq f(n)$.

Die Klasse $\text{FSPACE}(\mathcal{F})$ enthält alle Funktionen $g \colon \Sigma^* \to \Sigma^*$, so dass eine Turingmaschine M und ein $f \in \mathcal{F}$ existieren mit

1. M berechnet g und
2. für alle $n \in \mathbb{N}$ gilt $S_M(n) \leq f(n)$.

4-12 Wichtige Platzklassen

▶ Definition
$$\begin{aligned}
\text{REG} &= \text{SPACE}\big(0\big) = \text{SPACE}\big(O(1)\big),\\
\text{L} &= \text{SPACE}\big(O(\log n)\big),\\
\text{L}^2 &= \text{SPACE}\big(O(\log^2 n)\big),\\
\text{polyL} &= \text{SPACE}\big(O(\log^{O(1)} n)\big),\\
\text{LINSPACE} &= \text{SPACE}\big(O(n)\big),\\
\text{PSPACE} &= \text{SPACE}\big(\text{poly}(n)\big),\\
\text{ESPACE} &= \text{SPACE}\big(2^{O(n)}\big),\\
\text{EXPSPACE} &= \text{SPACE}\big(2^{\text{poly}(n)}\big).
\end{aligned}$$

$$\text{FL} = \text{FSPACE}\big(O(\log n)\big).$$

4.3 Tiefe

4.3.1 Definition

Einschub: Wie man mit Schaltkreisen Sprachen akzeptiert.

4-13

Aufgrund ihrer Konstruktion können Schaltkreise nur Bitstrings fester Länge verarbeiten. Um nun *Sprachen* zu »entscheiden«, brauchen wir also *pro Eingabelänge einen Schaltkreis*.

▶ Definition: Schaltkreisfamilien

Sei $L \subseteq \{0,1\}^*$ eine Sprache. Eine *Schaltkreisfamilie für L* ist eine Folge $\mathcal{C} = (C_0, C_1, C_2, \dots)$ von Schaltkreisen, so dass

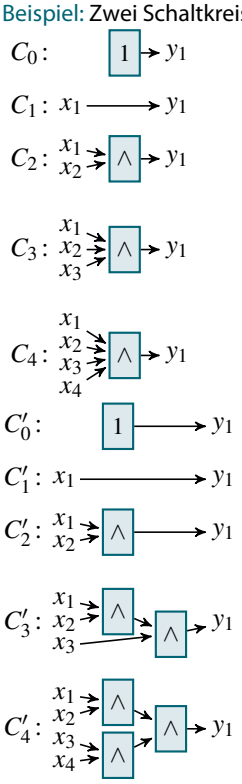
1. jedes C_n ein n-Eingaben-1-Ausgabe-Schaltkreis ist und
2. $C_{|x|}(x) = 1$ genau dann gilt, wenn $x \in L$.

Wir schreiben $\mathcal{C}(x)$ für $C_{|x|}(x)$.

Zwei sehr einfache Schaltkreisfamilien.

4-14

Beispiel: Zwei Schaltkreisfamilien für die Sprache $\{1\}^*$



Schaltkreisfamilien für Funktionen

4-15

Will man statt Sprachen *Funktionen* $f\colon \{0,1\}^* \to \{0,1\}^*$ mit Schaltkreisen berechnen, so haben die Schaltkreise *statt nur einem, viele Ausgabegatter.* Problematisch ist dabei, dass Wörter gleicher Länge (wie 00 und 11) auf Wörter unterschiedlicher Länge abgebildet werden könnten (zum Beispiel $f(00) = 1$ und $f(11) = 101$). Dies verbieten wir einfach.

▶ Definition: Schaltkreisfamilien für Funktionen

Sei $f\colon \{0,1\}^* \to \{0,1\}^*$ eine Funktion, so dass die Länge von $f(x)$ gleich ist für alle x gleicher Länge. Eine *Schaltkreisfamilie für f* ist eine Folge $\mathcal{C} = (C_0, C_1, C_2, \dots)$ von Schaltkreisen, so dass:

1. Jedes C_n ist ein n-Eingaben-m-Ausgabe-Schaltkreis mit $m = |f(0^n)|$ und
2. $C_{|x|}(x) = f(x)$ für alle x.

Wir schreiben auch $\mathcal{C}(x)$ für $f(x)$.

4-16

Uniformität: Die hässliche Seite von Schaltkreisfamilien

Schaltkreisfamilien können prinzipiell beliebig »wild« sein: C_{100} könnte Schach spielen, C_{101} könnte bestimmte Instanzen das Haltproblems lösen, C_{102} hingegen bestimmte Instanzen von SAT. Da Schaltkreisfamilien prinzipiell fast beliebige Funktionen von $\mathbb{N}$ in die Menge aller Schaltkreise sein können, sind die *allermeisten Schaltkreisfamilien nicht berechenbar*. Wir müssen also irgendwie sicherstellen, dass sich für gegebenes i der Schaltkreis C_i *»sehr leicht«* berechnen lässt.

▶ **Definition:** Uniforme Schaltkreisfamilien
Eine Schaltkreisfamilie $(C_i)_{i\in\mathbb{N}}$ heißt *(logspace-)uniform*, wenn eine Funktion $f \in \mathrm{FL}$ existiert mit $f(1^i) = \mathrm{code}(C_i)$.

4-17

Schaltkreistiefe = Zeit

Für einen Schaltkreis C entspricht seine *Tiefe* der Rechenzeit – denn so lange müssten wir warten, wäre er in Hardware implementiert.

▶ **Definition:** Tiefe und Größe einer Schaltkreisfamilie
Sei $\mathcal{C} = (C_i)_{i\in\mathbb{N}}$ eine Schaltkreisfamilie. Dann ist

- $\mathrm{depth}_{\mathcal{C}}(n)$ die Anzahl der Gatter außer den Eingabe und Ausgabegattern auf dem längsten Pfad in C_n und
- $\mathrm{size}_{\mathcal{C}}(n)$ die Anzahl der Gatter in C_n.

Beispiel
Für $n \geq 2$ gilt für die Schaltkreisfamilien aus dem Beispiel $\mathrm{depth}_{\mathcal{C}}(n) = 1$ und $\mathrm{depth}_{\mathcal{C}'}(n) = \lceil \log_2 n \rceil$.

4.3.2 Klassen

4-18

Klassifikation von Problemen gemäß ihre Schaltkreiskomplexität

▶ **Definition:** Schaltkreisklassen
Sei $i \in \mathbb{N}$. Die *Schaltkreisklassen* AC^i *und* FAC^i enthalten alle Sprachen $L \subseteq \{0,1\}^*$ beziehungsweise Funktionen $f\colon \{0,1\}^* \to \{0,1\}^*$, für die uniforme Schaltkreisfamilie $\mathcal{C}$ existieren mit

1. $\mathcal{C}$ entscheidet L beziehungsweise berechnet f,
2. $\mathrm{depth}_{\mathcal{C}}(n) = O(\log^i n)$ und
3. $\mathrm{size}_{\mathcal{C}}(n) = n^{O(1)}$.

Weitere Klassen ergeben Sie durch Variation dieser »Basisdefinition«:

- Für NC^i *und* FNC^i müssen alle Gatter kleine sein (Fan-In höchstens 2).
- Für TC^i *und* FTC^i sind hingegen auch Threshold-Gatter zugelassen.
- Für $\mathrm{Mod_kC}^i$ *und* $\mathrm{FMod_kC}^i$ sind Modulo-k-Gatter zugelassen.

4.4 Inklusionen zwischen Klassen

4-19

Warum uns Inklusionen zwischen Klassen interessieren.

Eine Menge Energie wird in der Komplexitätstheorie darauf verwandt, $\mathrm{C}_1 \subseteq \mathrm{C}_2$ zu beweisen (oder eben $\mathrm{C}_1 \not\subseteq \mathrm{C}_2$). Solche Inklusionen erlauben uns, einen Komplexitätsaspekt eines Problems zu untersuchen und es dann *automatisch auch auf eine andere Art lösen zu können.*

Beispiel
Man kann zeigen, dass $\mathrm{L} \subseteq \mathrm{AC}^1$ gilt. Falls wir also einen platzeffizienten Algorithmus für ein Problem finden können, so *ist das Problem automatisch parallelisierbar.*

Beispiel
Man kann zeigen, dass $\mathrm{NC}^1 \subseteq \mathrm{L}$. Weiter kann man zeigen, dass die Division in NC^1 liegt. Folglich kann man in logarithmischem Platz dividieren. (Versuchen Sie mal, das direkt zu zeigen…)

4.4.1 Triviale Inklusionen

Folgende Inklusionen folgen direkt aus den Definitionen. 4-20

$$\begin{aligned}
\mathrm{P} &\subseteq \mathrm{E} \subseteq \mathrm{EXP} \subseteq \mathrm{EE} \subseteq \mathrm{ELEMENTARY},\\
\mathrm{L} &\subseteq \mathrm{polyL} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPSPACE},\\
\mathrm{AC}^0 &\subseteq \mathrm{AC}^1 \subseteq \mathrm{AC}^2 \subseteq \cdots \subseteq \mathrm{AC},\\
\mathrm{NC}^0 &\subseteq \mathrm{NC}^1 \subseteq \mathrm{NC}^2 \subseteq \cdots \subseteq \mathrm{NC},\\
\mathrm{TC}^0 &\subseteq \mathrm{TC}^1 \subseteq \mathrm{TC}^2 \subseteq \cdots \subseteq \mathrm{TC},\\
\mathrm{NC}^i &\subseteq \mathrm{AC}^i \subseteq \mathrm{TC}^i.
\end{aligned}$$

4.4.2 Nichttriviale Inklusionen

Inklusionen zwischen Zeit und Platz. 4-21

▶ Lemma
Für jede vernünftige Funktion f mit $f(n) \geq \log(n)$ gilt

$$\mathrm{TIME}(f) \subseteq \mathrm{SPACE}(O(f)) \subseteq \mathrm{TIME}(2^{O(f)}).$$

Beweis. Für die erste Inklusion beachte man, dass man in $f(n)$ Schritten nicht mehr als $bf(n)$ Zellen beschreiben kann, wobei b die Anzahl der Bänder ist.
Für die zweite Inklusion sei $L \in \mathrm{SPACE}(f)$ via M. Dann ist die Größe des Konfigurationsgraphen von M bei Eingabe x mit Länge $n = |x|$ höchstens

$$S = \underbrace{O(}_{\text{Zustände}} \underbrace{n}_{\text{Eingabekopfpositionen}} \cdot \underbrace{f(n)^{O(1)}}_{\text{Arbeitsbandkopfpositionen}} \cdot \underbrace{2^{O(f(n))}}_{\text{Bandinhalt}}).$$

Wenn M nicht nach S Schritten angehalten hat, wird M nie halten. □

Inklusionen zwischen Zeit- und Platzklassen. 4-22

▶ Folgerung
Es gelten

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXPSPACE} \subseteq \mathrm{ELEMENTARY}$$

und

$$\mathrm{ELEMENTARY} = \mathrm{TIME}(2^{2^{\vdots^{2^n}}}) = \mathrm{SPACE}(2^{2^{\vdots^{2^n}}}).$$

Inklusionen zwischen Schaltkreisklassen. 4-23

▶ Lemma
Für alle $i \geq 0$ gilt $\mathrm{TC}^i \subseteq \mathrm{NC}^{i+1}$.

Beweis. Es genügt zu zeigen, dass wir ein Majority-Gatter mit n Eingängen durch einen Schaltkreis der Tiefe $O(\log n)$ simulieren können, der nur kleine Gatter besitzt.
Dies geht so: Wir summieren die n Eingabebits der Gatters. Dazu müssen wir n Zahlen addieren, wobei jede maximal Länge n hat (sogar nur Länge $\log n$). Dies kann man in Tiefe $O(\log n)$ erreichen, siehe Übung 4.6. Überprüfe am Ende, ob das erste Bit 1 ist. □

▶ Folgerung

$$\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{TC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{AC}^1 \subseteq \mathrm{TC}^1 \subseteq \mathrm{NC}^2 \subseteq \ldots$$

4.4.3 Echte Inklusionen

4-24

Echte Inklusionen zwischen Komplexitätsklassen

Intuition

Wenn wir mehr Zeit haben, *sollten* wir in der Lage sein, mehr Probleme zu lösen. Andererseits sieht man leicht (siehe Übungsaufgabe 4.7), dass beispielsweise eine Verdopplung der erlaubten Zeit *keinen Effekt hat.*

▶ Satz: Zeithierarchiesatz
Sei $f(n) \geq n$ *eine sinnvolle Funktion. Dann gilt* $\mathrm{TIME}(f) \subsetneq \mathrm{TIME}(f(n)^3)$.

▶ Satz: Platzhierarchiesatz
Sei $f(n) \geq \log n$ *eine sinnvolle Funktion und* $f \in o(g)$. *Dann gilt* $\mathrm{SPACE}(f) \subsetneq \mathrm{SPACE}(g)$.

Beweis des Zeithierarchiesatzes. Betrachte die folgende *Diagonalisierungssprache*:

$$D_f = \{\mathrm{code}(M) \mid M \text{ akzeptiert } \mathrm{code}(M) \text{ nicht in } f(|\mathrm{code}(M)|) \text{ Schritten}\}.$$

Wir behaupten $D_f \in \mathrm{TIME}(f(n)^3) \setminus \mathrm{TIME}(f)$:

- $D_f \in \mathrm{TIME}(f(n)^3)$ via einer *Simulation von* M auf der Eingabe $\mathrm{code}(M)$ für maximal $f(|\mathrm{code}(M)|) = f(n)$ Schritte.
- *Nehmen wir an,* $D_f \in \mathrm{TIME}(f)$ gelte via M. Sei $x = \mathrm{code}(M)$. Dann gilt:
  1. Wenn M das Wort x akzeptiert, gilt $x \in D_f$. Also akzeptiert M das Wort x nicht in $f(|x|)$ Schritten (ein *Widerspruch*).
  2. Wenn M das Wort x nicht akzeptiert, gilt $x \notin D_f$. Dann braucht M entweder mehr als $f(|x|)$ Schritte (ein *Widerspruch*) oder M akzeptiert x (noch ein *Widerspruch*).

  Die Widersprüche zeigen, dass $D_f \notin \mathrm{TIME}(f)$. □

4-25

Echte Inklusionen zwischen Zeit- und Platzklassen.

▶ Lemma

$$\mathrm{LINTIME} \subsetneq \mathrm{P} \subsetneq \mathrm{E} \subsetneq \mathrm{EXP} \subsetneq \mathrm{EE} \subsetneq \mathrm{ELEMENTARY},$$
$$\mathrm{L} \subsetneq \mathrm{polyL} \subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE}.$$

▶ Lemma
In der Inklusionskette

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE}$$

ist mindestens eine Inklusion echt, es gilt also $\mathrm{L} \neq \mathrm{P}$ *oder* $\mathrm{P} \neq \mathrm{PSPACE}$ *(wir wissen nur nicht, welches).*

Zusammenfassung dieses Kapitels

▶ Klassen und ihre Inklusionsstruktur

$$\begin{aligned}
\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{TC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{AC}^1 \subseteq \mathrm{TC}^1 \subseteq \mathrm{NC}^2 \subseteq \ldots,\\
\mathrm{LINTIME} \subsetneq \mathrm{P} \subsetneq \mathrm{E} \subsetneq \mathrm{EXP} \subsetneq \mathrm{EE} \subsetneq \mathrm{ELEMENTARY},\\
\mathrm{L} \subsetneq \mathrm{polyL} \subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE},\\
\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXPSPACE} \subseteq \mathrm{ELEMENTARY}\\
= \mathrm{TIME}(2^{2^{\vdots^{2^n}}}) = \mathrm{SPACE}(2^{2^{\vdots^{2^n}}}).
\end{aligned}$$

Zum Weiterlesen

[1] J. Hartmanis and R. E. Stearns, On the computational complexity of algorithms, *Transactions of the* ACM, 117:285–306, 1965

[2] J. Hartmanis, P. L. Lewis II, and R. E. Stearns, Hierarchies of memory-limited computations, *Proceedings of the Sixth Annual* IEEE *Symposium on Switching Circuit Theory and Logic Design,* 179–190, 1965

Diese beiden Artikel sind die »Pioniere« betreffend die Definition von Zeit- und Platzkomplexität. Die Hierarchiesätze werden dort bewiesen. Beachten Sie, dass diese Artikel »ziemlich alt sind« (Quicksort wurde 1962 »entdeckt«).

Übungen zu diesem Kapitel

Für die folgenden Aufgaben benötigen wir eine Definition:

▶ Definition: Zeitkonstruierbar

Eine (totale) Funktion $f\colon \mathbb{N} \to \mathbb{N}$ heißt *zeitkonstruierbar* wenn es eine Turing-Maschine M gibt mit $f = T_M$.

Übung 4.1 Einfache zeitkonstruierbare Funktionen, einfach

Zeigen Sie, dass $f(n) = n+1$ und $g(n) = \lfloor 1{,}5n \rfloor$ zeitkonstruierbar sind.

Übung 4.2 Nichtzeitkonstruierbare Funktionen, mittel

Zeigen Sie, dass es eine nichtzeitkonstruierbare Funktion $f \in O(n)$ gibt.

Übung 4.3 Wir können annehmen, dass Maschinen immer anhalten, mittel

Sei M eine Turingmaschine und f zeitkonstruierbar. Nehmen wir weiter an, dass für alle $x \in \Sigma^*$ entweder $t_M(x) \leq f(|x|)$ oder $t_M(x) = \infty$ gilt. Zeigen Sie, dass eine Turingmaschine M' existiert mit $L(M) = L(M')$ und $T_{M'}(n) \leq f(n)+n$ für alle $n \in \mathbb{N}$.

Tipp: Machen Sie sich zunächst klar, warum die Bedingunge $T_{M'}(n) \leq f(n)+n$ bedeutet, dass M' auf allen Eingaben anhält. Wie können Sie dies garantieren? Nutzen Sie weitere Bänder.

Übung 4.4 Sublogarithmischer Platz, mittel

Zeigen Sie $\mathrm{SPACE}(\log\log n) \neq \mathrm{REG}$.

Tipp: Betrachten Sie

$$L = \{\mathrm{code}(1)\#\mathrm{code}(2)\#\ldots\#\mathrm{code}(n) \mid n \in \mathbb{N}\}.$$

Übung 4.5 Sublogarithmischer Platz, schwer

Zeigen Sie $\mathrm{SPACE}(\log\log\log n) = \mathrm{REG}$.

Tipp: Erweitern Sie das Argument aus dem Kapitel über Zwei-Wege-Automaten im Skript zur Theoretischen Informatik, in dem mit Hilfe von Crossing-Sequenzen gezeigt wird, dass sich jeder Zwei-Wege-Automat durch einen Ein-Wege-Automat simulieren lässt. Erweitern Sie aber nun den Begriff des Zustands so, dass auch die Inhalte der Arbeitsbänder eingeschlossen sind. Sie sollten für jede Wortlänge einen Automaten erhalten, der doppelt-exponentiell viele Zustände im Platzverbrauch der Maschine hat. Argumentieren Sie nun weiter wie folgt: Betrachte die kürzeste Eingabe x, für die die Maschine $S > 0$ Platz benötigt. Dann muss der Automat für alle Präfixe dieser Eingabe in einem anderen Zustand sein, denn sonst gäbe es ein kürzeres Wort, für das auch nur S Platz benötigt würde (warum?). Argumentieren Sie, dass Sie nun einen Widerspruch bezüglich der Länge von x haben.

Übung 4.6 Iterierte Addition in NC^1, schwer

Die *iterierte Addition* ist das Funktionsproblem f-ITERATED-ADDITION, das Strings der Form

$$\mathrm{code}(x_1)\#\mathrm{code}(x_2)\#\ldots\#\mathrm{code}(x_n)$$

auf $\sum_{i=1}^{n} x_i$ abbildet. Zeigen Sie, dass f-ITERATED-ADDITION $\in \mathrm{FNC}^1$.

Tipp: Lesen Sie das Kapitel »Grundrechenarten« im Skript zu »Parallelverarbeitung«.

Übung 4.7 Linearer Speedup, schwer

Beweisen Sie den folgenden *Linearen-Speedup-Satz: Sei* $L \in \mathrm{TIME}(f)$. *Dann gilt für alle* $\varepsilon > 0$, *dass* $L \in \mathrm{TIME}(f')$ *mit* $f'(n) = \varepsilon f(n) + n + 2$.

Tipp: Benutzen Sie den »Register-Trick«. Alle paar Jahre werden Computer schneller, weil man statt 8-Bit-Computern irgendwann 16-Bit-Computer, dann 32-Bit-Computer, dann 64-Bit-Computer und so weiter eingeführt hat. Übersetzt auf Turing-Maschinen bedeutet dies: Führen Sie ein größeres Alphabet ein, bei dem ein Symbol in diesem Alphabet viele Symbole des Originalalphabets kodiert. Die Maschine läuft dann einmal über die Eingabe, um diese zu »komprimieren« (daher der $n+2$ Part). Danach können viele Schritte der alten Maschine durch einen Schritt der neuen Maschine ersetzt werden.

Gut ist diese Simulation, wenn die zu simulierende Maschine immer viele Symbole hintereinander weg liest; problematisch ist es, wenn sie an einer »Wortgrenze« immer hin- und herwechselt. Um dies zu verhindern, muss die simulierende Maschine immer zwei nebenanderstehende Symbol lesen.

Übung 4.8 Platzhierarchiesatz, schwer

Beweisen Sie den Platzhierarchiesatz.

Tipp: Gehen Sie genauso vor wie beim Zeithierarchiesatz. Der Unterschied liegt lediglich in der Simulation, wo Sie argumentieren müssen, dass g Platz ausreicht, um f Platz zu simulieren, wenn g echt schneller wächst als f.

5-1

Kapitel 5
Vergleich von Komplexität
Der Färbbarkeitswolf im Erfüllbarkeitsschafpelz

5-2

Lernziele dieses Kapitels

1. Genaues Verständnis von Reduktionen und Vollständigkeit
2. Eigene Vollständigkeitsbeweise führen können
3. Eine Reihe vollständiger Probleme kennen

Inhalte dieses Kapitels

Worum es heute geht

Wenn Sie mal ein Kind haben sollten, fragen Sie bitte keinen Komplexitätstheoretiker und auch keine Komplexitätstheoretikerin um Rat betreffend mögliche Namen für ihren Sprössling. Komplexitätstheoretiker tendieren dazu, reichlich verwirrliche Namen für die Dinge zu wählen, die ihnen lieb und teuer sind. Die Namen der Komplexitätsklassen aus dem vorigen Kapitel sind gute Beispiele. Den Unterschied zwischen E und EXP muss man schon suchen; schlimmer noch: über die Jahre ändert sich, was womit gemeint ist. Früher hieß P mal PTIME, wohingegen E früher EXPTIME hieß (und nicht etwa ETIME).

Sind die meisten in der Komplexitätstheorie genutzten Klassennamen lediglich verwirrlich, sind die drei zentralen Begriff »Reduktion«, »Schwere« und »Vollständigkeit« katastrophal schlecht gewählt: Sie verwirren Anfängerinnen und Anfänger in der Regel dermaßen, dass sie die Lust an der Materie verlieren. Eine Reduktion macht ein Problem nicht kleiner, ein schweres Problem muss nicht schwer zu lösen sein und ein vollständiges Problem ist nicht das Gegenteil eines unvollständigen Problems. Natürlich ist es mittlerweile zu spät, die Dinge neu zu benennen, genausowenig wie wir die lustigen, aber ebenfalls verwirrlichen Namen »Körper« oder »Ring« in der Mathematik noch ändern können.

Was wären aber bessere Bezeichungen gewesen? Statt einer »Reduktion« zwischen Problemen sollte man besser von einem »Vergleich« von Problemen sprechen: Wenn A auf B reduziert werden kann, bedeutet dies »die Komplexität von A ist nicht höher als die von B«. Statt zu sagen »A ist schwer für eine Klasse C« sollte man lieber sagen, »A ist hilfreich für eine Klasse C«: Wenn A schwer ist für C, so kann man alle Probleme in C lösen, wenn man denn A gelöst bekommt. Schließlich könnte man statt von »einem vollständigen Problem für eine Klasse C« von einem »hilfreichen Problem in der Klasse C« sprechen: Der einzige Unterschied zwischen »schwer« und »vollständig« ist, dass vollständige Probleme in der Klasse sein müssen, für die sie schwer sind.

5.1 Einleitung

5-4

Relative statt absolute Komplexität.

»Absolute« Komplexität

Die Maße »Zeit«, »Platz«, »Tiefe« und so weiter sind »absolute« Maße. Wir nutzen sie für Aussagen wie »Problem Foo hat Zeitkomplexität $\Theta(bar)$ und Platzkomplexität $\Theta(blub)$.« Leider sind gerade die unteren Schranken fast nie bekannt.

»Relative« Komplexität

Bei der *relativen Komplexität* eines Problems *vergleichen* wir es »lediglich« mit einem anderen Problem. Die *absolute Komplexität* beider Problem muss dabei gar nicht klar sein.

5.2 Reduktionen

5.2.1 Grundideen

5-5

Was ist eine Reduktion?

Idee

Reduktionen erlauben uns, die Komplexität zweier Probleme zu vergleichen. Ziel ist es, Aussagen der folgenden Form zu zeigen:

> Problem A kann leicht gelöst werden *unter der Annahme, dass Problem B leicht gelöst werden kann.*

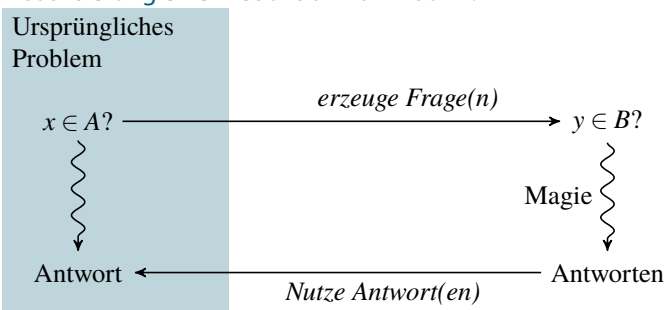
Die Idee ist, hierfür die Frage »Ist $x \in A$?« in Fragen der Form »Ist ein $y \in B$?« umzuwandeln. Wir sagen dann, *A reduziert sich auf B.*

Merke

Bei einer Reduktion ist es unerheblich, *wie schwer B ist* und wir *versuchen nicht, B zu lösen*. Vielmehr *nehmen wir einfach an B* wäre auf magische Weise beliebig schnell lösbar.

5-6

Visualisierung einer Reduktion von A auf B.



Reduktionen unterscheiden sich darin, wie schwierig die Fragen zu erzeugen sind, wie viele Fragen wir stellen dürfen und was wir mit den Antworten machen dürfen.

5-7

✎ Zur Übung

Geben Sie möglichst einfache Reduktion an

1. von REACH auf DISTANCE,
2. von DISTANCE auf REACH und
3. von REACH auf SUCCINCT-REACH.

Achten Sie auf die Richtung!

5.2.2 Logspace-Many-One-Reduktionen

Die Logspace-Many-One-Reduktionen

5-8

Steckbrief

Was darf eine Logspace-Many-One-Reduktion, die die Frage »Ist $x \in A$?« beantworten soll?

Anzahl Fragen Nur eine Frage »Ist $y \in B$?« darf gestellt werden.

Zeit-/Platzschranke Nur logarithmisch viel Platz zur Berechnung von y.

Erlaubte Auswertung Die Antwort auf »Ist $y \in B$?« muss übernommen werden.

▶ **Definition**

Seien $A, B \subseteq \Sigma^*$ Sprachen. Wir sagen, A *lässt sich logspace-many-one-reduzieren* auf B, geschrieben $A \leq_{\mathrm{m}}^{\log} B$, wenn

- eine Funktion $f \in \mathrm{FL}$ existiert, so dass
- für alle $x \in \Sigma^*$ gilt $x \in A \iff f(x) \in B$.

Ein wichtiges Lemma

5-9

▶ **Lemma:** Das Logspace-Kompositionslemma

Die Klasse FL *ist unter Kompositionen abgeschlossen, das heißt, aus* $f, g \in \mathrm{FL}$ *folgt* $f \circ g \in \mathrm{FL}$.

▶ **Folgerung**

Sei $A \leq_{\mathrm{m}}^{\log} B$ *und* $B \leq_{\mathrm{m}}^{\log} C$. *Dann gilt* $A \leq_{\mathrm{m}}^{\log} C$.

Beweisskizze. Sei x eine Eingabe. Wir wollen $f(g(x))$ in logarithmischem Platz berechnen. Wir können *nicht* einfach erst $g(x)$ berechnen und dann f auf das Ergebnis anwenden, da $g(x)$ *viel zu lang ist, als dass wir es zwischenspeichern könnten.* Der Trick ist, ein »virtuellen« Arbeitsband zu nutzen: Wir beginnen die Berechnung von f auf Eingabe $g(x)$. Wir kennen $g(x)$ gar nicht, tun aber einfach so. Früher oder später wird die Berechnung nicht weitergehen, wenn wir nicht ein bestimmtes Bit von $g(x)$ kennen. An dieser Stelle starten wir eine Simulation von g auf der (echten) Eingabe x. Die Ausgabe der Simulation (welche $g(x)$ wäre) wird *weggeschmissen mit Ausnahme des uns interessierenden Bits.* □

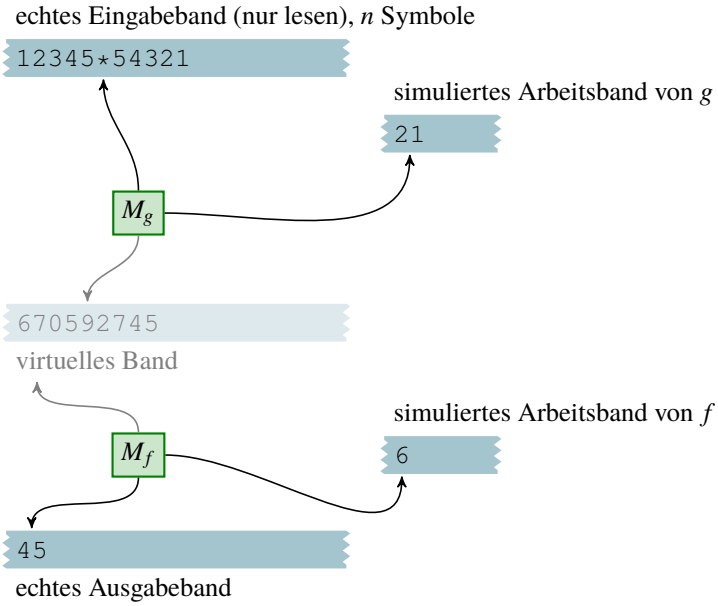
Beispiel für die Komposition zweier logspace-berechenbaren Funktionen.

5-10

Sei $g \in \mathrm{FL}$ die Multiplikation, also $g = f\text{-MULITPLICATION}$, und sei M_g eine Logspace-Maschine, die g berechnet. Sei f die Funktion, die eine Zahl auf die Summe ihrer Ziffern abbildet, und sei M_f eine Logspace-Maschine, die f berechnet. Dann bildet die Berechnung $f \circ g$ das Wort `12345*54321` auf `45` ab, da $12345 \cdot 54321 = 670592745$ und die Quersumme von 670592745 gerade 45 ist.

Visualisierung des »Virtuellen Bands.«

5-11



5.2.3 Polynomialzeit-Turing-Reduktionen

5-12

Die Polynomialzeit-Turing-Reduktion

Steckbrief
Was darf eine Polynomialzeit-Turing-Reduktion, die die Frage »Ist $x \in A$?« beantworten soll?

Anzahl Fragen Beliebig viele Fragen der Form »Ist $y \in B$?« dürfen adaptiv gestellt werden.

Zeit-/Platzschranke Es darf polynomiell viel Zeit genutzt werden.

Erlaubte Auswertung Die Antworten dürfen beliebig in die Berechnung einfließen.

Eine genaue Definition wird später gegeben, wenn wir Orakel-Turing-Maschinen behandeln.

5.2.4 Projektionsreduktionen

5-13

Die schwächsten Reduktionen: Projektionen

Steckbrief
Was darf eine Projektionsreduktion, die die Frage »Ist $x \in A$?« beantworten soll?

Anzahl Fragen Eine Frage der Form »Ist $y \in B$?« darf gestellt werden.

Zeit-/Platzschranke Jedes Bit von y muss sich aus einem Bit von x direkt ergeben.

Erlaubte Auswertung Die Antwort muss übernommen werden.

▶ **Definition: Projektion**
Eine *Projektion* ist eine Schaltkreisfamilie $(C_0, C_1, C_2, \dots)$, so dass jeder Schaltkreis Tiefe maximal 1 hat und nur Eingabe-, Ausgabe-, Negations- und Konstantengatter enthält.

▶ **Definition: Projektionsreduktionen**
Seien $A, B \subseteq \Sigma^*$ Sprachen. Wir sagen, A *lässt sich mit einer Projektion reduzieren* auf B, geschrieben $A \leq_{\text{proj}} B$, wenn

- es eine logspace-uniforme Projektion $\mathcal{C}$ gibt, so dass
- für alle $x \in \Sigma^*$ gilt $x \in A \iff \mathcal{C}(x) \in B$.

5.2.5 Abschlusseigenschaften

5-14

Reduktionsabschlüsse.

▶ **Definition: Reduktionsabschluss**
Sei C eine Sprachklasse. Dann ist der (logspace-many-one) *Reduktionsabschluss* von C die Klasse $\mathrm{R}_{\mathrm{m}}^{\log}(C) = \{A \mid A \leq_{\mathrm{m}}^{\log} B \in C\}$.

▶ **Definition**
Eine Sprachklasse C heißt *abgeschlossen* unter (logspace-many-one) Reduktionen, wenn $C = \mathrm{R}_{\mathrm{m}}^{\log}(C)$.

▶ **Lemma**
Die Klassen L, P, PSPACE *und* EXP *sind abgeschlossen unter Reduktionen.*

Beweis. Siehe Übung 5.2. □

5.3 Schwere und Vollständigkeit

5.3.1 Idee

Auf der Suche nach wirklich schweren Problemen.

5-15

$A \leq_{\mathrm{m}}^{\log} B$ bedeutet, dass B »mindestens so schwer wie A« ist. Ein wirklich schwieriges Problem wird nicht nur schwerer als A, sondern auch schwerer als *viele andere Probleme* sein. Es kann sogar schwerer sein als *alle Probleme einer Klasse* und wir sagen dann *es ist schwer für diese Klasse*.

Beispiel

- Das Halteproblem ist schwer für L.
- Das Halteproblem ist auch schwer für P.
- Das Halteproblem ist auch schwer für ELEMENTARY.

Schwer, schwerer, am schwersten

5-16

Probleme wie das Halteproblem sind für praktisch jede Klasse schwer. Interessanter sind Probleme, die schwer sind für eine Klasse und trotzdem *in der Klasse* liegen: die *vollständigen Probleme.*

Wenn Probleme Menschen wären

Stellen wir uns vor, Probleme wären Menschen und $A \leq B$ bedeutet »A ist nicht größer als B«. Ein *schweres* Problem für eine Klasse von Leuten ist eine beliebige Person, die mindestens so groß wie alle in der Klasse ist. Die *vollständigen* Probleme für eine Klasse sind genau die größten Personen in der Klasse (es kann mehrere geben).

5.3.2 Definitionen

Die Definitionen von Schwere und Vollständigkeit

5-17

▶ Definition

Sei C eine Sprachklasse und A ein Problem. Wie sagen

1. A ist *schwer für C (unter Logspace-Many-One-Reduktionen)* oder *C-schwer,* wenn $C \subseteq \mathrm{R}_{\mathrm{m}}^{\log}(\{A\})$ und
2. A ist *vollständig für C (unter Logspace-Many-One-Reduktionen)* oder *C-vollständig,* wenn A schwer ist für C und $A \in C$.

Ein praktisches Lemma

5-18

▶ Lemma: Das Kollapslemma

Sei $C \subseteq D$, seien C und D unter Reduktionen abgeschlossen und sei A schwer für D. Dann folgt aus $A \in C$, dass $C = D$.

Beweis. Trivial (folgt direkt aus den Definitionen). □

▶ Folgerung

Wenn ein PSPACE*-vollständiges Problem in* P *ist, so folgt* P = PSPACE.

▶ Folgerung

Die Klasse L *enthält keine* PSPACE*-schweren Probleme.*

Beweis. Nach dem Platzhierarchiesatz gilt L ≠ PSPACE. □

5.3.3 P-Vollständigkeit

5-19

Wie man Vollständigkeit beweist

Es gibt zwei Methoden zu zeigen, dass ein Problem B schwer ist für eine Klasse C:

Bootstrapping Man konstruiert für jedes Problem $A \in C$ eine Reduktion von A auf B.

Reduktionsmethode Wenn ein C-vollständiges Problem X bekannt ist, zeige $X \leq_{\mathrm{m}}^{\log} B$.

5-20

Ein vollständiges Problem für P.

▶ **Satz**

CVP *ist* P*-vollständig.*

Beweisplan. CVP $\in$ P ist leicht zu sehen. Wir zeigen Schwere durch Bootstrapping. Sei $A \in$ P via M, wir zeigen $A \leq_{\mathrm{m}}^{\log}$ CVP: Sei x eine Eingabe, für die wir die Frage »Ist $x \in A$?« beantworten sollen. Wir müssen (in logarithmischem Platz) einen Schaltkreis C bauen, so dass

1. wenn $x \in A$, so wertet C zu 1 aus, und
2. wenn $x \notin A$, so wertet C zu 0 aus.

Der Schaltkreis bestehen aus einer Reihe von *Schichten*. Jede Schicht erhält eine (kodierte) Konfiguration von M als Eingabe und liefert die Nachfolgekonfiguration als Ausgabe. Bei der letzten Schicht überprüfen wir einfach, ob eine akzeptierende Endkonfiguraiton erreicht wurde. □

5-21

Setting the Stage: Die im Folgenden verwendeten Namen.

