de Gruyter Lehrbuch

Jansen/Margraf · Approximative Algorithmen
und Nichtapproximierbarkeit

Klaus Jansen
Marian Margraf

Approximative Algorithmen und Nichtapproximierbarkeit

Walter de Gruyter
Berlin · New York

Prof. Dr. Klaus Jansen
Institut für Informatik
Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4
24118 Kiel
E-Mail: kj@informatik.uni-kiel.de

Dr. Marian Margraf
Bundesamt für Sicherheit in der Informationstechnik
Kompetenzbereich Kryptographie
E-Mail: mma@informatik.uni-kiel.de

Mathematics Subject Classification 2000: 68Rxx, 68W25, 68Q05

♾ Gedruckt auf säurefreiem Papier, das die US-ANSI-Norm über Haltbarkeit erfüllt.

ISBN 978-3-11-020316-5

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Printed in Germany.
Konvertierung von LaTeX-Dateien der Autoren: Kay Dimler, Müncheberg.
Einbandgestaltung: Martin Zech, Bremen.
Druck und Bindung: Hubert & Co. GmbH & Co. KG, Göttingen.

Vorwort

Algorithmen zum Lösen von Optimierungsproblemen spielen heutzutage in der Industrie eine große Rolle. So geht es zum Beispiel bei dem Problem MIN JOB SCHEDULING, das uns während der Lektüre dieses Buches immer wieder begegnen wird, darum, zu einer gegebenen Anzahl von Maschinen und Jobs bzw. Prozessen eine Verteilung der Jobs auf die Maschinen so zu finden, dass alle Jobs in möglichst kurzer Zeit abgearbeitet werden. Natürlich soll auch das Finden einer Verteilung nicht allzu viel Zeit in Anspruch nehmen. Das Ziel ist somit, einen Algorithmus zu konstruieren, der effizient, d. h. schnell, arbeitet und eine optimale Lösung findet.

Allerdings ist dieses Ziel für viele Probleme nicht erreichbar. Wir werden sehen, dass es zum Beispiel für das Problem MIN JOB SCHEDULING höchstwahrscheinlich keinen effizienten Algorithmus gibt, der zu jeder Eingabe eine optimale Verteilung konstruiert. Ein Ausweg besteht dann darin, sogenannte Approximationsalgorithmen zu finden, d. h. Algorithmen, die zwar eine effiziente Laufzeit haben, aber dafür Lösungen konstruieren, die nicht optimal, jedoch schon ziemlich nahe an einer Optimallösung liegen.

Damit ist die Absicht, die diesem Buch zugrunde liegt, schon beschrieben. Wir werden uns also hauptsächlich mit Optimierungsproblemen beschäftigen, die schwer, d. h. nicht optimal, in effizienter Zeit zu lösen sind und stattdessen Approximationsalgorithmen für diese Probleme konstruieren und analysieren. Dabei lernen wir, sozusagen nebenbei, eine Vielzahl von Techniken für das Design und auch die Analyse solcher Algorithmen kennen.

Das Buch richtet sich gleichermaßen an Studierende der Mathematik und Informatik, die ihr Grundstudium bereits erfolgreich absolviert haben. Weiterführende Kenntnisse, wie zum Beispiel der Wahrscheinlichkeitstheorie, Logik oder Maschinenmodelle, werden nicht vorausgesetzt. Wo immer wir diese Begriffe benötigen, werden sie kurz erläutert. Allerdings erheben wir dabei keinen Anspruch auf Vollständigkeit, sondern führen die benötigten Theorien nur bis zu einem Grad ein, der für das Verständnis der hier behandelten Algorithmen und insbesondere deren Analyse notwendig ist.

Weiter haben wir versucht, weitestgehend auf Literaturverweise zu verzichten, sondern wollen stattdessen die Beweise der meisten benötigten Aussagen selbst vorstellen, um der Leserin und dem Leser eine vollständige Einführung in das behandelte Thema zu ermöglichen. Dass dies nicht immer haltbar ist, zeigt sich zum Beispiel an der umfangreichen Theorie zum Lösen linearer oder semidefiniter Programme. Die Vorstellung dieser Algorithmen ist Bestandteil einer Vorlesung über effiziente Algorithmen und selbst in diesen werden diese Themen häufig nicht vollständig behandelt. Wir werden deshalb an einigen Stellen nicht darum herumkommen, einige Ergebnisse

der aktuellen Forschung ohne Beweise zu zitieren. Dem Verständnis sollte dies aber keinen Abbruch tun. Die Alternative wäre gewesen, wichtige Errungenschaften unbehandelt zu lassen und so dem Anspruch, einen möglichst großen Überblick über die Theorie approximativer Algorithmen zu geben, nicht zu genügen.

Im Einzelnen gehen wir wie folgt vor: Im ersten Abschnitt werden zunächst die wichtigsten Begriffe und Notationen eingeführt und diese anhand von einfach zu formulierenden Optimierungsproblemen und Approximationsalgorithmen motivieren.

Der zweite Abschnitt dieses Buches dient der Definition der Komplexitätsklassen P und NP. Da diese beiden Klassen eine zentrale Bedeutung bei der Behandlung von Approximationsalgorithmen spielen (insbesondere liefert uns diese Theorie das Werkzeug dafür, von Optimierungsproblemen zeigen zu können, dass sie mit großer Wahrscheinlichkeit nicht effizient lösbar sind), werden wir sowohl besonderen Wert darauf legen, sehr formale Definitionen zu liefern, was sich leider als etwas kompliziert herausstellt, als auch verschiedene Interpretationen dieser Definitionen zu besprechen. Wir lernen in diesem Kapitel weiterhin den berühmten Satz von Cook kennen. Dieser Satz zeigt, dass das Entscheidungsproblem SAT unter der Voraussetzung $\mathrm{P} \neq \mathrm{NP}$ nicht effizient lösbar ist, und ist die erste bewiesene Aussage dieser Art überhaupt. Darauf aufbauend sind wir dann in der Lage, von einer Reihe von Problemen nachzuweisen, dass es für diese keine effizienten Algorithmen gibt, die optimale Lösungen bestimmen.

Bevor wir uns dann ab Kapitel 4 mit approximativen Algorithmen, die eine multiplikative Güte garantieren, beschäftigen, wollen wir zunächst in Kapitel 3 den Begriff der additiven Güte kennen lernen und an drei graphentheoretischen Beispielen vertiefen. Zusätzlich behandeln wir in diesem Kapitel erste Nichtapproximierbarkeitsresultate, die ebenfalls einen wichtigen Bestandteil dieses Buches bilden. Die Kapitel 17–20 sind allein diesem Thema gewidmet.

Bis einschließlich Kapitel 7 konstruieren wir meist sogenannte Greedy-Algorithmen, d. h. Algorithmen, die die Strategie verfolgen, zu jedem Zeitpunkt den lokal besten Gewinn zu erzielen (greedy (engl.) = gierig). Diese Algorithmen sind oft einfach zu formulieren, allerdings ist ihre Analyse häufig erheblich komplizierter.