Wir legen zunächst einige wichtige Bezeichner fest:

- A Die Sprache in P, die wir auf CVP reduzieren wollen.
- M Eine Polynomialzeitmaschine, die A entscheidet.
- p Ein Polynom, das die Laufzeit von M beschränkt.
- x Eine Eingabe, für die die Reduktionsfunktion die Frage »Ist $x \in A$?« in die Frage »Ist $\mathrm{code}(C) \in$ CVP?« umwandeln soll.
- n Die Länge von x, also $n = |x|$.
- C Der Schaltkreis, der von der Reduktion berechnet wird.

5-22

Von der Maschine zum Schaltkreises.

Wir beginnen mit ein paar Vereinfachungen:

- Die Maschine M habe nur *ein Band*.
- Das Band der Maschine M sei *einseitig beschränkt*.
- Es gibt *genau eine* akzeptierende Endkonfiguration.
- Bei dieser Endkonfiguration ist der *Kopf am Anfang*.

✎ **Zur Diskussion**

Wie kann man jeweils Maschinen, die eine der obigen Eigenschaften nicht haben, umwandeln, so dass die Eigenschaften erfüllt werden?

5-23

Grober Aufbau des Schaltkreises.

Wir bauen den Schaltkreis C wie folgt:

- Er besteht aus $p(n)$ *Schichten*.
- An den Ausgabegattern der k-ten Schicht liegt *kodiert die k-te Bandkonfiguration an*.
- An den Eingabegattern der k-ten Schicht liegt *kodiert die $(k-1)$-ste Bandkonfiguration an*.

Die Kodierung einer Bandkonfiguration funktioniert wie folgt: Um eine Bandzelle zu kodieren, benutzen wir mehrere Leitungen. Ist das Bandalphabet Γ, so können in einer Zelle $|\Gamma|$ verschiedene Symbole stehen; wir benutzen deshalb $\log|\Gamma|$ Leitungen zur Kodierung des Zelleninhalts. Zusätzlich benutzt man noch pro Zelle eine Leitung, um zu signalisieren, ob der Kopf dort ist. Schließlich benutzen wir noch $\log|Q|$ Leitungen, um den aktuellen Zustand der Turingmaschine zu kodieren.

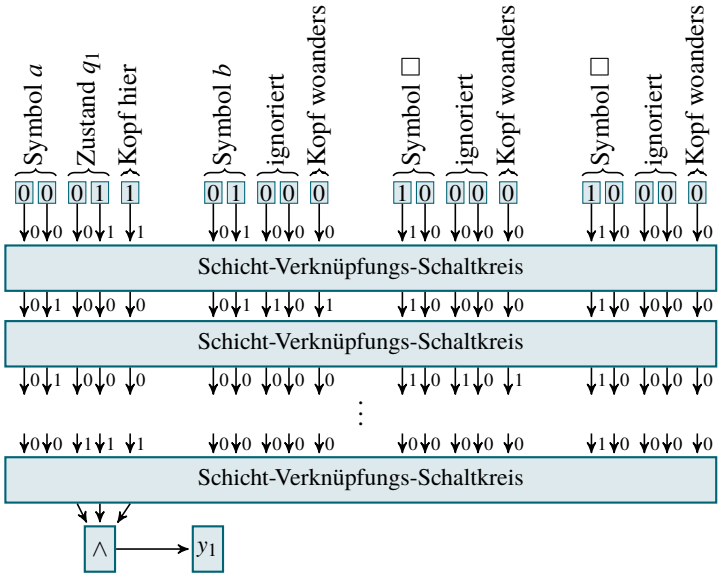
Eine Beispielsituation.

5-24

- Die Symbole des Bandalphabets $\Gamma = \{a, b, \square\}$ werden kodiert durch 00, 01, 10.
- Es gibt vier Zustände $\{q_0, q_1, q_2, q_3\}$, kodiert durch 00, 01, 10 und 11.
- Der initiale Bandinhalt ist $ab\square\square$, der initiale Zustand ist q_1 und der Kopf ist initial auf erstem Zeichen (dem a).
- Nach einem Schritt ist der Bandinhalt $bb\square\square$, der Zustand ist q_2 und der Kopf auf dem zweiten Zeichen.
- Nach zwei Schritten ist der Bandinhalt $ba\square\square$, der Zustand ist q_2 und der Kopf auf dem dritten Zeichen.
- Nach $p(n) - 1$ Schritten ist der Bandinhalt $aaa\square$, der Zustand ist q_3 und der Kopf auf dem ersten Zeichen.
- Akzeptierender Zustand ist genau q_3.

Visualisierung der Schichtenstruktur des Schaltkreises.

5-25



Was leistet der Schaltkreis?

5-26

- Betrachtet man die Leitungen, die in die *erste* Schicht hineinführen, so kodieren sie genau die Anfangskonfiguration.
- Betrachtet man die Leitungen, die in die *zweite* Schicht hineinführen, so kodieren sie genau den Bandinhalt nach einem Schritt.
- Betrachtet man die Leitungen, die in die *dritte* Schritt hineinführen, so kodieren sie genau den Bandinhalt nach zwei Schritten.
- Betrachtet man die Leitungen, die in die $p(n)$*-te* Schritt hineinführen, so kodieren sie genau den Bandinhalt der Endkonfiguration.
- Schaltet man also noch einen winzigen Schaltkreis nach, der überprüft, ob diese Endkonfiguration akzeptierend ist, so leistet der Schaltkreis das Gewünschte:
  Er wertet zu 1 aus genau dann, wenn die Maschine die Eingabe akzeptiert.

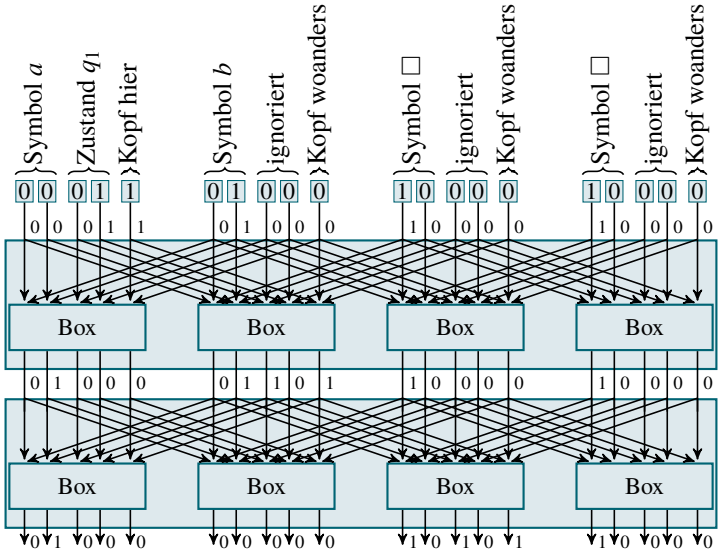
Was in den Schicht-Verknüpfungs-Schaltkreisen passiert.

5-27

Wir wollen einen (flachen) Schaltkreis angeben, der *Schicht i mit Schicht $i+1$ verknüpft*. Dieser Schaltkreis besteht aus $p(n)$ vielen *Boxen*. Jede Box kümmert sich darum, den Bandinhalt einer bestimmten Zelle z zu berechnen. Dazu muss sie lediglich *die Inhalte der Zelle z in der vorherigen Schicht kennen, sowie die Inhalte der Zelle links und rechts davon.*

5-28

Aufbau eines Schicht-Verknüpfungs-Schaltkreises.



5-29

Wie schwierig ist der Schaltkreis zu bauen?

Was wir erreicht haben:

- Zu jeder beliebigen Eingabe x können wir einen Schaltkreis konstruieren, der genau dann zu 1 auswertet, wenn M das Wort x akzeptiert.

Was fehlt:

- Wie schwierig ist es, den Schaltkreis zu konstruieren?

Antwort hierauf:

- Dies ist einfach, da *alle Boxen identisch sind*.
- Der Schaltkreis besteht im Wesentlichen aus $p(n)^2$ Boxen, die in einfacher Weise verdrahtet sind.
- Es ist leicht, in logarithmischem Platz in einer Schleife $p(n)^2$ Boxen (fester Größe!) auszugeben.
- Die Verdrahtung ist etwas fummeliger, aber auch nicht schwierig.

Zusammenfassung dieses Kapitels

5-30

▶ Reduktionen

Reduktionen übersetzen die Frage »Ist $x \in A$?« in die Frage(n) »Ist $y \in B$?«. Wir haben drei Arten von Reduktionen betrachtet (nach Mächtigkeit sortiert):

1. Polynomialzeit-Turing-Reduktionen ($\leq_{\mathrm{T}}^{\mathrm{p}}$),
2. Logspace-Many-One-Reduktionen ($\leq_{\mathrm{m}}^{\log}$),
3. Projektionsreduktionen ($\leq_{\mathrm{proj}}$),

▶ Abschluss, Schwere, Vollständigkeit

1. Für eine Klasse C ist $\mathrm{R}_{\mathrm{m}}^{\log}(C) = \{A \mid A \leq_{\mathrm{m}}^{\log} B \in C\}$.
2. Ein Problem A heißt *schwer für* C wenn $C \subseteq \mathrm{R}_{\mathrm{m}}^{\log}(\{A\})$.
3. Ein Problem A heißt *vollständig für* C wenn $A \in C$ und A ist schwer für C.

▶ Kollapslemma

Wenn $C \subseteq D$ und C und D abgeschlossen sind unter Reduktion und A vollständig ist für D, so folgt aus $A \in C$, dass $C = D$.

▶ CVP ist P-vollständig

Das Circuit-Value-Problem ist vollständig für P.

Übungen zu diesem Kapitel

Übung 5.1 Logspace-many-one-Reduktionen sind in L nutzlos, leicht

Zeigen Sie, dass für alle $A \in \mathrm{L}$ gilt $A \leq_{\mathrm{m}}^{\log} \{1\}$.

Übung 5.2 Abschlusseigenschaften, mittel

Zeigen Sie, dass die folgenden Klassen unter logspace-Many-One-Reduktionen abegschloosen sind:

1. L (Tipp: Nutzen Sie das Kompositionslemma).
2. P (Tipp: Nutzen Sie, dass $\mathrm{FL} \subseteq \mathrm{FP}$).
3. PSPACE.
4. EXP.

Übung 5.3 PSPACE-Schwere, schwer

Zeigen Sie, dass SUCCINCT-REACH schwer ist für PSPACE. (Es ist sogar vollständig für die Klasse, aber das soll nicht gezeigt werden.)

Tipp: Die Frage »Ist $x \in A$?« reduziert sich auf ein Erreichbarkeitsproblem im Konfigurationsgraphen der PSPACE-Maschine, die A entscheidet. Dieser Graph hat zwar exponentielle Größe, aber er kann knapp beschrieben werden (im Wesentlichen durch den Schaltkreis aus dem Schichtenlemma).

Übung 5.4 Untere Schranke für ein Problem, leicht

Zeigen Sie SUCCINCT-REACH $\notin$ L.

Tipp: Nutzen Sie Übung 12.1.

Für die nächste Aufgabe brauchen wir eine Definition.

▶ **Definition: Graphisomorphie-Problem, GI**

Instanzen Zwei ungerichtete Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$.

Frage Sind G_1 und G_2 isomorph? (Gibt es eine Bijektion $\iota : V_1 \to V_2$ mit $\{x,y\} \in E_1 \iff \{\iota(x), \iota(y)\} \in E_2$).

Beispiel:  ist isomorph zu  .

Übung 5.5 Colored-GI reduziert sich auf GI, schwer

Ein *gefärbter Graph* ist ein Graph, in dem alle Knoten eine *Farbe* haben (benachbarte Knoten *dürfen* dieselbe Farbe haben). Ein Isomorphismus zwischen gefärbten Graphen ist ein Isomorphismus, der die Farben »erhält« (rote Knoten werden auf rote Knoten abgebildet).

Zeigen Sie COLORED-GI $\leq_{\mathrm{m}}^{\log}$ GI.

Tipp: Kleben Sie sehr lange Pfade an die Knoten, deren Längen die Farben kodieren.

Übung 5.6 Reduktionsabschlüsse von E und EXP, schwer

1. Zeigen Sie, dass $\mathrm{EXP} = \mathrm{R}_{\mathrm{m}}^{\log}(\mathrm{E})$.
2. Zeigen Sie, dass E nicht unter Reduktionen abgeschlossen ist.
   Tipp: Nutzen Sie den Zeithierarchiesatz.

Teil V
Berechnungen mit Hilfe

Wenn man sich einen Großteil der Konzepte der Komplexitätstheorie anschaut, dann könnte man auf die Idee kommen, dass Komplexitätstheoretiker sehr hilfsbereite Menschen sein müssen. Zumindest Maschinen helfen sie gerne: Statt ihre Turing-Maschinen einfach hilflos ihre Berechnungen durchführen zu lassen, sie die Literatur voll von Ideen, wie man ihnen helfen kann. Am bekanntesten ist zweifelsohne der Nichtdeterminismus, dessen Definition schon auf Turings Arbeit von 1936 zurückgeht. Turing stellte sich vor, dass an bestimmten Stellen der Berechnung die Maschine, wenn sie nicht mehr recht weiter weiß, ihren Zustandswechsel von einem Menschen durchführen lassen kann. Heute formalisieren wir dies mittels »Bändern mit nichtdeterministischen Bits«.

Neben nichtdeterministischen Bits gibt es aber noch viele weitere »Hilfestellungen«, die in der Literatur betrachtet werden: Man kann Maschinen mit Zufallsbits helfen. Man kann Ratschlagsbits (Advice-Bits) zur Verfügung stellen. Man kann die Ergebnisse von Vorberechnungen nutzen. Und wenn alle Stricke reißen, so kann man Orakel befragen.

6-1

Kapitel 6
Nichtdeterminismus
Betrügerische Maschinen

6-2

Lernziele dieses Kapitels

1. Unterschiedliche Definitionen von Nichtdeterminismus kennen und vergleichen können
2. Den Satz von Cook kennen und beweisen können

Inhalte dieses Kapitels

Worum es heute geht

Ist es moralisch verwerflich, zu schummeln? Für Menschen sicherlich – und es kann äußerst unangenehme Konsequenzen haben, wie Ihnen mittlerweile eine ganze Reihe ehemaliger Bundesminister(innen) werden bestätigen können. Für Maschinen ist die Situation etwas anders: Es ist nicht verwerflich, wenn Maschinen schummeln dürfen – es führt lediglich zu Berechnungsmodellen, die nichts bringen.

Wie können Maschinen »schummeln«? Nun, zunächst können sie (Aus)wahlbänder haben. Dies sind extra Bänder, auf denen die Maschine Hinweise bekommt. Stellen Sie sich vor, Sie sitzen in einer Klausur und Ihr Nachbar steckt Ihnen klandestin einen Bogen mit den Antworten für die Klausur zu. Sehr praktisch. Jedoch hat die Sache einen Haken: Sie können sich leider nicht sicher sein, dass Sie dieselbe Klausur wie Ihr Nachbar haben; Sie müssen die Antworten erst noch überprüfen. Wie Sie aber aus eigener Erfahrung wissen, ist das Überprüfen von Lösungen im Allgemeinen wesentlich einfacher, als die Lösungen selbst zu finden.

Auswahlbänder und »Hinweise« sind zugegebenermaßen etwas dubios, sie sind schwer zu motivieren und schwer zu definieren. Trotzdem brauchen wir sie, denn sie stehen im Zentrum des berühmtesten Problems der Informatik überhaupt: dem P-NP-Problem. Formuliert man es als »Schummel-Problem« so ist die Frage, ob Polynomialzeitmaschinen die Hinweise ihrer Nachbarn etwas nützen oder ob diese Hinweise nicht auch immer durch »ehrliche« Berechnungen ersetzt werden können.

6.1 Sichten auf Nichtdeterminismus

6-4

Nichtdeterminismus – Ein kurzer Überblick.

Die Idee des Nichtdeterminismus geht auf Turings Originalarbeit zu Turingmaschinen zurück. Er nennt nichtdeterministische Maschine »automatisch«. In diesem Kapitel betrachten wir Nichtdeterminismus von drei Blickwinkeln:

1. *Prover-Verifier-Protokolle.*
   Dies ist eine »moderne« Sicht, die zu Durchbrüchen wie dem PCP-Satz geführt hat.
2. *(Aus)wahlbänder.*
   Hier ist der Nichtdeterminismus besonders »explizit«, was viele Argumente leichter macht.
3. *Transitionsrelationen.*
   Dies ist die klassische Sicht, die sich leicht auf viele Maschinenmodelle verallgemeinern lässt.

6.1.1 Sicht I: Beweiser und Überprüfer

6-5

Die Struktur der vieler Probleme.

Viele Entscheidungsprobleme haben grundsätzlich folgende Struktur: Gegeben ist eine Eingabeinstanz, für die wir nach einer *Lösung* suchen. Falls eine solche Lösung existiert, so ist die Eingabe eine »Ja«-Instanz (ist in der Sprache), sonst nicht.

Beispiele

1. Für SAT sind Lösungen *erfüllende Belegungen*.
2. Für CLIQUE sind Lösungen *hinreichend große Cliquen*.
3. Für das Faktorisierungsproblem sind Lösungen die Faktoren der Eingabe.
4. Für das Halteproblem sind Lösungen die Anzahl der Schritte, nach der die Maschine anhält.

6-6

Warum sind Probleme schwer?!

Interessanterweise kann es viele Gründe geben, warum Probleme schwer sind:

- Es ist unklar, *was Lösungen überhaupt sein könnten.*
  Beispiel: Das Primzahlproblem oder das Erfüllbarkeitsproblem für prädikatenlogische Formeln.
- Lösungen können *unglaublich groß sein*.
  Beispiel: Halteproblem.
- Lösungen können *sehr große sein*.
  Beispiel: SUCCINCT-REACH.
- Lösungen können *extrem schwer zu finden sein*.
  Beispiel: SAT, CLIQUE.
- Lösungen können *schwer zu finden sein*.
  Beispiel: Faktorisierung, Graphisomorphie.

6-7

Prover-Verifier-Spiele.

Die Idee

In einem *Prover-Verifier-Spiel* (auch Prover-Verifier-*Protokoll* genannt) gibt es zwei Spieler, genannt *Prover* und *Verifier*. Der Job von Prover ist, zu Eingaben Lösung zu produzieren. Der Job von Verifier ist, Lösungen zu überprüfen.

▶ **Definition**

Sei $L \subseteq \Sigma^*$. Ein *Prover-Verifier-Protokoll für L* ist ein Paar (P,V) mit $P\colon \Sigma^* \to \Sigma^*$ und $V \subseteq \Sigma^* \times \Sigma^*$, so dass

1. für alle $x \in L$ gilt $\big(x, P(x)\big) \in V$ und
2. für alle $x \notin L$ und alle $y \in \Sigma^*$ gilt $(x,y) \notin V$.

Die Ausgaben von Prover heißen *Zertifikate*. Die erste Bedingung sagt, dass für Wörter in der Sprache Prover Verifier immer »überzeugen« kann. Die zweite Bedingung sagt, dass für Wörter, die nicht in der Sprache sind, Verifier nie überzeugt werden kann.

✎ Zur Übung 6-8
Geben Sie Prover-Verifier-Protokolle an für

1. SAT
2. das Halteproblem.

Verifier mit eingeschränkten Fähigkeiten. 6-9

▶ Definition: NP-Prover-Verifier-Protokoll
Ein Prover-Verifier-Protokoll (P,V) für eine Sprache L ist ein NP-*Prover-Verifier-Protokoll*, wenn

1. P polynomiell beschränkt ist, also $|P(x)| \leq |x|^{O(1)}$ und
2. V in polynomieller Zeit entscheidbar ist, also $V \in \mathrm{P}$.

Man beachte, dass Prover *»allmächtig«* ist!

Beispiel: NP-Prover-Verifier-Protokoll für SAT

Prover Bildet jedes φ auf ein β mit $\beta \models \varphi$ ab, wenn ein solches β existiert.
Verifier Überprüft, ob $\beta \models \varphi$. (Also code$(\varphi,\beta) \in V$ genau dann, wenn $\beta \models \varphi$.)

Beispiel: NP-Prover-Verifier-Protokoll für CLIQUE

Prover Bildet (G,k) auf eine Clique C der Größe k in G ab.
Verifier Überprüft für $\big((G,k),C\big)$, ob C eine Clique der Größe k in G ist.

Beispiel: NP-Prover-Verifier-Protokoll für beliebiges $A \in \mathrm{P}$

Prover Macht nichts.
Verifier Bei Eingabe (x,y), akzeptiere genau dann, wenn $x \in A$.

✎ Zur Übung 6-10
Geben Sie NP-Prover-Verifier-Protokolle an für

1. GI (Graph-Isomorphie-Problem),
2. VERTEX-COVER,
3. PRIMES (schwierig!).

6.1.2 Sicht II: Auswahlbänder

Eine zweite Sicht auf NP: Auswahlbänder. 6-11

▶ Definition: Wahlband
Ein *Wahlband* ist ein spezielles Band für eine Turingmaschine mit:

1. In der Anfangskonfiguration ist das Band nicht leer, sondern mit einem String $b \in \{0,1\}^*$ initialisiert. Wir schreiben hierfür $C^b_{\text{init}}(x)$.
2. Der Kopf ist anfangs auf dem ersten Bit von b.
3. Der Kopf darf *nie nach links gehen* auf diesem Band (ein Einweg-Band).

Für die Zeit- und Platzkomplexität einer solchen Maschine für eine Eingabe x wird jeweils der maximale Zeit- oder Platzverbrauch über alle möglichen b genommen. Das Band zählt nicht beim Platz.

6-12

Wie Wahlbänder bei Berechnungen helfen.

▶ **Definition**
Sei M eine Turingmaschine mit einem Wahlband. Sei x eine Eingabe. Wir sagen, *M akzeptiert x*, wenn es ein $b \in \{0,1\}^*$ *gibt*, so dass $C^b_{\text{init}}(x) \vdash \cdots \vdash C_m$, wobei C_m eine akzeptierende Endkonfiguration ist.

Die Bits auf dem Wahlband sind »Tipps«, die die Maschine durch die Berechnung »leiten«. Wir nennen sie auch *nichtdeterministische Entscheidungen*.

Beispiel
Die Sprache SAT kann mit durch eine Turingmaschine mit Wahlband schnell entschieden werden: Bei Eingabe φ mit n Variablen kopiert die Maschine zunächst die ersten n Bits vom Wahlband auf ein Arbeitsband. Sie interpretiert dann diese Bits als eine Belegung β und akzeptiert, wenn $\beta \models \varphi$.

✎ **Zur Diskussion**
Wie schnell arbeitet diese Maschine?

6-13

Nichtdeterministische Klassen

▶ **Definition**
Die *nichtdeterministischen Klassen* $\text{NTIME}(\mathcal{F})$ und $\text{NSPACE}(\mathcal{F})$ sind wie $\text{TIME}(\mathcal{F})$ und $\text{SPACE}(\mathcal{F})$ definiert, nur dass die Maschinen Wahlbänder haben.

▶ **Definition:** Die wichtigsten nichtdeterministischen Klassen

$$\begin{aligned} \text{NL} &= \text{NSPACE}(O(\log n)), \\ \text{NP} &= \text{NTIME}(n^{O(1)}), \\ \text{NE} &= \text{NTIME}(2^{O(n)}), \\ \text{NEXP} &= \text{NTIME}(2^{p(n)}). \end{aligned}$$

6-14

Wahlbänder versus Prover-Verifier-Protokolle.

▶ **Satz**
Eine Sprache ist in NP *genau dann, wenn es ein* NP*-Prover-Verifier-Protokoll für sie gibt.*

Beweis.

1. Für die Rückrichtung, sei (P,V) ein NP-Prover-Verifier-Protokoll für L. Die gesuchte Maschine kopiert bei Eingabe x einfach den Inhalt des Wahlbandes hinter x und simuliert dann V.
2. Sei nun M eine Maschine mit Wahlband, die L akzeptiert. Prover bildet nun eine Eingabe $x \in L$ auf ein kürzestes b ab, so dass $C^b_{\text{init}}(x) \vdash^* C$ für eine akzeptierende Endkonfiguration C. Ein solches b muss es geben und es muss polynomiell lang sein, da M polynomiell zeitbeschränkt ist. Verifier simuliert auf Eingabe (x,y) einfach M für Eingabe x und Wahlbandinhalt $b = y$. □

6.1.3 Sicht III: Transitionsrelationen

Von Programmen zu Transitionsrelationen. 6-15

- Zur Erinnerung: Ein *Programm für eine Turingmaschine* ist eine Funktion

$$\delta : Q \times \Gamma^h \to Q \times \Gamma^h \times \{l, r, n\}^h.$$

- Wir definieren nun ein *nichtdeterministisches Programm*, indem wir diese *Funktionen* durch *Relationen* ersetzen:

$$\Delta \subseteq Q \times \Gamma^h \times Q \times \Gamma^h \times \{l, r, n\}^h.$$

- Die Relation $\vdash$ auf Konfigurationen wird nun manchmal eine Konfiguration C mit mehreren Nachfolgekonfigurationen in Beziehung setzen.
- Ein Wort wird nun akzeptiert, wenn es eine akzeptierende Berechnung *gibt*.

Im Westen nichts Neues. . . 6-16

▶ Satz
Eine Sprache ist in NP *genau dann, wenn sie von einer Turingmaschine mit nichtdeterministischem Programm akzeptiert wird, bei der alle Berechnungen höchstens polynomielle Länge haben.*

Beweis.

1. Für die Rückrichtung sei M eine Maschine mit nichtdeterministischem Programm P. Eine deterministische Maschine M' mit Wahlband simuliert diese nun, indem sie immer dann, wenn M eine von k möglichen nichtdeterministische Entscheidung fällen muss, $\lceil \log k \rceil$ Bits vom Wahlband liest.
2. Für die andere Richtung sei M' eine Maschine mit Wahlband. Eine nichtdeterministische Maschine M kann M' simulieren, indem sie zunächst nichtdeterministisch einen Inhalt b für das Wahlband »rät« und dann M' normal laufen lässt. □

6.2 Klassenstruktur

Zusammenhand zwischen deterministischen und nichtdeterministischen Klassen. 6-17

▶ Satz: Triviale Inklusionen

$$\begin{aligned} \text{TIME}(f) &\subseteq \text{NTIME}(f), \\ \text{SPACE}(f) &\subseteq \text{NSPACE}(f). \end{aligned}$$

▶ Satz
Für alle sinnvollen f mit $f(n) \geq \log(n)$ gilt

$$\begin{aligned} \text{NTIME}(f) &\subseteq \text{SPACE}(f), \\ \text{NSPACE}(f) &\subseteq \text{TIME}(2^{O(f)}). \end{aligned}$$

Beweis.

1. In Zeit f können wir nur $f(|x|)$ Bits vom Wahlband lesen. Folglich können wir deterministisch in Platz f einfach alle möglichen Inhalte des Wahlbandes durchgehen und stoppen, wenn wir einen finden, bei dem die Maschine akzeptiert.
2. Sei $L \in \text{NSPACE}(f)$ via M. Bei Eingabe x schreiben wir zunächst den Konfigurationsgraphen von M auf, wobei wir uns vom Wahlband aber immer nur das aktuelle Zeichen merken brauchen. Der Graph hat dann Größe $O(nf(n)^{O(1)}2^{O(f(n))})$. Hier müssen wir nun Erreichbarkeit lösen, was quadratischen Zeitaufwand hat und somit gilt die behauptete Schranke. □

6.3 Der Satz von Cook

6-18

NP hat ein vollständiges Problem.

▶ Definition: CIRCUIT-SAT

Eingaben Schaltkreis C mit einem Ausgabegatter und n Eingabegatter.

Frage Gibt es eine Eingabe $x \in \{0,1\}^n$, so dass $C(x) = 1$?

▶ Satz

CIRCUIT-SAT *ist* NP-*vollständig*.

Beweis. Um CIRCUIT-SAT $\in$ NP zu sehen, betrachte folgendes Prover-Verifier-Protokoll:

Prover Bilde C auf ein x ab mit $C(x) = 1$.

Verifier Bei Eingabe (C,x), überprüfe, ob $C(x) = 1$.

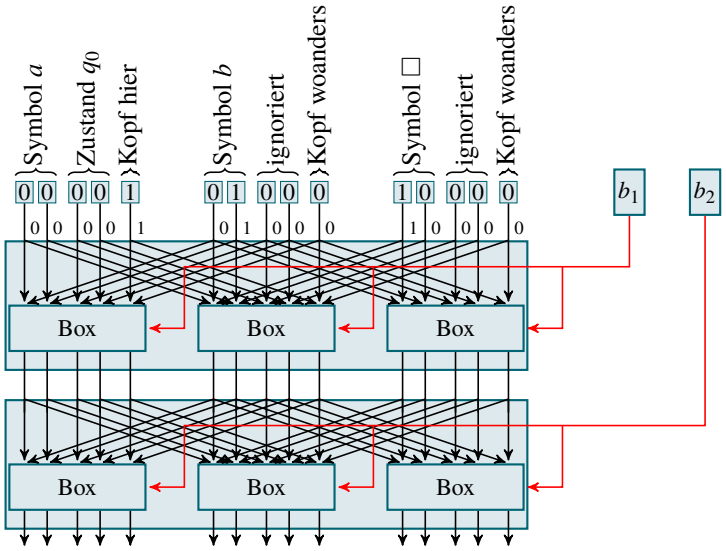
Schwere zeigen wir durch Bootstrapping: Sei $L \in$ NP via M mit einem Wahlband. Wir können annehmen, dass M *genau ein Bit* auf dem Wahlband in jedem Schritt liest. Sei nun x eine Eingabe für die Reduktion auf CIRCUIT-SAT. Die Reduktion bildet x nun auf den Schaltkreis C aus dem Vollständigkeitsbeweis von CVP für P ab, jedoch gibt es für jede Schicht ein zusätzliches Eingabegatter, das das Bit des Wahlbandes enthält, das in diesem Schritt gelesen wird. Dieses Eingabegatter ist mit allen Boxen auf der Schicht verbunden. Nun gilt:

> Für eine Eingabe b für den Schaltkreis gilt $C(b) = 1$ genau dann, wenn b ein Inhalt für das Wahlband ist, so dass M für diesen akzeptiert.

□

6-19

Der konstruierte Schaltkreis.



6-20

Satz von Cook.

▶ Satz

SAT *ist* NP-*vollständig*.

Beweis. Eine Reduktion von CIRCUIT-SAT auf SAT. Siehe die Übung 6.3. □

Zusammenfassung dieses Kapitels

▶ Definitionen von NP: Sicht I

NP ist die Klasse aller Sprachen L, für die es Prover-Verifier-Protokolle gibt mit

- Provers Beweise sind nur polynomiell lang und
- Verifier liegt in P.

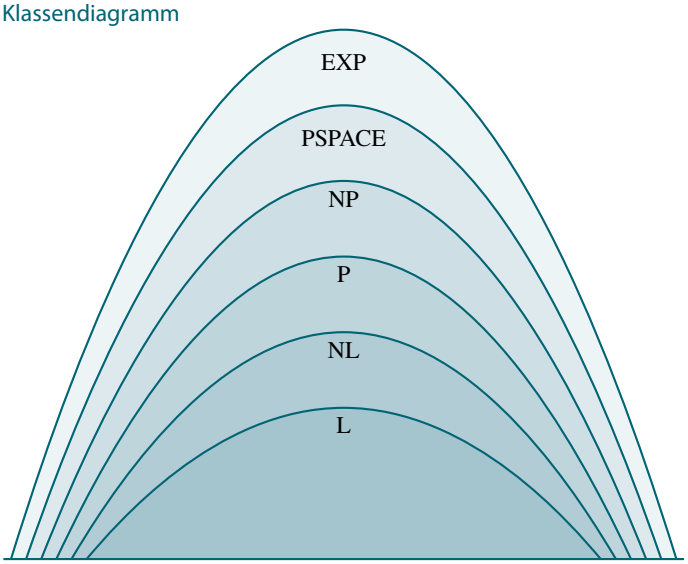
▶ Definitionen von NP: Sicht II

NP ist die Klasse aller Sprachen L, die von deterministischen Turingmaschinen M mit Wahlband entschieden werden können, die nur polynomiell lange rechnen.

▶ Definitionen von NP: Sicht III

NP ist die Klasse aller Sprachen L, die von nichtdeterministischen Turingmaschinen M entschieden werden können, die nur polynomiell lange rechnen.

▶ Klassendiagramm



Zum Weiterlesen

[1] Steven A. Cook, The complexity of theorem-proving procedures. In *Proceedings of STOC' 71,* 151–158, 1971.

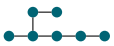
Dies ist die Originalarbeit, in der Cook zeigt, dass das Erfüllbarkeitsproblem vollständig ist für NP. Da er direkt auf SAT reduziert, ist der Beweis etwas undurchsichtig. Weiterhin benutzt Cook auf keine Logspace-Many-One-Reduktion, sondern eine Polynomialzeit-Turing-Reduktion.

Übungen zu diesem Kapitel

▶ Definition: Graph-Automorphie-Problem, GA

Eingabe Ein ungerichteter Graph $G = (V, E)$.

Frage Hat G einen nichtrivialen Automorphismus? (Gibt es eine Bijektion $\iota : V \to V$, *die nicht die Identitiät ist,* mit $\{x, y\} \in E \iff \{\iota(x), \iota(y)\} \in E$).

Beispiel:  hat einen nichttrivialen Automorphismus, aber  nicht.

Übung 6.1 Reduktion von GA auf GI, schwer

Zeigen Sie $\text{GA} \leq^{\text{p}}_{\text{T}} \text{GI}$.

Tipp: Reduzieren Sie auf das gefärbte Isomorphieproblem. Für jedes Paar $u, v \in V$ mit $u \neq v$ im Graphen G erzeugen Sie zwei Versionen von G, in der einen ist u schwarz gefärbt und alle anderen weiß, in der anderen ist v schwarz gefärbt und alle anderen weiß.

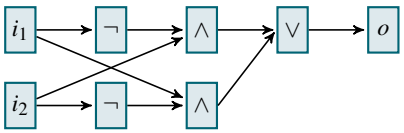
Übung 6.2 Schaltkreise in Formeln umwandeln, schwer

Wir wollen Schaltkreise in Formeln umwandeln ohne sie stark zu vergrößern. Sei C ein Schaltkreis mit Fan-In 2. Wir bauen eine Formeln φ wie folgt: Für jedes Gatter g_i führen wir eine Variable g_i ein. Ziel ist, folgendes zu erreichen:

> $(*)$ Es gibt eine erfüllende Belegung β für φ mit $\beta(g_i) = 1$ genau dann, wenn es eine Eingabe für C gibt, so dass g_i eine 1 ausgibt.

Wir können $(*)$ beispielsweise für ein Und-Gatter g_i durch folgende Teilformel garantieren: $g_i \leftrightarrow (g_j \wedge g_k)$, wobei g_j und g_k die Vorgänger von g_i sind. Für ein Oder-Gatter würden wir $g_i \leftrightarrow (g_j \vee g_k)$ zu φ hinzufügen.

1. Beschreiben Sie, wie die Konstruktion für die anderen Gatterarten (Negations-, Konstanten-, Eingabe- und Ausgabegatter) funktioniert.
2. Wenden Sie die Idee auf folgenden Schaltkreis an:



3. Welche Form hat die Formel, die entsteht? Ist sie in einer Normalform? Wenn ja, in welcher?
4. Argumentieren Sie, weshalb tatsächlich $(*)$ gilt bei der Konstruktion. (*Tipp:* Eine Induktion.)

Übung 6.3 Beweis des Satzes von Cook, mittel

Zeigen Sie den Satz von Cook (SAT ist NP-vollständig). Gehen Sie wie folgt vor:

1. Zeigen Sie SAT $\in$ NP, indem Sie ein NP-Prover-Verifier-Protokoll angeben.
2. Zeigen Sie CIRCUIT-SAT $\leq_{\mathrm{m}}^{\log}$ SAT. Nutzen Sie die Idee aus der vorherigen Übung.

Wenn Ihnen das zu einfach ist: Zeigen Sie, dass 3-SAT auch NP-vollständig ist.

Übung 6.4 Schwache Verifier, schwer

Betrachten Sie folgende spezielle Art von Prover-Verifier-Protokoll (P,V): Prover ist polynomiell beschränkt und $V \in \mathrm{L}$ (statt $V \in \mathrm{P}$, wie es sonst üblich wäre). Zeigen Sie, dass für jede Sprache in NP ein Prover-Verifier-Protokoll dieser Art existiert.

Tipp: Man kann leicht in logarithmischem Platz überprüfen, ob $\beta \models \varphi$ gilt, wenn φ in 3-KNF ist.

7-1

Kapitel 7
Relativierung
Grenzen gängiger Beweismethoden

Lernziele dieses Kapitels

1. Die Polynomielle Hierarchie und wichtige Probleme in ihr kennen
2. Konzept der Orakelmaschine und der Relativierung kennen
3. Orakel kennen, relativ zu denen P und NP zusammenfallen und nicht zusammenfallen

7-2

Inhalte dieses Kapitels

Worum es heute geht

Versetzen Sie sich bitte in den Kopf einer Komplexitätstheoretikerin Mitte der 1970er Jahre, die sich über die P-NP-Frage Gedanken macht. Ihre Überlegung würde in etwa wie folgt ablaufen: »Stephen hat ja vor kurzem gezeigt, dass aus $\text{SAT} \in \text{P}$ schon $\text{P} = \text{NP}$ folgt, und ich sehe gar nicht, weshalb SAT einen Polynomialzeitalgorithmus haben sollte. Bis zur nächsten Konferenz-Deadline zeige ich dann mal $\text{P} \neq \text{NP}$. Das ist ein ›Klassentrennungsproblem‹ und die sind ja alte Hüte. Schon Turing hat doch mittels *Diagonalisierung* das Halteproblem von den rekursiven Sprachen getrennt; und vor ein paar Jahren haben Dick, Juris und Phil P und EXP mittels Diagonalisierung getrennt. Also: Ich konstruiere einfach eine Sprache, die *definitiv* nicht in P liegt, weil jede Polynomialzeit-Maschine sich bei wenigstens einem Wort ›irrt‹. Das ist schonmal ziemlich einfach. Dann muss ich nur noch zeigen, dass die Sprache in NP liegt. Das mache ich dann morgen, … « … was dann aber nie geschah.