Ist ein Approximationsalgorithmus für ein Optimierungsproblem mit einer multiplikativen Güte gefunden, so stellt man sich automatisch die Frage, ob dieses Ergebnis noch verbessert werden kann, d. h., ob es einen Approximationsalgorithmus gibt, der eine bessere Güte garantiert, oder ob sich ein Problem vielleicht sogar beliebig gut approximieren lässt. Dies führt zum Begriff der sogenannten Approximationsschemata, d. h. Folgen von Approximationsalgorithmen $(\mathsf{A}_\varepsilon)_{\varepsilon>0}$ so, dass A_ε eine Approximationsgüte von $(1+\varepsilon)$ garantiert. Wir werden in Kapitel 8 eine sehr einfach zu beschreibende Technik einführen, die, so die Voraussetzungen dafür erfüllt sind, häufig zum Ziel führt.

Allerdings ist das Problem bei der obigen Konstruktion oft, dass sich die Laufzeiten der Algorithmen so stark erhöhen, dass sie zwar immer noch unserem Effizienzbegriff

nicht widersprechen, aber nicht mehr praktikabel sind. Die Aussagen sind also nur noch von theoretischer Natur. Kapitel 9 behandelt deshalb sogenannte vollständige Approximationsschemata, die dieses Problem beheben. Auch die in diesem Kapitel eingeführte Technik lässt sich leicht erläutern und auch theoretisch untermalen, was wir am Ende von Kapitel 9 tun werden.

Einen völlig neuen Ansatz zur Konstruktion approximativer Algorithmen lernen wir schließlich in Kapitel 10 kennen. Hier behandeln wir sogenannte randomisierte Algorithmen, d. h. Algorithmen, die Zufallsexperimente durchführen können. Die für die Behandlung solcher Algorithmen benötigte Theorie, insbesondere die wahrscheinlichkeitstheoretischen Grundlagen, werden wir dort einführen. Es wird sich zeigen, dass randomisierte Algorithmen häufig einfach zu beschreiben sind. Allerdings liefern diese für eine Eingabe unterschiedliche Ergebnisse (eben abhängig vom Ausgang des Zufallsexperimentes). Wir werden deshalb einen zweiten Begriff von Güte definieren müssen, die sogenannte erwartete Güte, und sehen, dass dies in der Praxis oft schon ausreicht. Darüber hinaus spielt die Methode der bedingten Erwartungswerte, oder auch Derandomisierung, eine große Rolle. Ziel dieser Technik ist es, aus einem randomisierten Algorithmus wieder einen deterministischen zu konstruieren, so dass die Güte erhalten bleibt.

Nahtlos daran schließt sich die Betrachtung der LP-Relaxierung an, die wir in Kapitel 11 einführen. Viele klassische Optimierungsprobleme lassen sich, wie wir noch sehen werden, in natürlicher Weise als ganzzahliges lineares Programm schreiben. Damit reduzieren wir die Behandlung vieler Optimierungsprobleme auf das Lösen dieser linearen Programme, was allerdings nicht unbedingt leichter ist. Verzichtet man aber auf die Ganzzahligkeit, d. h., relaxiert man das Programm, dann erhält man ein Problem, für das es effiziente Algorithmen gibt. Unglücklicherweise ist die so erhaltene Lösung im Allgemeinen nicht mehr ganzzahlig, immerhin aber eine gute Näherung. Es gibt nun zwei Möglichkeiten, aus einer Lösung des relaxierten Problems eine ganzzahlige Lösung zu gewinnen. Man betrachtet die Technik des randomisierten Rundens und erhält einen randomisierten Algorithmus, oder man rundet deterministisch. Beides muss so geschickt gemacht werden, dass die so erhaltene Lösung auch eine zulässige ist. Ist dies nicht der Fall, so kann man sich natürlich die Frage stellen, wie groß die Wahrscheinlichkeit dafür ist, auf diese Weise eine zulässige Lösung zu erhalten. Dies führt dann zum Begriff des probabilistischen Algorithmus, d. h., hier ist die Wahrscheinlichkeit dafür, eine zulässige Lösung zu erhalten, hinreichend groß.

Nachdem wir in Kapitel 11 lineare Programme zum Lösen von Optimierungsproblemen benutzt haben, stellen wir die Theorie der Linearen Programmierung in Kapitel 12 noch einmal genauer vor. Wir werden die drei verschiedenen Versionen der Linearen Programmierung kennen lernen und zeigen, dass alle drei äquivalent zueinander sind. Hauptziel dieses Kapitels ist aber die Formulierung des sogenannten Dualitätssatzes, der zu den bedeutendsten Sätzen der klassischen Mathematik gehört. Aufbauend auf diesen berühmten Satz werden wir eine weitere Technik für das Design

und die Analyse von Approximationsalgorithmen an einem schon bekannten Optimierungsproblem kennen lernen.

Das Kapitel 13 widmet sich ganz dem Problem MIN BIN PACKING. Wir werden sogenannte asymptotische Approximationsschemata, unter anderem ein vollständiges, für dieses Problem behandeln. Hier ist das Ziel, einfach gesprochen, eine gute multiplikative Güte für Instanzen zu garantieren, deren optimale Lösungen große Werte annehmen. Die Idee für das Design dieser Approximationsalgorithmen ist etwas kompliziert und nutzt einige Techniken, die wir im Laufe der Lektüre dieses Buches kennen gelernt haben werden.

Weiter vertiefen werden wir unser Wissen anhand des bereits mehrfach behandelten Problems MIN JOB SCHEDULING. Wir geben am Anfang von Kapitel 14 eine umfangreiche, aber nicht vollständige Einführung in verschiedene Versionen dieses Problems (die sogenannte 3-Felder-Notation) und behandeln insbesondere drei dieser Versionen genauer.

In Kapitel 15 lernen wir einen Algorithmus kennen, der konkave Funktionen auf einer konvexen kompakten Menge approximativ maximiert. Es zeigt sich, dass dieser Algorithmus dazu genutzt werden kann, eine große Klasse sogenannter Überdeckungsprobleme approximativ zu lösen. Wir werden dies am Beispiel des Problems MIN STRIP PACKING diskutieren, genauer konstruieren wir ein vollständiges asymptotisches Approximationsschema für dieses Problem.

Semidefinite Programmierung, die wir in Kapitel 16 behandeln, kann man als Verallgemeinerung der Linearen Programmierung auffassen. Es ist leicht einzusehen, dass sich jedes lineare Programm auch als semidefinites schreiben lässt. Außerdem beruhen die Ideen zum Lösen semidefiniter Programme auf denen zum Lösen linearer Programme. Der Trick, mit Hilfe der semidefiniten Programmierung eine Lösung für ein Optimierungsproblem zu gewinnen, ist einfach, aber sehr geschickt. Überraschenderweise lässt sich dieser Trick auf eine ganze Reihe höchst unterschiedlicher Optimierungsprobleme wie zum Beispiel MAXSAT, MAXCUT und MIN NODE COLORING anwenden.

Damit ist die Einführung neuer Techniken zum Design und zur Analyse abgeschlossen. Bevor wir uns nun vollständig mit Nichtapproximierbarkeitsresultaten beschäftigen können, benötigen wir zunächst eine genauere Definition des Begriffs Optimierungsproblem, speziell um auch für diese Klasse geeignete Reduktionsbegriffe einzuführen. Wie sich zeigen wird, sind die Erläuterungen aus Kapitel 2 dafür nicht ausreichend. Wir werden also in Kapitel 17 noch einmal genauer darauf eingehen.

Ab Kapitel 18 beschäftigen wir uns dann ausgiebig mit Nichtapproximierbarkeitsresultaten bezüglich der multiplikativen Güte. Solche haben wir schon in vorangegangenen Kapiteln behandelt, wir wollen aber nun damit beginnen, diese Theorie etwas systematischer vorzustellen. Unter anderem lernen wir in diesem Kapitel einen neuen Reduktionsbegriff, die sogenannten lückenerhaltenen Reduktionen, kennen, die auch Nichtapproximierbarkeitsresultate übertragen können (im Gegensatz zu den in Ka-

pitel 2 definierten Reduktionen, die zwar die nichteffiziente Lösbarkeit übertragen, allerdings keine Aussage über Nichtapproximierbarkeit garantieren).

Anknüpfungspunkt für alle in den letzten Kapiteln dieses Buches behandelten Nichtapproximierbarkeitsresultate ist eine neue Charakterisierung der Klasse NP, das berühmte PCP-Theorem. PCP seht für *probabilistic checkable proof* und wir werden Kapitel 19 ganz diesem Thema widmen. Leider wird es uns nicht möglich sein, den Beweis des PCP-Theorems vollständig vorzustellen, die Ausführungen werden aber trotzdem eine Ahnung davon geben, was die Ideen hinter dieser Aussage sind.

Mit Hilfe dieses Theorems können wir dann zeigen, dass das Problem MAX3SAT kein polynomielles Approximationsschema besitzt. Ausgehend von diesem Resultat weisen wir in Kapitel 20 von einer ganzen Reihe höchst unterschiedlicher Optimierungsprobleme ähnliche und bei weitem stärkere Ergebnisse nach.

Wir formulieren in diesem Buch Algorithmen größtenteils umgangssprachlich, d. h., wir beschreiben die einzelnen Schritte im sogenannten Pseudocode. Die zugrundeliegenden Maschinenmodelle führen wir im Anhang sowohl für deterministische und nichtdeterministische als auch für randomisierte Algorithmen ein. Wir werden dort einen groben Überblick über verschiedenste Versionen von Turingmaschinen geben und wollen die Definitionen an mehr oder weniger komplizierten Beispielen festigen.

Es liegt in der Natur eines Lehrbuches, dass die meisten präsentierten Ergebnisse nicht auf wissenschaftliche Erkenntnisse der Autoren zurückgehen, sondern hierin vielmehr bereits seit langem bekanntes Wissen lediglich in neuer Form dargestellt wird. Das vorliegende Buch ist aus einem Skript zur Vorlesung Approximative Algorithmen I und II hervorgegangen, die von Klaus Jansen regelmäßig und von Marian Margraf im Sommersemester 2003 an der Universität zu Kiel gehalten wurden. Bei der Vorbereitung dieser Vorlesungen wurde neben den bekannten Büchern zu diesem Thema, die im Literaturverzeichnis aufgelistet sind, eine Reihe von im Internet frei verfügbaren Vorlesungsskripten anderer Hochschuldozenten genutzt, beispielsweise von J. Blömer und B. Gärtner (ETH Zürich), M. Lübbecke (TU Berlin) und R. Wanka (Universität Erlangen-Nürnberg).

Die wenigsten Bücher sind frei von Fehlern. Wir werden daher auf der Seite

`www.informatik.uni-kiel.de/mma/approx/`

regelmäßig wichtige Korrekturen auflisten. Zusätzlich sind wir dankbar, wenn Sie uns über Fehler, die Ihnen während der Lektüre dieses Buches auffallen, informieren.

Viele Personen haben zur Entstehung dieses Lehrbuches beigetragen. Neben zahlreichen Durchsichten der Manuskripte über viele Jahre und die daraus resultierenden Tipps zu Verbesserungen von Florian Diedrich, Ralf Thöle und Ulrich Michael Schwarz bedanken wir uns noch einmal besonders bei Florian Diedrich für die Erstellung von Kapitel 15 und Abschnitt 14.3, bei Ulrich Michael Schwarz für die Bereitstellung vieler LaTeX-Makros und einer grundlegenden Überarbeitung des Layouts, sowie bei Sascha Krokowski für die Erstellung vieler Graphiken. Auch beim Verlag

de Gruyter, speziell bei den Mitarbeitern Herrn Albroscheit und Herrn Dr. Plato und dem externen Mitarbeiter Herrn Dimler, möchten wir uns für die große Unterstützung bedanken.

Kiel und Bonn, Januar 2008 K. Jansen und M. Margraf

Inhaltsverzeichnis

Teil I

Approximative Algorithmen

Kapitel 1
Einführung

Ein Ziel bei der Konstruktion von Algorithmen für Optimierungsprobleme ist häufig, wie bereits im Vorwort beschrieben, die Laufzeit der Algorithmen so gering wie möglich zu halten, um Lösungen schnell berechnen zu können, Dabei muss man oft, wie wir noch sehen werden, darauf verzichten, optimale Lösungen zu bestimmen.

Wir starten mit zwei motivierenden Beispielen und untersuchen zunächst sehr einfache Approximationsalgorithmen für diese Probleme, an denen wir die Hauptideen zur Konstruktion solcher Algorithmen erläutern werden. Dass diese Probleme tatsächlich schwer zu lösen sind, können wir allerdings erst im folgenden Kapitel zeigen, da uns die dafür benötigte Theorie hier noch nicht zur Verfügung steht.

Nachdem wir dann schon zwei Optimierungsprobleme und zugehörige Approximationsalgorithmen kennen, wollen wir in Abschnitt 1.2 die wichtigsten für die Lektüre dieses Buches benötigten Definitionen einführen und anhand der bereits vorgestellten Beispiele erläutern. Dazu gehört neben der Definition von Optimierungsproblem, Algorithmus, Laufzeit und Approximationsgüte (die beschreibt, wie nahe eine gefundene Lösung am Optimum liegt) auch eine sehr kurze Einführung in die Komplexitätsklassen P und NP, die für ein grobes Verständnis der Problematik genügen sollte. Für einen vertiefenden Einblick sei allerdings auf Kapitel 2 verwiesen, das sich ausgiebig mit diesem Thema beschäftigen wird.