Alle Versuche, $\text{P} \neq \text{NP}$ mittels Diagonalisierung zu beweisen, scheiterten – und dies hat einen tieferen Grund, der erstmals im Jahr 1975 von Baker, Gill und Solovay [1] in ihrer Arbeit *Relativizations of the* P=NP*? Question* erläutert wurde und den wir in diesem Kapitel kennen lernen wollen.

7.1 Orakel

7.1.1 Die Idee

Eine kleine Geschichte zu RISC und CISC.

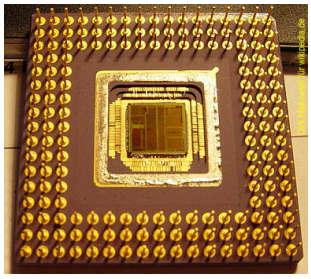
In der technischen Informatik tobt seit Urzeiten ein Streit, ob nun RISC oder CISC besser sei:

Alan Turing: Pioneer des RISC

Die RISC-Jünger möchten den in Hardware implementierte Befehlssatz ihrer Prozessoren klein halten, komplexere Befehle sollen doch bitte »ausprogrammiert« werden. Ihr radikalster Vertreter war (ohne es zu wissen) Alan Turing, denn die Turing-Maschine ist die ultimative RISC-Maschine.

Public domain

7-4


Public domain


Public domain

Pentium: Eine fette CISC-Maschine

Die bekannteste CISC-Maschine ist der Pentium mit seinen »mächtigen« Befehlen wie FPATAN zur Berechnung des Arkustangens einer Zahl in einem (konzeptionellen) Schritt. Hierdurch werden Berechnungen schnell, bei denen der Arkustangens oft gebraucht wird; hingegen wird ein Pentium beispielsweise bei einer normalen Rekursionen hierdurch keine Vorteile haben. Der Pentium besitzt beispielsweise auch solch lustige Befehle wie AAA (»adjust after addition«), einer von vier (!) Befehlen allein zum Arbeiten mit Binary-Coded-Decimals.

CISC mit wirklich mächtige Befehlen

Was würde passieren, wenn jemand einen Prozessor mit, sagen wir, einem Befehl ONGM für »optimal next go move« bauen würde, der in einem Schritt den optimalen nächsten Zug für eine Position im Brettspiel Go berechnet?

7-5

Auf dem Weg zur Definition von Orakeln.

Reale Prozessoren und ihre Befehlssätze sind für theoretische Betrachtungen eher ungeeignet. Um die Auswirkungen von »extrem mächtigen Befehle« exakt zu untersuchen, übertragen wir daher die Idee auf die besser handhabbaren Turingmaschinen. Das Resultat sind die recht originell benannten »Orakel-Turingmaschinen«. Die Idee ist, die »mächtigen Befehle« formal als Sprachen aufzufassen, sie aber »Orakel« zu nennen und sie dann »zu befragen«.

Beispiel: Ein Befehl aus der Kryptologie als Orakel

Ein »Befehl«, den man in der Kryptologie gerne hätte, lautet »teste, ob eine Zahl prim ist«. Dies formalisieren wir als Orakel PRIMES $= \{10, 11, 101, 111, 1011, \dots\}$, das alle Primzahlen in Binärdarstellung enthält.

Beispiel: Der hypothetischen Maschinen-Befehl ONGM (»optimal next go move«) als Orakel

Unseren hypothetischen ONGM Befehl können wir als Orakel GO formalisieren. Diese Sprache enthält alle (Codes von) Go-Spielbrettern zusammen mit einer Feldposition, falls die Feldposition für Weiß ein optimaler nächster Zug ist.

7.1.2 Die Definition

7-6

Definition von Orakel-Turingmaschinen.

Definition: Orakel

Ein *Orakel* ist eine Sprache.

Definition: Orakel-Turingmaschinen

Eine *Orakel-Turingmachine* ist eine Maschine $M^?$ mit den drei speziellen Zuständen

1. *Frage-Zustand* $q_?$ (*query state*),
2. *Ja-Zustand* q_{ja} und dem
3. *Nein-Zustand* q_{nein}

sowie einem speziellen *Orakel-Band*, das ein Nur-Schreiben-Band ist.

Definition: Semantik einer Orakel-Turingmaschine

Sei $M^?$ eine Orakel-Turingmaschine und X ein Orakel. Die Berechnung von $M^?$ bei Eingabe x *relativ zum Orakel* X ist eine normale Berechnung mit folgender Ausnahme: Wenn $M^?$ in den Frage-Zustand wechselt, passiert folgendes:

- Falls der aktuelle Inhalt des Orakel-Bandes ein Wort in X ist, so ist in der nächsten Konfiguration der Zustand der Ja-Zustand, sonst der Nein-Zustand.
- In der nächsten Konfiguration ist das Orakel-Band komplett geleert.
- Keine anderen Zellen werden verändert und keine Köpfe bewegt.

Eine Anfrage an das Orakel zu stellen ist »kostenlos«, sie kostet weder Zeit noch Platz.

Definition: Spezielle Regeln betreffend Nichtdeterminismus und Orakel-Maschinen

Wenn eine Orakel-Maschine auch ein Wahlband hat, so gibt es folgende Sonderregel:

> Sie darf ihren Kopf auf dem Wahlband nur dann bewegen, wenn das Orakelband komplett leet ist.

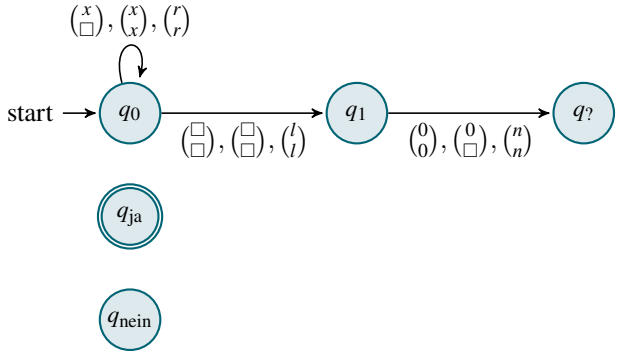
Mit anderen Worten: Die Maschine darf keine nichtdeterministischen Schritt machen, während sie ihre Orakelanfrage aufschreibt.

Eine einfache Orakel-Maschine.

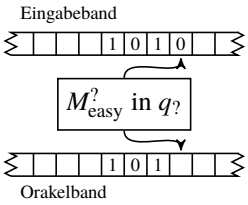
7-7

Wir bauen eine Turing-Maschine, die alle (Binärcodes von) Zahlen akzeptiert, die das Doppelte von Primzahlen sind. Normalerweise ist das kniffelig; wenn wir aber Zugriff auf das Orakel PRIMES haben, so leistet folgende Orakel-Turingmaschine das Gewünschte: Sie kopiert ihre Eingabe auf das Orakelband, löscht aber die 0 am Ende. Nun fragt sie das Orakel und akzeptiert (nur) im Zustand q_{ja}.

Die Maschine $M^?_{\text{easy}}$



Konfiguration beim Erreichen von $q_?$ bei Eingabe 1010.



Von Orakel-Maschinen akzeptierte Sprachen

7-8

▶ Definition: Die relativ zu einem Orakel akzeptierte Sprache

Für eine Orakel-Turingmaschine $M^?$ und ein Orakel X schreiben wir $L(M^X)$ für die Menge aller von $M^?$ *relativ zu dem Orakel X akzeptierten Wörter.* Eine andere gängige Schreibweise ist $L(M^?, X)$.

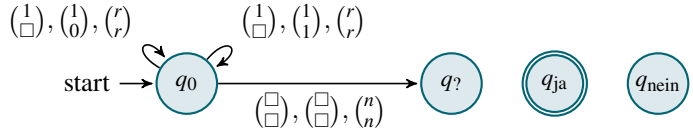
✎ Zur Übung

Geben Sie $L(M^{\text{PRIMES}}_{\text{easy}})$ und $L(M^Y_{\text{easy}})$ mit $Y = \{1^n 0^n \mid n \in \mathbb{N}\}$ an für $M^?_{\text{easy}}$ von oben an.

Eine nichtdeterministische Orakel-Turingmaschine

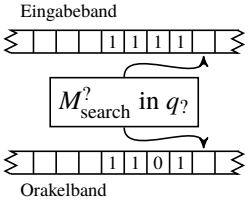
7-9

Die Suchmaschine $M^?_{\text{search}}$



Die Maschine läuft in einer Schleife nach rechts über die Eingabe, solange diese nur aus 1en besteht. Währenddessen schreibt sie *nichtdeterministisch* auf das Orakelband einen Bitstring, der *genauso lang wie die Eingabe ist.* Sie akzeptiert offenbar überhaupt nur Wörter der Form 1^n, denn sonst schafft sie es nie nach $q_?$. Darüber hinaus muss es dann noch wenigstens eine nichtdeterministische Berechnung geben, bei der auf dem Orakelband auch ein Wort steht, zu dem das Orakel »Ja« antwortet.

Vorletzte Konfiguration einer akzeptierenden Berechnung bei Eingabe 1111 für das Orakel PRIMES



✎ Zur Übung
Wie lauten $L(M^X_{\text{search}})$ für $X = \text{PRIMES}$ und für $X = \{ww^{-1} \mid w \in \{0,1\}^*\}$?

7.2 Relativierung

7.2.1 Relativierte Klassen

7-10

Die Frage nach dem Effekt von ONGM ist brisant.
Standard-Beweisverfahren sind sehr »CISC-tolerant«: Wenn man mit ihnen zeigt, dass zwei Berechnungsmodelle (wie »deterministische polynomielle Zeit« und »nichtdeterministische polynomielle Zeit«) unterschiedlich sind, dann klappt diese Trennung *unabhängig davon, wie schnell bestimmte Befehle ausgeführt werden können.*
Baker, Gill und Solovay [1] haben 1975 in der Arbeit *Relativizations of the* P=NP*? Question* gezeigt, dass genau diese beiden Berechnungsmodelle *gleich* sind, wenn (nur) »ONGM« als Ein-Schritt-Befehl zur Verfügung steht, und sie *ungleich* sind, wenn (nur) »TRWC« als Ein-Schritt-Befehl zur Verfügung steht. (TRWC soll für »test really weird condition« stehen.)
Also lässt sich $\text{P} \neq \text{NP}$ *gar nicht* mittels Standard-Beweisverfahren zeigen, da diese Trennung ja für jeden Befehlssatz funktionieren müsste, was nicht der Fall ist.

7-11

Relativierte Klassen

▶ Definition: Relativierte Zeitklassen
Sei X ein Orakel und $\mathcal{F} \subseteq \{f \mid f\colon \mathbb{N} \to \mathbb{N}\}$ eine Klasse von Zeitschranken. Die Klasse $\text{TIME}(\mathcal{F})^X$ enthält alle Sprachen L, so dass eine Orakel-Turingmaschine $M^?$ und ein $f \in \mathcal{F}$ existieren mit

1. $L(M^X) = L$ und
2. für alle $n \in \mathbb{N}$ gilt $T_M(n) \leq f(n)$.

Die Klassen $\text{NTIME}^X(\mathcal{F})$, $\text{SPACE}^X(\mathcal{F})$ und $\text{NSPACE}^X(\mathcal{F})$ sind analog definiert.

Schreibweisen
Wir schreiben P^X für $\text{TIME}^X(n^{O(1)})$ und NP^X für $\text{NTIME}^X(n^{O(1)})$. Für eine ganze Klasse D von Problemen schreiben wir C^D für $\bigcup_{X \in \text{D}} \text{C}^X$. Beispielsweise steht NP^P für »Sprachen entscheidbar von NP-Maschinen, die eine Sprache in P befragen dürfen«.

7-12

Ein mächtiges Orakel.
Wie bei echten Prozessoren auch sind einige Orakel (= »mächtige Befehle«) zwar nice-to-have, sie ändern aber nichts grundsätzlich an der Mächtigkeit der Maschinen. So gilt $\text{P}^X = \text{P}$ für das Orakel $X = \{0^n 1^n \mid n \geq 1\}$, denn statt X zu befragen, kann eine Polynomialzeitmaschine die Frage »Ist $x \in X$?« bei diesem einfachen Orakel gleich selbst beantworten. Das folgende Beispiel zeigt, dass ein ONGM-Befehl hingegen sehr nützlich wäre.

▶ Lemma
Für das Orakel $X = \text{GO}$ *gilt* $\text{PSPACE} \subseteq \text{P}^X$.

Beweis. Man kann zeigen, dass GO ein PSPACE-vollständiges Problem ist. Sei nun L in PSPACE und somit $L \leq^{\log}_{\text{m}} X$ via einer Funktion $f \in \text{FL} \subseteq \text{FP}$. Eine Orakel-Turingmaschine rechnet bei Eingabe w das Wort $f(w)$ aus, kopiert $f(w)$ auf das Orakelband und *fragt dann das Orakel, ob* $f(w) \in \text{GO}$ *gilt.* Ist die Antwort »Ja«, so akzeptiert sie, sonst verwirft sie. □

7-13

Was Relativierungen mit Beweisen zu tun haben.
Die mit Standardbeweisverfahren aus der Komplexitätstheorie erzielte Ergebnisse gelten *relativ zu jedem Orakel*. Ein einfaches Beispiel ist das (triviale) Ergebnis $\text{P} \subseteq \text{NP}$. In der Tat gilt auch $\text{P}^X \subseteq \text{NP}^X$ für jedes Orakel: In der Literatur lesen Sie dann etwa »P ⊆ NP and this result relativizes.«

Zentrale Beobachtung
Wenn $\text{P}^X = \text{NP}^X$ für ein Orakel X und $\text{P}^Y \neq \text{NP}^Y$ für ein anderes Orakel Y, so kann man weder $\text{P} = \text{NP}$ noch $\text{P} \neq \text{NP}$ beweisen mit einem Verfahren, das relativ zu jedem Orakel funktioniert.

7.2.2 P = NP relativ zu einem Orakel

Ein Orakel, relativ zu dem P = NP gilt.

7-14

Wie könnte ein Orakel X aussehen, bei dem Nichtdeterminismus keinen »Vorteil« mehr darstellt, wenn man X befragen kann? Der Trick ist, X so mächtig zu wählen, dass die »feinen Unterschiede« zwischen den Modellen »plattgewalzt« werden.

▶ **Satz**
Es gibt ein Orakel X mit $\mathrm{P}^X = \mathrm{NP}^X$.

Beweis. Sei $X = \mathrm{GO}$.[1] Nach dem Lemma gilt $\mathrm{PSPACE} \subseteq \mathrm{P}^X$. Weiter gilt natürlich immer $\mathrm{P}^X \subseteq \mathrm{NP}^X$. Nun gilt aber auch $\mathrm{NP}^X \subseteq \mathrm{PSPACE}$, denn in polynomiellem Platz kann man

1. alle möglichen Berechnungen einer NP-Maschine nacheinander durchgehen und
2. dabei die Orakelanfragen an X »selbst« beantworten, da $X \in \mathrm{PSPACE}$.

Also gilt $\mathrm{PSPACE} \subseteq \mathrm{P}^X \subseteq \mathrm{NP}^X \subseteq \mathrm{PSPACE}$. □

Kommentare zum Beweis

[1] Es kann auch jedes andere PSPACE-vollständige Problem als Orakel gewählt werden. Übung ?? zeigt aber, dass man hier *nicht* $X = \mathrm{SAT}$ wählen kann.

7.2.3 P ≠ NP relativ zu einem Orakel

Ein Orakel, relativ zu dem P ≠ NP gilt.

7-15

Wie könnte nun umgekehrt ein Orakel Y aussehen, das nichtdeterministische Maschinen »bevorteilt«? Was können diese besonders gut? »Suchen!« Sie sind unschlagbar, wenn es darum geht, in einem mathematischen Heuhaufen eine Nadel zu finden. Wir werden deshalb in Y »Hinweise« verstecken, die einer solchen Maschine helfen, die aber (nachweislich) für eine deterministische Maschine nicht zu finden sind.

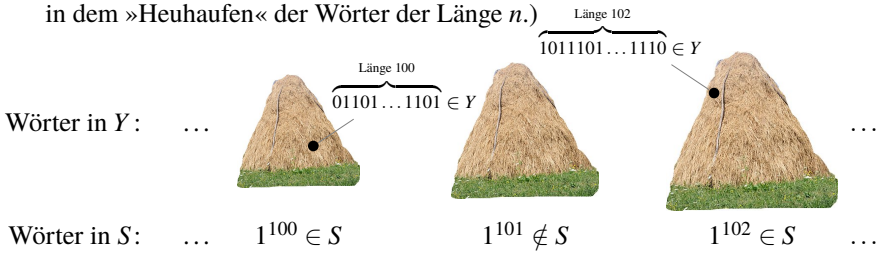
▶ **Satz**
Es gibt ein Orakel Y mit $\mathrm{P}^Y \neq \mathrm{NP}^Y$.

Der Beweisplan.

7-16

Wir konstruieren gleichzeitig das Orakel Y und eine Sprache S mit folgenden Eigenschaften:

1. S wird von keiner *deterministischen* Polynomialzeitmaschine relativ zu Y akzeptiert. (Diagonalisierung!)
2. S enthält nur Wörter, die nur aus 1en bestehen, und auch nur einige von ihnen. (S ist also eine Tally-Sprache.)
3. Ist $1^n \in S$, so enthält Y genau ein Wort der Länge n, sonst keines. (Das ist die »Nadel« in dem »Heuhaufen« der Wörter der Länge n.)



Der Beweis.

7-17

Die trennende Sprache S liegt in NP^Y.

Wir können jetzt schon zeigen (bevor wir Y und S überhaupt genauer definiert haben), dass $S \in \mathrm{NP}^Y$ gilt. Betrachten wir dazu die Orakel-Turingmaschine $M^?_{\text{search}}$: Diese akzeptiert nur Wörter der Form 1^n und auch nur solche, für die Y *wenigstens ein Wort der Länge n enthält* – was genau S ergibt.

Die Maschinen, gegen die wir diagonalisieren.

Um nun $S \notin \mathrm{P}^Y$ zu erreichen, diagonalisieren wir gegen alle »Kandidaten« $M^?$, für die $S = L(M^Y)$ gelten könnte. Dazu nummerieren wir die Kandidaten durch als $M^?_1, M^?_2, M^?_3, \ldots$. Da wir nur an Polynomialzeitmaschinen interessiert sind, statten wir jede Maschine $M^?_n$ mit einem »Wecker« aus, der ihre Berechnung nach n^{k_n} Schritten abbricht. Dabei gilt $k_n = \lfloor \log_2 n \rfloor$ (je andere langsam wachsende Funktion ginge auch). Jetzt ist garantiert, dass jede Maschine, die $S \in \mathrm{P}^Y$ zeigen könnte, in der Liste irgendwo auftaucht und nicht von ihrem »Wecker« unterbrochen wird.

Wir garantieren, dass $S \notin \mathrm{P}^Y$.

Um nun $S \notin \mathrm{P}^Y$ zu garantieren, sorgen wir für $n = 1, 2, 3, \ldots$ dafür, dass $M_n^?$ sich relativ zu Y bei dem Wort 1^n »irrt«: Wir haben bereits S und Y für Wörter bis zur Länge $n-1$ festgelegt. Nun betrachten wir, was $M_n^?$ relativ zu dem bisher festgelegten Orakel Y und der Eingabe 1^n macht. *Dabei beantworten wir alle Orakelfragen betreffend Wörter, die Länge n oder größer haben, mit »Nein«.* Falls nun $M_n^?$ das Wort 1^n akzeptiert, so sei $1^n \notin S$, sonst sei $1^n \in S$. Das soll garantieren, dass $S \notin \mathrm{P}^Y$.

Wie wir das Orakel Y wählen müssen.

Es bleibt, die Nadel in Y zu verstecken, falls $1^n \in S$: Wir müssen dann genau ein Wort im »Heuhaufen« $\{0,1\}^n$ in Y haben. Wir können jedes wählen, solange die deterministischen Maschinen »nichts davon mitbekommen«: Ihnen gegenüber soll Y ja ab Wortlänge n immer mit »Nein« antworten. Wir dürfen also kein Wort wählen, dass *irgendeine der Maschinen* $M_1^?, M_2^?, \ldots, M_n^?$ *bei ihrer Berechnung fragt*. Diese Wörter sind »verbrannt«. Da $M_i^?$ maximal i^{n_i} Schritte macht, werden maximal $\sum_{i=1}^{n} i^{k_i} \ll 2^n$ unterschiedliche Wörter verbrannt. Wir packen also einfach eines der (irrsinnig vielen) Wörter aus $\{0,1\}^n$ nach Y, das nicht gefragt wurde.

Zusammenfassung dieses Kapitels

7-18

1. Bei einer *relativierten Klasse* haben die Turingmaschinen Zugriff auf ein Orakel, das ihnen in konstanter Zeit antwortet.
2. Es gibt ein Orakel X mit $\mathrm{P}^X = \mathrm{NP}^X$ und ein Orakel Y mit $\mathrm{P}^Y \neq \mathrm{NP}^Y$. Deshalb kann man die P-NP-Fragen mit »Standardverfahren« nicht beantworten.

Seit Baker, Gill und Solovay 1975 einem schnellen Beweis von $\mathrm{P} \neq \mathrm{NP}$ einen Riegel vorgeschoben haben, hat man sich redlich bemüht, eben gerade *nicht* relativierbare Beweisverfahren zu entwickeln. Leider haben sich dann wieder neue Probleme aufgetan, wie Sie den folgenden Literaturangaben entnehmen können. Derzeit ist die Trickkiste der Theorie leer.

Zum Weiterlesen

[1] Theodore Baker, John Gill und Robert Solovay. Relativizations of the P=NP? Question, SIAM *Journal on Computing,* 4(4):431–442, 1975.

Diese Arbeit prophezeite, dass Relativierbarkeit ein zentrales Problem bei der Lösung des P-NP-Problems sein wird. Wer in diese Arbeit etwas genauer schauen möchte, sei aber gewarnt: Den Beweis, dass es ein Orakel X gibt mit $\mathrm{P}^X \neq \mathrm{NP}^X$, haben die Autoren nur aufgenommen »to illustrate the basic techniques«. Ihre wirkliche Beweisleidenschaft gilt in der Arbeit wesentlich abgehobeneren Orakelkonstruktionen.

[2] Alexander Razborov und Steven Rudich. Natural Proofs, *Journal of Computer and System Sciences,* 55:24–35, 1997.

Gut zwanzig Jahre nach der Arbeit von Baker, Gill und Solovay zeigen Razborov und Rudich hier eine neue Barriere auf: In den 1980er Jahren gab es eine Reihe von wichtigen unteren Schranken für die Tiefe von Schaltkreisen, die schwierige Sprachen entscheiden. Entscheidend war, dass diese unteren Schranken *nicht* relativierbar waren und somit einen möglichen Weg zu einem Beweis von $\mathrm{P} \neq \mathrm{NP}$ aufzeigen könnten. In dieser Arbeit wird nun aber gezeigt, dass der Weg über Schaltkreise aller Voraussicht nach zum Scheitern verurteilt ist. Ähnlich wie die Arbeit von Baker, Gill und Solovay ist der Hauptverdienst dieser Arbeit, vorhandene Ideen einheitlich zu fassen und aufzuzeigen, wie sie sich auf die P-NP-Frage auswirken.

[3] Scott Aaronson und Avi Wigderson. Algebrization: A New Barrier in Complexity Theory, *Proceedings of the 40th Annual* ACM *Symposium on Theory of Computer (STOC 2008),* ACM, Seiten 731–740, 2008.

Gut zehn Jahre nachdem Razborov und Rudich die große Hoffnung der Theoretikergemeinde aus den '80ern zerstört hatten (eine Lösung des P-NP-Frage durch Schaltkreisbetrachtungen), zeigen Aaronson und Wigderson in dieser Arbeit auf, dass die in den '90er Jahren entwickelte Technik der *Algebraisierung* von Sprachen das P-NP-Problem auch nicht wird lösen können. Auf dieses Verfahren war viel Hoffnung gesetzt worden, da Shamir 1992 mit dieser Technik zeigen konnte $\mathrm{IP} = \mathrm{PSPACE}$. Wie die Klasse IP definiert ist, ist hier nicht wichtig, entscheidend ist aber, dass es Orakel gibt mit $\mathrm{IP}^X \neq \mathrm{PSPACE}^X$; der Beweis von Shamir durchbricht also die

Relativierungsbarriere. Leider zeigen Aaronson und Wigderson in dieser Arbeit, dass man eine Art »algebraische Relativierung« einführen kann – und prompt zeigt sich, dass die P-NP-Frage mit Mitteln der Algebraisierung nicht beantwortet werden kann.

8-1

Kapitel 8
Die Polynomielle Hierarchie
Jenseits von NP (?)

8-2

Lernziele dieses Kapitels

1. Die Definition der Polynomiellen Hierarchie kennen
2. Die Stockmeyer-Charakterisierung kennen
3. Ein Kollapsresultat kennen

Inhalte dieses Kapitels

Worum es heute geht

Die Klasse NP ist zugegebenermaßen ziemlich groß. Praktisch jedes Problem, das einem in Theorie und Praxis über den Weg läuft, ist in NP. Aber, ist das wirklich so? Es lohnt sich, etwas genauer hinzuschauen: In vielen Fällen sind Probleme eigentlich nur deshalb in NP, »weil wir so formulieren, dass sie gerade in NP liegen«. Beispielsweise ist die Frage beim Handlungsreisendenproblem nicht nur, ob es eine Rundreise mit Kosten »kleiner oder gleich einem gegebenen Budget« gibt. Vielmehr möchte man vielleicht auch wissen, ob ein gegebenes Budget gerade *gleich* den minimalen Kosten ist. Ähnlich beim Cliquen-Problem: Neben der in NP liegenden Frage, ob ein Graph eine Clique der Größe k hat, kann man sich auch die berechtigte Frage stellen, ob die größte Clique im Graphen Größe k hat.

Wenn Sie etwas darüber nachdenken, so werden Sie feststellen, dass diese Problemvarianten gar nicht in NP zu sein scheinen. Andererseits wird man auch nicht erwarten, dass ihre Komplexität nun »wahnsinnig anders« ist aber die der ursprünglichen Varianten – jedenfalls wäre es überraschend, wenn die Frage nach der Größe der größten Clique in einem Graphen nun plötzlich PSPACE-vollständig wäre. Aber mehr als NP scheint es schon zu sein.

An dieser Stelle kommt die Polynomielle Hierarchie ins Spiel. Mit ihr lassen sich Problemvarianten wie die eben skizzierten gut untersuchen; beispielsweise liegt die Frage nach der Größe der größten Clique in einem Graphen »auf der zweiten Stufe« und damit »etwas« oberhalb von NP.

Die Polynomielle Hierarchie heißt so, weil man vermutet, dass sie eine Hierarchie von »immer schwierigeren« Problemklassen darstellt. Dabei ist jedoch nicht klar, ob die Probleme wirklich immer schwieriger werden: Ist beispielsweise $\mathrm{P} = \mathrm{NP}$, so »kollabiert« die Hierarchie und alle Probleme in ihr sind gleich schwierig. Wie wir sehen werden, ist die Aussage »die Polynomielle Hierarchie ist tatsächlich eine Hierarchie« eine viel stärkere Vermutung als $\mathrm{P} \neq \mathrm{NP}$; letzteres ist nämlich einfach nur die Aussage »die Polynomielle Hierarchie ist bis zur ersten Stufe tatsächlich eine Hierarchie«.

Genau wie die meisten Theoretikerinnen und Theoretiker davon ausgehen, dass $\mathrm{P} \neq \mathrm{NP}$, gehen sie auch davon aus, dass die Polynomielle Hierarchie echt ist. Widerspricht eine Annah-

me dem, so »glaubt« man nicht daran. Wir werden beispielsweise sehen, dass die Annahme, SAT ließe sich von Schaltkreisen polynomieller Größe entscheiden, die Polynomielle Hierarchie kollabieren lässt – weshalb man in der Theorie davon ausgeht, dass sich SAT gerade *nicht* von Schaltkreisen polynomieller Größe entscheiden lässt.

8.1 Die Polynomielle Hierarchie

8.1.1 Die Idee

Manche Probleme bleiben schwer, selbst wenn man ein SAT-Orakel hat. 8-4

Was wäre wenn?

Stellen wir uns vor, wir hätten ein SAT-Orakel zur Verfügung. Dann können wir Probleme wie SAT leicht lösen. Ebenso können wir TAUTOLOGY leicht lösen. Aber was ist mit Problemen wie SHORTER-FORMULA?

Die Sprache SHORTER-FORMULA

Instanzen Formel φ.

Frage Gibt es eine *kürzere* Formel ψ mit $\varphi \equiv \psi$?

Beobachtungen:

1. Mit einem SAT-Orakel können wir eine Formel ψ »raten« und dann etwas überprüfen.
2. Mit einem SAT-Orakel können wir auch die Frage beantworten, ob $\psi \equiv \varphi$ gilt.
3. Wir können aber *nicht* beides gleichzeitig!

Die Idee hinter der Polynomiellen Hierarchie. 8-5

Wir wollen *systematisch untersuchen,* was ein Orakel wie SAT nun bringt: Welche Sprachen liegen in $\mathrm{P}^{\mathrm{SAT}}$? Genau wie der Wechsel von P zu NP uns erlaubt, viele neue Probleme zu untersuchen, kann man auch von $\mathrm{P}^{\mathrm{SAT}}$ zu $\mathrm{NP}^{\mathrm{SAT}}$ wechseln. Dies führt zu neuen Problemen, die man wiederum als Orakel nutzen kann, was wieder zu neuen Klassen führt: Es entsteht die *Polynomielle Hierarchie*.

8.1.2 Die Definition

Die Definition der Polynomiellen Hierarchie. 8-6

▶ **Definition**

$$\begin{aligned}\Delta_1\mathrm{P} &:= \mathrm{P},\\ \Sigma_1\mathrm{P} &:= \mathrm{NP},\\ \Pi_1\mathrm{P} &:= \mathrm{coNP}.\end{aligned}$$

Für $i \geq 1$ definieren wir:

$$\begin{aligned}\Delta_{i+1}\mathrm{P} &:= \mathrm{P}^{\Sigma_i\mathrm{P}} = \mathrm{P}^{\Pi_i\mathrm{P}},\\ \Sigma_{i+1}\mathrm{P} &:= \mathrm{NP}^{\Sigma_i\mathrm{P}} = \mathrm{NP}^{\Pi_i\mathrm{P}},\\ \Pi_{i+1}\mathrm{P} &:= \mathrm{coNP}^{\Sigma_i\mathrm{P}} = \mathrm{coNP}^{\Pi_i\mathrm{P}}.\end{aligned}$$

Weiter ist

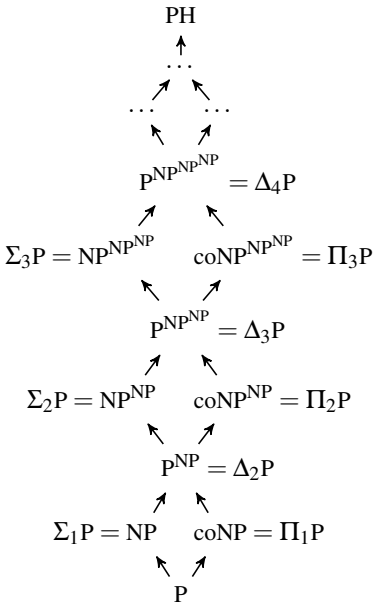
$$\mathrm{PH} = \bigcup_i \Sigma_i\mathrm{P} = \bigcup_i \Pi_i\mathrm{P}.$$

Es gilt also beispielsweise:

$$\begin{aligned}\Sigma_2\mathrm{P} &= \mathrm{NP}^{\mathrm{NP}} = \mathrm{NP}^{\mathrm{coNP}},\\ \Sigma_3\mathrm{P} &= \mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}},\\ \Pi_4\mathrm{P} &= \mathrm{coNP}^{\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}}}.\end{aligned}$$

8-7

Wie die Klassen zueinander liegen.



8-8

Beispiele von Sprachen in der Polynomiellen Hierarchie.

▶ **Lemma**
SHORTER-FORMULA $\in \Sigma_2 P = NP^{NP} = NP^{coNP}$.

Beweis. Wir bauen eine Maschine mit Wahlband, die TAUTOLOGY als Orakel benutzt.[1] Sei φ die Eingabe, für die wir prüfen müssen, ob es ein kürzeres äquivalentes ψ gibt: Die Maschine liest ψ vom Wahlband. Dann fragt sie ihr Orakel, ob $\varphi \leftrightarrow \psi$ eine Tautologie ist. □

Kommentare zum Beweis

[1] Es gilt ja $NP^{NP} = NP^{coNP}$, weshalb wir tatsächlich dieses Orakel nehmen können.

Sei SHORTEST-FORMULA die Sprache, die alle Formeln φ enthält, für die es *keine* kürzere äquivalente Formel gibt.

▶ **Lemma**
SHORTEST-FORMULA $\in \Pi_2 P = coNP^{NP} = coNP^{coNP}$.

Beweis. Die Sprache ist gerade das Komplement von SHORTER-FORMULA (bis auf syntaktisch inkorrekte Worte, die man leicht herausfiltern kann). □

Sei EXACT-TSP die Sprache, die (die Codes von) einem gewichteten Graphen G und einer Zahl k enthält, so dass die kürzeste Rundreise in G genau Länge k hat.

▶ **Lemma**
EXACT-TSP $\in \Delta_2 P = P^{NP} = P^{coNP}$.

Beweis. Bei Eingabe (G,k) fragen wir zunächst das Orakel TSP, ob $(G,k) \in$ TSP, ob also die kürzeste Rundreise Länge *höchstens k hat.* Lautet die Antwort »Ja«, fragen wir noch, ob auch $(G,k-1) \in$ TSP. Lautet die Antwort wieder »Ja«, so verwerfen wir, sonst akzeptieren wir. □

8-9

✎ **Zur Übung**
Ordnen Sie folgendes Problem in die Polynomielle Hierarchie PH ein:

▶ **Problem:** Sprache CHROMATIC-NUMBER

Instanzen Ein ungerichteter Graph $G = (V,E)$ und eine Zahl k.

Frage Ist die chromatische Zahl von G gleich k?

(Die chromatische Zahl eines Graphen ist die kleineste Anzahl an Farben, die nötig ist, um den Graphen zu färben.)

8.1.3 Die Stockmeyer-Charakterisierung: Erste Stufen

Die Idee hinter der Stockmeyer-Charakterisierung

8-10

Wenn man mit NP arbeiten möchte. . .

Obwohl NP mittels nichtdeterministischen Berechnungen definiert ist, nutzt man zum »Arbeiten« mit NP viel lieber Prover-Verifier-Protokolle. Will man beispielsweise »FOOBAR $\in$ NP« zeigen, so zeigt man »für eine Instanz für FOOBAR sind Lösungen Foos – und Foos kann man natürlich in polynomieller Zeit überprüfen; qed«.

Wenn man mit PH arbeiten möchte. . .

Obwohl die Klassen in PH mittels nichtdeterministischen Berechnungen plus Orakeln definiert sind, nutzt man zum »Arbeiten« mit PH viel lieber erweiterte Prover-Verifier-Protokolle.

Zur Erinnerung: Die ersten Level der Hierarchie

8-11

Charakterisierung von NP und coNP (per Definition)

Eine Sprache L ist genau dann in $\mathrm{NP} = \Sigma_1\mathrm{P}$, wenn es eine deterministische Polynomialzeit-Turingmaschine M mit Wahlband gibt, so dass $x \in L$ genau dann gilt, wenn es einen Inhalt für das Wahlband *gibt,* so dass M bei Eingabe x akzeptiert.
Analog ist L genau dann in $\mathrm{coNP} = \Pi_1\mathrm{P}$, wenn *. . . (wie oben) . . .*, wenn *für jeden* Inhalt für das Wahlband M bei Eingabe x akzeptiert.

Charakterisierung der zweiten Stufe der Hierarchie

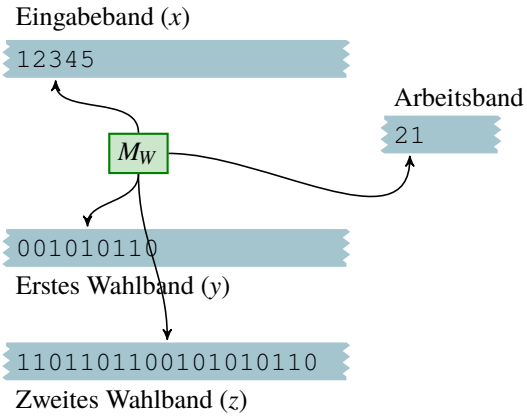
8-12

▶ Satz: Charakterisierung der zweiten Stufe der Hierarchie
Eine Sprache L ist genau dann in $\mathrm{NP}^{\mathrm{NP}} = \Sigma_2\mathrm{P}$, wenn

1. *es eine deterministische Polynomialzeit-Turingmaschine M_W mit zwei Wahlbändern gibt, so dass $x \in L$ genau dann gilt, wenn*
2. *es einen Inhalt y für das erste Wahlband gibt, so dass für jeden Inhalt z für das zweite Wahlband M_W bei Eingabe x akzeptiert.*

Analog ist L genau dann in $\mathrm{coNP}^{\mathrm{NP}} = \Pi_2\mathrm{P}$, wenn

1. . . . (wie oben) . . .
2. *für jeden Inhalt y für das erste Wahlband es einen Inhalt z für das zweite Wahlband gibt, so dass M_W bei Eingabe x akzeptiert.*



Beweis der einfachen Richtung.