1.1 Zwei Beispiele (MIN JOB SCHEDULING und MAXCUT)

Das folgende Problem wird uns während der Lektüre des Buches immer wieder begegnen.

Beispiel 1.1 (MIN JOB SCHEDULING). Gegeben seien m Maschinen $M_1, \ldots, M_m$, n Jobs $J_1, \ldots, J_n$ und zu jedem Job J_i, $i \in \{1, \ldots, n\}$, die Laufzeit $p_i \in \mathbb{N}$, die jede Maschine benötigt, um den Job J_i auszuführen.

Weiter machen wir die folgenden Einschränkungen: Zu jedem Zeitpunkt kann jede Maschine nur einen Job ausführen, und ein einmal angefangener Job kann nicht abgebrochen werden. Gesucht ist dann eine Verteilung (ein sogenannter *Schedule*) der n Jobs auf die m Maschinen, die den obigen Bedingungen genügt und möglichst schnell alle Jobs abarbeitet.

Ein Schedule wird durch n Paare (s_k, M_i), $k \in \{1, \ldots, n\}$, $i \in \{1, \ldots, m\}$, beschrieben. Dabei bedeute (s_k, M_i), dass der Job J_k zum Zeitpunkt s_k auf der i-ten

Maschine M_i gestartet wird. Für einen Schedule

$$((s_1, M_{i_1}), \ldots, (s_n, M_{i_n})), \quad i_j \in \{1, \ldots, m\},$$

sei

$$C_{\max} := \max\{s_k + p_k; k \in \{1, \ldots, n\}\}$$

der sogenannte *Makespan* des Schedules, also die Zeit, die die Maschinen bei vorgegebenem Schedule benötigen, um alle Jobs abzuarbeiten. Es geht also darum, den Makespan zu minimieren.

Allgemein werden wir Probleme wie folgt formulieren:

Problem 1.2 (MIN JOB SCHEDULING).
Eingabe: Maschinen $M_1, \ldots, M_m$, $m \in \mathbb{N}$, Jobs $J_1, \ldots, J_n$, $n \in \mathbb{N}$, und Ausführungszeiten $p_1, \ldots, p_n$ für jeden Job.
Ausgabe: Ein Schedule mit minimalem Makespan.

Wir betrachten als Beispiel für das obige Problem zwei Maschinen M_1, M_2 und fünf Jobs $J_1, \ldots, J_5$ mit Laufzeiten $p_1 = 1$, $p_2 = 2$, $p_3 = 2$, $p_4 = 4$ und $p_5 = 1$. Wie leicht zu sehen ist, ist ein optimaler Schedule

$$((0, M_1), (3, M_1), (1, M_1), (0, M_2), (4, M_2)),$$

siehe Abbildung 1.1.

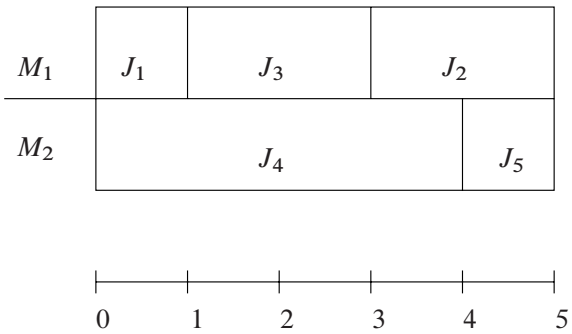

Abbildung 1.1. Ein optimaler Schedule zu den Jobs aus Beispiel 1.1.

Wie wir später noch sehen werden, ist es im Allgemeinen sehr schwer, einen optimalen Schedule zu finden. Das bedeutet, dass es höchstwahrscheinlich keinen effizienten Algorithmus gibt, der für jede Eingabe des Problems einen optimalen Schedule konstruiert.

Abhilfe schafft ein Approximationsalgorithmus, d. h. ein Algorithmus, der effizient ist, also polynomielle Laufzeit hat, dafür aber keinen optimalen Wert liefert, sondern einen, der nicht zu sehr vom Optimum abweicht. All die oben schon benutzten Begriffe wie schweres Problem, Algorithmus, Effizienz und polynomielle Laufzeit werden

wir in den nächsten Abschnitten noch einmal präzisieren. Hier soll es erst einmal genügen, mit der Anschauung, die diesen Begriffen zugrunde liegt, zu arbeiten.

Wir werden nun einen Algorithmus für das Problem MIN JOB SCHEDULING kennen lernen, der eine ziemlich gute Annäherung an das Optimum findet. In der Analyse dieses Algorithmus sehen wir dann schon einige wichtige Techniken, die wir auch später benutzen werden.

Algorithmus LIST SCHEDULE$(m, p_1, \ldots, p_n)$

```
1  S := ∅
2  for i = 1 to m do
3    a_i := 0
4  od
5  for k = 1 to n do
6    Berechne das kleinste i mit a_i = min{a_j; j ∈ {1,...,m}}.
7    s_k := a_i
8    a_i := a_i + p_k
9    S := S ∪ {(s_k, M_i)}
10 od
11 return S
```

Der Algorithmus verfolgt also die Strategie, die Jobs auf die gerade frei werdende Maschine zu verteilen, ohne sich die gesamte Eingabe überhaupt anzusehen. Wir werden im Laufe der Lektüre noch weitere Approximationsalgorithmen für dieses Problem kennen lernen, die bei weitem besser sind, allerdings ist deren Analyse dann auch komplizierter.

Für unser oben angegebenes Beispiel berechnet der Algorithmus den in Abbildung 1.2 dargestellten Schedule.

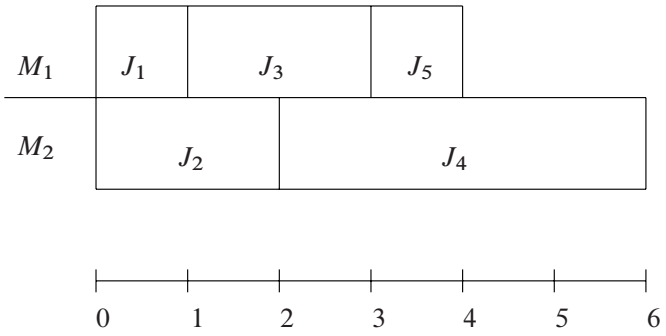

Abbildung 1.2. Das Ergebnis von LIST SCHEDULE für die Instanz aus Beispiel 1.1.

Bevor wir im nächsten Abschnitt zeigen, dass dieser Algorithmus tatsächlich effizient ist, wozu wir zunächst noch einige Definitionen benötigen, beweisen wir erst,

dass die vom Algorithmus LIST SCHEDULE erzeugte Lösung höchstens doppelt so lang ist wie die eines optimalen Schedules. Der obige Algorithmus, ebenso wie die folgende Analyse, stammen von Graham, siehe [80].