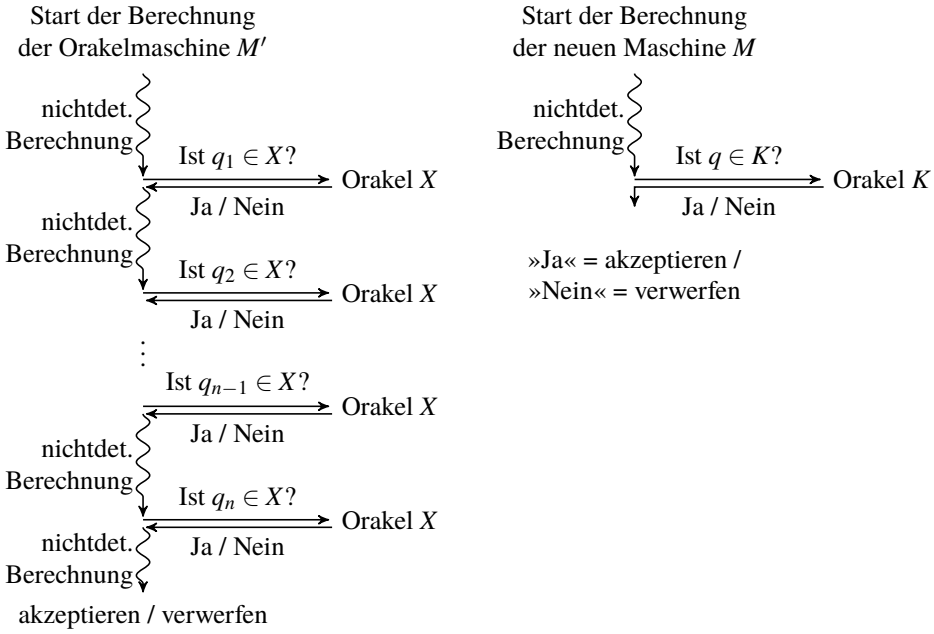
8-13

Offenbar reicht es, nur die erste Behauptung zu zeigen.
Nehmen wir an, L wird von einer Maschine M_W mit zwei Wahlbändern entschieden. Betrachten wir folgende Sprache $K = \{x\#y \mid$ für alle möglichen Inhalte z für das zweite Wahlband gilt, dass M_W das Eingabewort x akzeptiert, wenn auf dem ersten Wahlband y steht und auf dem zweiten $z\}$. Jetzt gilt $K \in \mathrm{coNP}$. Dann gilt auch $L \in \mathrm{NP}^K$: Bei Eingabe x »raten« wir zunächst nichtdeterministisch einen Inhalt y für das erste Wahlband und fragen dann das Orakel K, ob $x\#y \in K$ gilt.

8-14

Vorüberlegungen zur zweiten Beweisrichtung.

Der Beweis der ersten Beweisrichtung zeigte, dass die Arbeitsweise der Maschine M grob folgende ist: »Rate nichtdeterministisch ein y, frage dann einmal ein coNP-Orakel K und übernimm die Antwort.« Für die andere Beweisrichtung haben wir nun aber eine Orakel-Maschine M', die wie folgt arbeitet: »Rate nichtdeterministisch etwas, frage ein coNP-Orakel X (oder NP-Orakel, das ist egal), rate wieder etwas, frage wieder X, rate wieder, und so weiter.« Was wir also zeigen müssen, ist, dass wir statt *vieler Fragen an das* coNP-*Orakel* mit *einer einzigen* auskommen (und wir auch die Antwort direkt übernehmen).



8-15

Umwandlung vieler Fragen in eine Frage: Das Setting

Das Setting

Wir haben eine Orakel-Maschine M' gegeben, die für ein Wort x viele Fragen an ein Orakel X stellt. Die Maschine M' ist nichtdeterministisch, hat also ein Wahlband mit einem Inhalt b. Für jedes mögliche b stellt die Maschine Fragen $q_1, q_2, \ldots, q_n$. Welche Fragen gestellt werden, kann von b abhängen.

Unser Ziel

Wir möchten nun eine neue Orakel-Maschine M bauen, die für ein Wort x nur noch eine Frage q an ein $K \in$ coNP stellt und die Antwort übernimmt.

8-16

Der Trick: Wie man die Fragen loswird

Die Maschine M versucht, *die Fragen an das Orakel so weit wie möglich zu »erahnen« und dann »selbst zu beantworten«.* Sie »rät« dazu am Anfang (formell: sie interpretiert den Anfang ihres Wahlbandes als)

1. den Inhalt b des Wahlbandes von M' (also, welche nichtdeterministischen Entscheidungen M' fällen wird), gefolgt von
2. den Fragen $q_1, \ldots, q_n$, die M' stellen wird,
3. und den Antworten, die das Orakel X hierauf geben wird.

Nun simuliert sie M' deterministisch für dieses Wahlband und diese Fragen und Antworten und überprüft, ob M' tatsächlich diese Fragen stellen würde und dann akzeptieren würde. *Ist dies der Fall,* so muss M noch überprüfen, ob die sie am Anfang tatsächlich *die richtigen Antworten geraten hat.* (Das machen wir gleich)

8-17

Wie man die Fragen überprüft.

Wo wir stehen

Eine Maschine M hat »Fragen $q_1, \ldots, q_n$ an ein Orakel $X \in$ NP« und »erwartete Antworten $a_1, \ldots, a_n \in \{\text{ja}, \text{nein}\}$« von ihrem Wahlband gelesen. Die Maschine muss nun noch überprüfen, ob die Antworten stimmen. Das Problem von M ist, dass sie nur *eine* Orakel-Frage stellen darf (mit n Fragen wäre die Überprüfung ja leicht).

Was M für Fragen q_i mit $a_i =$ ja macht

Das Orakel X sei in NP via einer Maschine M_X. Für alle Wörter in X gibt es dann einen Inhalt für das Wahlband von M_X, so dass M_X akzeptiert (in polynomieller Zeit). Die Maschine M macht daher folgendes: Für alle Fragen q_i mit $a_i =$ ja liest sie von *ihrem* Wahlband ein Zertifikat und überprüft, ob M_X nun tatsächlich q_i akzeptiert.
Die entscheidende Beobachtung ist: Stehen auf dem Wahlband von M gerade die richtigen Zertifikate hintereinander, so werden die Simulationen bestätigen, dass tatsächlich $q_i \in X$ für alle i mit $a_i =$ ja gilt.

Was M für Fragen q_i mit $a_i =$ nein macht

Als nächstes muss M nun noch folgendes beantworten:

> Ist es wahr, dass für *alle* q_i mit $a_i =$ nein die Maschine M_X für *alle* möglichen Zertifikate verwirft?

Dies ist »eine coNP-Frage«: Genauer liegt die Sprache $Y = \{q_1\#\ldots\#q_m \mid q_i \notin X \text{ für } i \in \{1,\ldots,n\}\}$ in coNP. Also kann M die obige Frage mit *einer Frage* an Y beantworten und M kann sogar die Antwort *übernehmen*.

Ein Beispiel

8-18

Betrachten wir als Beispiel für die Konstruktion der Stockmeyer-Charakterisierung die folgende (etwas künstliche) Sprache:

▶ Problem: Sprache ONE-ISOMORPH

Instanzen Ein (kodierter) Graph G und eine (kodierte) Menge $M = \{G_1,\ldots,G_n\}$ von Graphen.

Frage Gibt es genau einen Graphen in M, der isomorph zu G ist?

Diese Sprache ist in Σ_2P (sogar in Δ_2P) via folgendem Algorithmus:

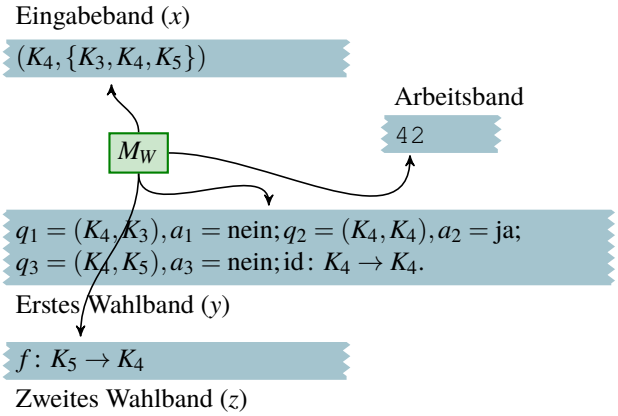
```
1 input G, M = {G_1,...,G_n}
2 c ← 0
3 for i ∈ {1,...,n} do
4   frage das Orakel GRAPH-ISOMOPHIE, ob G und G_i isomorph sind
5   if Antwort = ja then
6     c ← c +1
7 if c = 1 then akzeptiere else verwerfe
```

Betrachten wir nun, was die Maschine M bei der Eingabe $(K_4, \{K_3, K_4, K_5\})$ macht (der K_n ist die vollständige Graph mit n Knoten):

Beispiel einer akzeptierenden Berechnung von M

Zunächst »rät« M (= auf dem Wahlband von M steht), dass die Orakelanfragen an das Orakel GI sein werden (K_4, K_3), dann (K_4, K_4) und dann (K_4, K_5). Weiter »rät« M, dass die Antworten sein werden »nein«, »ja«, »nein«. Dann stellt simuliert M den Algorithmus von M' und stellt fest, dass tatsächlich genau diese Fragen gestellt werden und dass M' bei diesen Antworten akzeptiert.
Nun überprüft M die »Ja«-Antworten. Dazu »rät« sie für die einzige »Ja«-Antwort einen Isomorphismus zwischen K_4 und K_4 (was nicht schwierig ist…). Da dies geklappt hat, überprüft M nun die »Nein«-Antworten, indem sie das Orakel Y mit einer Frage fragt, ob sowohl K_3 und K_4 nicht isomorph sind und auch gleichzeitig K_5 und K_4.

Aus M wird M_W: Die Orakelanfrage wird durch das zweite Wahlband ersetzt



8-19 Alternative Formulierung der Charakterisierung.

▶ Satz: Alternative Formulierung der Charakterisierung
Es gilt $L \in \Sigma_2\mathrm{P}$ genau dann, wenn ein $W \in \mathrm{P}$ existiert mit

$$L = \{x \mid \exists y \forall z\colon x\#y\#z \in W, |y| = |z| = |x|^{O(1)}\}.$$

Es gilt $L \in \Pi_2\mathrm{P}$ genau dann, wenn ein $W \in \mathrm{P}$ existiert mit

$$L = \{x \mid \forall y \exists z\colon x\#y\#z \in W, |y| = |z| = |x|^{O(1)}\}.$$

Beweis. Die Maschinen M_W aus der originalen Charakterisierung sind alle deterministisch und polynomiell zeitbeschränkt. Für jeden konkreten Inhalt der Wahlbänder fällt M_W also in polynomieller Zeit eine Entscheidung. Insbesondere ist die Sprache $W = \{x\#y\#z \mid$ die Maschine M_W akzeptiert x, wenn das erste Wahlband mit y belegt ist und das zweite mit $z\}$ in der Klasse P. □

8.1.4 Die Stockmeyer-Charakterisierung: Höhere Stufen

8-20 Die Stockmeyer-Charakterisierung für beliebige Stufen.

▶ Satz
Es gilt $L \in \Sigma_i\mathrm{P}$ genau dann, wenn ein $W \in \mathrm{P}$ existiert mit

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \forall y_4 \cdots \mathrm{Q}\, y_i\colon x\#y_1\#\ldots\#y_n \in W, |y_i| = |x|^{O(1)}\}.$$

Es gilt $L \in \Pi_i\mathrm{P}$ genau dann, wenn ein $W \in \mathrm{P}$ existiert mit

$$L = \{x \mid \forall y_1 \exists y_2 \forall y_3 \exists y_4 \cdots \mathrm{Q}\, y_i\colon x\#y_1\#\ldots\#y_n \in W, |y_i| = |x|^{O(1)}\}.$$

Dabei steht »Q« passend für »∃« oder »∀«.

Beweis. Per Induktion über i; der Induktionsschritt ist genau das Argument für $i = 2$ von oben. □

8.2 Kollapsresultate

8.2.1 Kollapslemma

Ist die Hierarchie eine echte Hierarchie?

8-21

▶ Satz

Falls $\Sigma_i\mathrm{P} = \Sigma_{i+1}\mathrm{P}$*, so gilt* $\Sigma_i\mathrm{P} = \Sigma_j\mathrm{P}$ *für alle* $j \geq i$*. Genauso folgt aus* $\Pi_i\mathrm{P} = \Pi_{i+1}\mathrm{P}$ *auch* $\Pi_i\mathrm{P} = \Pi_j\mathrm{P}$ *für alle* $j \geq i$*.*

Beweis. Dies folgt unmittelbar aus den Definitionen. □

Aufgrund diese »Kollapslemmas« wissen wir:

1. Die polynomielle Hierarchie kann entweder komplett eine »echte Hierarchie sein« (also $\Sigma_i\mathrm{P} \subsetneq \Sigma_{i+1}\mathrm{P}$ für alle i) oder
2. sie ist nur echt bis zu einem bestimmten i und danach sind alle Stufen gleich.

Im Fall 2 sagt man, *»die Polynomielle Hierarchie kollabiert auch die i-te Stufe«* und auch *»die Polynomielle Hierarchie kollabiert«.*

8.2.2 Hat SAT polynomiell große Schaltkreise?

Hat SAT polynomiell große Schaltkreise?

8-22

▶ Lemma

Falls SAT *von einer logspace-uniformen Schaltkreisfamilie polynomieller Größe entschieden werden kann, so gilt* $\mathrm{P} = \mathrm{NP}$*.*

Beweis. Sei C die Schaltkreisfamilie. Für eine Eingabeformel φ kann eine Polynomialzeitmaschine zunächst den Schaltkreis $C_{|\varphi|}$ ausrechnen (wegen der Logspace-Uniformität) und dann $C_{|\varphi|}(\varphi)$ auswerten (wegen $\mathrm{CVP} \in \mathrm{P}$). Also gilt $\mathrm{SAT} \in \mathrm{P}$. □

Große offene Frage

Gilt das Lemma auch, wenn man »logspace-uniform« streicht?

Was passiert, wenn SAT polynomiell große Schaltkreise hat.

8-23

▶ Satz

Falls SAT *von einer (nicht unbedingt uniformen) Schaltkreisfamilie polynomiell Größe entschieden werden kann, so kollabiert die Polynomielle Hierarchie auf die zweite Stufe.*

Beweis. Sei $L \in \Sigma_3\mathrm{P}$. Dann ist $L = L(M_1^X)$ für ein X mit $X = L(M_2^{\mathrm{SAT}})$ für zwei NP-Orakelmaschinen $M_1^?$ und $M_2^?$. Wir wollen zeigen, dass $L \in \Sigma_2\mathrm{P}$ via einer Maschine $M^?$ gilt. Sei dazu x eine Eingabe. Zunächst »rät« $M^?$ Schaltkreise $C_0, C_1, \ldots, C_{p(n)}$ bis zu einer geeigneten Länge $p(n)$, von denen sie »hofft«, dass sie SAT korrekt für Formeln bis Länge $p(n)$ entscheiden. *Nach Voraussetzung gibt es solche Schaltkreise.*
Wir schreiben nun $C(\varphi)$ für die Ausgabe $C_{|\varphi|}(\varphi)$, also dafür, ob die Schaltkreise »behaupten«, dass $\varphi \in \mathrm{SAT}$. Die Maschine $M^?$ fragt nun ihr Orakel die folgende Frage:

> *Ist es wahr, dass für alle Formeln* φ *der Länge höchstens* $p(n)$ *gilt:*
>
> 1. *Enthält* φ *keine Variablen, so gilt* $C(\varphi) = 1$ *genau dann, wenn* φ *zu wahr auswertet.*
> 2. *Enthält* φ *eine Variable* x*, so gilt* $C(\varphi) = 1$ *genau dann, wenn* $C(\varphi[x \hookrightarrow \mathtt{f}]) = 1$ *oder* $C(\varphi[x \hookrightarrow \mathtt{w}]) = 1$*.*

Die entscheidende Beobachtung ist, dass (a) diese Frage von einem coNP-Orakel beantwortet werden kann und (b) wenn sie mit ja beantwortet wird, die Schaltkreise tatsächlich SAT korrekt für alle Formeln der Länge maximal $p(n)$ entscheiden.
An dieser Stelle hat nun die Maschine $M^?$ effektiv in Form der Schaltkreise eine Methode, SAT deterministisch in polynomieller Zeit zu entscheiden. Nun galt $X = L(M_2^{\mathrm{SAT}})$. Die Orakelanfragen von M_2 an SAT kann $M^?$ nun aber »selbst, deterministisch« beantworten, so dass M_2 nur noch eine reine NP-Berechnung durchführt. Dann kann aber die Berechnung von M_2 wiederum auf SAT reduziert werden und somit kann $M^?$ die gesamte Berechnung von M_2 »selbst, deterministisch« durchführen. Schließlich wird damit die Berechnung von M_1 eine normale NP-Berechnung und ist damit schließlich auch selbst wieder von $M^?$ direkt simulierbar. □

Zusammenfassung dieses Kapitels

8-24

▶ Definition der Polynomiellen Hierarchie

$$\begin{aligned}\Delta_1\mathrm{P} &:= \mathrm{P},\\ \Sigma_1\mathrm{P} &:= \mathrm{NP},\\ \Pi_1\mathrm{P} &:= \mathrm{coNP},\\ \Delta_{i+1}\mathrm{P} &:= \mathrm{P}^{\Sigma_i\mathrm{P}} = \mathrm{P}^{\Pi_i\mathrm{P}},\\ \Sigma_{i+1}\mathrm{P} &:= \mathrm{NP}^{\Sigma_i\mathrm{P}} = \mathrm{NP}^{\Pi_i\mathrm{P}},\\ \Pi_{i+1}\mathrm{P} &:= \mathrm{coNP}^{\Sigma_i\mathrm{P}} = \mathrm{coNP}^{\Pi_i\mathrm{P}}\\ \mathrm{PH} &:= \bigcup_i \Sigma_i\mathrm{P} = \bigcup_i \Pi_i\mathrm{P}.\end{aligned}$$

▶ Die Stockmeyer-Charakterisierung

Es gilt $L \in \Sigma_i\mathrm{P}$ genau dann, wenn ein $W \in \mathrm{P}$ existiert mit

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \forall y_4 \cdots \mathrm{Q}y_i \colon x\#y_1\#\ldots\#y_n \in W, |y_i| = |x|^{O(1)}\}.$$

Es gilt $L \in \Pi_i\mathrm{P}$ genau dann, wenn ein $W \in \mathrm{P}$ existiert mit

$$L = \{x \mid \forall y_1 \exists y_2 \forall y_3 \exists y_4 \cdots \mathrm{Q}y_i \colon x\#y_1\#\ldots\#y_n \in W, |y_i| = |x|^{O(1)}\}.$$

▶ Hat NP polynomiell große Schaltkreise?

Falls SAT von logspace-uniformen Schaltkreisen polynomieller Größe entschieden werden kann, so ist $\mathrm{P} = \mathrm{PH}$. (Die Polynomielle Hierarchie kollabiert auf die nullte Stufe.)
Falls SAT von Schaltkreisen polynomieller Größe entschieden werden kann, so ist $\mathrm{NP}^{\mathrm{NP}} = \mathrm{PH}$. (Die Polynomielle Hierarchie kollabiert auf die zweite Stufe.)

Zum Weiterlesen

[1] L. J. Stockmeyer. The polynomial hierarchy, *Theoretical Computer Science,* 4:1–22, 1976.

Die Definition der Polynomiellen Hierarchie ist schon etwas älter. Die Stockmeyer-Charakterisierung findet sich auch in dieser Arbeit.

Übungen zu diesem Kapitel

Übung 8.1 Komplexität des exakten TSP-Problems, schwer

Sei UNIQUE-TSP die Sprache, die alle (Codes von) Instanzen für das Travelling-Salesperson-Problem enthält, die genau eine minimale Rundreise haben. Zeigen Sie UNIQUE-TSP $\in \Delta_2\mathrm{P}$.

Übung 8.2 Komplexität von knappen Graphproblemen, mittel

Bestimmen Sie, in welchen Stufen der Polynomiellen Hierarchie die folgenden Sprachen liegen:

▶ Problem: Sprache SUCCINCT-SMALL-DIAMETER

Instanzen Ein Schaltkreis C mit $2n$ Eingängen, der einen ungerichteten Graphen $G = (\{0,1\}^n, E)$ kodiert, und eine Zahl $k \leq n$.

Frage Gibt es von jedem Knoten von G zu jedem anderen einen Weg der Länge höchstens k?

▶ Problem: Sprache SUCCINCT-SMALL-RADIUS

Instanzen Ein Schaltkreis C mit $2n$ Eingängen, der einen ungerichteten Graphen $G = (\{0,1\}^n, E)$ kodiert, und eine Zahl $k \leq n$.

Frage Ist der Radius von G höchstens k?

(Der Radius eines Graphen ist der kleinste Zahl r, so dass es einen Knoten gibt, von dem alle Knoten in r Schritten erreichbar sind.)

Übung 8.3 Vollständigkeit von knappen Graphproblemen, sehr schwer

Zeigen Sie, dass die Probleme aus Übung 8.2 sogar vollständig sind für die Stufen der Polynomiellen Hierarchie, die Sie identifiziert haben. Sie brauchen Vollständigkeit nur für Polynomialzeit-Many-One-Reduktionen zeigen (es geht aber auch für Logspace-Many-One-Reduktionen).

Tipp: Nutzen Sie die Stockmeyer-Charakterisierung und Bootstrapping. Nehmen Sie für SUCCINCT-SMALL-DIAMETER eine beliebige Sprache L her, die sich schreiben lässt als $\{x \mid \forall y \exists z\colon x\#y\#z \in W, |y| = |z| = |x|^{O(1)}\}$. Bauen Sie nun einen großen Graphen, der aus drei Cliquen besteht. Die erste Clique enthält einen Knoten für jedes mögliche y, die zweite Clique enthält einen Knoten für jedes mögliche z, die dritte »Clique« besteht nur aus einem Knoten v. Es gibt nun eine Kanten von v zu jedem »z-Knoten«. Weiter gibt es eine Kante von einem »y-Knoten« zu einem »z-Knoten« genau dann, wenn $x\#y\#z \in W$ gilt. Der resultierende Graph hat Durchmesser 2 genau dann, wenn $x \in L$.

Für SUCCINCT-SMALL-RADIUS müssen Sie die Konstruktion anpassen und viele solche Graphen nebeneinanderlegen und über einen zentralen Knoten verbinden, der dann als einziger als Mittelpunkt des Graphen in Frage kommen darf.

Wer diese Dinge genauer wissen möchte, dem sei ein Blick in folgendes Paper empfohlen:

Literatur

[1] Edith Hemaspaandra, Lane A. Hemaspaandra, Till Tantau und Osamu Watanabe. On the complexity of kings. *Theoretical Computer Science*, 411(2010):783–798, 2010.

Übung 8.4 Die Polynomielle Hierachie und polynomieller Platz, leicht

Zeigen Sie PH $\subseteq$ PSPACE.

Tipp: Nutzen Sie die Stockmeyer-Charakterisierung.

Übung 8.5 Strecken der Polynomiellen Hierarchie lässt sie kollabieren, schwer

Zeigen Sie, dass aus PH = PSPACE folgt, dass die Polynomielle Hierarchie kollabiert.

Tipp: Sie dürfen ausnutzen, dass es PSPACE-vollständige Probleme gibt.

Übung 8.6 Charakterisierung von P mit Schaltkreisen polynomieller Größe, mittel

Sei POLYSIZE die Klasse aller Sprachen, die von logspace-uniformen Schaltkreisen polynomieller Größe entschieden werden (die Klasse ist also definiert wie NC^i und AC^i, nur ohne die Einschränkung an die Tiefe). Zeigen Sie POLYSIZE = P.

Teil VI

Lösen schwerer Probleme

Sie sind von einem kleinen Forschungs-Startup als Informatik-Spezialistin eingestellt worden. Sie lieben die Firma, die Leute sind cool, die Forschungsthemen spannend und – mit Abstand am wichtigsten – es gibt Double Chocolaty Chip Frappuccino® Blended Crème in der Variante Venti Decaf Lowfat umsonst! Nach ein paar Tagen stellt sich heraus, dass Sie insbesondere deshalb eingestellt wurden, da ihre Chefin in Ihrem Zeugnis gesehen hat, dass Sie eine Vorlesung zur Komplexitätstheorie besucht haben: Die Forschungsabteilung hat ein mathematisches Optimierungsproblem, das sie gerne gelöst hätte und Sie werden gebeten, das doch mal in Angriff zu nehmen. (Welches Problem dies genau ist, kann ich hier nicht verraten, da Sie jede Menge Non-Disclosure-Agreements unterschrieben haben. Nennen wir seine Entscheidungsvariante einfach BIG-SECRET.)

Ihre – zumindest im Vergleich zur Bezahlung im öffentlichen Dienst – üppige Bezahlung ist in der Tat nicht ganz ungerechtfertig: Sie haben in der Vorlesung gut aufgepasst und stellen schnell fest, dass BIG-SECRET leider NP-vollständig ist. Dementsprechend gehen Sie bald zu Ihrer Chefin und teilen ihr befriedigt mit »Ah, übrigens, BIG-SECRET ist NP-vollständig.« Die Antwort »Ja, und?« ist dann doch etwas überraschend. Etwas nervös antworten Sie »Also, das heißt, dass das Problem, nun ja, nicht gelöst werden kann.« Etwas sicherer fahren Sie fort: »Also, ich meine, es hat noch *niemand* einen effizienten für dieses oder für tausende ähnlich gelagerte Probleme gefunden.« Ihre Chefin schaut Sie etwas länger an und teilt schließlich mit: »Wir haben dich angestellt, das Problem zu lösen. Entweder du findest eine neue Lösung für das Problem oder wir finden eine neue Lösung für deinen Arbeitsplatz.«

In diesem Teil geht es darum, wie Sie Ihren Job behalten. Wenn ein Problem NP-vollständig ist, heißt dies noch nicht, dass es nicht praktisch gelöst werden kann. Es bedeutet einfach, dass unter der Annahme $\mathrm{P} \neq \mathrm{NP}$ das Problem nicht für *alle* Eingaben in polynomieller Zeit *perfekt* gelöst werden kann. Es kann aber sehr wohl sein, dass es für *Ihre* Eingaben leicht lösbar ist oder dass es *fast* perfekt lösbar ist.

9-1

Kapitel 9
Fixed-Parameter-Algorithmen

Wie man Problem schnell löst, die sich nicht schnell lösen lassen

Lernziele dieses Kapitels

1. Konzept der Problemparameter verstehen
2. Beispiele von FPT-Algorithmen kennen
3. Konzept der FPT-Intractability kennen

9-2

Inhalte dieses Kapitels

Worum es heute geht

In diesem Kapitel gehen wir einen Pakt mit dem Teufel ein, wie Rod Downey und Mike Fellows es beschreiben (beide sind wirklich nette Menschen, von denen man eigentlich nicht erwarten würde, dass sie Pakte mit Personen wie dem Teufel eingehen). Um bestimmte NP-vollständige Probleme zu lösen, geben wir dem Teufel, was er möchte: eine exponentielle Laufzeit. Jedoch bringt *uns* der Deal auch etwas: Die Laufzeit ist nur exponentiell in Bezug auf einen *festen Parameter* (daher der Name »Fixed-Parameter-Algorithmen«), der in der Praxis (hoffentlich) sehr klein ist – selbst dann, wenn die Eingabe ansonsten riesig groß ist. Dieser faustische Pakt ermöglicht es uns, einige NP-vollständige Probleme – wie VERTEX-COVER – in Sekunden zu lösen auf Eingaben mit Tausenden oder gar Millionen an Knoten, falls denn die gesuchte Knotenüberdeckung nicht allzu groß ist.

Wenn man einmal damit begonnen hat, nach »kleinen Parametern« in Problemen zu suchen, entdeckt man sie überall. Für Färbbarkeit ist beispielsweise die Anzahl der Farben ein natürlicher und in der Regel sehr kleiner Parameter (meist suchen wir ja 3- oder 4-Färbungen). Bei Problemen aus der Bioinformatik wird die Anzahl an Fehleinträgen in den Eingaben hoffentlich klein sein (ist sie sehr groß, kann man Daten eh' wegschmeißen). Bei Anfragen an eine relationale Datenbank werden in den SQL-Ausdrücken sicherlich nur wenige Verschachtelungen von Subqueries zu finden sein, wohingegen die Datenbanken, auf die sich die Anfragen beziehen, ja riesig sein können. In diesen und vielen anderen Fällen ist jedesmal unsere Hoffnung, einen Pakt mit dem Teufel eingehen zu können, so dass die »kombinatorische Explosion« sich auf den (kleinen) Parameter einschränken lässt und uns das Problem nicht komplett um die Ohren fliegt.

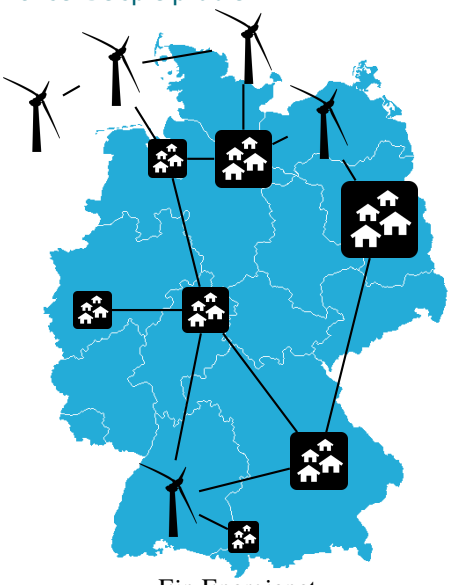
Leider zeigt eine genauere Untersuchung des Kleingedruckten in den Verträgen mit dem Teufel, dass dieser, nun ja, teuflisch ist. Es stellt sich nämlich heraus, dass für manche Probleme – wie Färbbarkeit – die kombinatorische Explosion nicht vermieden werden kann (es sei denn natürlich, dass $P = NP$ gilt). Bei anderen – wie eben VERTEX-COVER – kann sie vermieden werden. Schließlich gibt es noch viele Probleme – wie das Cliquen-Problem – bei denen wir weder das eine noch das andere wissen.

9.1 Einleitung

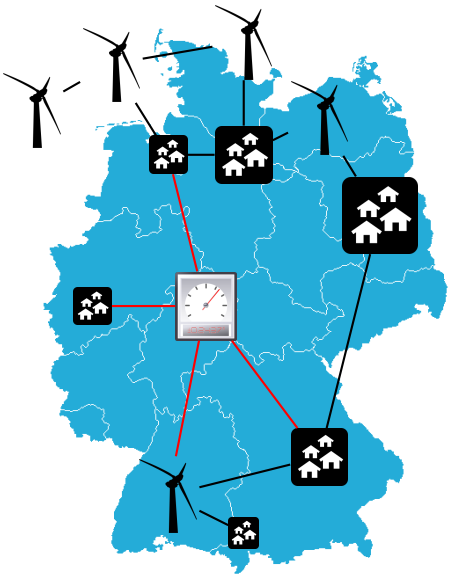
9.1.1 Das Schlüsselproblem: Vertex-Cover

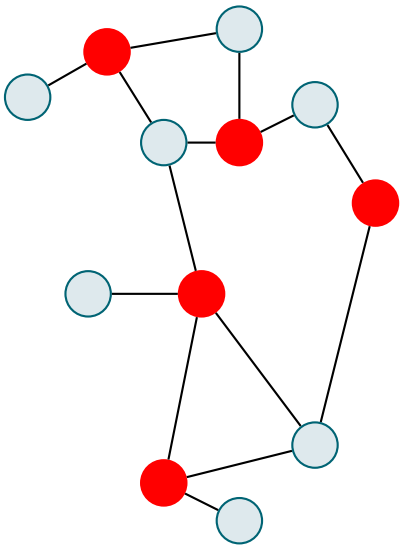
9-4

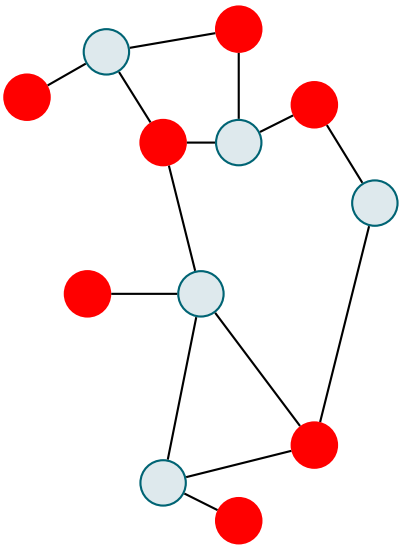
Unser Beispielproblem



Ein Energienetz


Eine Kontrollstation


Eine Knotenüberdeckung (Vertex-Cover)


Eine unabhängige Menge

- Ein Netzbetreiber möchte *Kontrollstationen* für Stromleitungen installieren.
- Eine Kontrollstation kann alle *angrenzenden Leitungen* kontrollieren.

Das Vertex-Cover-Problem

Für einen Graph G und eine Zahl k entscheide: Gibt es k Knoten, so dass jede Kante einen dieser Knoten als Endpunkt hat?

Szenarien, wo man kleine Vertex-Covers sucht

- Netzwerk-Monitoring
- Suche nach großen unabhängigen Mengen
- Suche nach großen Cliquen, speziell in der Bioinformatik:
  Gene-Clustering, Phylogenie-Analyse, Gen-Motiv-Analyse, ...

Vertex-Cover ist ein schwieriges Problem

9-5

Vertex-Cover ist eines der 21 NP-*vollständigen Probleme* von Richard Karp.

Public Domain

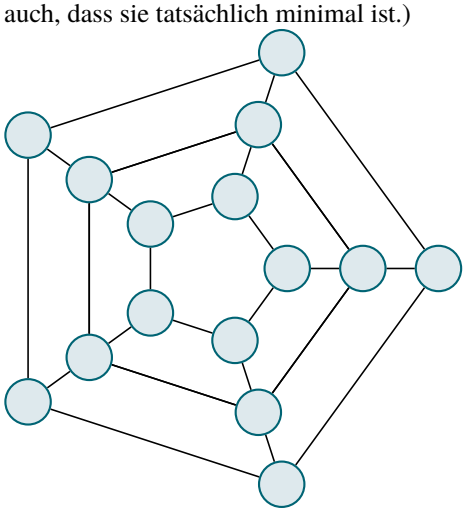
Wenn wir es *trotzdem* lösen wollen:

- Bei Graphen mit n Knoten und m Kanten kann man es offenbar in Zeit $O(n^k m)$ lösen: Probiere alle Möglichkeiten durch, k Knoten auszuwählen.
- Für $n = 1000$ und $k = 50$ ist dies sequentiell hoffnungslos. (Und parallel auch.)

✎ Zur Übung

9-6

Finden Sie eine minimale Knotenüberdeckung des folgenden Graphen (argumentieren Sie auch, dass sie tatsächlich minimal ist.)



9.1.2 Die Idee: Feste Parameter

Problemparameter für das Vertex-Cover-Problem.

9-7

Zur Erinnerung: *Problemparameter* sind Zahlen, die Eingabeinstanzen zugeordnet werden. Für das Vertex-Cover-Problem sind folgenden Parameter »natürlich«:

1. Die Länge l der Eingabe.
2. Die Anzahl n an Knoten im Graphen.
3. Die Anzahl m an Kanten im Graphen.
4. Die Größe k der gesuchten Knotenüberdeckung.

✎ Zur Übung

Geben Sie Algorithmen an, die VERTEX-COVER lösen in Zeit

1. $O(2^n l^{O(1)})$,
2. $O(3^m l^{O(1)})$,
3. $O(2^l l^{O(1)})$.

9.2 Das Vertex-Cover-Problem

9.2.1 Einfacher Fixed-Parameter-Algorithmus

9-8

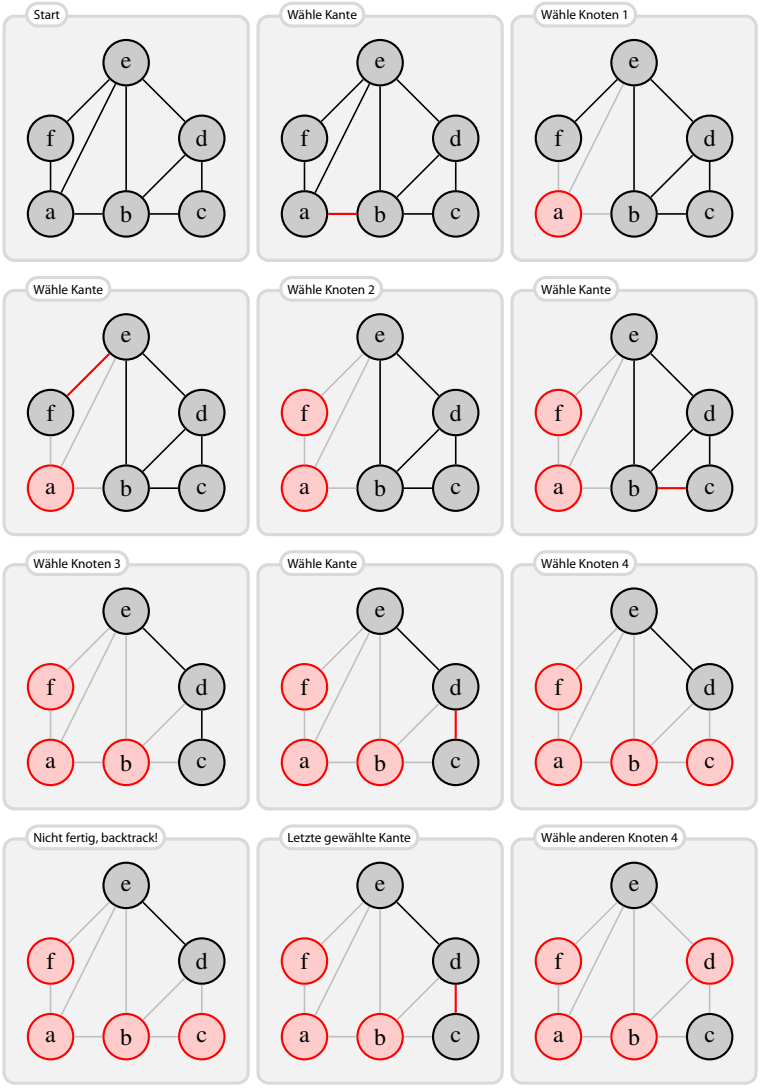
Eine einfache, aber brillante Idee: Backtracking entlang der *Kanten*.

Fellows Algorithmus

```
1  procedure vc-backtrack-edges (G, k)
2      if k < 0 then
3          return false
4      else if G has no edges then
5          return true
6      else
7          pick any edge {x,y} ∈ E
8          foreach v ∈ {x,y} do
9              G' ← G − v
10             if vc-backtrack-edges (G', k − 1) then
11                 return true
12         return false
```

9-9

Fellows Algorithmus in Aktion für $k = 4$.



Die Laufzeit von Fellows Algorithmus.

9-10

▶ Satz
Der Algorithmus ist korrekt und arbeitet in Zeit $O(2^k nm)$.

Beweis. Korrektheit: Wir zeigen die Korrektheit der Ausgabe durch Induktion über die Anzahl an Kanten in G.

- Wenn G keine Kanten hat, ist die Ausgabe offenbar korrekt.
- Wenn G eine Kante $\{x,y\} \in E$ hat, dann hat G eine Knotenüberdeckung der Größe k genau dann, wenn $G-x$ oder $G-y$ Knotenüberdeckungen der Größe $k-1$ haben.

Laufzeit:

- Die Rekursionstiefe ist k.
- Die Anzahl Knoten im Rekurionsbaum ist $N(k) \leq 2N(k-1)+1$ und $N(0)=0$, also $N(k) \leq 2^k - 1$.
- Wir brauchen schließlich Zeit $O(mn)$ pro Knoten. □

9.2.2 Fortgeschrittener Fixed-Parameter-Algorithmus

Eine neue Idee.

9-11

Intuitiv ist es »wahrscheinlicher«, dass ein Knoten mit hohem Grad in einer minimalen Knotenüberdeckung ist als ein Knoten mit kleinem Grad: Hat x Grad 100, so ist entweder x in der Überdeckung *oder alle 100 Nachbarn*. Es scheint also eine gute Idee zu sein, mit Knoten mit hohem Grad anzufangen.
Richtig nett wird die Sache, wenn wir keine Knoten mit hohem Grad mehr haben: Wenn ein Graph keine Knoten mit Grad 3 oder höher mehr enthält, so ist er »ziemlich trivial«: Er besteht nur noch aus *Pfaden und Kreisen*. Für einen solchen Graph ist eine minimal Knotenüberdeckung leicht zu berechnen ohne Rekursion:

- Ein Pfad der Länge n kann optimal mit $\lfloor n/2 \rfloor$ Knoten überdeckt werden,
- ein Kreis der Länge n benötigt $\lceil n/2 \rceil$ Knoten.

Backtracking entlang der *Knoten mit hohem Grad*.

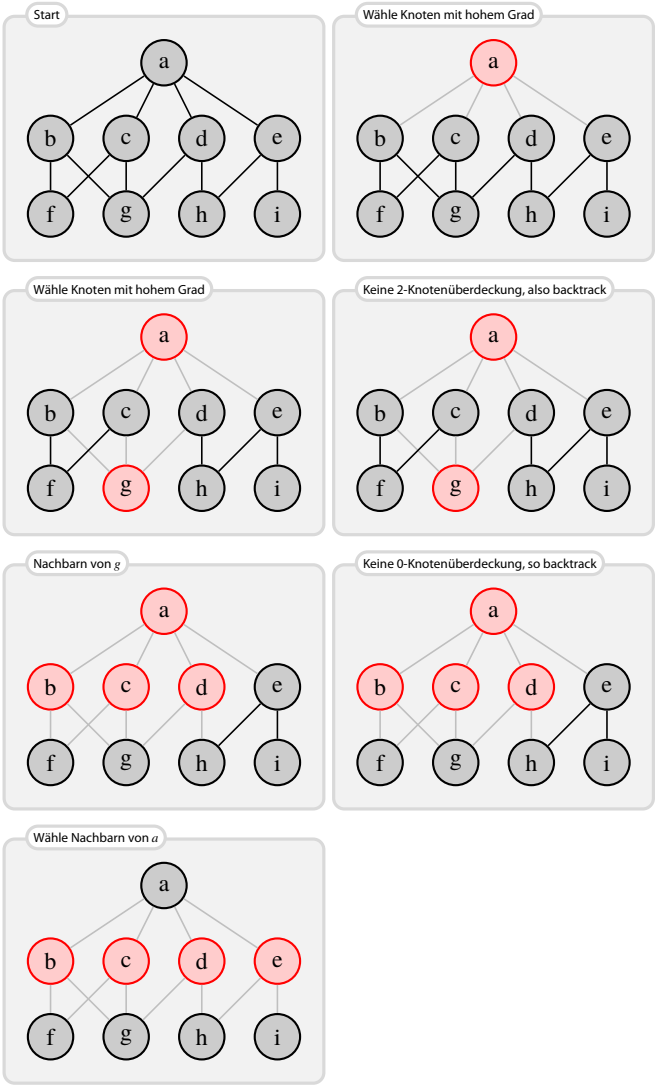
9-12

Backtracking entlang Knoten mit hohem Grad

```
1  procedure vc-backtrack-high-degree (G, k)
2      if k < 0 then
3          return false
4      else if G has no vertex of degree at least 3 then
5          compute the size s of the smallest vertex cover of G
6          return k ≥ s
7      else
8          pick a node x ∈ V of degree 3 or more
9          N ← {v | {x,v} ∈ E}, that is, the neighbors of x
10         foreach S ∈ {{x},N} do
11             if vc-backtrack-high-degree (G − S, k − |S|) then
12                 return true
13         return false
```

9-13

Der Algorithmus in Aktion für $k = 4$.



9-14

Die Laufzeit des Algorithmus.

▶ **Satz**
Der Algorithmus ist korrekt und läuft in Zeit $O(1.49535^k nm)$.

Beweis. Die Korrektheit sollte klar sein. Für die Laufzeit betrachte die Anzahl Knoten im Rekursionsbaum. Sie lässt sich wie folgt abschätzen:

$$\begin{aligned} N(k) &\leq N(k-1) + N(k-3) + 1, \\ N(2) &= 1, \\ N(1) &= 0, \\ N(0) &= 0, \end{aligned}$$

da wir zwei rekursive Aufrufe machen, einen für $k-1$ und einen für $k-|S| \leq k-3$.
Wir behaupten, dass $N(k) \leq 5^{k/4} - 1$ eine Lösung von $N(k) \leq N(k-1) + N(k-3) + 1$ ist: Für $k = 0, 1, 2$ stimmt dies. Für den Induktionsschritt von $k-1$ auf k betrachte:

$$\begin{aligned} N(k) &\leq N(k-1) + N(k-3) + 1 \\ &\leq 5^{k/4-1/4} + 5^{k/4-3/4} - 1 \\ &= \underbrace{(5^{-1/4} + 5^{-3/4})}_{\approx 0.968 < 1} 5^{k/4} - 1. \end{aligned}$$

□

State-of-the-Art-Laufzeit für das Vertex-Cover-Problem. 9-15

▶ Satz: Chen, Kanj, Xia 2006
VERTEX-COVER *kann in Zeit* $O(1.2738^k + nm)$ *gelöst werden.*

Man beachte, dass $1.2738^k < 10^9$ for $k \leq 85$ und dass $O(nm)$ ein *additiver* und *kein multiplikativer* Term ist. Folglich lässt sich VERTEX-COVER für $n = 1000$, $m = 10000$ und $k = 80$ *im Bruchteil einer Sekunde (!) lösen.*

9.3 Allgemein Theorie

9.3.1 Fixed-Parameter-Tractability

Die »Zutaten« der allgemeinen Theorie der Fixed-Parameter-Algorithmen. 9-16

Was wir erreicht haben

Für das Vertex-Cover-Problem haben wir es geschafft, die »kombinatorische Explosion« auf einen einzigen Parameter k zu *beschränken*. In Bezug auf die anderen Parameter hat der Algorithmus eine »vertretbare« Laufzeit. Insbesondere gilt: Ist k »fest«, so ist die Laufzeit ein festes Polynom, dessen Grad nicht von k abhängt.

Übergang zu einer allgemeinen Theorie

Um die obigen Ideen zu verallgemeinern, müssen wir definieren, was

- ein *parametrisiertes Problem* ist und
- was dann *Fixed-Parameter-Tractability* bedeuten soll.

Definition von Parametrisierte Probleme. 9-17

▶ Definition
Ein *parametrisiertes Problem* ist ein Paar (L, κ), wobei

- $L \subseteq \Sigma^*$ eine Sprache ist und
- $\kappa\colon \Sigma^* \to \mathbb{N}$ ein *Parameter*.

Wir schreiben parametrisierte Problem auf die folgende Weise auf:

Beispiel: p_k-VERTEX-COVER

Eingabe Ein Graph G und eine Zahl k.
Parameter k
Frage Hat G eine Knotenüberdeckung der Größe k?

Welchen Parameter wir betrachten, schreiben wir als Index bei »p«, wobei wir den Index auch weglassen, wenn wir den »natürlichen« Parameter betrachten.

Beispiele von parametrisierten Problemen. 9-18

Beispiel: p-CLIQUE

Eingabe Ein Graph G und eine Zahl k.
Parameter k
Frage Enthält G eine Clique der Größe k?

Beispiel: p-COLORABLE

Eingabe Ein Graph G und eine Zahl χ.
Parameter χ
Frage Können wir G mit χ Farben korrekt färben?

Beispiel: p-WEIGHTED-SAT

Eingabe Eine aussagenlogische Formel φ und eine Zahl k.
Parameter k
Frage Gibt es eine erfüllende Belegung von φ, in der genau k Variablen wahr sind?

9-19

Fixed-Parameter-Tractable Probleme.

▶ **Definition**
Ein parametrisiertes Problem (L, κ) ist in der Klasse FPT, wenn eine Turingmachine M, eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und einen Konstante c existieren, so dass

- $L(M) = L$ und
- bei Eingabe x hält die Maschine nach $f(\kappa(x))|x|^c$ Schritten an.

Beispiel
p-VERTEX-COVER $\in$ FPT: Für Fellows Algorithmus ist f die Funktion $f(k) = 2^k$.

Beachte: Das Polynom $|x|^c$ darf *nicht* vom Parameter abhängen. Die Funktion f darf hingegen beliebig schnell wachsen, so darf sogar *nichtrekursiv* sein. In der Regel ist sie aber einfach- oder doppeltexponentiell.

9.3.2 Fixed-Parameter-Intractability

9-20

Welche Probleme sind Fixed-Parameter-Tractable?

✎ **Zur Übung**
Zeigen Sie, dass p-COLORABLE $\in$ FPT genau dann, wenn P = NP.

- Manche Probleme, wie p-COLORABLE, können nicht in FPT liegen, außer P = NP.
- Manche Probleme, wie p-VERTEX-COVER liegen in FPT.
- Für manche Probleme, wie p-CLIQUE, glaubt man nicht, dass sie in FPT liegen, auch wenn dies nicht P = NP zu implizieren scheint.
  Um dies im Detail zu untersuchen, kann man *Reduktionen zwischen parametrisierten Problemen* definieren und *weitere Klassen*.

Zusammenfassung dieses Kapitels

9-21

▶ **Parametrisiertes Problem**
Ein *parametrisiertes Problem* ist ein Paar (L, κ), wobei

- $L \subseteq \Sigma^*$ eine Sprache ist und
- $\kappa\colon \Sigma^* \to \mathbb{N}$ ein *Parameter*.

▶ **Die Klasse FPT**
Ein parametrisiertes Problem ist in FPT, wenn es gelöst werden kann in Zeit

$$f(\kappa(x))|x|^c.$$

Hierbei kann f eine beliebig schnell wachsende Funktion sein.

▶ **Vertex-Cover ist schnell lösbar**
Das NP-vollständige Problem VERTEX-COVER kann effizient für Graphen mit *Millionen an Knoten* gelöst werden, solange $k \leq 85$.

Zum Weiterlesen

[1] J. Chen, I. Kanj and G. Xia, Improved Parameterized Upper Bounds for Vertex Cover. In *Proceedings of Mathematical Foundations of Computer Science 2006,* 238–249, 2006.

10-1

Kapitel 10
Kernelisierung
Des Pudels Kern

10-2

Lernziele dieses Kapitels

1. Die Idee der Kernelisierung verstehen
2. Beispiele von Kernelisierungen kennen
3. Kernelisierungen entwerfen können

Inhalte dieses Kapitels

Worum es heute geht

Kernelisierung ähnelt ziemlich dem Lösen eines Sudokus. Bei diesem Zahlenpuzzle wird man als erstes versuchen, an den »offensichtlichen« Stellen passende Zahlen einzufüllen. Wenn dies erfolgreich war, so gibt es meist wieder »offensichtliche« Stellen, wo man Zahlen eintragen kann; und so weiter. Mit etwas Glück löst sich das gesamte Probleme durch eine solche Kette von »offensichtlichen Vereinfachungen« in Luft auf.

Wie Sie vielleicht schon aus eigener Erfahrung leidvoll feststellen mussten, hat man nicht immer Glück im Leben. Es gibt durchaus Situationen beim Sudoku, bei denen man nur weiterkommt, wenn man zwei unterschiedliche Möglichkeiten »ausprobiert«. Danach ist meist wieder alles »offensichtlich«, bis doch nach einiger Zeit wieder zwei Möglichkeiten ausprobiert werden müssen. Situationen, die so vertrackt sind, dass man etwas »ausprobieren« muss, nennt man *Problem-Kernel*, da sie den »harten Kern« des Problems darstellen.

Für viele Probleme können wir »offensichtliche Vereinfachungen« erstaunliche lange durchführen, bevor wir auf einen Kernel stoßen. Ein wichtiges Beispiel ist (mal wieder) das Knotenüberdeckungsproblem: Nehmen wir an, wir suchen für einen Graphen eine Knotenüberdeckung mit 10 Knoten. Dann gibt es einen einfachen Algorithmus, der *jeden* Graphen einer *beliebigen* Größe auf einen Graphen mit maximal 100 Knoten reduziert, der einen Knotenüberdeckung mit 10 Knoten hat genau dann, wenn der »große« Graph eine hatte. Wie wir in diesem Kapitel sehen werden, gilt allgemeiner, dass wir »kernelisieren« können *genau* dann, wenn das Problem fixed-parameter-tractable ist, also in FPT liegt.

Zur Erinnerung: Unser Beispielproblem

10-4

Das Vertex-Cover-Problem

Für einen Graph G und eine Zahl k entscheide: Gibt es k Knoten, so dass jede Kante einen dieser Knoten als Endpunkt hat?

10-5

Wiederholung: Das Problem der kombinatorischen Explosion

Der *Suchaum* für das Vertex-Cover-Problem *»explodiert«:* Er hat Größe n^k.

Diese »kombinatorische Explosion« ist bei NP-vollständigen Problemen *nicht zu vermeiden.*

Es ist aber nicht klar, *was hier explodieren muss*.

10-6

Wiederholung: Fixed-Parameter-Algorithmen

Die Idee von Downey und Fellows

- Beschränke die kombinatorische Explosion auf einen *Parameter,* der in der Praxis *klein* bleibt.
- Für eine Eingabe x ist das Ziel eine Laufzeit von

$$\underbrace{f(\kappa(x))}_{\text{»kleine« Explosion}} \cdot \underbrace{|x|^{O(1)}}_{\text{polynomiell}}.$$

10.1 Der Kern des Problems

10.1.1 Die Idee: Erstmal vereinfachen

10-7

Die Idee der Vorverarbeitung

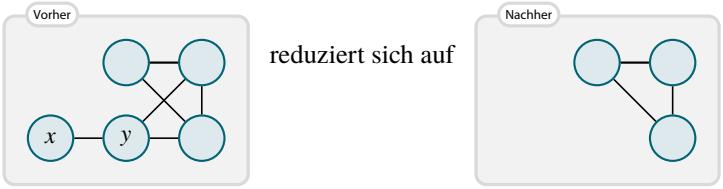
Bevor wir den Suchraum durchsuchen, ersetzen wir die Eingabe durch eine *vereinfachte* Eingabe – idealerweise eine mit kleinerem Parameter.
Diese Idee funktioniert erstaunlich oft. Die Regeln, nach denen wir Eingaben vereinfachen (»reduzieren«), nennt man *Reduktionsregeln.*

10-8

Beispiel einer Reduktionsregel.

Beispiel: Reduktionsregel für p-VERTEX-COVER
Sei (G,k) eine Eingabe für p-VERTEX-COVER und $x \in V$ ein Knoten vom Grad 1. Dann reduziert sich (G,k) auf $(G',k-1)$, wobei $G' = G - \{x,y\}$ und y der Nachbar von x ist.

Diese Regel ist »korrekt«, da jede Knotenüberdeckung von G entweder x oder y oder beide enthalten muss und es »nie falsch ist«, einfach nur y zu nehmen.



10-9

✎ Zur Übung

1. Geben Sie eine Reduktionsregel für p-VERTEX-COVER für den Fall an, dass G zwei benachbarte Knoten vom Grad 2 enthält.
2. Geben Sie ein Reduktionsregel für p-CLIQUE an.

10.1.2 Definition: Kernel

Reduktionen etwas formaler 10-10

Der Reduktionsprozess

Bei Eingabe einer Instanz für ein Problem wenden wir so lange wie irgend möglich Reduktionsregeln an. Wenn dies für eine Eingabe nicht mehr möglich ist, so nennen wir das Ergebnis eine *nichtreduzierbare Instanz*.

▶ Definition: Reduktionsregel

Sei (L, κ) ein parametrisiertes Problem. Eine *Reduktionsregel* ist eine in polynomieller Zeit berechenbare Funktion $f \in \mathrm{FP}$, die

1. als Eingaben Instanzen $x \in \Sigma^*$ bekommt und
2. entweder ausgibt »keine Reduktion möglich« oder
3. $x \in L \iff f(x) \in L$ und $\kappa(f(x)) \leq \kappa(x)$ und $|f(x)| < |x|$.

Kernel sind kleine nichtreduzierbare Instanzen. 10-11

▶ Definition: Kernelisierung

Ein parametrisiertes Problem *lässt sich kernelisieren*, wenn es eine Menge von Reduktionsregeln gibt, so dass für alle Eingaben x die wiederholte Anwendung der Regeln immer einer nichtreduzierbare Instanz liefert, *deren Größe nur von* $\kappa(x)$ *abhängt*. Die nichtreduzierbare Instanz heißt *Kernel*.

Man beachte: Welche Reduktionsregel wir anwenden (müssen), ändert sich ständig. Jedoch müssen wir »immer« einen Kernel erreichen, egal in welcher Reihenfolge wir Regeln anwenden.

Beispiel

Wir werden gleich sehen, dass das Knotenüberdeckungsproblem einen *quadratischen Kernel* hat, also einen Kernel der Größe $O(k^2)$. Für $k = 10$ bedeutet dies folgendes: Es gibt Reduktionsregeln, so dass wir für jeden Graphen G mit einer Milliarde Knoten sehr schnell einen Graphen G' mit maximal 100 Knoten und eine Zahl $k' \leq 10$ berechnen können mit folgender Eigenschaft: Der reduzierte Graph G' mit 100 Knoten hat genau dann eine Knotenüberdeckung der Größe k', wenn der Graph G mit einer Milliarde Knoten eine der Größe k hatte.

Kernelisierbar = Fixed-Parameter-Tractable 10-12

▶ Satz

Ein Problem ist genau dann kernelisierbar, wenn es einen Fixed-Parameter-Algorithmus hat.

Beweis. Siehe Übungen 10.1 und 10.2. □

10.2 Kernel

10.2.1 . . . für Vertex-Cover

Reduktionsregeln für das Knotenüberdeckungsproblem. 10-13

Reduktionsregeln für das Knotenüberdeckungsproblem

Bei Eingabe (G,k) tue:

1. Wenn G einen isolierten Knoten hat, entferne ihn.
2. Wenn G einen Knoten v vom Grad mindestens $k+1$ hat, gib $(G-\{v\},k-1)$ aus.
3. Wenn die obigen Regeln nicht anwendbar sind, aber G mehr als $k(k+1)$ Knoten hat, gib $(\bullet\!\!-\!\!\bullet, 0)$ aus (also »nein«).

▶ Lemma
Die Reduktionsregeln sind korrekt.

Beweis.
1. Regel 1 ist trivial.
2. Für Regel 2 beachte, dass ein Knoten vom Grad $k+1$ *immer* in einer Knotenüberdeckung der Größe k enthalten sein muss.
3. Für Regel 3 betrachte zum Zwecke des Widerspruchs einen Graphen mit mehr als $k(k+1)$ Knoten, der eine Knotenüberdeckung C der Größe k besitzt. Da Regel 2 nicht anwendbar ist, kann jeder Knoten in C nur k Kanten abdecken. Es gibt im Graphen folglich einen Knoten v, der mit keinem Knoten in C verbunden ist. Da Regel 1 nicht anwendbar ist, hängt an v eine nicht überdeckte Kante. □

10-14 Kernelisierung des Knotenüberdeckungsproblems.

▶ Lemma
Die Reduktionsregeln liefert einen Kernel der Größe höchstens $k(k+1)$.

Beweis. Trivial. □

▶ Folgerung
p-VERTEX-COVER kann in Zeit $O(|G|+2^k k^2)$ gelöst werden.

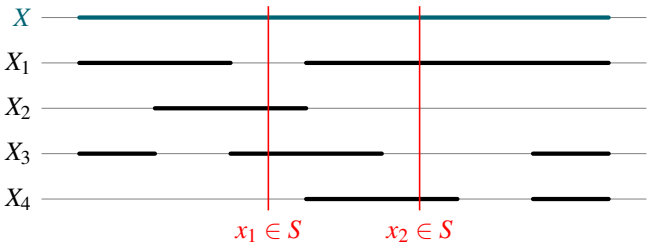
▶ Folgerung
p-VERTEX-COVER kann in Zeit $O(|G|+1.49535^k k^2)$ gelöst werden.

Beweis. In beiden Fällen gilt: Erst kernelisieren, dann die Algorithmen aus dem vorigen Kapitel anwenden. □

10.2.2 . . . für Unique-Hitting-Set

10-15 Das Unique-Hitting-Set-Problem

▶ Problem: p-UNIQUE-HITTING-SET
Eingaben Eine Menge X und k Teilmengen $X_1, \ldots, X_k$ von X.
Parameter k.
Frage Gibt es eine Teilmenge $S \subseteq X$, so dass für alle i gilt $|S \cap X_i| = 1$?



10-16 Kernelisierung des Unique-Hitting-Set-Problems.

Reduktionsregel
Bei Eingabe $(X, X_1, \ldots, X_k)$ tue:
- Falls es zwei unterschiedliche $x, y \in X$ gibt mit $\{i \mid x \in X_i\} = \{i \mid y \in X_i\}$, so gib $(X - \{y\}, X_1 - \{y\}, \ldots, X_k - \{y\})$ aus.
- Sonst gib »keine Reduktion möglich« aus.

▶ Lemma
Die Reduktion ist korrekt und liefert einen Kernel der Größe 2^k.

Beweis. Wir werden nie sowohl x als auch y in unser Hitting-Set wählen – und wenn wir eines wählen, könnten wir genausogut das andere wählen. Folglich schadet es nicht, y zu entfernen. Da es nur 2^k Möglichkeiten für die Menge $\{i \mid x \in X_i\}$ gibt, wird die Reduktion nicht stoppen, wenn es noch mehr als 2^k Elemente in X gibt. □

10.2.3 . . . für Bushy-Spanning-Tree

Das Gerüstproblem mit einer beschränkten Anzahl Blätter. 10-17

▶ Problem: p-BUSHY-SPANNING-TREE

Instanzen Ungerichtete Graphen G und eine Zahl k.

Parameter k.

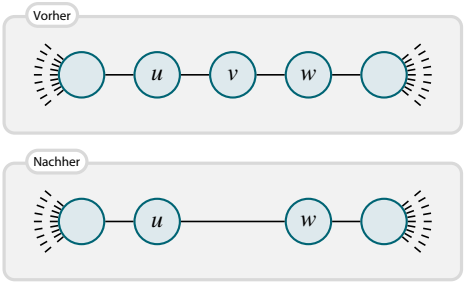
Frage Hat G ein Gerüst (»spanning tree«) *mit mindestens k Blättern*?

Beispiele

- Ein Pfad hat ein Gerüst mit 2 Blättern, aber nicht mit 3.
- Ebenso für Zyklen.
- Eine k-Clique hat ein Gerüst mit $k-1$ Blättern.
- Ebenso ein Stern mit $k-1$ Blättern.

Die Reduktionsregeln 10-18

1. *Zusammenhangsregel:* Wenn der Graph unzusammenhängend ist, gib $(\bullet\!\!-\!\!\bullet, 3)$ aus (also »nein«).
2. *Hoher-Grad-Regel:* Gibt es sonst einen Knoten vom Grad mindestens k, gib $(\bullet\!\!-\!\!\bullet, 2)$ aus (also »ja«).
3. *Unnützer-Knoten-Regel:* Gibt es sonst einen Knoten v vom Grad 2, dessen Nachbarn u und w auch Grad 2 haben, entferne v und füge eine Kanten zwischen u und w hinzu.



4. *Die Großer-Graph-Regel:* Lässt sich keine der ersten drei Regeln anwenden und hat G mehr als $N = 4(k+2)(k+1)$ Knoten, gib $(\bullet\!\!-\!\!\bullet, 2)$ aus (also »ja«).

Die Korrektheit der ersten drei Regeln sollte klar sein. Falls auf die vierte Regel ist (was wir gleich zeigen), so gilt:

▶ Satz
p-BUSHY-SPANNING-TREE *hat quadratische Kernel.*

▶ Folgerung
p-BUSHY-SPANNING-TREE *kann in Zeit $O(|G| + k^{O(k)})$ gelöst werden.*

Beweis. Bei einer Eingabe (G,k) kernelisiere. Löse dann den Kernel durch »rohe Gewalt«. Dies dauert nur *exponentiell lange in der Größe des Kernels.* □

Korrektheit der Großer-Graph-Regel. 10-19

▶ Satz
Die Großer-Graph-Regel ist korrekt.

Intuition

Nehmen wir an der Graph sei sehr groß. Er kann keine langen Pfade haben (wegen der Unnützer-Knoten-Regel). Dann muss es »viele Verzweigungen« geben und somit ein Gerüst mit vielen Blättern.

10-20

Beweisplan

Was wir zeigen müssen

- Wir haben eine Graphen G mit mehr als $N = 4(k+1)(k+2)$ gegeben.
- Weiter ist der Graph zusammenhängend, hat keine Knoten mit hohem Grad und keine unnützen Knoten.
- Wir sollen zeigen, dass er ein Gerüst T mit mindestens $b \geq k$ Blättern hat.

Hauptwiderspruchsannahme

Das Gerüst T mit der maximalen Anzahl Blätter hat nur $b < k$ Blätter.

10-21

Einige Beobachtungen zur Topologie von T.

▶ **Lemma:** Das Puschel-Lemma
Ein Baum mit m Knoten vom Grad mindestens 3 hat mindestens $m+2$ Blätter.

Beweis. Per Induktion. □

Betrachten wir nun die Grade der Knoten von T: Der Baum T hat N Knoten, aber nur $b < k$ Blätter. Nach dem Puschel-Lemma kann T nur $b-2$ Knoten vom Grad 3 oder mehr haben. Es müssen also fast alle (genau $N-b+2-b$) Knoten von T Grad 2 haben. Wir halten fest: Es gibt viel ($\approx 4k^2$) Knoten im Baum T, die Grad 2 haben.

10-22

Kanten im Baum versus Kanten im Graph.

▶ **Lemma:** Das Blaue-Knoten-Lemma
Es gibt mindestens $3(k+1)(k+3)+2$ Knoten in G mit:

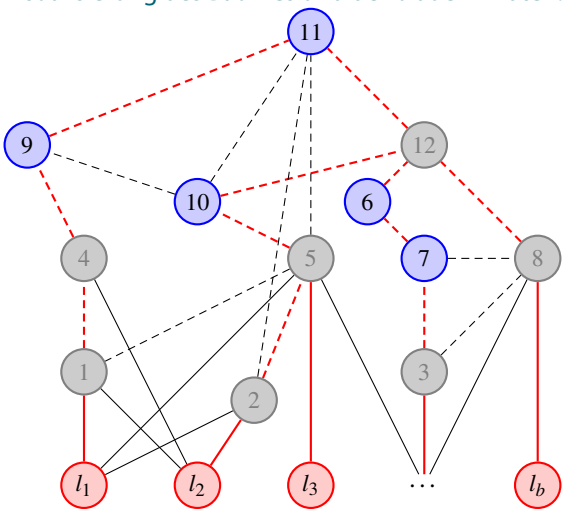
1. *In T haben sie Grad 2.*
2. *In G haben sie keine Kante zu einem Blatt von T.*

Solche Knoten bezeichnen wir als *blaue Knoten*.

Beweis. Jeder Knoten in G hat ja Grad höchstens $k-1$ (wegen der Hoher-Grad-Regel). Folglich kann jedes der $b \leq k-1$ Blätter von T mit höchstens $k-1$ Knoten in G verbunden sein. Es können also höchstens $(k-1)^2 \approx k^2$ Knoten von G mit Blättern von T verbunden sein; aber es gab ja $4k^2$ Knoten, die in T Grad 2 haben. Damit bleiben etwa $4k^2 - k^2 = 3k^2$ Knoten übrig, die blau sein müssen. (Ihre genaue Anzahl ist $3(k+1)(k+3)+2$.) □

10-23

Visualisierung des Baumes und der blauen Knoten.



- Der Baum ist rot gezeigt, ebenso sein Blätter.
- Durchgezogene Kanten enden an einem Blatt, gestrichelte Kanten nicht.

Welchen Grad können blaue Knoten haben?

10-24

Ein blauer Knoten hat (per Definition) Grad 2 in T, kann aber höheren Grad in G haben.

▶ Definition

- Wir nennen einen blauen Knoten *dunkelblau*, wenn er Grad 3 oder mehr in G hat.
- Sonst nennen wir ihn *hellblau* (er hat dann Grad 2 auch in G.)

▶ Lemma

Es gibt mindestens $k-2$ dunkelblaue Knoten.

Beweis. Wir machen eine *Nebenwiderspruchsannahme*: Nehmen wir an, es gäbe nur $k-3$ dunkelblaue Knoten. Dann muss es $3(k+1)(k+3)+2-(k-3)$ hellbaue Knoten geben. Aufgrund der Unnützer-Knoten-Regel müssen alle hellblauen Knoten mit einem Knoten vom Grad mindestens 3 in G verbunden sein. Nennen wir diesen Knoten den *Partner* des hellblauen Knoten. Beispielsweise ist 7 ein Partner des hellblauen Knoten 6 im Beispiel vorher. Partner können selber nicht hellblau sein (warum?). Aufgrund der Hoher-Grad-Regel kann jeder Partner mit maximal $k-1$ hellblauen Knoten in G verbunden sein. Es muss also mindestens $\big(3(k+1)(k+3)+2-(k-3)\big)/(k-1) \geq 3k+1$ Partner geben.
Was wir über Partner wissen:

- Keiner von ihnen ist ein Blatt von T (da er mit einem blauen Knoten verbunden ist).
- Jeder hat Grad mindestens 3 in G.
- Es gibt mindestens $3k+1$ von ihnen.
- Höchstens $k-3$ von ihnen sind dunkelblau (die Nebenwiderspruchsannahme).
- Da sie auch nicht hellblau sein, können höchstens $k-3$ von ihnen blau sein.
- Also können höchstens $k-3$ von ihnen Grad 2 in T haben.

Insgesamt erhalten wir, dass mindestens $3k+1-(k-3) = 2k+4$ Knoten von T Grad 3 oder mehr haben. Nach dem Puschel-Lemma müsse T also mehr als k Blätter haben – ein Widerspruch zur Hauptwiderspruchsannahme. □

Wo wir stehen.

10-25

- Es gibt mindestens $k-2$ dunkelblaue Knoten v in G.
- Seien u und w die Nachbarn von v in T. Sei weiter x ein weiterer Nachbar von v in G (in T hat v nur zwei Nachbarn, in G aber mindestens 3).
- Betrachte einen Pfad von v nach x in T.
- Dann muss für ein geeignetes v auf diesem Pfad ein Knoten y mit Grad 2 in T liegen. (Sonst gäbe es $k-2$ innere Knoten in T vom Grad 3, was dem Puschel-Lemma widersprechen würde.)

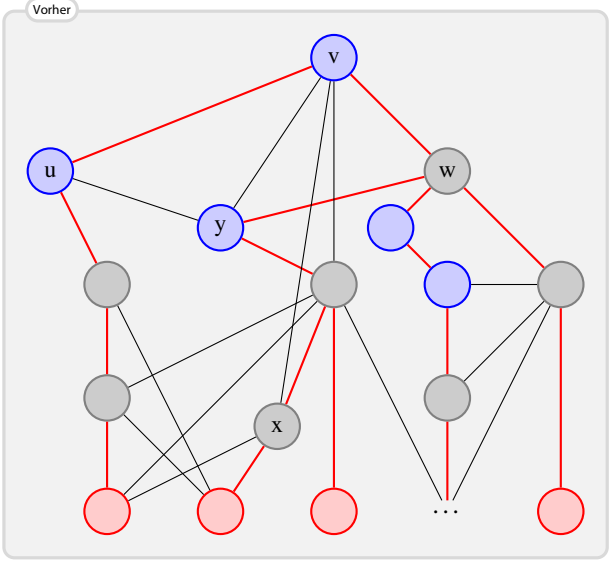
Wir können nun T *modifizieren,* um die Anzahl der Blätter zu erhöhen:

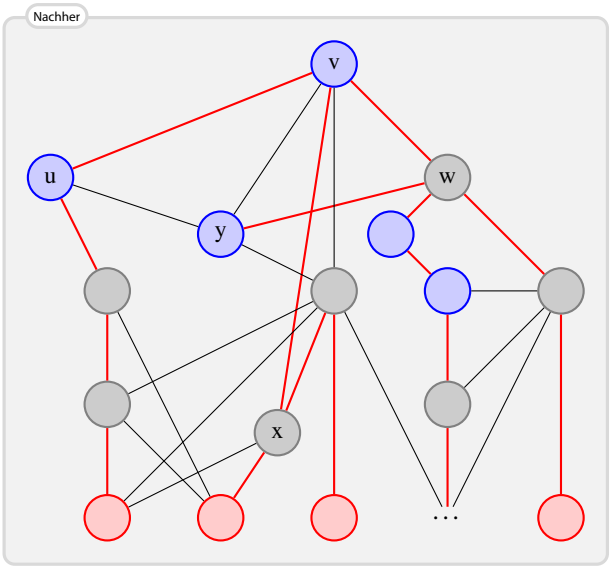
- Füge eine Kante zwischen v und x hinzu.
- Entferne die Kante zwischen y und seinem Kind.

Der modifizierte Baum hat dann *ein Blatt mehr (nämlich y)*, was der Hauptwiderspruchsannahme widerspricht.

Wie man die Anzahl der Blätter erhöht

10-26





Zusammenfassung dieses Kapitels

10-27

▶ **Reduktionsregel**
Eine *Reduktionsregel* für ein parametrisiertes Problem (L, κ) ist eine Abbildung $f \in \mathrm{FP}$, so dass für alle Worte $w \in \Sigma^*$ gilt:

1. $f(w)$ ist »keine Reduktion möglich« oder
2. $w \in L \iff f(w) \in L$ und $|f(w)| < |w|$ und $\kappa(f(w)) \leq \kappa(w)$.

▶ **Kernelisierbar, Kernel**
Ein Problem (L, κ) heißt *kernelisierbar*, wenn es eine Menge von Reduktionsregeln gibt, die jede Eingabe w immer reduzieren auf ein Wort w', dessen Länge nur vom Parameter der Eingabe abhängt. Wir nennen w' einen *Kernel* von w.

▶ **Satz**
$(L, \kappa) \in \mathrm{FPT}$ genau dann, wenn (L, κ) kernelisierbar ist.

Zum Weiterlesen

[1] R. G. Downey und M. R. Fellows, Parametrized Computational Feasibility, In *Proceedings of Feasible Mathematics II,* 219–244, Birkhäuser, 1995.

Dieser Artikel stellt die Kernelisierung für das Bushy-Spanning-Tree-Problem vor. Man beachte aber, dass sich dort ein kombinatorischer Fehler eingeschlichen hat.

Übungen zu diesem Kapitel

Übung 10.1 Kernelisierbare Problem sind in FPT, mittel
Zeigen Sie: Ist L rekursiv und kann (L, κ) kernelisiert werden, so gilt $(L, \kappa) \in \mathrm{FPT}$.
Tipp: Zeigen Sie, dass L entschieden werden kann in Zeit

$$O\Big(\underbrace{|x|^{O(1)}}_{\text{Kernelisierung}} + \overbrace{g\big(\underbrace{h(\kappa(x))}_{\text{Kernel-Größe}} \big)}^{\text{Rohe Gewalt}} \Big)$$

für geeignete Funktionen g und h.

Übung 10.2 Alle Probleme in FPT haben Kernel, schwer
Zeigen Sie: Wenn $(L, \kappa) \in \mathrm{FPT}$, so kann (L, κ) kernelisiert werden.
Tipp: Benutzen Sie folgende eigentlich reichlich nutzlose Reduktionsregel: Für Eingaben x mit $|x|^c < f(\kappa(x))$ gib »keine Reduktion möglich« aus und sonst löse x direkt und gib ein festes Wort $x_1 \in L$ aus, wenn $x \in L$ gilt, und sonst ein festes Wort $x_0 \notin L$.

11-1

Kapitel 11
Approximation

Lieber eine gute Lösung als gar keine Lösung

Lernziele dieses Kapitels

1. Grundbegriffe der Approximationstheorie kennen
2. Beispiele von Approximationsalgorithmen und -klassen kennen
3. Entscheiden können, ob ein Problem approximierbar ist

11-2

Inhalte dieses Kapitels

Worum es heute geht

Eine Menge Zeit und Geld in das Finden optimaler Lösungen zu investieren, ist für viele Probleme, mit Verlaub gesagt, bescheuert. Nehmen wir an, Sie sind die Logistikleiterin einer Speditionsfirma (als Sie noch im Rahmen Ihres Informatik-Studiums in Vorlesungen wie »Komplexitätstheorie« saßen, hätten Sie sich nicht träumen lassen, eines Tages diesen Job zu machen, aber was tut man nicht alles aus Liebe). Ihre Aufgabe ist es, täglich möglichst gute Touren für die Lieferwagen der Firma zu erstellen. Natürlich war Ihnen schon am ersten Tag klar, dass dies ein NP-vollständiges Problem ist; die Reduktion von EUCLIDEAN-TSP hat Sie förmlich angesprungen. Was also tun? Sich mit Fixed-Parameter-Algorithmen an die Problematik machen? (Aber, was ist der Parameter?) Mehr Hardware kaufen?