Satz 1.3. *Der vom Algorithmus* LIST SCHEDULE *erzeugte Schedule hat einen Makespan, der höchstens doppelt so lang ist wie der Makespan eines optimalen Schedules.*

Beweis. Gegeben seien m Maschinen und n Jobs mit Laufzeiten $p_1, \ldots, p_n$. Weiter sei OPT der Makespan eines optimalen Schedules und C der vom Algorithmus erzeugte Makespan. Wie im Algorithmus sei s_k die Startzeit des k-ten Jobs in dem vom Algorithmus erzeugten Schedule und $C_k = s_k + p_k$ die Zeit, in der der k-te Job endet. Schließlich sei J_l der Job, der als letzter beendet wird. Es gilt also $C = C_l$.

Bis zum Zeitpunkt s_l ist jede Maschine durchgehend ausgelastet (sonst hätte J_l früher gestartet werden können). Damit ist die durchschnittliche Laufzeit jeder Maschine bis zum Start des letzten Jobs J_l höchstens

$$\frac{1}{m} \sum_{k \in \{1,\ldots,n\} \setminus \{l\}} p_k .$$

Es gibt also mindestens eine Maschine, deren Laufzeit höchstens von dieser Größe ist, und es folgt

$$s_l \leq \frac{1}{m} \sum_{k \in \{1,\ldots,n\} \setminus \{l\}} p_k .$$

Insbesondere erhalten wir

$$C = C_l = s_l + p_l \leq \frac{1}{m} \sum_{k \in \{1,\ldots,n\} \setminus \{l\}} p_k + p_l = \frac{1}{m} \sum_{k \in \{1,\ldots,n\}} p_k + \left(1 - \frac{1}{m}\right) p_l .$$

Weiter ist $\frac{1}{m} \sum_{k=1}^{n} p_k$ die durchschnittliche Laufzeit jeder Maschine. Damit gilt also $\mathrm{OPT} \geq \frac{1}{m} \sum_{k=1}^{n} p_k$. Da weiter $\mathrm{OPT} \geq p_l$, folgt insgesamt

$$\begin{aligned} C &\leq \frac{1}{m} \sum_{k=1}^{n} p_k + \left(1 - \frac{1}{m}\right) p_l \\ &\leq \mathrm{OPT} + \left(1 - \frac{1}{m}\right) \mathrm{OPT} \\ &\leq \left(2 - \frac{1}{m}\right) \mathrm{OPT} \\ &\leq 2\,\mathrm{OPT} . \end{aligned}$$

□

Wir können also sogar zeigen, dass der Algorithmus LIST SCHEDULE eine Güte von $\left(2 - \frac{1}{m}\right)$ garantiert, wobei m die Anzahl der Maschinen ist.

Zunächst scheint es schwierig zu sein, den Wert der Lösung eines Approximationsalgorithmus mit dem optimalen Wert zu vergleichen, da wir diesen gar nicht kennen. Der Trick dabei ist, gute untere Schranken für die optimale Lösung zu suchen. Das Finden solcher Schranken ist häufig der entscheidende Punkt in den Analysen von Approximationsalgorithmen. In dem obigen Beweis sind diese Schranken

- $\text{OPT} \geq \frac{1}{m} \sum_{k=1}^{n} p_k$, und
- $\text{OPT} \geq p_l$.

Als zweites Beispiel lernen wir nun ein *Maximierungsproblem* kennen und analysieren einen einfachen Approximationsalgorithmus für dieses Problem.

Problem 1.4 (MAXCUT).
Eingabe: Ein ungerichteter Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge E.
Ausgabe: Eine Partition $(S, V \setminus S)$ der Knotenmenge so, dass die Größe $w(S)$ des *Schnittes*, also die Zahl der Kanten zwischen S und $V \setminus S$, maximiert wird.

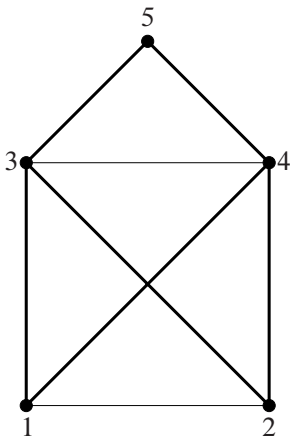

Abbildung 1.3. Ein Beispiel für einen Schnitt.

Wir betrachten das Beispiel aus Abbildung 1.3: Der Schnitt $S = \{3, 4\}$ hat eine Größe von 6 und ist, wie man leicht überlegt, ein maximaler Schnitt (im Bild sind die Schnittkanten jeweils fett gezeichnet).

Auch dieses Problem ist, wie schon das Problem MIN JOB SCHEDULING, schwierig (dies wurde von Karp in [128] gezeigt). Wir werden jetzt einen Approximationsalgorithmus kennen lernen, der eine effiziente Laufzeit hat und dessen Lösungen mindestens halb so groß sind wie die Optimallösungen. Dieser Algorithmus und die Analyse findet man in Sahni und Gonzales [178].

Algorithmus LOCAL IMPROVEMENT(G)

```
1  S := ∅
2  while ∃v ∈ V : w(S Δ {v}) > w(S) do
3    S := S Δ {v}
4  od
5  return S
```

Dabei bezeichnet Δ die *symmetrische Differenz* zweier Mengen, für unseren speziellen Fall heißt dies

$$S \,\Delta\, \{v\} = \begin{cases} S \cup \{v\}, & \text{falls } v \notin S, \\ S \setminus \{v\}, & \text{sonst.} \end{cases}$$

Die Idee des Algorithmus ist sehr einfach. Man startet mit einer Menge S von Knoten. Solange es noch einen Knoten gibt, dessen Hinzunahme oder Wegnahme von S den aktuellen Schnitt vergrößert, wird S entsprechend angepasst (local improvement). Andernfalls wird S ausgegeben.

Wir kommen nun zur angekündigten Analyse.

Satz 1.5. *Der Algorithmus* LOCAL IMPROVEMENT *liefert zu jeder Eingabe einen Schnitt, für dessen Größe* $w \geq \frac{1}{2}\,\mathrm{OPT}$ *gilt, wobei* OPT *die Größe eines optimalen Schnittes bezeichnet.*

Beweis. Sei S der von dem Algorithmus berechnete Schnitt. Dann gilt für alle $v \in V$, dass mindestens die Hälfte der Kanten durch v Schnittkanten bezüglich S sind (andernfalls wäre $S \,\Delta\, \{v\}$ ein größerer Schnitt). Also sind mindestens die Hälfte aller Kanten Schnittkanten bezüglich S, woraus $2w \geq |E|$ folgt. Da aber offensichtlich $\mathrm{OPT} \leq |E|$, erhalten wir $2w \geq |E| \geq \mathrm{OPT}$. □

Wie schon bei der Analyse des Algorithmus LIST SCHEDULE haben wir für den obigen Beweis eine Schranke für die optimale Lösung angegeben, nur im Gegensatz zum Ersten eine obere, da es sich bei MAXCUT um ein Maximierungsproblem handelt.