Wenn man etwas darüber nachdenkt, so stellt man schnell fest, dass es eine völlig falsche Zielsetzung ist, das Speditionsproblem *exakt* lösen zu wollen: Die *Eingaben* sind ja schon mit reichlich Unsicherheit versehen (Entfernungen zwischen Orten sind sicherlich bestenfalls nur auf den Kilometer genau). Wer dann viel algorithmische Energie aufwendet, gefundene Touren noch um ein paar Millimeter zu verkürzen, hat das Problem nicht verstanden. Es reicht völlig aus, eine Tour zu finden, die »fast« die kürzeste ist; in der Realität ist diese vielleicht sogar kürzer als eine vermeintlich »optimale«.

Leider könnte es ja schon ein schwieriges Problem sein, eine »fast« optimale Tour zu finden. Um hierüber eine sinnvolle Aussage treffen zu können, brauchen wir einen formalen Begriff der »Approximierbarkeit« von Problemen. Und wo wir schon dabei sind, können wir gleich mal ein paar Klassen definieren. Und Reduktionen. Oder vielleicht lieber doch nicht – Reduktionen zwischen Optimierungsproblemen sind definitorische Monster, weshalb wir uns in diesem Kapitel lieber ein paar wichtige Approximationsalgorithmen genauer anschauen.

11.1 Die Idee

11-4

Die 80/20-Regel (Pareto-Regel).
Die 80/20-Regel ist eine *angebliche empirische Beobachtung* betreffend reale Projekte:

- In vielen Projekten werden 80% der Arbeit in 20% der Zeit erledigt.
- Dementsprechend verschlingen 20% der Arbeit 80% der Zeit.

Beispiel
Wenn Sie eine Abschlussarbeit schreiben, werden 80% des Textes recht schnell geschrieben sein. Aber 80% der Zeit geht für die »Perfektionierung« drauf.

Beispiel
Wenn Sie ein Programm schreiben, sind 80% des Codes schnell geschrieben. Aber 80% der Zeit geht dafür drauf, dass der Code funktioniert.

11-5

Was uns die 80/20-Regel lehrt.
Auch wenn man an die 80/20-Regel nicht »glaubt« (ich tue es nicht), so lehrt sie uns doch etwas Wichtiges:

- Oft ist es leicht, eine »80%-perfekte« Lösung zu finden.
- Hieraus eine »100%-perfekte« Lösung zu machen, ist schwierig.

Dies lässt sich auf Algorithmen übertragen:

- Es kann leicht sein, eine »80%-perfekte« Lösung zu finden.
- Es kann hingegen unglaublich schwierig sein, ein »100%-perfekte« Lösung zu finden. (Na und?)

11.2 Der formale Rahmen

11-6

Der formale Rahmen der (algorithmischen) Apprxomiationstheorie.
Wir müssen zunächst die folgenden Fragen beantworten:

- Was, genau, ist eine »80%-perfekte Lösung«?
- Ist eine »70%-perfekte Lösung« auch gut genug?
- Lassen sich Problem danach klassifizieren, »wie perfekte« Lösungen wir schnell berechnen können?

11.2.1 Optimierungsprobleme

11-7

Wiederholung von Optimierungsproblemen.

▶ **Definition:** Optimierungsproblem
Sei Σ ein Alphabet. Ein *Optimierungsproblem auf* Σ^* ist ein Tupel bestehend aus

1. einer *Lösungsrelation* $S \subseteq \Sigma^* \times \Sigma^*$,
2. einer *Maßfunktion* $m\colon S \to \mathbb{N}$ und
3. einem *Typ* $t \in \{\min, \max\}$.

Für eine Instanz $x \in \Sigma^*$ und ein Paar $(x,s) \in S$ sagen wir, *s sei eine Lösung für x.*

Das Maß misst, wie »gut« eine Lösung ist. Der Typ beschreibt, ob große Maßzahlen oder kleine Maßzahln »gut« sind.

Beispiel: MIN-VERTEX-COVER

Instanzen	Ungerichtete Graphen G.
Lösungen	S ist die Menge aller (kodierten) Paare (G,C), wobei C eine Knotenüberdeckung von G ist
Maß	$m(G,C) = \lvert C\rvert$.
Typ	Minimierung.

11.2.2 Approximationsrate

Lösungen, optimale Lösungen und approximative Lösungen. 11-8

▶ **Definition**

Für ein gegebenes Optimierungsproblem (S,m,t) und eine Instanz $x \in \Sigma^*$ sagen wir:

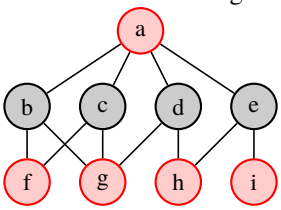
- $s \in \Sigma^*$ ist *eine Lösung für x,* wenn $(x,s) \in S$.
- Der Wert $m(x,s)$ ist das *Maß der Lösung s für x*.
- Eine Lösung s für x ist *optimal,* wenn ihr Maß minimal (für $t = \max$ entsprechend maximal) ist unter allen Lösungen für x.
  Wir schreiben $\mathrm{opt}(x)$ für ihr Maß.
- Die *Güte* (*approximation ratio*) einer Lösung s ist der Quotient

$$\mathrm{ratio}(s,x) = \frac{\max\{m(x,s),\mathrm{opt}(x)\}}{\min\{m(x,s),\mathrm{opt}(x)\}}.$$

Man beachte, dass $\mathrm{ratio}(s,x) \geq 1$ immer gilt und $\mathrm{ratio}(s,x) = 1$ genau dann, wenn s eine optimale Lösung ist.
Manchmal wird die Güte auch gerade als Kehrwehrt des obigen Wertes definiert.

Beispiel einer Instanz, einer Lösung und ihrer Güte. 11-9

Man betrachte die folgende Lösung für das Knotenüberdeckungsproblem (rote Knoten):



- Die *Instanz* ist der Graph.
- Die *Lösung* ist die Knotenüberdeckung (die Menge der roten Knoten).
- Das *Maß* dieser Lösung ist 5 (Anzahl der Knoten).
- Das *optimale Maß* ist 4, da es eine Knotenüberdeckung der Größe 4 gibt.
- Die *Güte* der Lösung ist deshalb $5/4 = 1{,}25$.

11.2.3 Klassen von Optimierungsproblemen

Die Klasse der interessanten Optimierungsprobleme. 11-10

Bis jetzt haben wir keine besonderen Bedingungen an Optimierungsprobleme gestellt. Insbesondere könnten Lösungen *schrecklich groß* und / oder *schrecklich schwierig zu überprüfen* sein.
Wir wollen uns deshalb nur auf »sinnvolle« Probleme konzentrieren:

▶ **Definition: Die Klasse NPO**

Ein Optimierungsproblem (S,m,t) ist in der Klasse NPO, wenn

- für alle $(x,s) \in S$ gilt $|s| \leq |x|^{O(1)}$ und
- $S \in \mathrm{P}$, das heißt, S ist in polynomieller Zeit entscheidbar.

Der Name »NPO« wird gleich noch klarer werden. Alle von uns betrachteten Approximationsklassen werden Teilmengen von NPO sein.

Die Klasse der schnell lösbaren Optimierungsprobleme. 11-11

Bevor wir die *schwierigen* Probleme betrachten, definieren wir zunähst die Klasse der *schnell optimal lösbaren* Optimierungsprobleme:

▶ **Definition**

Ein Problem $(S,m,t) \in \mathrm{NPO}$ ist in PO, falls

- eine Funktion $f \in \mathrm{FP}$ existiert,
- so dass für alle $x \in \Sigma^*$, für die eine Lösung s existiert,
- gilt $(x, f(x)) \in S$ und $m(x, f(x)) = \mathrm{opt}(x)$.

Oder kürzer: Für Probleme in PO kann man optimale Lösungen in polynomieller Zeit ausrechnen.

11-12

Warum die Klassen PO und NPO so heißen.

▶ Definition
Für ein Maximierungsproblem $Q = (S, m, \max)$ definieren wir die folgenden Entscheidungsprobleme:

$$Q_{\exists\,\mathrm{sol}} = \{x \mid \text{es gibt eine Lösung } s \text{ für } x\},$$
$$Q_{\mathrm{sol}>} = \{(x,k) \mid \text{es gibt eine Lösung } s \text{ für } x \text{ mit } m(x,s) \geq k\}.$$

▶ Satz

1. *Für jedes* $Q \in \mathrm{NPO}$ *gilt* $Q_{\mathrm{sol}>} \in \mathrm{NP}$.
2. *Für jedes* $Q \in \mathrm{PO}$ *gilt* $Q_{\mathrm{sol}>} \in \mathrm{P}$.
3. *Für jedes* $L \in \mathrm{NP}$ *gibt es ein* $Q \in \mathrm{NPO}$ *mit* $L = Q_{\exists\,\mathrm{sol}}$.
4. *Für jedes* $L \in \mathrm{P}$ *gibt es ein* $Q \in \mathrm{PO}$ *mit* $L = Q_{\exists\,\mathrm{sol}}$.

Beweis.

1. Sei $Q \in \mathrm{NPO}$. Um $Q_{\mathrm{sol}>} \in \mathrm{NP}$ zu zeigen, rate bei Eingabe (x,k) eine Lösung s und überprüfe, ob $m(x,s) \geq k$.
2. Sei $Q \in \mathrm{PO}$. Um $Q_{\mathrm{sol}>} \in \mathrm{P}$ zu zeigen, berechne bei Eingabe (x,k) eine optimale Lösung s und überprüfe, ob $m(x,s) \geq k$.
3. Sei $L \in \mathrm{NP}$. Definiere Q wie folgt:
   - Die Lösungen s für ein x sind gerade die Zertifikate für $x \in L$.
   - Das Maß ist irrelevant (immer 1).

   Dann ist $Q \in \mathrm{NPO}$ und $L = Q_{\exists\,\mathrm{sol}}$.
4. Sei $L \in \mathrm{P}$. Definiere Q wie folgt:
   - Falls $x \in L$, dann ist »1« die einzige Lösung für x; sonst gibt es keine Lösung für x.
   - Das Maß ist irrelevant (immer 1).

   Dann ist $Q \in \mathrm{PO}$ und $Q_{\exists\,\mathrm{sol}} = L$. □

11.3 Konstante Approximation

11.3.1 Die Klasse APX

11-13

Konstante Approximation.

▶ Definition
Sei $r \geq 1$ eine Zahl. Ein Problem $(S,m,t) \in \mathrm{NPO}$ ist in r-APX, falls

- eine Funktion $f \in \mathrm{FP}$ existiert, die *Approximationsfunktion* genannt,

so dass für alle $x \in \Sigma^*$, für die eine Lösung existiert, gilt,

- $(x, f(x)) \in S$ und
- $\mathrm{ratio}(x, f(x)) \leq r$.

Offenbar gilt $1\text{-APX} = \mathrm{PO}$.

▶ Definition
$\mathrm{APX} = \bigcup_{r>1} r\text{-APX}$.

11.3.2 Bin-Packing

Das Bin-Packing-Problem. 11-14

→

Das Bin-Packing-Problem lässt sich gut approximieren. 11-15

▶ Problem: MIN-BIN-PACKING

Instanzen Liste von Objektgrößen $(g_1, \ldots, g_n)$ und eine Eimergröße b (*bin size*).

Lösungen Zuordnung von Objekten zu Eimern, so dass alle Eimer höchstens mit Objekten einer Gesamtgröße von b gefüllt sind.

Maß Anzahl der Eimer.

Typ Minimierung.

▶ Satz

MIN-BIN-PACKING $\in$ 2-APX.

Ein Algorithmus für das Bin-Packing-Problem. 11-16

First-Fit-Algorithmus

```
1 for i = 1,2,3,...,n do
2   for j = 1,2,3,... do
3     if Objekt i passt noch in Eimer j then
4       platziere Objekt i in Eimer j
5       break inner for loop and continue with next object
```

Zentrale Beobachtung: *Für je zwei von First-Fit »angebrochenen« Eimer muss in einer optimalen Lösung mindestens ein Eimer genutzt werden.*
Hieraus folgt sofort, dass der Algorithmus eine 2-Approximation liefert.

11.3.3 Vertex-Cover

Der naheliegende Greedy-Algorithmus für das Knotenüberdeckungsproblem. 11-17

Greedy-Vertex-Cover

```
1 while es gibt eine Kante im Graphen do
2   bestimmte einen Knoten v mit maximalem Grad
3   füge v zur Knotenüberdeckung hinzu
4   lösche v aus dem Graphen
```

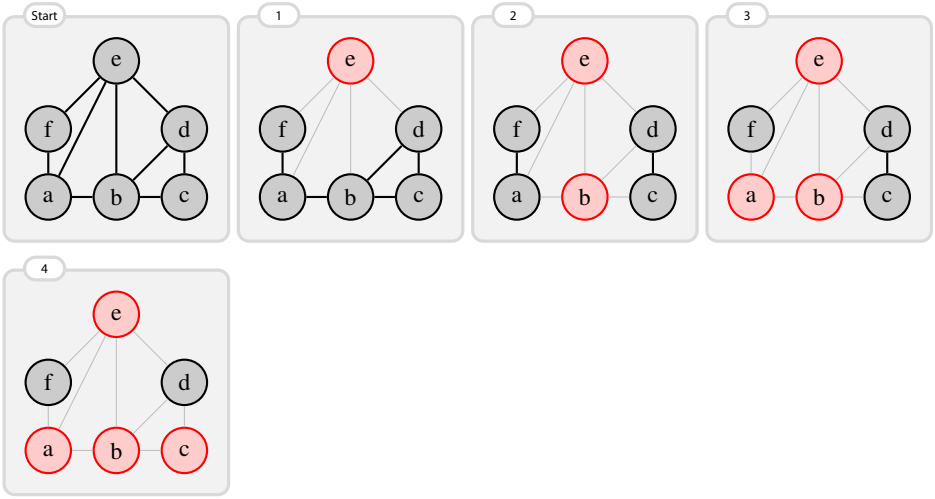
▶ Satz

Der Algorithmus liefert keine konstante Approximationsrate.

Beweis. In Übung 11.1 wird gezeigt, dass die Approximationsrate nur $\ln n$ ist (und folglich nicht konstant). □

11-18

Der Greedy-Algorithmus in Aktion.



11-19

Ein lächerlich einfacher Algorithmus.

Trivialer Algorithmus

```
1 while es gibt noch Kanten im Graph do
2   wähle eine beliebige Kante {u,v}
3   füge u und v zur Knotenüberdeckung hinzu
4   lösche beide aus dem Graphen
```

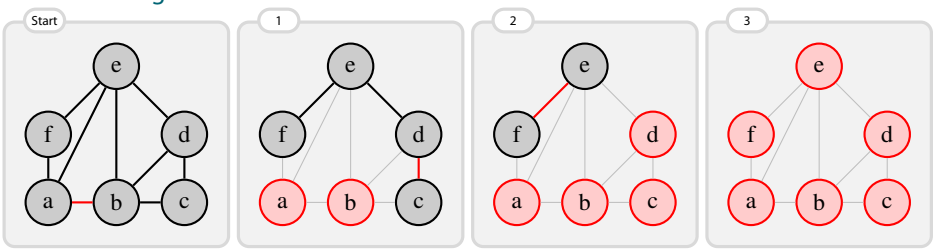
▶ **Satz**
MIN-VERTEX-COVER $\in$ 2-APX *via dem trivialen Algorithmus.*

✎ **Zur Übung**
Beweisen Sie den Satz.

Keine besseren Approximationsraten sind bekannt (und es gibt gute Gründe zu glauben, dass keine bessere möglich ist).

11-20

Der triviale Algorithmus in Aktion.



11.3.4 Travelling-Salesperson

11-21

Gute Nachrichten, schlechte Nachrichten.

▶ **Problem:** MIN-TSP

Instanzen	Ein vollständiger, gerichteter Graph und eine Gewichtsfunktion $w\colon E \to \mathbb{N} - \{0\}$
Lösungen	Eine Tour $(v_1, v_2, \ldots, v_n, v_1)$ der Knoten (eine Permutation ohne den letzten Knoten).
Maß	Gewicht $w(v_n, v_1) + \sum_{i=2}^{n} w(v_{i-1}, v_i)$ der Tour
Typ	Minimierung

▶ **Problem:** MIN-Δ-TSP
Genau wie MIN-TSP, aber die Gewichtsfunktion muss der Dreiecksungleichung $w(x,z) \leq w(x,y) + w(y,z)$ gehorchen.

▶ Satz: Schlechte Nachrichten
MIN-TSP $\in$ APX *genau dann, wenn* P $=$ NP.

▶ Satz: Gute Nachrichten
MIN-Δ-TSP $\in 3/2$-APX.

Beweis der schlechten Nachrichten.

11-22

Beweis der Nichtapproximierbarkeit. Eine Richtung ist einfach: Falls P $=$ NP, so gilt PO $=$ NPO und somit MIN-TSP $\in$ PO $\subseteq$ APX.
Für die andere Richtung sei MIN-TSP $\in$ APX. Sei A ein r-Approximationsalgorithmus für ein festes r. Wir zeigen, dass dann HAMILTON $\in$ P. Bei Eingabe $G = (V,E)$ betrachte folgende Instanz für MIN-TSP:

- Der Graph G' hat (wie G) die Knotenmenge V.
- Setze $w(\{u,v\}) = 1$, falls $\{u,v\} \in E$, und $w(\{u,v\}) = rn+1$ sonst.

Wir lassen nun A auf diesem Graphen laufen und akzeptieren das ursprüngliche G genau dann, wenn die ausgegebene Tour eine hamiltonsche Tour in G ist. Dies ist korrekt, denn: Gibt es eine hamiltonsche Tour in G, so hat die optimale Tour in G' Gewicht n. Also muss A eine Tour mit Gewicht maximal rn ausgeben. Diese Tour kann also nur aus Kanten von G bestehen. □

Der Algorithmus hinter der guten Nachricht: Christofides' Algorithmus.

11-23

Christofides' Algorithmus für MIN-Δ-TSP

```
1 input (G,w)
2 berechne ein minimales Gerüst T von G
3 betrachte den Teilgraph H von G, der von den Knoten mit ungeradem Grad in T induziert
      wird
4 berechne ein perfektes Matching M von H mit minimalem Gewicht
5 sei K die Vereinigung von T und M (als Graph mit Mehrfachkanten zwischen Knoten)
6 berechne eine Euler-Tour S von K
7 while S besucht einen Knoten v mehrfach do
8   sei (u,v,w) ein Tourstück in S, wo v besucht wrid
9   ersetze es durch (u,w)
```

▶ Lemma
Christofides' Algorithmus liefert eine $3/2$*-Approximation.*

Beweis. Betrachte eine optimale Tour mit Gewicht $x := \mathrm{opt}(G,w)$. Wir analysieren zunächst das *Gewicht des Gerüstes* T:

1. Entfernt man aus einer Tour eine Kante, so erhält man ein Gerüst (ein etwas merkwürdiges Gerüst, aber ein Gerüst).
2. Folglich ist das Gewicht des minimalen Gerüstes echt kleiner als x.

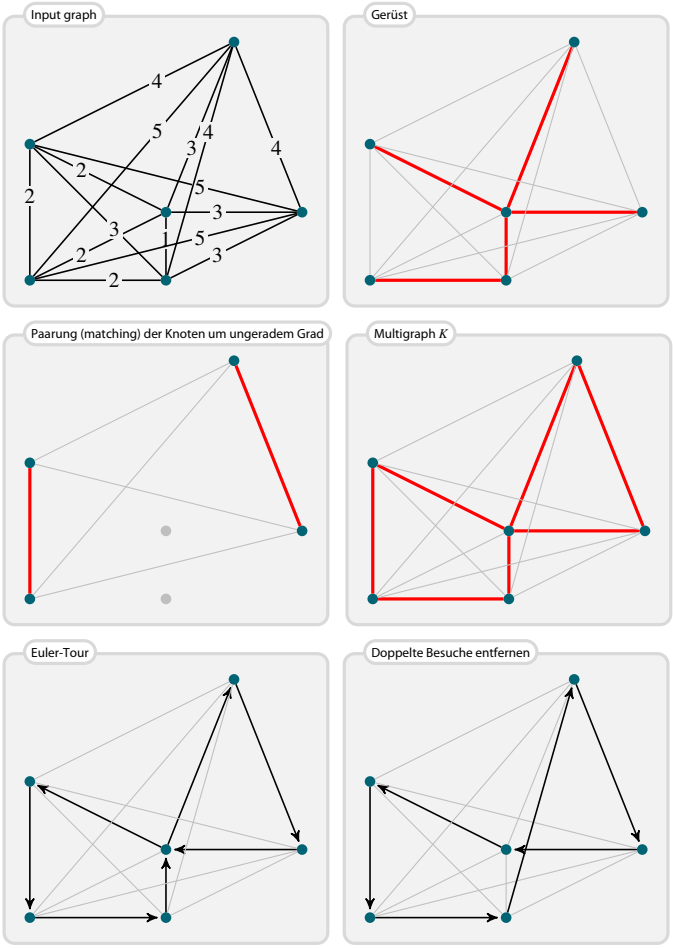
Nun betrachten wir die *Paarung* M (*matching*):

1. Betrachte eine optimale Tour auf H.
2. Ihr Gewicht ist höchstens x, da H induziert ist und die Dreiecksungleichung gilt.
3. Nun müssen entweder die ungerade oder die geraden Kanten auf der Tour ein Gewicht von höchstens $x/2$ haben.
4. Folglich hat eine Paarung mit minimalem Gewicht auf H ein Gewicht von höchstens $x/2$.

Dies zeigt, dass das Gewicht der Euler-Tour S höchstens $3x/2$ ist. Die Schleife am Ende verkürzt dann die Tour nur noch. □

11-24

Christofides' Algorithmus in Aktion



11.4 Approximationsschemata

11.4.1 Die Klasse PTAS

11-25

Noch besser als konstante Approximation: Approximationsschemata

- Für manche Probleme, wie MIN-VERTEX-COVER, können wir nicht unter eine bestimmte Approximationsrate kommen (falls nicht P = NP).
- Für manche Probleme ist aber möglich, »beliebig gut« zu approximieren, also eine Approximationsrate von $1+\varepsilon$ für jedes $\varepsilon > 0$ zu erreichen.

▶ Definition: Polynomialzeit-Approximationsschema
PTAS $= \bigcap_{r>1} r$-APX.

Man beachte: Die Entscheidungsvariante eines Probleme Q könnte NP-vollständig sein, obwohl $Q \in$ PTAS gilt. Der Grund ist, dass die Laufzeit *exponentiell von* $1/\varepsilon$ *abhängen kann*.

11.4.2 Rucksack-Problem

Das Rucksack-Problem.

11-26

Copyright by Marieke Kuijjer, Creative Commons Attribution-ShareAlike

→

Copyright by user Joadl, Creative Commons Attribution-ShareAlike

Ein Approximationsschema für das Rucksack-Problem.

11-27

▶ Problem: MAX-KNAPSACK

Eingabe	Eine Folge $(w_1,\ldots,w_n)$ von *Werten* und eine Folge $(g_1,\ldots,g_n)$ von *Gewichten* und ein Maximalgewicht m.
Lösungen	Eine Menge $I \subseteq \{1,\ldots,n\}$, so dass $\sum_{i\in I} g_i \leq m$.
Maß	Die Summe $\sum_{i\in I} w_i$.
Typ	Maximierung

▶ Satz

MAX-KNAPSACK $\in$ PTAS.

Wiederholung aus Algorithmendesign: Das Problem lässt sich mit dynamischer Programmierung lösen

11-28

Schritt 1: Rekursion

```
1 function rucksack(w_1,...,w_n,g_1,...,g_n,m) : int
2   if n = 0 then
3     return 0
4   else
5     if m < g_n then // Passt nicht?
6       return rucksack(w_1,...,w_{n-1},g_1,...,g_{n-1},m)
7     else // Passt
8       return max{rucksack(w_1,...,w_{n-1},g_1,...,g_{n-1},m),
9                  w_n + rucksack(w_1,...,w_{n-1},g_1,...,g_{n-1},m-g_n)}
```

In Java:

```
int rucksack(int[] w, int[] g, int n, int m)
{
  if (n == 0)
    return 0;
  else if (m < g[n])
    return rucksack(w,g,n-1,m);
  else
    return Math.max(rucksack(w,g,n-1,m),
                    w[n]+rucksack(w,g,n-1,m-g[n]));
}
```

11-29

Wiederholung aus Algorithmendesign: Das Problem lässt sich mit dynamischer Programmierung lösen

Schritt 2: Memoization

```
boolean[][] cached;
int[][]     table;

int rucksack(int[] w, int[] g, int n, int m)
{
  if (!cached[m][n]) {
    if (n == 0)
      table[m][n] = 0;
    else if (m < g[n])
      table[m][n] = rucksack(w,g,n-1,m);
    else
      table[m][n] = Math.max(rucksack(w,g,n-1,m),
                             w[n]+rucksack(w,g,n-1,m-g[n]));
    cached[m][n] = true;
  }
  return table[m][n];
}
```

11-30

Wiederholung aus Algorithmendesign: Das Problem lässt sich mit dynamischer Programmierung lösen

Schritt 3: Als Iteration

```
int rucksack(int[] w, int[] g, int n, int m)
{
  int[][] table = new int[m+1][n+1];

  for (int k=0; k<=n; k++) {
    for (int i=0; i<=m; i++) {
      if (k == 0)
        table[i][k] = 0;
      else if (i < g[k])
        table[i][k] = table[i][k-1];
      else
        table[i][k] = Math.max(table[i][k-1],
                               w[k] + table[i-g[k]][k-1]);
    }
  }
  return table[m][n];
}
```

- Das Rucksack-Problem kann in Zeit $O(mn)$ gelöst werden.
- Leider ist m eine binär als Teil der Eingabe gegebene Zahl und folglich ist m potenziell *exponentiell groß in Bezug auf die Eingabelänge*.

11-31

Wie wir die Laufzeit reduzieren.

Der Genauigkeitstrick

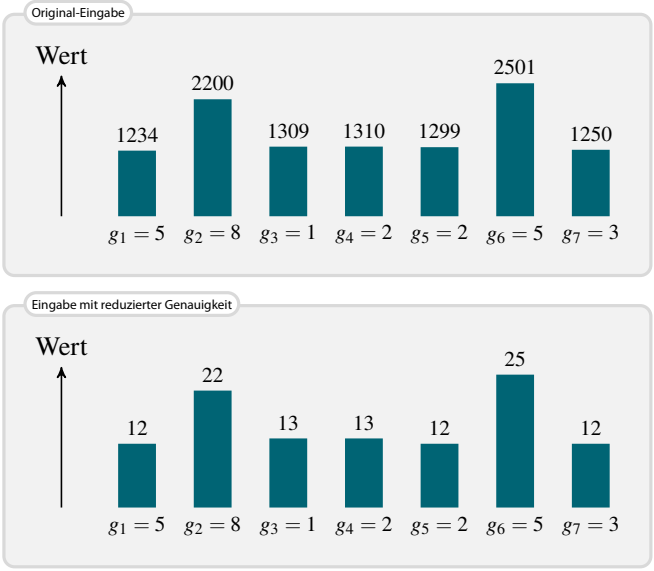
Da wir nur eine *approximative Lösung* suchen, reicht es, nur die *ersten paar Bits jeder Zahl zu betrachten*. Je genauer die Lösung sein muss, desto mehr Bits muss man betrachten.

Das Approximationsschema

1 ***input*** $(w_1,\ldots,w_n),(g_1,\ldots,g_n),m$ *und* ε
2 $k \leftarrow 1+1/\varepsilon$
3 $\mu \leftarrow \max_i w_i$
4 ***for*** $i \leftarrow 1$ ***to*** n ***do***
5 $\quad w'_i \leftarrow \lfloor w_i \frac{nk}{\mu} \rfloor$
6 ***return*** *die Menge* I*, die der exakte Algorithmus bei* (w',m) *ausgibt*

Darstellung des Genauigkeitstricks.

11-32



Analyse des Approximationsschemas.

11-33

▶ Lemma

Der Algorithmus gibt immer eine Lösung aus, deren Wert höchstens $1+\varepsilon$ vom Optimum abweicht.

Beweis. Sei J eine Lösung für die Originaleingabe und $x = \sum_{j\in J} w_j$. Sei I eine optimale Lösung für (w', m).

$$\begin{aligned}\sum_{i\in I} w_i &= \frac{\mu}{nk}\sum_{i\in I} w_i \frac{nk}{\mu} \geq \frac{\mu}{nk}\sum_{i\in I}\left\lfloor w_i \frac{nk}{\mu}\right\rfloor \\ &\geq \frac{\mu}{nk}\sum_{j\in J}\left\lfloor w_j \frac{nk}{\mu}\right\rfloor \geq \frac{\mu}{nk}\sum_{j\in J}(w_j\frac{nk}{\mu} - 1) \\ &\geq x - \frac{n\mu}{nk} \geq x - x/k = x/(1+\varepsilon).\end{aligned}$$

□

▶ Lemma

Der Algorithmus läuft in Zeit $O(n^2)$ für jedes feste ε.

Beweis. Die n Zahlen, auf die der exakte Algorithmus aufgerufen wird, liegen alle zwischen 1 und $O(n)$. □

Zusammenfassung dieses Kapitels

11-34

Approximationsrate
Ein Optimierungsproblem heißt *r-approximierbar*, wenn ein Algorithmus für jede Eingabe in polynomieller Zeit eine Lösung der Güte höchstens r berechnen kann.

Approximationsklassen

NPO
↓
APX
↓
3-APX
↓
2-APX
↓
1,5-APX
↓
1,01-APX
↓
PTAS
↓
PO

Unterschiedliche Approximierbarkeiten
Falls P $\neq$ NP, so gilt

- MAX-KNAPSACK $\in$ PTAS $\setminus$ PO.
- MIN-VERTEX-COVER $\in$ APX $\setminus$ PTAS.
- MIN-TSP $\in$ NPO $\setminus$ APX.

Zum Weiterlesen

[1] Irit Dinur and Shmuel Safra, The Importance of Being Biased, In *Proceedings of STOC 2002*, 33–42, 2002

Dies ist ein typischer Artikel zur (Nicht-)Approximierbarkeit. Es wird gezeigt, dass MIN-VERTEX-COVER nicht besser als mit Rate 1,36 approximiert werden kann, wenn nicht P = NP. Dieses Resultat baut auf dem PCP-Theorem auf, das in vielen Nichtapproximierbarkeitsresultaten verwendet wird und das einen der am schwierigsten zu beweisenden Sätze der Komplexitätstheorie darstellt.

Übungen zu diesem Kapitel

Übung 11.1 Greedy-Heuristik für Vertex-Cover, schwer

Zeigen Sie, dass die Greedy-Heuristik von Seite 11-17 eine Worst-Case-Approximationsrate von $\Theta(\ln n)$ hat. Zeigen Sie dazu:

1. Für jedes n gibt es einen Graphen G_n, so dass der Greedy-Algorithmus einen Lösung der Güte $\Omega(\ln n)$ ausgibt.
   Tipp: Wählen Sie für G_n einen bipartiten Graphen. Ein Ufer besteht aus n Knoten umfassen, die eine Knotenüberdeckung bilden (in einem bipartiten Graphen ist jedes Ufer eine Knotenüberdeckung). Das andere Ufer besteht aus $N-1$ Blöcken B_2, B_3, …, B_n, wobei B_i gerade $\lfloor N/i \rfloor$ Knoten hat und mit i Knoten im ersten Ufer verbunden ist.
2. Zeigen Sie, dass der Greedy-Algorithmus immer eine Lösung findet, die eine Güte von $\ln n$ hat.
   Tipp: In einer optimalen Knotenüberdeckung gilt: Ist d der maximale Grad, so muss die Überdeckung mindesten $|E|/d \leq \text{opt}(G)$ Knoten enthalten (warum?). Wenn E_i die nach Runde i verbleibende Kantenmenge ist und d_i der maximale Knotengrad nach Runde i, dann gilt $|E_{i+1}| \leq |E_i| - d_i \leq |E_i| - |E_i|/\text{opt}(G) = |E_i|(1-1/\text{opt}(G))$. Dies zeigt $|E_i| \leq |E|(1-1/\text{opt}(G))^i$. Um den Beweis abzuschließen, nutzen Sie $(1-x)^i \leq e^{-ix}$ für ein geeignetes x.

Übung 11.2 Optimierung versus Entscheidung, mittel

Zeigen Sie, dass NPO = PO genau dann, wenn NP = P.

Teil VII

Die große Welt des kleinen Platzes

Die Welt zwischen deterministischem und nichtdeterministischem logarithmischen Platz hat mich schon immer fasziniert. Hier liegen viele Dinge anders als man es sonst im Komplexitätszoo gewohnt ist.

Machen wir uns zunächst klar, wie wenig *logarithmisch viel* Platz wirklich ist. Bei Eingaben der Länge n haben wir $O(\log n)$ Bits an Platz zur Verfügung. Das »O« verdeckt hier eine Konstante, wir haben also genauer $c \cdot \log_2 n$ Bits zu Verfügung für eine Konstante c, die beispielsweise 3 sein könnte – dies ist durchaus typisch bei vielen Algorithmen. Wenn nun unsere Eingaben 128 Bits lang sind, so haben wir $3 \cdot \log_2 128 = 21$ Bits an Speicher zur Verfügung. Haben wir hingegen ein Terabyte an Daten (also etwa den Inhalt einer Harddisk), so stehen $3 \cdot \log_2(8 \cdot 1024^4) = 129$ Bits zur Verfügung – oder etwas mehr als zwei 64 Bit Register. Haben wir schließlich »das ganze Universum als Eingabe«, also jedes Atom des Universums entweder eine 0 oder eine 1 speichert, so können wir $3 \cdot \log_2(8 \cdot 10^{80}) \approx 807$ Bits oder dreizehn 64-Bit-Register nutzen. Noch einmal: Für Eingaben von der Größe des Universums brauchen Logspace-Algorithmen nie mehr als dreizehn 64-Bit-Register. Sie werden zustimmen müssen, dass dies wahrlich ein kleiner Platzbedarf ist.

Eine andere Sicht auf logarithmischen Platz ist ebenfalls »erhellend«: Wenn man das Arbeitsband, auf dem ja $c \log_2 n$ Bits stehen, in c Blöcke von $\log_2 n$ Bits aufteilt, so kann jeder Block genau eine Position in der Eingabe speichern. Dies ist kein Zufall: Man kann zeigen, dass die Klasse L von in logarithmischem Platz akzeptierbarer Sprachen genau die Klasse aller Sprachen ist, die von 2-Wege-Mehrkopf-Automaten akzeptiert werden. Von diesem Standpunkt aus mutiert logarithmischer Platz zu »Berechnungen, bei denen lediglich eine feste Zahl an Zeigern auf der unveränderlichen Eingabe deterministisch verschoben werden«. Es erscheint dann schon ziemlich erstaunlich, dass man auf diese Art beispielsweise das Erreichbarkeitsproblem in ungerichteten Graphen lösen kann.

In diesem Teil sollen zwei zentrale Resultate über logarithmischen Platz genauer untersucht werden: Der Satz von Savitch und der Satz von Immerman-Szelepcsényi. Der Satz von Savitch ist recht einfach zu zeigen, hat aber wichtige und überraschende Konsequenzen (im Wesentlichen, dass nichtdeterministischer und deterministischer polynomieller Platz dasselbe sind). Der Satz von Immerman-Szelepcsényi ist wesentlich schwieriger zu beweisen und hat ebenfalls überraschende Konsequenzen: Die kontextsensitiven Sprachen sind unter Komplementbildung abgeschlossen. Beiden Resultate zeigen, dass Sätze über logarithmischen Platz Anwendungen außerhalb der Welt zwischen L und NL haben.

12-1

Kapitel 12
Der Satz von Savitch

Eine einfache Idee mit weitreichenden Folgen

12-2

Lernziele dieses Kapitels

1. Den Satz von Savtich kennen
2. Den Beweis verstehen
3. Konsequenzen für polynomiellen Platz kennen

Inhalte dieses Kapitels

Worum es heute geht

Im Jahre 1970 machte Walter Savtich eine unglaubliche Entdeckung: Um in einem Graphen von einem Startknoten s zu einem Zielknoten t zu gelangen, muss man einen Knoten v durchlaufen, der (fast) die gleiche Entfernung von s und t hat. Noch unglaublicher ist allerdings, dass man, um von s nach v zu gelangen, *wiederum* einen Knoten durchlaufen muss, der etwa die gleiche Entfernung von s und v hat! Und das gilt auch für v und t! Und nun gilt, dass jeder dieser vier Pfade *wiederum* einen Mittelknoten besitzen muss! (Wie bei so vielen Entdeckungen in der Mathematik erscheinen diese immer reichlich trivial, wenn man sie viele Jahre später in einem Lehrbuch liest wie diesem Text. Aber glauben Sie mir: Darauf zu kommen ist alles andere als einfach.)

Savitch' Beobachtung lässt sich leicht rekursiv formulieren: Jeder Pfad von s nach t durchläuft einen Knoten v, der die gleiche Entfernung von s und t hat (oder die Entfernungen unterscheiden sich um genau 1), und dies gilt rekursiv dann auch für die Pfade von s zu v und von v zu t. Hieraus entsteht ein Algorithmus (Savitch' Algorithmus) für das Erreichbarkeitsproblem, der (a) nur extrem wenig Platz benötigt, da wir nur einen sehr kleinen Aufrufstack speichern müssen, und (b) irrsinnig langsam ist.

Wirklich erstaunlich am Satz von Savitch ist allerdings nicht wie oben behauptet die dem Algorithmus zugrundeliegende Beobachtung sondern die vielfältigen Konsequenzen, die er in scheinbar völlig anderen »Bereichen« der Theorie hat. Erstens folgt aus ihm, dass nichtdeterministischer logarithmischer Platz in deterministischem Platz $O(\log^2 n)$ enthalten ist. Zweitens folgt aus ihm, dass nichtdeterministischer und deterministischer polynomieller Platz gleich sind (!). Drittens werden wir ihn nutzen, um zu zeigen, dass eine Variante des Erfüllbarkeitproblems vollständig für PSPACE ist.

12.1 Das Satz von Savitch

Ein netter Satz.

12-4

▶ Satz

REACH $\in$ SPACE$\big(O(\log^2 n)\big)$.