1.2 Notationen und Definitionen

Wie oben versprochen, wollen wir nun einige im letzten Abschnitt bereits benutzten Begriffe an dieser Stelle präzisieren und die Definitionen an einigen Beispielen motivieren.

Optimierungsprobleme

Ein *Optimierungsproblem* $\Pi = (\mathcal{I}, F, w)$ ist definiert durch:

- Eine Menge $\mathcal{I}$ von *Instanzen*.
- Zu jeder Instanz $I \in \mathcal{I}$ existiert eine Menge $F(I)$ von *zulässigen Lösungen*,
- Jeder Lösung $S \in F(I)$ ist ein Wert $w(S) \in \mathbb{Q}_+$ zugeordnet.

Das Ziel ist dann, bei gegebener Instanz eine zulässige Lösung so zu finden, dass $w(S)$ möglichst groß (*Maximierungsproblem*) oder möglichst klein ist (*Minimierungsproblem*).

Im MIN JOB SCHEDULING zum Beispiel wird eine Instanz I spezifiziert durch die Anzahl der Maschinen, die Anzahl der Jobs und die Laufzeiten der einzelnen Jobs. Die Menge $F(I)$ der zulässigen Lösungen ist dann die Menge der Schedules zur Eingabe I. Weiter bezeichnet für jede zulässige Lösung, also jeden Schedule $S \in F(I)$ der Wert $w(S)$ den Makespan. Da wir den Makespan möglichst klein haben wollen, handelt es sich hierbei also um ein Minimierungsproblem.

Entscheidungsprobleme

Darüber hinaus können wir aber auch folgendes *Entscheidungsproblem* betrachten.

Problem 1.6 (JOB SCHEDULING).
Eingabe: m Maschinen $M_1, \ldots, M_m$, n Jobs $J_1, \ldots, J_n$, Ausführungszeiten $p_1, \ldots, p_n$ für jeden Job und $k \in \mathbb{N}$.
Frage: Existiert ein Schedule mit einem Makespan kleiner gleich k?

Allgemein besteht ein Entscheidungsproblem $\Pi = (\mathcal{I}, Y_{\mathcal{I}})$ aus einer Menge $\mathcal{I}$ von Instanzen und einer Teilmenge $Y_{\mathcal{I}} \subseteq \mathcal{I}$ von JA-Instanzen. Das Problem kann dann wie folgt formuliert werden:

Problem 1.7 ($\Pi = (\mathcal{I}, Y_{\mathcal{I}})$).
Eingabe: $x \in \mathcal{I}$.
Frage: Gilt $x \in Y_{\mathcal{I}}$?

Die oben angegebene Konstruktion, also zu einem Optimierungsproblem ein Entscheidungsproblem zu konstruieren, lässt sich natürlich verallgemeinern.

Definition 1.8. Sei $\Pi = (\mathcal{I}, F, w)$ ein Optimierungsproblem. Dann heißt $\Pi' = (\mathcal{I}', Y_{\mathcal{I}'})$ mit $\mathcal{I}' = \mathcal{I} \times \mathbb{Q}$ und

$$Y_{\mathcal{I}'} = \{(I, x) \in \mathcal{I}'; \text{es existiert } S \in F(I) \text{ mit } w(t) \geq x\}$$

bzw.

$$Y_{\mathcal{I}'} = \{(I, x) \in \mathcal{I}'; \text{es existiert } S \in F(I) \text{ mit } w(t) \leq x\},$$

wenn Π ein Maximierungs- bzw. ein Minimierungsproblem ist, das zu Π *zugehörige Entscheidungsproblem*.

Ein Algorithmus für ein Optimierungsproblem löst auch sofort das zugehörige Entscheidungsproblem. Dies ist umgekehrt nicht immer so. Wir werden im nächsten Kapitel noch einmal näher darauf eingehen.

Um zwischen Optimierungsproblemen und den zugehörigen Entscheidungsproblemen unterscheiden zu können, werden wir Optimierungsprobleme entsprechend ihrer Zielfunktion mit dem Präfix Min bzw. Max bezeichnen und beim zugehörigen Entscheidungsproblem dieses einfach weglassen. Beispielsweise bezeichnet MIN JOB SCHEDULING also das Optimierungsproblem und JOB SCHEDULING die Entscheidungsvariante.

Kodierung

Damit ein Algorithmus ein Optimierungsproblem bearbeiten kann, werden Instanzen und zulässige Lösungen über einem endlichen Alphabet Σ kodiert, d. h., sowohl die Menge $\mathcal{I}$ der Instanzen eines Optimierungsproblems Π als auch die Menge der zulässigen Lösungen $F(I)$ für eine Instanz $I \in \mathcal{I}$ sind Sprachen, und somit Teilmengen von Σ^*. Mit anderen Worten ist ein Optimierungsproblem Π also eine Relation $\Pi \subseteq \Sigma^* \times \Sigma^*$ mit einer Gewichtsfunktion w auf der Menge der zulässigen Lösungen. Dabei bedeutet $(I, S) \in \Pi$, dass S eine zulässige Lösung zur Instanz I ist.

Damit ist die *Länge* $|I|_\Sigma$ *einer Instanz* $I \in \mathcal{I}$ (oder auch Eingabe) die Anzahl der Zeichen aus Σ, die für die Kodierung von I benutzt werden. Ist das Alphabet bekannt oder spielt keine Rolle, so schreiben wir auch kurz $|I|$.

Da wir uns, wie wir noch sehen werden, häufig nur für die Laufzeit eines Algorithmus „bis auf Konstanten“ interessieren, ist es egal, welche Kodierung wir für ein Problem wählen. Um dies einzusehen, betrachten wir die Alphabete

$$\begin{aligned} \Sigma_1 &= \{0, \ldots, 9\} \text{ und} \\ \Sigma_2 &= \{0, 1\}. \end{aligned}$$

Jeder Buchstabe aus dem Alphabet Σ_1 lässt sich auch als Bitstring, d. h. als Wort, über dem Alphabet Σ_2 schreiben, wobei für jeden Buchstaben $x \in \Sigma_1$ höchstens $\lceil \log_2 10 \rceil$ Bits benötigt werden (Σ_1 besteht aus genau zehn Buchstaben). Dies bedeutet, dass die Längen einer Instanz I kodiert über Σ_1 und Σ_2 nur um eine Konstante voneinander abweichen. Genauer gilt

$$|I|_{\Sigma_1} \leq c \cdot |I|_{\Sigma_2},$$

wobei $c = \lceil \log_2 |\Sigma_1| \rceil = \lceil \log_2 10 \rceil$. Die Konstante c ist also insbesondere unabhängig von I. Üblicherweise wählen wir die Binärkodierung mit Trennsymbol als Standard, d. h. $\Sigma = \{0, 1, |\}$.

Für MIN JOB SCHEDULING bedeutet dies: Eine Eingabe I der Form m Maschinen, n Jobs mit Laufzeiten $p_1, \ldots, p_n \in \mathbb{N}$ wird beschrieben durch den String

$$(\text{bin}(m)|\,\text{bin}(p_1)|\ldots|\,\text{bin}(p_n)) \in \{0, 1, |\}^*,$$

wobei bin(x) die Binärdarstellung einer natürlichen Zahl $x \in \mathbb{N}$ beschreibe. Die Länge von I ist somit

$$\begin{aligned}&(\lfloor \log_2 m \rfloor + 1) + (\lfloor \log_2 p_1 \rfloor + 1) + \cdots + (\lfloor \log_2 p_n \rfloor + 1) + n \\ &= \lfloor \log_2 m \rfloor + \sum_{i=1}^{n} \lfloor \log_2 p_1 \rfloor + 2n + 1.\end{aligned}$$

Die Kodierungslänge von natürlichen bzw. rationalen Zahlen wird häufig mit dem Symbol $\langle \cdot \rangle$ bezeichnet. Dabei ist

$$\begin{aligned}\langle n \rangle &:= \log_2 n + 1 \text{ für eine natürliche Zahl } n \in \mathbb{N}, \\ \langle q \rangle &:= \langle n \rangle + \langle m \rangle \text{ für eine rationale Zahl } q = \tfrac{n}{m} \in \mathbb{Q}, \\ \langle b \rangle &:= \sum_{i=1}^{n} \langle b_i \rangle \text{ für einen Vektor } b = (b_1, \ldots, b_n) \in \mathbb{Q}^n, \text{ und} \\ \langle A \rangle &:= \sum_{i=1}^{n} \sum_{j=1}^{m} \langle a_{ij} \rangle \text{ für eine Matrix } A = (a_{ij}) \in \mathbb{Q}^{n \times m}.\end{aligned}$$

Wie man andere Arten von Instanzen, wie zum Beispiel Graphen, kodiert, lernen wir am Ende dieses Kapitels kennen.

Algorithmus und Laufzeit

Formal gesehen ist ein Algorithmus eine deterministische Turingmaschine, siehe Kapitel 2. Wir werden aber in diesem Buch Algorithmen umgangssprachlich formulieren.

Ein Algorithmus A für ein Problem Π berechnet zu jeder Eingabe I von Π einen Wert $\mathsf{A}(I)$. Für Optimierungsprobleme ist dies eine zulässige Lösung, d. h. ein Element aus $F(I)$, bei Entscheidungsproblemen ein Element aus der Menge $\{\texttt{wahr}, \texttt{falsch}\}$. A berechnet damit eine Funktion über der Instanzenmenge von Π. Ist eine Funktion

$$f : X \longrightarrow Y$$

gegeben, so sagen wir, dass A die Funktion f berechnet, wenn $f(x) = \mathsf{A}(x)$ für alle $x \in X$.

Die *Laufzeit* $T_{\mathsf{A}}(I)$ eines Algorithmus A zu einer gegebenen Instanz I ist die Anzahl der elementaren Operationen, die der Algorithmus (und damit die zugehörige *Turingmaschine*) auf der Eingabe I durchführt. Weiter bezeichnen wir mit

$$T_{\mathsf{A}}(n) = \max\{T_{\mathsf{A}}(I); |I| = n\}$$

die *Zeitkomplexität*, oder auch worst-case-Rechenzeit des Algorithmus A für Instanzen der Länge $n \in \mathbb{N}$. Wir sagen dann, dass ein Algorithmus A *polynomielle Laufzeit* hat, wenn es ein Polynom $p : \mathbb{N} \longrightarrow \mathbb{R}$ so gibt, dass

$$T_{\mathsf{A}}(n) \leq p(n)$$

für alle $n \in \mathbb{N}$.

Betrachten wir noch einmal das Problem MIN JOB SCHEDULING und den für dieses Problem konstruierten Algorithmus LIST SCHEDULE. Zunächst setzt der Algorithmus m Werte $a_1, \ldots, a_m$ auf null. In jedem der n Schleifendurchläufe wird ein Minimum von m Elementen bestimmt, was eine Laufzeit von mindestens m Schritten benötigt, siehe Übungsaufgabe 1.10. Bis auf eine multiplikative Konstante erhalten wir also eine Laufzeit von $m + n \cdot m$. Die Eingabelänge einer Instanz von MIN JOB SCHEDULING, in der alle Jobs eine Ausführungszeit von eins haben, ist, wie wir auf Seite 10 gesehen haben, gerade $\log_2 m + 2n + 1$. Zunächst hat dieser Algorithmus also keine polynomielle Laufzeit.[1] Überlegt man sich aber, dass man sowieso nur Instanzen betrachten muss, bei der die Anzahl der Maschinen kleiner als die Anzahl der zu verteilenden Jobs ist (alles andere ist trivial), so erhalten wir eine Laufzeit kleiner gleich $n + n \cdot n$, was polynomiell in der Länge der Eingabe ist. Insbesondere werden wir im Folgenden für jede Version des Problems MIN JOB SCHEDULING immer annehmen, dass die Anzahl der Maschinen kleiner ist als die Anzahl der Jobs, ohne dies explizit zu erwähnen.

$\mathcal{O}(\cdot)$-Notation

Wie weiter oben bereits beschrieben, interessieren wir uns häufig nicht für die genaue Laufzeit eines Algorithmus, sondern nur für die Laufzeit „bis auf Konstanten“.

Seien $f, g : \mathbb{N} \to \mathbb{N}$ Funktionen. Dann gilt

- $f \in \mathcal{O}(g)$, wenn $c > 0, n_0 \in \mathbb{N}$ existieren mit $f(n) \leq cg(n)$ für alle $n \geq n_0$.
- $f \in \Omega(g)$, wenn $c > 0, n_0 \in \mathbb{N}$ existieren mit $f(n) \geq cg(n)$ für alle $n \geq n_0$.
- $f \in \Theta(g)$, wenn $f \in \mathcal{O}(g)$ und $f \in \Omega(g)$.
- $f \in \widetilde{\mathcal{O}}(g)$, wenn $n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}_{>1}$ existieren mit $f(n) \leq c_1 g(n)(\log n)^{c_2}$ für alle $n \geq n_0$.
- $f \in o(g)$, wenn $\lim_{n\to\infty} f(n)/g(n) = 0$.

Wir sagen, dass ein Algorithmus A polynomielle Laufzeit hat, wenn $T_{\mathsf{A}} \in \mathcal{O}(p)$ für ein Polynom p.

Die Laufzeit von MIN JOB SCHEDULING kann durch $\mathcal{O}(|I|^2)$ abgeschätzt werden, ist also polynomiell.

[1] Man beachte, dass $m + n \cdot m$ für große Zahlen m exponentiell in $\log_2 m + 2n + 1$ ist.