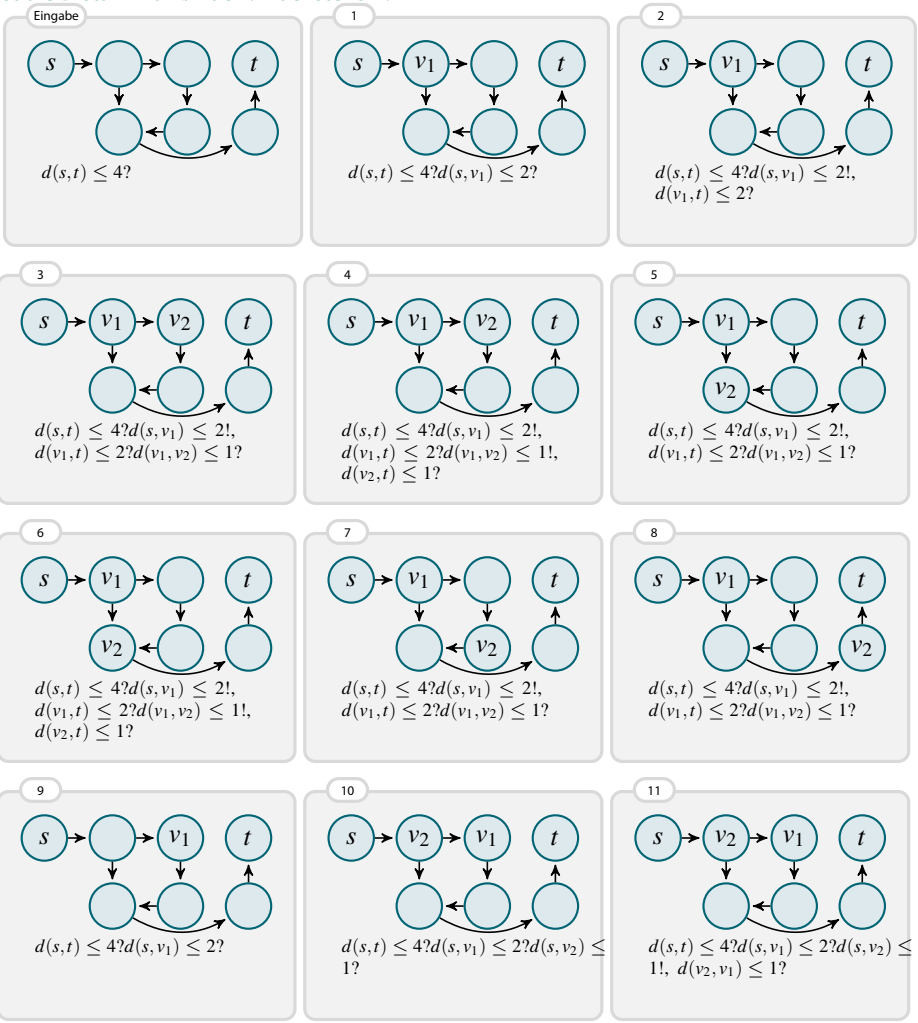
Der Beweis nutzt folgenden Algorithmus:

```
1  function isDistanceAtMost(G,s,t,k) : boolean
2    // Überprüft, ob die Distanz von s nach t in G höchstens k ist
3    if s = t then // s und t sind gleich
4      return true
5    else if (s,t) ∈ E then // Kante von s nach t
6      return true if k ≥ 1, otherwise false
7    else if k ≥ 2 then
8      foreach v ∈ V − {s,t} do // Suche »Mittelknoten«
9        if isDistanceAtMost(G,s,v,⌊k/2⌋) = true then
10         if isDistanceAtMost(G,v,t,⌈k/2⌉) = true then
11           return true
12   return false
```

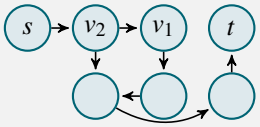
Savitch' Algorithmus in Aktion.

12-5

Ist die Distanz von s nach t höchstens 4?

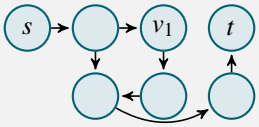


12



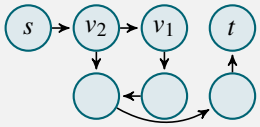
$d(s,t) \leq 4?d(s,v_1) \leq 2?d(s,v_2) \leq 1!,\ d(v_2,v_1) \leq 1!$

13



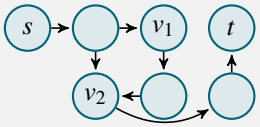
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?$

14



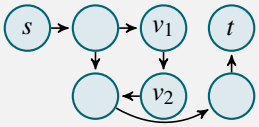
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

15



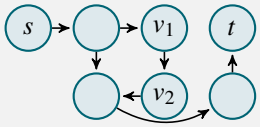
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

16



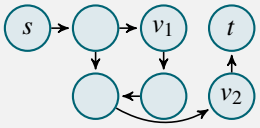
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

17



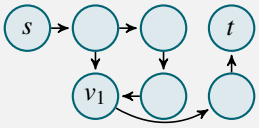
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1!,\ d(v_2,t) \leq 1?$

18



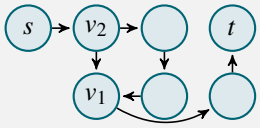
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

19



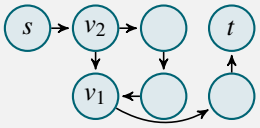
$d(s,t) \leq 4?d(s,v_1) \leq 2?$

20



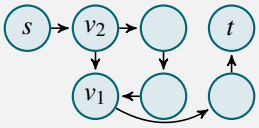
$d(s,t) \leq 4?d(s,v_1) \leq 2?d(s,v_2) \leq 1?$

21



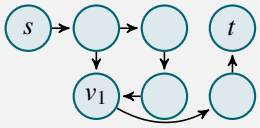
$d(s,t) \leq 4?d(s,v_1) \leq 2?d(s,v_2) \leq 1!,\ d(v_2,v_1) \leq 1?$

22



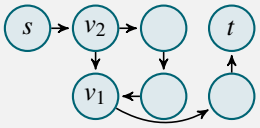
$d(s,t) \leq 4?d(s,v_1) \leq 2?d(s,v_2) \leq 1!,\ d(v_2,v_1) \leq 1!$

23



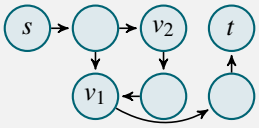
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?$

24



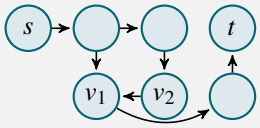
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

25



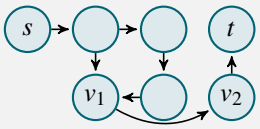
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

26



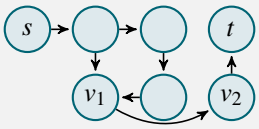
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

27



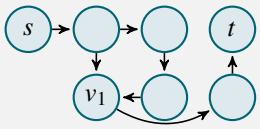
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1?$

28



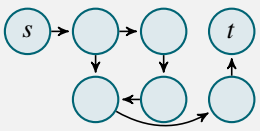
$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2?d(v_1,v_2) \leq 1!,\ d(v_2,t) \leq 1!$

29



$d(s,t) \leq 4?d(s,v_1) \leq 2!,\ d(v_1,t) \leq 2!$

30



$d(s,t) \leq 4!$

12-6

Beweis. Wir zeigen per Induktion, dass isDistanceAtMost$(G,s,t,k) =$ true genau dann gilt, wenn $\text{dist}(s,t) \leq k$.

1. Für $k \leq 1$ ist die Ausgabe korrekt.
2. Für $k > 1$, betrachte einen kürzesten Pfad von s nach t der Länge $x \leq k$. Sei v der $\lfloor x/2 \rfloor$-te Knoten auf dem Pfad. Dann gilt $\text{dist}(s,v) \leq \lfloor x/2 \rfloor \leq \lfloor k/2 \rfloor$ und $\text{dist}(v,t) \leq \lceil x/2 \rceil \leq \lceil k/2 \rceil$. Also gibt nach Induktionsvoraussetzung der Algorithmus »true« zurück.
3. Gibt umgekehrt der Algorithmus »true« zurück, so gibt es einen Knoten v mit $\text{dist}(s,v) \leq \lfloor k/2 \rfloor$ und $\text{dist}(v,t) \leq \lceil k/2 \rceil$, was $\text{dist}(s,t) \leq k$ zeigt.

Weiter gilt, dass der Algorithmus höchstens $O(\log^2 |V|)$ Platz braucht: In jedem rekursiven Aufruf brauchen wir $O(\log |V|)$ Platz, um v und die Rücksprungadresse zu speichern. Die Rekursionstiefe ist $\log |V|$. Also benötigt der Call-Stack $O(\log^2 |V|)$ Bits an Speicher. □

12.2 Konsequenzen für . . .

12.2.1 NL

Das Verhältnis von deterministischem und nichtdeterministischem *logarithmischem* Platz. 12-7

▶ Satz
REACH *ist vollständig für* NL.

✎ Zur Übung
Skizzieren Sie den Beweis:

1. Warum gilt nochmal REACH $\in$ NL?
2. Zeigen Sie durch Bootstrapping, dass alle Sprachen $A \in$ NL sich auf REACH reduzieren lassen.

Ein Folgerung aus dem Satz von Savitch 12-8

▶ Folgerung
$\text{NL} \subseteq \text{SPACE}\big(O(\log^2 n)\big)$.

Nach dem Satz gilt also $\text{L} \subseteq \text{NL} \subseteq \text{L}^2$. Geht's auch besser? Gilt $\text{NL} \subseteq \text{SPACE}\big(O(\log^{1.9999} n)\big)$? Dies ist mit das größte offene Problem im Bereich der kleinen Platzklassen.

12.2.2 NPSPACE

Das Verhältnis von deterministischem und nichtdeterministischem *polynomiellen* Platz. 12-9

Überraschenderweise folgt aus dem Satz von Savitch auch etwas für *polynomiellen* Platz. Der Grund ist, dass *»Klasseninklusionen sich nach oben propagieren«*:

▶ Definition: Padding-Sprachen
Sei Σ ein Alphabet, das # nicht enthält. Sei $A \subseteq \Sigma^*$. Sei $f\colon \mathbb{N} \to \mathbb{N}$. Dann ist die *Padding-Sprache* $\text{pad}_f(A) \subseteq (\Sigma \cup \{\#\})^*$ die Sprache

$$\text{pad}_f(A) = \{x\#^{f(|x|)} \mid x \in A\}.$$

Mit anderen Worten: Wir hängen jede Menge Hashmarks an die Wörter von A an.

▶ Lemma: Padding-Lemma
Sei A eine Sprache und $f, g\colon \mathbb{N} \to \mathbb{N}$ sinnvolle Funktionen. Dann gilt:

1. $A \in \text{SPACE}(g(n + f(n))) \iff \text{pad}_f(A) \in \text{SPACE}(g(n))$.
2. $A \in \text{NSPACE}(g(n + f(n))) \iff \text{pad}_f(A) \in \text{NSPACE}(g(n))$.
3. $A \in \text{TIME}(g(n + f(n))) \iff \text{pad}_f(A) \in \text{TIME}(g(n))$.
4. $A \in \text{NTIME}(g(n + f(n))) \iff \text{pad}_f(A) \in \text{NTIME}(g(n))$.

12-10

Beweis des Padding-Lemmas.

Beweis. Wir zeigen nur die erste Behauptung, die anderen gehen sehr ähnlich. Wie zeigen zwei Richtungen:

1. Sei $A \in \text{SPACE}(g(n+f(n)))$ via einer Maschine M. Wir müssen $\text{pad}_f(A) \in \text{SPACE}(g(n))$ via einer Maschine M' zeigen. Bei Eingabe $y = x\#^{f(|x|)}$ simuliert M' gerade M auf Eingabe x und ignoriert die Hashmarks. Dies benötigt Platz $g(|x|+f(|x|))$ und $|x|+f(|x|) = |y|$ ist gerade die Länge unserer Eingabe.
2. Sei nun $\text{pad}_f(A) \in \text{SPACE}(g(n))$ via M'. Wir müssen zeigen, dass $A \in \text{SPACE}(g(n+f(n)))$ via einer Maschine M. Bei Eingabe x hängt M an x den String $\#^{f(|x|)}$ an und lässt M' auf $y = x\#^{f(|x|)}$ laufen. Dann braucht M' Platz $g(|y|) = g(|x|+f(|x|))$. Folglich braucht die Simulation Platz $g(n+f(n))$. □

12-11

Folgerungen aus dem Satz von Savitch und dem Padding-Lemma.

▶ **Folgerung**
$\text{NSPACE}(n^k) \subseteq \text{SPACE}(n^{2k})$.

Beweis. Seien $f(n) = 2^{n^k} - n$ und $g_1(n) = \log n$ und $g_2(n) = \log^2 n$. Dann gilt $g_1(n+f(n)) = n^k$ und $g_2(n+f(n)) = n^{2k}$. Sein nun $A \in \text{NSPACE}(n^k)$. Nach dem Padding-Lemma gilt $\text{pad}_f(A) \in \text{NSPACE}(\log n)$. Nach dem Satz von Savitch gilt $\text{pad}_f(A) \in \text{SPACE}(\log^2 n)$. Nach dem Padding-Lemma gilt $A \in \text{SPACE}(n^{2k})$. □

▶ **Folgerung**
NPSPACE = PSPACE.

12.2.3 PSPACE-Vollständigkeit

12-12

Next-Generation-Erfüllbarkeit.

Was unterscheidet eigentlich CIRCUIT-SAT und CIRCUIT-TAUT?

- Bei CIRCUIT-SAT ist die Frage, ob eine Belegung *existiert,* so dass der Schaltkreis 1 ausgibt.
- Bei CIRCUIT-TAUT ist die Frage, ob *jede* Belegung den Schaltkreis zu 1 auswerten lässt.

Dies scheint auf eine *allgemeinere Idee* hinzudeuten: »Existiert« und »für alle« werden typischerweise mittels *Quantoren* ausgedrückt. Allgemeiner wollen wir nun *pro Eingabegatter* regeln, ob dort eine Belegung »existieren« muss oder »für alle« Belegungen dieses Gatters etwas gelten muss. Wenn alle Quantoren $\exists$ sind, so ergibt sich gerade CIRCUIT-SAT; wenn alle $\forall$ sind, so ergibt sich CIRCUIT-TAUT.

12-13

Formale Definition des Quantified-Boolean-Circuit-Problems.

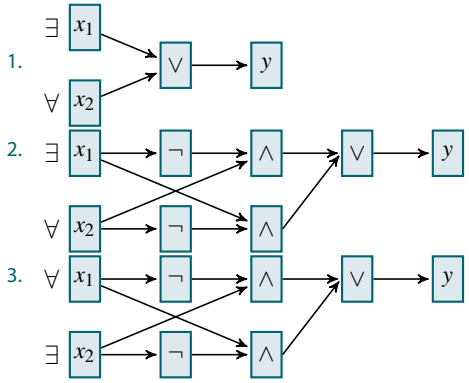
▶ **Definition:** Schaltkreis mit Quantoren
Syntaktisch ist ein *Schaltkreis mit Quantoren* ein Schaltkreis mit Eingabegattern $x_1, \ldots, x_n$, einem Ausgabegatter und einer Beschriftung, die jedem Eingabegatter einen der Quantoren $\exists$ oder $\forall$ zuordnet. Semantisch definieren wir induktiv, wann ein solcher Schaltkreis *wahr ist:*

1. Wenn C keine Eingabegatter hat, so ist er wahr, wenn er zu 1 auswertet.
2. Wenn das erste Eingabegatter x_1 mit $\exists$ beschriftet ist, so ist der Schaltkreis wahr, wenn C_0 oder C_1 wahr ist. Hierbei entsteht C_i aus C, indem Gatter x_1 durch das i-Konstanten Gatter ersetzt wird (und die Gatter neu nummeriert werden, so dass die Nummerierung wieder bei 1 beginnt.)
3. Wenn das erste Eingabegatter x_1 mit $\forall$ beschriftet ist, so ist der Schaltkreis war, wenn sowohl C_0 und C_1 wahr sind.

Die Sprache QBC ist die Menge aller wahren Schaltkreise mit Quantoren.

✎ Zur Übung 12-14

Welche der folgenden Schaltkreise sind wahr (sind Element von QBC)?



Das Satz von Savitch bildet den Kern der PSPACE-Vollständigkeit von QBC. 12-15

▶ Satz

QBC *ist vollständig für* PSPACE.

Beweis. Wir zeigen:

1. QBC $\in$ PSPACE.
2. QBC ist schwer für PSPACE.

□

Inklusion von QBC in PSPACE 12-16

Wir behaupten, dass QBC $\in$ PSPACE via:

```
1  function solveQBC (C) : boolean
2    // Gibt »true« zurück, wenn C wahr ist
3    if C hat keine Eingabegatter then
4      return »true«, falls C zu 1 auswertet, sonst »false«
5    else
6      C_0 ← C mit x_1 substituiert durch das 0-Konstanten Gatter
7      C_1 ← C mit x_1 substituiert durch das 1-Konstanten Gatter
8      b_0 ← solveQBC(C_0)
9      b_1 ← solveQBC(C_1)
10     if die Beschriftung von x_1 ist ∀ then
11       return b_0 ∧ b_1
12     else
13       // die Beschriftung ist ∃
14       return b_0 ∨ b_1
```

Die Rekursionstiefe des Algorithmus ist linear. In jedem rekursiven Aufruf müssen wir nur linear viel speichern. Insgesamt brauchen wir also quadratisch viel Platz.

Reduktion auf QBC. 12-17

Wir müssen nun zeigen, dass jedes Problem in PSPACE sich auf QBC reduzieren lässt. Dazu reduzieren wir von SUCCINCT-REACH, von dem wir aus Übung 12.1 schon wissen, dass es PSPACE-vollständig ist.

▶ Lemma

SUCCINCT-REACH $\leq_m^{\log}$ QBC.

Beweisskizze 12-18

Die Situation

Für die Reduktion sei C ein Schaltkreis, der einen Graphen G beschreibt mit zwei Knoten s und t darin. Die Frage ist nun, ob es einen Pfad von s nach t gibt. Dazu sollen wir einen Schaltkreis D mit Quantoren bauen, der genau dann zu wahr auswertet, wenn es den Pfad gibt.

Die Idee

Savitch' Algorithmus auf G angewandt lautet wie folgt:

- Es gibt einen Pfad von s nach t, falls *ein Knoten* v_1 *existiert,* so dass *für beide Paare* (s,v_1) *und* (v_1,t) ein Pfad der halben Länge existiert.
- Für ein solches Paar (s',t') gibt es einen Pfad von s' zu t', falls *ein Knoten* v_2 *existiert,* so dass *für beide Paare* (s',v_2) *und* (v_2,t') ein Pfad der halben Länge existiert.
- Für ein solches Paar (s'',t'') gibt es einen Pfad von s'' zu t'', falls *ein Knoten* v_3 *existiert,* so dass *für beide Paare* (s'',v_3) *und* (v_3,t'') ein Pfad der halben Länge existiert.
- $\ldots$
- Es gibt einen Pfad von $s'^{\cdots\prime}$ zu $t'^{\cdots\prime}$ der Länge höchstens 1, falls es eine direkte Kante zwischen ihnen gibt oder sie gleich sind.

Die Eingabe-Gatter

Der Schaltkreis D nutzt einen Block von $\exists$-Gattern, um den Knoten v_1 zu »raten«. Es folgt *ein* $\forall$-Gatter, das entscheidet, ob wir das erste oder das zweite Paar von (s,v_1) und (v_1,t) überprüfen. Dann folgt in D wieder ein Block von $\exists$-Gattern, um den Knoten v_2 zu »raten«. Dann kommt wieder ein $\forall$-Gatter, das entscheidet, ob wir nun (s',v_2) oder (v_2,t') weiter überprüfen. Dann folgt in D wieder ein Block, um v_3 zu raten, gefolgt von einem $\forall$-Gatter und so weiter. Insgesamt gibt es $(n+1)n$ Eingabegatter.

Der Überprüfungsschaltkreis

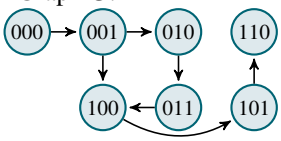
»Nachdem alles geraten wurde«, muss nun überprüft werden, ob es eine Kante von $s'^{\cdots\prime}$ nach $t'^{\cdots\prime}$ gibt (oder diese identisch sind, was aber einfach ist). Wenn wir nun $s'^{\cdots\prime}$ und $t'^{\cdots\prime}$ »kennen«, so ist leicht zu überprüfen, ob es eine Kante in G gibt: Der Schaltkreis C beantwortet genau diese Frage. Wir »bauen ihn also ein« in unsere Ausgabe.
Die Werte von $s'^{\cdots\prime}$ und $t'^{\cdots\prime}$ berechnen sich wie folgt: Wir beginnen mit dem Paar (s,t) und wiederholen für aufsteigende i: Wenn das i-te $\forall$-Gatter 0 ist, ersetze die erste Komponente im Paar durch v_i, sonst ersetze die zweite durch v_i. Diese Schritte können durch einen »Auswahlschaltkreis σ« erledigt werden.

12-19

Beispiel, wie die Reduktion arbeitet.

Beispiel: Eingabe für die Reduktion

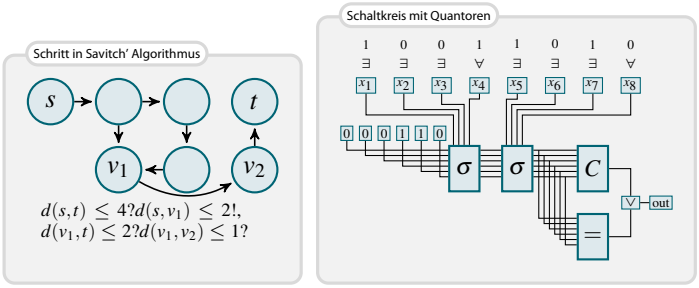
Graph G:



Kodierender Schaltkreis C:

- $C(000001) = 1$, da es eine Kante 000 nach 001 gibt.
- $C(001000) = 0$

Beispiel: Von Savitch' Algorithmus zu Schaltkreisen mit Quantoren



Zusammenfassung dieses Kapitels

▶ Satz von Savitch
REACH $\in$ SPACE$\big(O(\log^2 n)\big)$.

▶ Folgerung (oft auch Satz von Savitch genannt)
NL $\subseteq$ L^2.

▶ Folgerung
NPSPACE = PSPACE.

▶ Folgerung
QBF ist PSPACE-vollständig.

Zum Weiterlesen

[1] Walter Savitch, Relationship between nondeterministic and deterministic tape classes, *Journal of Computer System Sciences*, 4(2):177–192, 1970

In diesem Artikel präsentiert Savitch seinen Algorithmus. Wie man sieht, ist der Artikel recht »alt« und die »modernen« Begriffe wie logarithmischer Platz oder Logspace-Reduktionen waren noch nicht entwickelt. Dementsprechend ist es reichlich schwierig zu verstehen, was eigentlich in dem Artikel gezeigt wird, und erst recht, wie es gezeigt wird.

Übungen zu diesem Kapitel

Übung 12.1 Vollständigkeit von Succinct-Reach, mittel

Zeigen Sie, dass SUCCINT-REACH vollständig ist für NPSPACE.

Tipps: SUCCINCT-REACH $\in$ NPSPACE sieht man sehr leicht (raten Sie einfach einen Pfad).

Für die Reduktion, sei $A \in$ NPSPACE via M beliebig. Sei x eine Eingabe, die die Reduktion auf ein y abbilden soll, das genau dann in SUCCINCT-REACH liegt, wenn M das Wort x akzeptiert. Betrachten Sie den Konfigurationsgraphen G von M bei Eingabe x. Man kann davon ausgehen, dass M genau eine Start- und genau eine Endkonfiguration hat (warum?). Die Frage ist dann, ob es in G einen Weg von einem bestimmten Knoten s zu einem Knoten t gibt. Der Graph G wird sehr groß sein (wie groß?), aber er wird aber auch *sehr regelmäßig* sein. Insbesondere wird er so regelmäßig sein, dass man mit einem *kleinen Schaltkreis* entscheiden kann, ob es eine Kante von einer Konfiguration zu einer anderen gibt. Dies wird dann genau der gesuchte Schaltkreis sein.

Übung 12.2 Vollständigkeit von QBF, mittel

Das Problem QBF verhält sich zu QBC genau wie SAT zu CIRCUIT-SAT: Statt Schaltkreisen nutzen wir nun Formeln. Im Detail ist QBF wie folgt definiert:

▶ Definition: Quantified Boolean Formula Problem
Aussagenlogische Formeln mit Quantoren (die es in der Logik normalerweise *nicht* gibt) sind induktiv wie folgt definiert:
- Jede aussagenlogische Formeln ist eine.
- Wenn φ eine aussagenlogische Formel mit Quantoren ist, so auch $\exists a(\varphi)$ und $\forall a(\varphi)$, wobei a eine *aussagenlogische* Variable ist.

Die *Modellrelation* ist wie folgt definiert: Sei φ eine aussagenlogische Formeln mit Quantoren und $\beta : V \to \{0,1\}$ eine Belegung aller in φ vorkommenden Variablen (frei oder gebunden). Dann gilt $\beta \models \varphi$, falls:
- Wenn φ keine Quantoren enthält, so ist $\beta \models \varphi$ wie üblich definiert.
- Wenn $\varphi = \exists a(\psi)$, seien β_i mit $i \in \{0,1\}$ die Modifikationen von β, bei denen $\beta_i(a) = i$ gesetzt wurde. Dann gilt $\beta \models \varphi$, falls $\beta_0 \models \psi$ oder $\beta_1 \models \psi$.
- Für $\varphi = \forall a(\psi)$ gilt $\beta \models \varphi$, falls sowohl $\beta_0 \models \psi$ als auch $\beta_1 \models \psi$.

Die Sprache QBF enthält alle aussagenlogischen Formeln mit Quantoren, die ein Modell haben.

Zeigen Sie, dass QBF vollständig ist für PSPACE.

Tipp: Zeigen Sie QBC $\leq_m^{\log}$ QBF und QBF $\leq_m^{\log}$ QBC.

13-1

Kapitel 13
Der Satz von Immerman-Szelepcsényi
Sehr, sehr genaues Zählen

13-2

Lernziele dieses Kapitels

1. Den Satz von Immerman-Szelepcsényi kennen
2. Das Konzept des induktiven Zählens verstehen
3. Anwendungen des Satzes kennen

Inhalte dieses Kapitels

Worum es heute geht

Author Freddy Weber, public domain

Copyright by Pierre Brial, Creative Commons Attribution-Sharealike

Zu beweisen, dass etwas existiert, ist meist viel einfacher als zu beweisen, dass etwas nicht exstiert. Um zu zeigen, dass SAT ∈ P, müssen Sie »nur« einen Polynomialzeitalgorithmus für SAT angeben. Zu zeigen, dass *kein* solcher Algorithmus existiert, scheint schwieriger. Um zu zeigen, dass eine *Welwitschia mirabilis* existiert, müssen Sie einfach eine Photographie einer solchen Pflanze herumzeigen. Zu zeigen, dass weder die Hommingberger Gepardenforelle noch Nessie existieren, ist schwieriger. Um zu zeigen, dass Gott existiert, müssen Sie lediglich ein Wunder produzieren. Zu zeigen, dass er (sie? es?) nicht existiert, scheint schwieriger.

In diesem Kapitel werden wir viel Energie darauf verwenden zu beweisen, dass bestimmte Knoten eines Graphen *nicht* erreichbar sind von einem bestimmten Knoten in einer bestimmten Anzahl Schritten. Es ist, wie oben beschrieben, ziemlich schwierig, Menschen (oder nichtdeterministische Turingmaschinen) davon zu überzeugen, dass ein bestimmter Knoten *nicht* erreicht werden kann von einem anderen Knoten. (Wie viel einfacher ist es da, jemanden davon zu überzeugen, dass es einen Weg gibt: Man gibt ihn einfach an / rät ihn nichtdeterministisch.)

An dieser Stelle kommt ein genialer Trick ins Spiel, den Immerman und Szelepcsényi unabhängig voneinander praktisch gleichzeitig entdeckt haben: Nehmen wir an, ich *weiß ganz sicher*, dass von den 100 Knoten in einem Graphen *ganz genau* 42 erreichbar sind von einem Startknoten aus. Um nun zu »beweisen«, dass ein bestimmter Knoten x *nicht* erreichbar ist, zeigen wir, dass es Pfade von s zu 42 *unterschiedlichen, anderen* Knoten als x gibt. Dann ist klar, dass es *keinen* Pfad mehr zu x geben kann.

Nun stellt sich als neues Problem natürlich, wie wir herausfinden sollen, dass ausgerechnet 42 Knoten von s aus erreichbar sein sollen. Hier brauchen wir einen zweiten genialen Trick, der als *induktives Zählen* bekannt geworden ist: Die genaue Anzahl der Knoten mit Entfernung höchstens d von s kann man berechnen, wenn man die genaue Anzahl von Knoten mit Entfernung höchstens $d-1$ von s kennt. Man beginnt hier mit $k=0$ und der Anzahl 1 (s selbst) und »zählt dann induktiv«.

Lohn der ganzen Mühe ist dann der Satz von Immerman-Szelepcsényi, nachdem das Nichterreichbarkeitsproblem für Graphen in NL liegt. Dieses Resultat hat eine ganze Reihe von

Konsequenzen, die wir uns in diesem Kapitel auch ansehen werden.

13.1 Der Satz

13.1.1 Die Behauptung

Das Nichterreichbarkeitsproblem. 13-4

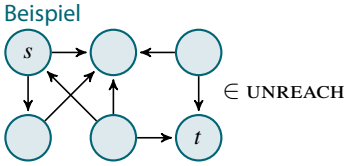
Die Sprache REACH

Eingabe Ein gerichteter Graph G, zwei Knoten s und t.
Frage Gibt es einen Pfad von s nach t?

Die Sprache UNREACH

Eingabe Ein gerichteter Graph G, zwei Knoten s und t.
Frage Gibt es *keinen* Pfad von s nach t?

Beispiel



$\in$ UNREACH

Der Satz von Immerman-Szelepcsényi 13-5

▶ **Satz**
UNREACH $\in$ NL.

Dieser Satz ist sehr überraschend, da die Definition von Nichtdeterminismus es ja nur leicht macht zu beweisen, dass etwas *existiert* – und nicht, dass etwas *nicht existiert.* Wir brauchen daher für den Beweis eine Methode, um zu »zertifizieren«, dass *kein* Pfad zwischen zwei Knoten existiert. Hierzu werden einige magische Tricks benötigt. Offiziell heißen diese »induktives Zählen« und, in der fortgeschrittenen Form, »doppeltinduktives Zählen«.

Beweisplan. 13-6

- Der NL-Algorithmus für UNREACH dreht sich hauptsächlich um etwas ganz anderes: *Finde heraus, wie viele Knoten von s erreichbar sind in d Schritte für unterschiedliche d.* (Deshalb heißt die Methode "induktives *Zählen*".)
- Wenn wir diese Anzahl für ein d errechnet haben, so werden wir daraus die Anzahl für $d+1$ errechnen.
  (Deshalb heißt die Methode "*induktives* Zählen".)
- Die zentrale Frage ist dann natürlich, wie das Wissen um die Anzahl der erreichbaren Knoten uns hilft herauszufinden, ob t von s aus erreichbar ist.

Einige Notationen. 13-7

▶ **Notation**

- G ist der gerichtete Eingabegraph.
- s ist der Startknoten laut Eingabe.
- t ist der Zielknoten laut Eingabe.
- Für eine Entfernung d sei $R_d \subseteq V$ die Menge aller in maximal d Schritten von s erreichbaren Knoten.

Offenbar gilt

- $R_0 = \{s\}$,
- $R_1 = \{s\} \cup \{v \mid v \text{ ist ein Nachbar von } s\}$,
- $R_{|V|} = R_\infty = \{v \mid v \text{ ist erreichbar von } s\}$.

Wir suchen nun Zertifikate für $t \notin R_\infty$.

13.1.2 Den Richter überzeugen

13-8

Das schreckliche Verbrechen

Ein schreckliches Verbrechen wurde verübt! Diebe sind in die Büros des Instituts für Theoretische Informatik eingebrochen und haben ihre Rucksäcke mit Theorieartikeln gefüllt. (Die gestohlenen Artikel haben einen maximalen kumulierten Impact-Factor unter allen in den Rucksack passenden Auswahlen an Artikeln, aber das wird im Folgenden irrelevant sein.) Inspektor Closure von der Sonderkommission *Elfenbeinturm* versucht nun, die Diebe zu fassen. Es gelingt ihm, eine Reihe von Verdächtigen zu ermitteln.

Die etwas überraschten Verdächtigen

Public domain, Tango Project

13-9

Die Verfahrensbeteiligten

Es gibt ein Verfahren, dem eine *Richterin* vorsitzt. Der *Inspektor* hat das Ziel, die Richterin zu überzeugen, dass die Verdächtigen *schuldig* sind. Der *Verteidiger* hat das Ziel, die Richterin von der *Unschuld* der Verdächtigen zu überzeugen.

13-10

Der Inspektor hat es leichter.

Wie der Inspektor die Richterin überzeugen kann

Um die Richterin zu überzeugen, dass ein bestimmter Verdächtiger (zum Beispiel der dritte) schuldig ist, lässt er dessen Wohnung durchsuchen und findet dort gestohlene Artikel, was die Richterin überzeugt.

Der Verteidiger hat es schwerer.

Für die Verteidigung ist es viel schwieriger, die Richterin von der Unschuld eines Verdächtigen *vollkommen zu überzeugen*. Werden bei der Wohnungsdurchsuchung *keine* Artikel gefunden, so *überzeugt dies die Richterin nicht* (die Artikel könnten woanders versteckt sein). Natürlich gilt »unschuldig bis zum Beweis der Schuld«, aber *überzeugt* von der Unschuld ist die Richterin hierdurch noch nicht. *Was kann die Verteidigung tun, um die Richterin abschließend von der Unschuld eines Angeklagten zu überzeugen?*

13-11

Warum die Anzahl der Diebe so nützlich ist.

Nehmen wir nun an, eine Überwachungskamera habe den Einbruch gefilmt. Da die Diebe maskiert waren, erkennt man ihre Gesichter nicht, aber es ist *vollkommen klar, dass es genau drei Diebe waren*. Diese Info ist *sehr nützlich für die Verteidigung:*

Strategie der Verteidigung

Die Verteidigung bittet den Inspektor zu sagen, welche drei Verdächtigten die Einbrecher seien. Dann lässt die Verteidigung die Wohnungen dieser drei durchsuchen. Nehmen wir an, an allen drei Orten werden gestohlene Artikel gefunden:

Da es genau drei Diebe gibt, wissen wir nun, dass *alle anderen* Verdächtigten *keine* Diebe waren:

13-12

Von Anwälten und Inspektoren zu NL-Maschinen.

Sei d eine feste Zahl.

- Die *Verdächtigten* entsprechen *Knoten.*
- Die *Diebe* sind die Knoten in R_d.
- Die *Unschuldigen* sind die Knoten in $V - R_d$.
- Der *Inspektor* kann die Richterin von $v \in R_d$ *überzeugen,* indem er einen *Pfad von s nach v der Länge maximal d* angibt.

– Die *Verteidigung* kann die Richterin von $v \notin R_d$ *überzeugen,* indem sie Pfade von s zu genau $|R_d|$ unterschiedlichen Knoten (die alle nicht v sind) angibt, die alle maximale Länge d haben.

Die Strategie der Verteidigung als Satz.

13-13

▶ **Definition**

Sei M eine Turingmaschine mit Eingabeband, Ausgabeband und einem Wahlband. Wir sagen, die Ausgabe von M *ist einstimmig* y, falls

1. für mindestens einen initialen Inhalte des Wahlbandes hält M bei Eingabe x in einem akzeptierenden Zustand an und y ist der Inhalt des Ausgabebands und
2. für jeden anderen initialen Inhalt des Wahlbandes gilt: hält M bei Eingabe x in einem akzeptierenden Zustand an, so ist y der Inhalt des Ausgabebandes.

Mit anderen Worten: Alle nichtdeterministischen Berechnungen von M brechen entweder ab oder geben das gleiche y aus (und mindestens eine Berechnung bricht nicht ab).

▶ **Satz**

Es gibt eine Logspace-Turingmaschine M mit Wahlband, so dass für alle Eingaben (G,s,t,r,d) gilt:

1. *Falls $r = |R_d|$ und $t \in R_d$, so gibt M einstimmt »Dieb« aus.*
2. *Falls $r = |R_d|$ und $t \notin R_d$, so gibt M einstimmig »unschuldig« aus.*

Die Maschine heiße »checkReachable«.

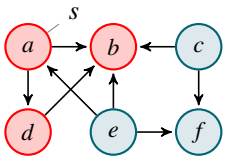
Beweis. Die Maschine funktioniert wie folgt: Das erste Symbol des Wahlbandes sagt der Maschine, ob ein Beweis von »$t \in R_d$« oder von »$t \notin R_d$« folgt.

1. Im ersten Fall überprüft M nun, ob der Rest des Wahlbandes ein Pfad von s nach t der Länge höchstens d ist und gibt »Dieb« aus, wenn dies der Fall ist.
2. Im zweiten Fall überprüft M, ob nun genau r Blöcke der folgenden Bauart folgen: Jeder Block ist ein Pfad der Länge maximal d von s zu einem Knoten, der nicht t ist und der eine höhere Nummer hat als der Endknoten des vorigen Pfades. Ist dies der Fall, so gibt M nun »unschuldig« aus.

Man beachte, dass es in beiden Fälle ($t \in R_d$ und $t \notin R_d$) jeweils einen Inhalt für das Wahlband gibt, so dass eine Ausgabe gemacht wird und dass die Ausgabe einstimmig ist. □

Die Wahlbandinhalte an einem Beispiel

13-14



Beispiel: Beweis, dass b ein Dieb ist

Der folgende Inhalt des Wahlbandes sorgt dafür, dass M bei Eingabe $(G,a,b,3,2)$ »Dieb« ausgibt:

```
Dieb b via a,d,b
```

Beispiel: Beweis, dass f unschuldig ist

Der folgende Inhalt des Wahlbandes sorgt dafür, dass M bei Eingabe $(G,a,f,3,2)$ »unschuldig« ausgibt:

```
1. Dieb a
2. Dieb b via a,d,b
3. Dieb d via a,d
```

13.1.3 Die Richterin induktiv überzeugen

13-15

Wo wir stehen

Unser Ziel ist weiterhin, UNREACH $\in$ NL zu zeigen. Nehmen wir nun an, bei Eingabe (G,s,t) könnten wir $r = |R_\infty|$ berechnen. Dann können wir durch eine NL-Maschine genau die (G,s,t) *akzeptieren*, für die $t \notin R_\infty$ gilt. (Wenn nämlich die Maschine checkReachable »Dieb« ausgibt, so verwerfen wir; gibt sie »unschuldig« aus, so akzeptieren wird. Eines von beiden passiert immer.)
Was jetzt noch geschehen muss:

- Wir müssen offensichtlich noch r ausrechnen.
- Dabei ist es aber nicht nötig, r deterministisch zu berechnen. Vielmehr reicht es doch, r einstimmig durch eine Maschine M bei Eingabe (G,s,t) berechnen zu lassen.

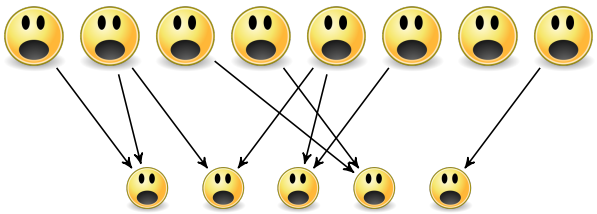
Neues Ziel

Finde eine Logspace-Turingmaschine mit Wahlband, die bei Eingabe (G,s,t) einstimmig $|R_\infty|$ ausgibt.

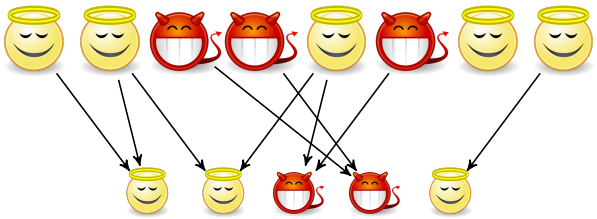
13-16

Sippenhaft: Die Kinder der Diebe.

Was bekannt ist: Die Verdächtigen haben Kinder (etwas patchworkfamilienhaft)



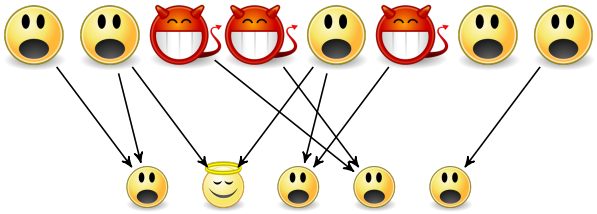
Was nicht bekannt ist: Welche Kinder haben mindestens einen Dieb als Eltern?



13-17

Wie man zeigen kann, dass ein Kind keine Diebe als Eltern hat.

Wie kann man die Richterin überzeugen, dass der zweite kleine Knoten *kein Kind eines Diebes ist*? Man beweist, dass alle »Eltern« des zweiten kleinen Knotens keine Diebe sind. Dafür zeigt man (wie vorher), dass *drei andere* Knoten Diebe sind.



13-18

Die Sippenhaft als Satz.

▶ Satz

Es gibt eine Logspace-Turingmaschine M mit Wahlband, die bei Eingabe (G,s,v,r,d) folgendes leistet:

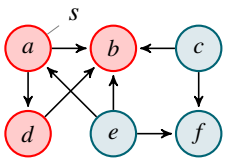
- *Falls $r = |R_d|$ und $v \in R_{d+1}$, so gibt M einstimmig aus »$v \in R_{d+1}$«.*
- *Falls $r = |R_d|$ und $v \notin R_{d+1}$, so gibt M einstimmig aus »$v \notin R_{d+1}$«.*

Beweis. Wieder entscheidet sagt erste Symbol des Wahlbandes, welchen der beiden Fälle oben die Maschine M überprüfen soll.

1. Im ersten Fall erwartet M auf dem Band einen Pfad von s nach v der Länge maximal $d+1$ und gibt »$v \in R_{d+1}$« aus, wenn sie einen solchen vorfindet.
2. Im zweiten Fall iteriert M über alle Vorgänger von v (also Knoten u, für die (u,v) eine Kante in G ist). Für jedes u überprüfe: Auf dem Wahlband kommen r Blöcke, von denen jeder ein Pfad der Länge maximal d zu einem Knoten mit größerer Nummer als im vorherigen Block ist und keiner u ist. Wenn für jedes u solche r Blöcke folgen, so gibt M nun »$v \notin R_{d+1}$« aus. Dies ist korrekt, da alle Vorgänger von v Entfernung mehr als d von s haben und v somit Entfernung mehr als $d+1$ haben muss. □

Beispielinhalt des Wahlbandes für den zweiten Fall

13-19



Beispiel: Beweis, dass $f \notin R_3$

Der folgende Inhalt des Wahlbandes »beweist« für die Eingabe $(G,a,f,3,2)$, dass $f \notin R_2$ gilt:

```
Vorgänger c ist unschuldig, da
 1. a ist ein Dieb
 2. b ist ein Dieb via a,d,b
 3. d ist ein Dieb via a,d
Vorgänger e ist unschuldig, da
 1. a ist ein Dieb
 2. b ist ein Dieb via a,b
 3. d ist ein Dieb via a,d
```

Der induktive Schritt.

13-20

Wir wollen immernoch $r = |R_\infty|$ bestimmen. Der Trick ist nun, dass wir nun für $d = 1,2,3,\ldots,|V|$ die Größen $|R_i|$ *induktiv* berechnen können. Mit unserem letzten Satz konnten wir nämlich schon für jeden Knoten v einstimmig bestimmen, ob $v \in R_{d+1}$ oder $v \notin R_{d+1}$ gilt, *wenn wir* $|R_d|$ *kennen.* Dann ist es aber für jedes d leicht, mit einer weiteren Schleife über alle Knoten in V zu iterieren und dabei zu *zählen*, wie viele davon in R_{d+1} liegen. Diese Anzahl ist dann gerade $|R_{d+1}|$.

Der induktive Schritt als Satz.

13-21

▶ **Satz**

Es gibt eine Logspace-Turingmaschine M mit Wahlband, die bei Eingabe (G,s,r,d) mit $r = |R_d|$ einstimmig $|R_{d+1}|$ ausgibt.

Beweis. In einer Schleife iteriert M über alle $v \in V$ und liest immer vom Wahlband einen Beweis für $v \in R_{d+1}$ oder für $v \notin R_{d+1}$. Am Ende wird die Anzahl der Knoten ausgegeben, für die ersteres der Fall war. □

```
1  function computeNextR(G,s,r,d) : integer
2    r' ← 0
3    foreach v ∈ V do
4      a ← next symbol from the choice tape
5      if a = in then
6        // the choice tape claims that v ∈ R_{d+1}. Verify this!
7        a ← next symbols from the choice tape
8        if a is a path from s to v of length at most d+1 then
9          r' ← r' + 1
10       else
11         reject // the choice tape contents is flawed
12     else
13       // the choice tape claims that v ∉ R_{d+1}. Verify this!
14       foreach u ∈ {u | (u,v) ∈ E} do
15         // Verify that u ∉ R_d
```

```
16          for i ← 1 to r do
17              a ← next symbols from the choice tape
18              if a is no path to a new node other than u then
19                  reject // choice tape contents is flawed
20  output r′
```

13.1.4 Das induktive Zählen

13-22

Der komplette Algorithmus.
Wir können nun aus $|R_d|$ die Zahl $|R_{d+1}|$ errechnen. In einer Schleife können wir dann, mit $|R_0| = 1$ beginnend, auch $R_\infty = R_{|V|}$ berechnen. Wir haben am Anfang schon gesehen, dass wir dann entscheiden können, ob ein Knoten von s aus erreichbar ist oder nicht.

Algorithmus für Nichterreichbarkeit

```
1  r ← 1 // Nur s ist von s in 0 Schritten erreichbar
2
3  foreach d ← 0 to n do
4      r ← computeNextR(G, s, r, d)
5
6  if checkReachable(G, s, t, r) = unschuldig then
7      accept
8  else
9      reject
```

13.2 Konsequenzen für . . .

13.2.1 NL

13-23

Nichtdeterministischer Platz ist unter Komplementbildung abgeschlossen.

▶ **Satz**
$\mathrm{NL} = \mathrm{coNL}$

Beweis.

1. Da REACH vollständig ist für NL, ist sein Komplement UNREACH vollständig für coNL, der Klasse aller Komplemente von Sprachen in NL.
2. Nach dem Satz von Immerman-Szelepcsényi ist $\mathrm{UNREACH} \in \mathrm{NL}$.
3. Hieraus folgt $\mathrm{coNL} \subseteq \mathrm{NL}$ und aus Symmetriegründen auch $\mathrm{NL} \subseteq \mathrm{coNL}$. □

▶ **Folgerung**
Die Logarithmische Hierarchie kollabiert auf die erste Stufe:

$$\mathrm{NL} \cup \mathrm{NL}^{\mathrm{NL}} \cup \mathrm{NL}^{\mathrm{NL}^{\mathrm{NL}}} \cup \cdots = \mathrm{NL}.$$

Beweis. Siehe Übung 13.1. □

13.2.2 Kontextsensitive Sprachen

Die kontextsensitiven Sprachen sind unter Komplementbildung abgeschlossen. 13-24

▶ Satz
CSL = coCSL.

Beweis. Man kann zeigen, dass CSL = NLINSPACE = NSPACE($O(n)$). Mit dem Padding-Lemma zeigt man nun coNSPACE($O(n)$) $\subseteq$ NSPACE($O(n)$) wie folgt:

1. Sei $A \in$ coNSPACE($O(n)$).
2. Dann gilt pad$_f(A) \in$ coNSPACE($O(\log n)$) nach dem Padding-Lemma für $g(n) = \log_2 n$ und $f(n) = 2^n - n$.
3. Dann gilt pad$_f(A) \in$ NSPACE($O(\log n)$) nach dem Satz von Immerman-Szelepcsényi.
4. Dann gilt $A \in$ NSPACE($O(n)$) nach dem Padding-Lemma. □

Zusammenfassung dieses Kapitels

▶ Satz von Immerman-Szelepcsényi 13-25
UNREACH $\in$ NL.

▶ Auch Satz von Immerman-Szelepcsényi
NL = coNL.

▶ Induktives Zählen
Der Beweis benutzt *induktives Zählen*, bei dem die Größe $|R_{d+1}|$ einstimmig errechnet wird aus dem Wert $|R_d|$.
Um hierbei zu zeigen, dass ein Knoten *nicht* erreichbar ist in d Schritten, gibt man $|R_d|$ Pfade zu *anderen* Knoten der Entfernung maximal d an.

▶ Folgerung
Das Komplement einer kontextsensitiven Sprachen ist kontextsensitiv.

Zum Weiterlesen

[1] Neil Immerman, Nondeterministic Space Is Closed under Complementation, *SIAM Journal of Computing*, 17(5):935–938, 1988
[2] Robert Szelepcsényi, The Method of Forced Enumeration for Nondeterministic Automata, *Acta Informatica*, 23:279–284, 1988

In diesen beiden Artikeln, die unabhängig voneinander veröffentlicht wurden, zeigen Immerman und Szelepcsényi beide den Beweis.

Übungen zu diesem Kapitel

Übung 13.1 Die logarithmische Hierarchie kollabiert, schwer
Zeigen Sie, dass die logarithmische Hierarchie kollabiert, dass also

$$\mathrm{NL} \cup \mathrm{NL}^{\mathrm{NL}} \cup \mathrm{NL}^{\mathrm{NL}^{\mathrm{NL}}} \cup \cdots = \mathrm{NL}.$$

Zur Erinnerung: Eine nichtdeterministische Orakelmaschine darf nur dann nichtdeterministische Entscheidungen fällen (etwas von ihrem Wahlband lesen), wenn das Orakelband leer ist.
Tipp: Es reicht, $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$ zu zeigen. Zeigen Sie dazu, wie eine NL-Maschine eine Orakelanfrage an eine andere NL-Maschine simulieren kann. Der schwierige Fall tritt ein, wenn die Antwort an das Orakel negativ ist. Nutzen Sie hierfür den Satz von Immerman-Szelepcsényi.

Übung 13.2 Erreichbarkeitsinvertierende Graphmodifikation, leicht
Zeigen Sie, dass es eine Funktion $f \in$ FL gibt, die jeden gerichteten Graphen G zusammen mit zwei Knoten s und t abbildet auf einen neuen Graphen G' und zwei Knoten s' und t', so dass es genau dann einen Pfad von s nach t in G gibt, wenn es keinen Pfad von s' nach t' in G' gibt.

Übung 13.3 Zusammenhand versus Unzusammenhang, mittel

Zeigen Sie, dass es eine Funktion $f \in \mathrm{FL}$ gibt, die jeden gerichteten Graph G auf einen Graphen G' abbildet, so dass G genau dann stark zusammenhängend ist, wenn G' es nicht ist.

Tipp: Zeigen Sie, dass CONNECTEDNESS vollständig für NL ist und nutzen Sie dann die Funktion aus Übung 13.2.

Für die nächsten Übungen brauchen wir eine Definition

▶ **Definition: Eindeutiger logarithmischer Platz (unambiguous logspace)**
Sei M eine Turingmaschine mit Wahlband, die eine Sprache $L = L(M)$ akzeptiert. Die Maschine heißt *eindeutig (unambiguous),* wenn für jede Einhabe *höchstens ein Wahlbandinhalt existiert*, der M akzeptieren lässt. (Mit anderen Worten: Für $x \notin L$ gibt es keinen akzeptierenenden Pfad und für $x \in L$ genau einen.)
Die Klasse UL enthält alle Sprachen, die von einer eindeutigen logarithmisch platzbeschränkten Turingmaschine entschieden werden.

Offenbar gilt $\mathrm{L} \subseteq \mathrm{UL} \subseteq \mathrm{NL}$.

▶ **Definition**
Ein gerichteter Graph $G = (V, E)$ hat die *Eindeutiger-kürzester-Pfad-Eigenschaft,* wenn für alle $u, v \in V$, für die überhaupt ein Pfad von u nach v existiert, es genau einen kürzesten Pfad von u nach v gibt.

Übung 13.4 Bäume haben die Eindeutiger-kürzester-Pfad-Eigenschaft, einfach

Zeigen Sie, dass Bäume die Eindeutiger-kürzester-Pfad-Eigenschaft haben.

Übung 13.5 Bestimmte Kreise haben die Eindeutiger-kürzester-Pfad-Eigenschaft, einfach

Zeigen Sie, dass Kreise ungerader Länge die Eindeutiger-kürzester-Pfad-Eigenschaft haben, Kreise gerader Länge hingegen nicht.

Übung 13.6 Doppeltes Zählen, schwer

Sei G ein Graph, der die Eindeutiger-kürzester-Pfad-Eigenschaft hat. Sei s ein Knoten im Graph. Nehmen wir an, wir kennen

1. die Zahl $r = |R_\infty|$ an von s aus erreichbaren Knoten und
2. auch die Zahl $S = \sum_{v \in R_\infty} \mathrm{d}(s, v)$ (die Summe der Distanzen von s zu allen von s aus erreichbaren Knoten).

Geben Sie nun eine logarithmisch platzbeschränkte Turingmaschine an, die für jeden Knoten t *eindeutig* entweder

1. $t \notin R_\infty$ zeigt oder
2. $t \in R_\infty$ zeigt. (Je nachdem, was nun gilt.)

Tipp: Um $t \notin R_\infty$ zu zeigen, iterieren Sie über alle $v \in R_\infty$ und lesen vom Wahlband den kürzesten Pfad von s nach v. Man zählt nun nicht nur die Anzahl dieser Knoten, sondern berechnet auch gleich die Summe der Längen dieser Pfade. Um zu garantieren, dass es einen *eindeutigen* Wahlbandinhalt gibt, der $t \notin R_\infty$ zeigt, muss die Summe der gelesenen Pfade genau S sein.
Tipp: Für den Beweis von $t \in R_\infty$ reicht es nicht, einen Pfad zu raten, da es viele Pfade geben kann. Stattdessen benutze man denselben Algorithmus wie oben, nur mit einer anderen Akzeptanzbedingung.

Übung 13.7 Doppeltinduktives Zählen, schwer

Zeigen Sie, dass das Erreichbarkeitsproblem eingeschränkt auf Graphen mit der Eindeutiger-kürzester-Pfad-Eigenschaft von eindeutigen Logspace-Maschinen gelöst werden kann.

Hint: Wenden Sie die Methode des doppelten Zählens induktiv wie im Satz von Immerman-Szelepcsényi an.

Bemerkung: Man kann zeigen, dass man *planaren* Graphen immer so modifizieren kann, dass sie die Eindeutiger-kürzester-Pfad-Eigenschaft haben ohne die Erreichbarkeitsrelation zu ändern. Hieraus folgt, dass PLANAR-REACH $\in$ UL gilt. Es ist keine bessere Schranke bekannt.

Teil VIII

Der Komplexitätszoo

Was gibt es schöneres, als eine Vorlesung mit einem Zoo-Besuch zu beenden? In den Kapiteln dieses Skriptes haben wir schon viele der merkwürdigen Spezies kennen gelernt, die den Zoo bevölkern, wir haben sie in ihrem natürlichen Habitat studiert und gesehen, wie sie sich in freier Wildbahn verhalten. Nun wird es Zeit, die vielen Probleme und Klassen in etwas »geordneter« Weise zu besichtigen.

In einem Dutzend Kapiteln ist es nicht möglich, alle Resultate der Komplexitätstheorie aufzuzeigen, ja überhaupt sie nur zu nennen. Auf Ihrer Tour wird Ihr Zooführer auf eine Reihe von Resultaten hindeuten, von denen Sie vorher noch nicht gehört haben. Nichtsdestotrotz kann ich Ihnen versichern, dass Sie am Ausgang des Zoos nicht einmal die Hälfte dessen gesehen haben werden, was es alles zu entdecken gibt. Daher: Kommen Sie bald wieder!

14-1

Kapitel 14
Der Komplexitätszoo

»Mama, schau mal, eine Baby-Polynomialzeitklasse«

14-2

Lernziele dieses Kapitels

1. Den Aufbau des Komplexitätszoos kennen
2. Probleme in die Klassen einordnen können

Inhalte dieses Kapitels

Worum es heute geht

Die systematische Untersuchung dessen, was Computer können und was nicht, begann mit Alan Turings Artikel über das Halteproblem. Zu einer Zeit, da man Computer noch gar nicht bauen konnte, hat Turing bereits gezeigt, dass für bestimmte Probleme (eben das Haltproblem) immer gelten wird »*does not compute*«. Dies was gleichzeitig die erste *Klassifikation* eines Problems: Das Halteproblem ist kein Element der Klasse der rekursiven Sprachen.

Die urzeitliche Komplexitätswelt des Jahres 1936 war allerdings noch wüst und leer. Es gab eine, vielleicht zwei Klassen: die rekursiven Sprachen und vielleicht noch die primitivrekursiven Sprachen. Die Theoretikerinnen und Theoretiker hatten dann knapp achzig Jahren Zeit (und nicht nur sieben Tage), diese Welt reichlich zu bevölkern. Hunderte Klassen und Tausende Probleme bevölkern das, was man allgemein den *Komplexitätszoo* nennt.

In den vorigen Kapiteln haben wir bereits viele der Hauptattraktionen des Zoos bestaunen können. Bei manchen von ihnen, wie NP oder SAT, muss man Schlange stehen, so viele Leute drängen sich um sie. Wie im echten Zoo verpasst aber ziemlich viel, wer sich immer nur die Raubtiere und die Elefanten anschaut; schauen Sie sich doch mal eine Ausstellung über Insekten an! Genauso haben NL oder SUCCINCT-PARITY sehr spannende Aspekte, die sich lohnen, genauer zu betrachtet werden.

Im diesem abschließenden Kapitel möchte ich Ihnen eine geführte Tour durch den Zoo geben. Die Tour wird mit kleinen Tierchen beginnen wie L, NL oder AC^0. Wir kommen dann zu den Arbeitstieren, die uns in der Praxis ständig begegnen: P, NC und Probleme wie Suchen oder Sortieren. Am Ende geht es dann zu den großen Exemplaren (bei Kindern immer sehr beliebt) wie NP und PSPACE. Den Leviathan der Komplexitätstheorie, EXP, heben wir uns für den Schluss auf.

14.1 Tour-Start

14.1.1 Kleine Klassen

Der Start der Tour durch den Zoo.

14-4

Wir beginnen unsere Tour durch den Zoo mit den kleinsten Tieren: den »kleinen Klassen« und »einfachen Problemen«.
Die folgenden Klassen enthalten Sprache, die gelöst werden können von…

- NC^i Schaltkreisen der Tiefe $O(\log^i n)$, polynomieller Größe und konstantem Fan-In.
- AC^i Schaltkreisen der Tiefe $O(\log^i n)$, polynomieller Größe und beliebigem Fan-In.

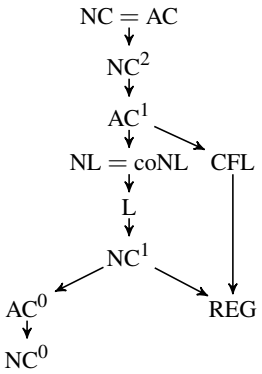
$\mathrm{SPACE}[O(1)]$ Turingmaschinen, die konstanten Platz benötigen (endliche Automaten).

$\mathrm{SPACE}[O(\log^i n)]$ Turingmaschinen, die (poly)logarithmischen Platz brauchen.

$\mathrm{NSPACE}[O(\log^i n)]$ nichtdeterministische Turingmaschinen, die (poly)logarithmischen Platz brauchen.

Inklusionsbeziehung zwischen den kleinen Klassen.

14-5



14.1.2 Einfache Probleme

Einige »einfache« Probleme

14-6

- PARITY Bitstrings, die eine ungerade Anzahl an Einsen enthalten.
- PALINDROMES Palindrome.
- DOUBLE Wörter der Form ww.
- REACH Das gerichtete Erreichbarkeitsproblem.
- UREACH Das ungerichtete Erreichbarkeitsproblem.
- UNREACH Das gerichtete Nicht-Erreichbarkeitsproblem.
- CONNECTED Die starkzusammenhängenden Graphen.
- DISTANCE Das Entfernungsproblem für gerichtete Graphen.
- UDISTANCE Das Entfernungsproblem für ungerichtete Graphen.
- TREE Die ungerichteten, azyklischen, zusammenhängenden Graphen.
- 2-COLORABLE Die 2-färbbaren (=bipartiten) Graphen.

✎ Zur Übung
Für jedes Problem, finden Sie die kleinste Klasse, in die es gehört.

14.1.3 Ausblicke

14-7

Ausblick 1: Logspace-Approximierbarkeit.
Genau wie für polynomielle Zeit kann man eine Theorie der *Approximierbarkeit* für die Welt der kleinen Platzklassen entwickeln. Es ist aber zu beachten, dass diese Theorie vom praktischen Standpunkt weniger wichtig ist, da alle im Folgenden betrachteten Problem in NL liegen und damit sowieso recht »einfach« sind.

Beispiel

- Das Problem TOURNAMENT-DISTANCE ist vollständig für NL.
- Es gibt aber ein *Logspace-Approximationsschema* für die Optimierungsvariante.
- Das Problem TOURNAMENT-REACH liegt in AC^0.

Hingegen gibt es kein Logspace-Approximationsschema für DISTANCE, es sei denn, L = NL.

Offenes Problem
Gibt es ein Logspace-Approximationsschema für UDISTANCE?

14-8

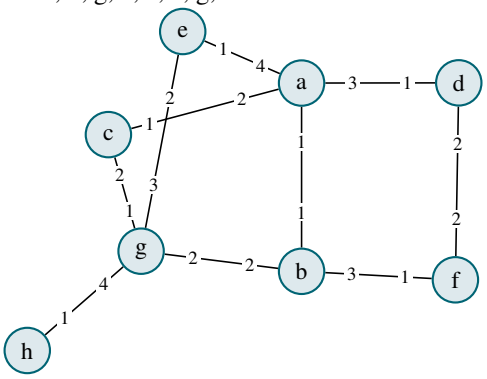
Ausblick 2: Universelle Traversierungsfolgen.

▶ **Definition:** Traversierungsfolge
Eine *Grad-d-Traversierungsfolge* ist eine Folge von Zahlen aus $\{1,\dots,d\}$.
Für eine gegebene Folge f, einen Graphen G vom Maximalgrad d und einen Startknoten s werden die folgenden Knoten *von t besucht:*

- Zunächst gilt natürlich $v_1 = s$ als besucht..
- Dann wird der $f[1]$-te Nachbar v_2 von v_1 besucht.
- Dann wird der $f[2]$-te Nachbar v_3 von v_2 besucht.
- Und so weiter.

Dabei sind die Nachbarn aller Knoten in einer festen Weise durchnummeriert. Hat ein Knoten weniger Nachbarn als die Zahl $f[i]$, so ist einfach $v_{i+1} = v_i$.

Beispiel
Die *Grad-4-Traversierungsfolge* $1,2,3,4,4,3,2,1$ besucht beim Knoten a startend die Knoten a, b, g, e, e, e, g, c.



▶ **Definition:** Universelle Traversierungsfolge
Eine Traversierungsfolge f heißt *universell* für eine Klasse C von Graphen, wenn für jedes $G \in C$ die Folge alle Knoten in G besucht.

Man kann sich eine universelle Traversierungsfolge als den »ultimativen Touristenführer« vorstellen: Es enthält eine Folge von Anweisungen der Form »geradeaus gehen« oder »scharf links drehen«, so dass man damit vom *jedem* Startpunkt zum eigenen Hotel kommt in *jeder beliebigen (!) Stadt*.

▶ **Satz**
Es gibt eine Funktion $f \in$ FL, die für jedes n das Wort 1^n auf eine universelle Traversierungsfolge für die Klasse der ungerichteten zusammenhängenden Graphen auf n Knoten abbildet.

▶ **Folgerung:** Satz von Reingold
UREACH $\in$ L.

Ausblick 3: Schwere des Graphisomorphieproblems. 14-9

Das Graphisomorphieproblem wird allgemein als schwierig angesehen. Man geht zwar nicht davon aus, dass es NP-vollständig ist (dann würde die Polynomielle Hierarchie kollabieren), jedoch hat es auch noch niemand geschafft, einen Polynomialzeitalgorithmus zu finden. Erstaunlicherweise ist die *(fast) beste untere Schranke* für dieses Problem die folgende:

▶ Satz
GI *ist schwer für* NL.

Dieser Satz ist erstaunlich schwierig zu beweisen, da es sich als ausgesprochen schwer herausstellt, algorithmische Probleme als Graphisomorphieprobleme zu kodieren.

14.2 Die Hauptattraktionen

14.2.1 Klassen von lösbaren Problemen

Die Klassen der lösbaren Probleme. 14-10

- NC Probleme, die parallel in polylogarithmischer Zeit gelöst werden können.
- P Probleme, die in polynomieller Zeit gelöst werden können.
- BPP Probleme, die in randomisierter polynomieller Zeit gelöst werden können.
- QBP Probleme, die in polynomieller Zeit gelöst werden können von einem Quantencomputer.

Die Klasse BPP ist wie folgt definiert:

▶ Definition: Polynomielle Zeit mit beschränkter Fehlerwahrscheinlichkeit
Eine Sprache A ist in BPP, wenn eine Polynomialzeit-Turingmaschine existiert mit einem Wahlband, so dass:

- Für $x \in A$ akzeptiert A für mindestens zwei Drittel aller möglichen Inhalte des Wahlbandes.
- Für $x \in A$ verwirft A für mindestens zwei Drittel aller möglichen Inhalte des Wahlbandes.

Verhältnis der Klassen. 14-11

14.2.2 Gut lösbare Probleme

Wichtige »gut lösbare« Probleme 14-12

- PRIMES Sprache der (Codes von) Primzahlen.
- FACTORIZE Entscheidungsvariante des Faktorisierungsproblems.
- CVP Das Schaltkreisauswertungsproblem.
- MON-CVP Das monotone Schaltkreisauswertungsproblem.
- INVERT Entscheidungsvariante des Matrixinvertierungproblems.
- LP Entscheidungsvariante des Lösens von linearen Gleichungssystemen.

(Bei den »Entscheidungsvarianten« ist eine Bitposition Teil der Eingabe und die Frage ist, ob die beschriebene Funktion an dieser Stelle eine 1 ausgibt.)

✎ Zur Übung
Für jedes Problem, finden Sie die kleinste Klasse, in die es gehört.

14.2.3 Ausblicke

14-13

Ausblick 1: Derandomisierung

Randomisierte Maschinen nutzen (per Definition) ihre Wahlbänder als Quellen von *Zufallsbits*. Man kann dabei leicht sicherstellen, dass die Maschinen für *die allermeisten Zufallsstrings* »korrekt« arbeiten, also Wörter in der Sprache akzeptieren und alle anderen verwerfen.

Ziel der Derandomisierungstheorie

Modifiziere eine randomisierte Maschine so, dass sie nicht nur für »die allermeisten« sondern für »alle« Zufallsstrings korrekt arbeitet.

Vermutung

$\mathrm{P} = \mathrm{BPP}$.

14-14

Warum allgemeine Derandomisierung möglich erscheint.

Auch wenn nicht bekannt ist, ob BPP derandomisiert werden kann, scheint folgender Algorithmus für alle Sprachen $A \in \mathrm{BPP}$ via einer Maschine M vielversprechend:

```
1  input x
2  c ← 0
3  for i ← 1 to |x|^O(1) do
4     r ← pseudoRandomNumber(i)
5     run M on x and the string r on the choice tape
6     if M accepts x with this choice tape then
7        c ← c + 1
8     else
9        c ← c − 1
10 if c > 0 then
11    accept
12 else
13    reject
```

Das Programm macht polynomiell viele *Stichproben* an pseudozufälligen Stellen. Daraufhin entscheidet es sich zwischen »akzeptieren« und »verwerfen« aufgrund einer *Mehrheitsentscheidung*. Die Ausgabe wäre für *wirklich zufällige Stichproben* immer korrekt. Nehmen wir nun an, das Programm »funktioniert nicht«. Dann gilt:

- Der Zufallszahlengenerator produziert nicht wirklich zufällige Zahlen.
- Probieren wir also einen anderen.
- Wenn der auch nicht funktioniert, noch einen anderen.
- Wenn der auch nicht, dann noch einen anderen; und so fort.

Falls *kein Pseudozufallszahlengenerator* funktioniert, so wissen wir, dass *man keine wirklich guten Zufallszahlen erzeugen kann*.

▶ Satz

Genau eine der folgenden Aussagen ist wahr:

1. $\mathrm{P} = \mathrm{BPP}$.
2. *Es gibt keine guten Pseudozufallszahlengeneratoren.*

14-15

Ausblick 2: Universelle Zufallsstrings

Zur Erinnerung: Randomisierte Maschinen verhalten sich für die allermeisten Zufallsstrings korrekt (akzeptieren also Wörter in der Sprache für die allermeisten und verwerfen alle anderen Wörter für die allermeisten). Hieraus folgt natürlich, dass bestimmte Inhalte des Wahlbandes bei »vielen Wörtern« dafür sorgen, dass sich die Maschine korrekt verhält.

▶ Definition: Universelle Zufallsstrings

Sei M eine Turingmaschinen mit Wahlband und sei A eine Sprache. Ein Wahlbandinhalt $c \in \{0,1\}^m$ heißt *universell* für eine Wortlänge n und die Maschine M, wenn für alle $x \in \Sigma^n$ gilt:

- Wenn $x \in A$, so M akzeptieren x für den Wahlbandinhalt c.
- Wenn $x \notin A$, so M verwirft x für den Wahlbandinhalt c.

Mit anderen Worten: Ein universeller Zufallsstring ist ein einziger String, der denselben »Effekt« hat wie die »Mehrheit aller wirklich zufälligen Strings«.

Universelle Zufallsstrings existieren. 14-16

▶ Satz
Sei $A \in$ BPP via M. Dann existiert für jede Wortlänge n ein universeller Zufallsstring für n und M.

▶ Folgerung
BPP $\subseteq$ POLYSIZE. *(Nichtuniforme Schaltkreise polynomieller Größe.)*

Beweis. Eine BPP-Maschine ist eine Polynomialzeitmaschine, die wir also mit polynomiell großen Schaltkreisen simulieren können, wenn wir den Inhalt des Wahlbandes festlegen. Da unsere Schaltkreisfamilie nicht uniform sein muss, können wir einfach für jede Wortlänge den universellen Zufallsstring »fest verdrahten«. □

▶ Folgerung
Lässt sich SAT *in randomisierter polynomieller Zeit entscheiden (*SAT $\in$ BPP*), so kollabiert die Polynomielle Hierarchie.*

Beweis. Aus SAT $\in$ BPP folgt NP $\subseteq$ BPP $\subseteq$ POLYSIZE, woraus SAT $\in$ POLYSIZE folgt. □

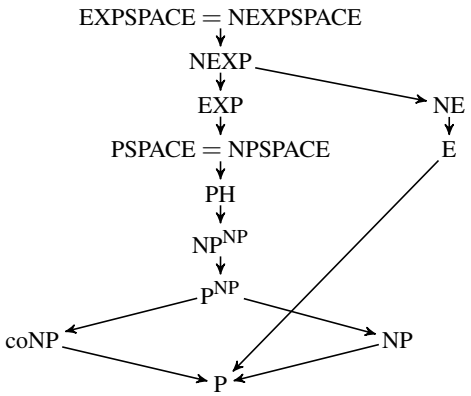
14.3 Die Leviathane

14.3.1 Große Klassen

Klassen schwieriger Probleme. 14-17

NP	Nichtdeterministische polynomielle Zeit
coNP	Komplemente von Sprachen in NP (*nicht* das Komplement von NP)
P^{NP}	Erste Δ-Stufe der polynomiellen Hierarchie
PH	Die Polynomielle Hierarchie
PSPACE	Polynomieller Platz
E	Linear-exponentielle Zeit
NE	Nichtdeterministische linear-exponentielle Zeit
EXP	polynomial-exponentielle Zeit
NEXP	Nichtdeterministische polynomiell-exponentielle Zeit
EXPSPACE	Exponentieller Platz

Beziehungen zwischen den großen Klassen. 14-18



14.3.2 Schwere Probleme

14-19

Schwere Probleme

SAT Das aussagenlogische Erfüllbarkeitsproblem.

TAUT Das aussagenlogische Tautologieproblem.

QBF Das aussagenlogische Erfüllbarkeitsproblem mit Quantoren.

SUCCINT-PARITY Knappes Paritätsproblem.

SUCCINT-REACH Knappes Erreichbarkeitsproblem.

SUCCINT-SAT Knappes aussagenlogisches Erfüllbarkeitsproblem.

CLIQUE Hat ein Graph eine Clique der Größe k?

EXACT-CLIQUE Hat die größte Clique eines Graphen die Größe k?

TSP Hat der Graph eine Rundreise vom Gewicht höchstens k?

EXACT-TSP Hat die kürzeste Rundreise in einem Graphen ein Gewicht von genau k?

✎ Zur Übung

Für jedes Problem, finden Sie die kleinste Klasse, in die es gehört.

14.3.3 Ausblicke

14-20

Ausblick: Extrem eingeschränkte Prover-Verifier-Protokolle

NP lässt sich mittels *Prover-Verifier-Protokollen* definieren: Hierbei ist Prover *allmächtig*, wohingegen Verifier sich in *polynomieller Zeit* entscheiden muss. Man kann nun überlegen, ob man Verifier den Zugriff auf die Zertifikate *einschränken kann*:

▶ Definition

Eine Sprache A ist in $\big(r(n), q(n)\big)$-PCP, wenn ein Prover-Verifier-Protokoll für A existiert, wobei Verifier für ein Wort x und ein Zertifikat c wie folgt arbeitet:

1. Verifier liest $r(n)$ Zufallsbits von einem Wahlband.
2. Verifier errechnet nur aufgrund von x und den Zufallsbits Bitpositionen $p_1, \ldots, p_{q(n)}$.
3. Verifier entscheidet dann in polynomieller Zeit, ob $x \in A$ gilt, wobei er neben x lediglich die Bits $c[p_1], \ldots, c[p_{q(n)}]$ kennt.

Der Verifier muss bei $x \notin A$ immer verwerfen und bei $x \in A$ für einen festen Anteil (wie »die Hälfte«) aller Wahlbandinhalte akzeptieren.

14-21

Einfache Beobachtungen

▶ Lemma: Kein Zugriff auf Zertifikate

1. $(0,0)$-PCP $=$ P.
2. $(O(\log n), 0)$-PCP $=$ P.
3. $(n^{O(1)}, 0)$-PCP $=$ RP $\subseteq$ BPP.

▶ Lemma: Kein Zufall

1. $(0,0)$-PCP $=$ P.
2. $(0, O(\log n))$-PCP $=$ P.
3. $(0, n^{O(1)})$-PCP $=$ NP.

▶ Lemma: Wenig Zugriff auf das Zertifikat

1. $(O(\log n), 1)$-PCP $=$ P.
2. $(O(\log n), 2)$-PCP $=$ P.

14-22

Das PCP-Theorem

▶ Satz

$(O(\log n), 3)$-PCP $=$ NP.

Der Beweis gehört zu den komplexesten in der gesamten Komplexitätstheorie. Entscheidend an diesem Satz ist, dass er viele Anwendungen in der *Nichtapproximierbarkeitstheorie* hat. Beispielsweise folgt aus ihm, dass Clique sich nicht besser als mit Faktor $n^{1-\varepsilon}$ approximieren lässt.

Zusammenfassung dieses Kapitels

1. Die große Anzahl an Klassen im Komplexitätszoo ist nötig, um die vielen in der Praxis auftretenden Probleme sinnvoll zu klassifizieren. 14-23
2. Der Grundaufbau des Zoos ist gut verstanden.
3. Das P-NP-Problem ist zwar das bekannteste Problem, die in der Theorie wirklich interessanten Probleme liegen aber oft abseits der Hauptattraktionen.