Approximationsalgorithmus

Sei A ein Algorithmus für ein Optimierungsproblem Π. Für eine Instanz $I \in \mathcal{I}$ sei $\mathrm{OPT}(I)$ der Wert einer optimalen Lösung der Instanz I. Weiter sei $\mathsf{A}(I)$ der Wert der vom Algorithmus A gefundenen zulässigen Lösung. Dann ist

$$\delta_{\mathsf{A}}(I) = \frac{\mathsf{A}(I)}{\mathrm{OPT}(I)}$$

die *Approximationsgüte* oder der *Approximationsfaktor* von A bei der Eingabe von I.

Weiter sei $\delta : \mathcal{I} \to \mathbb{Q}_+$ eine Funktion. Wir sagen, dass A *(multiplikative) Approximationsgüte* oder *Approximationsfaktor* δ hat, wenn für jede Instanz $I \in \mathcal{I}$ gilt

$$\delta_{\mathsf{A}}(I) \geq \delta(I) \qquad \text{bei einem Maximierungsproblem,}$$
$$\delta_{\mathsf{A}}(I) \leq \delta(I) \qquad \text{bei einem Minimierungsproblem.}$$

Häufig ist δ eine konstante Funktion oder hängt nur von der Eingabelänge ab. Mit dieser Notation können wir nun sagen, dass der Algorithmus LIST SCHEDULE einen Approximationsfaktor von 2 und der Algorithmus LOCAL IMPROVEMENT einen Approximationsfaktor von $\frac{1}{2}$ hat.

Man beachte, dass die Approximationsgüte für Approximationsalgorithmen von Maximierungsproblemen immer kleiner gleich 1 ist, im Gegensatz dazu ist diese für Minimierungsprobleme immer größer gleich 1. Um diese Asymmetrie aufzuheben, wird für Maximierungsprobleme häufig auch der Wert $\mathrm{OPT}(I)/\mathsf{A}(I)$ anstelle von $\mathsf{A}(I)/\mathrm{OPT}(I)$ als Approximationsgüte bezeichnet. Damit erhält man dann für alle Optimierungsprobleme eine Approximationsgüte größer gleich 1, unabhängig davon, ob es sich um Maximierungs- oder Minimierungsprobleme handelt.

Wir werden, da Verwechslungen ausgeschlossen sind, im Laufe dieses Buches beide Begriffe benutzen, wobei wir die zweite Definition hauptsächlich während der Behandlung von Nichtapproximierbarkeitsresulten in den Kapiteln 18–20 verwenden.

Die Komplexitätsklassen P und NP

Wir wollen in diesem Unterabschnitt eine sehr kurze Einführung in die Komplexitätsklassen P und NP geben, die zum Verständnis für die Lektüre dieses Buches ausreichen sollte. Eine genaue und vollständige Beschreibung der Klassen verschieben wir auf das folgende Kapitel.

Einfach gesagt besteht die Klasse P aus allen Entscheidungsproblemen Π, für die es einen polynomiell zeitbeschränkten Algorithmus gibt, der für jede Instanz I von Π korrekt entscheidet, ob I eine JA-Instanz ist oder nicht.

Schwieriger ist die Definition der Klasse NP. Diese besteht aus allen Entscheidungsproblemen $\Pi = (\mathcal{I}, Y_{\mathcal{I}})$, für die es einen polynomiell zeitbeschränkten Algorithmus A und ein Polynom p so gibt, dass für alle $I \in \mathcal{I}$ gilt:

$$I \in Y_{\mathcal{I}} \Longleftrightarrow \text{es ex. } y \in \Sigma^* \text{ mit } |y| \leq p(|I|) \text{ so, dass } \mathsf{A}(I, y) = \texttt{wahr}.$$

Man kann y dann als Hilfsvariable dafür ansehen, dass der Algorithmus sich für die korrekte Antwort entscheidet. Allerdings kann das Finden dieser Hilfsvariable sehr schwer sein. Wir werden im nächsten Kapitel weitere Interpretationen der Klasse NP kennen lernen.

Schauen wir uns aber zunächst ein Beispiel an. Bei der Entscheidungsvariante des Problems MIN JOB SCHEDULING kann zu einer Eingabe I der Form „m Maschinen, n Jobs mit Laufzeiten $p_1, \ldots, p_n$ und $k \in \mathbb{N}$“ und einem Schedule s in polynomieller Zeit getestet werden, ob

(i) s überhaupt eine zulässige Lösung ist, und

(ii) der von s erzeugte Makespan kleiner als k ist.

Der Schedule s ist dann also die „Hilfsvariable“, die dem Algorithmus hilft, sich für die richtige Antwort zu entscheiden.

Offensichtlich gilt $\mathrm{P} \subseteq \mathrm{NP}$, man setze die Hilfsvariable y in der Definition von NP einfach als leeres Wort. Die große Frage aber ist, ob auch die Umkehrung gilt, d. h., ob die beiden Klassen gleich sind. Trotz vieler Bemühungen in den letzten Jahrzehnten konnte diese Frage bisher nicht beantwortet werden, es wird allerdings angenommen, dass P eine echte Teilmenge von NP ist.

Ziel der Betrachtung der oben definierten Komplexitätsklassen, jedenfalls im Kontext des vorliegenden Buches, ist es, ein Verfahren in die Hand zu bekommen, mit dem man von Optimierungsproblemen zeigen kann, dass diese unter der Voraussetzung $\mathrm{P} \neq \mathrm{NP}$ nicht polynomiell gelöst werden können.

Offensichtlich ist ein Optimierungsproblem mindestens so schwer (schwer in Bezug auf polynomielle Lösbarkeit) wie das zugehörige Entscheidungsproblem. Haben wir nämlich einen polynomiellen Algorithmus für das Optimierungsproblem, so können wir auch sofort einen polynomiellen Algorithmus für die Entscheidungsvariante angeben (man löse zunächst das Optimierungsproblem und vergleiche den optimalen Wert mit der Zahl aus der Eingabe des Entscheidungsproblems).

Wir müssen uns also bei der Frage, ob ein Optimierungsproblem nicht polynomiell gelöst werden kann, nur auf das zugehörige Entscheidungsproblem konzentrieren. Kandidaten für Entscheidungsprobleme, die nicht in polynomieller Zeit gelöst werden können, sind die Probleme aus $\mathrm{NP} \setminus \mathrm{P}$, genauer werden wir, da wir ja gar nicht wissen, ob solche Elemente überhaupt existieren, sogenannte NP-vollständige Entscheidungsprobleme betrachten.

Ein Entscheidungsproblem Π ist, einfach gesprochen, NP-vollständig, wenn sich jedes Entscheidungsproblem aus NP auf Π in polynomieller Zeit reduzieren lässt, siehe Definition 2.7 für eine formale Definition. Dies bedeutet, dass, wenn Π polynomiell lösbar ist, jedes Entscheidungsproblem aus NP ebenfalls, dank der polynomiellen Reduktion, in polynomieller Zeit entschieden werden kann. Die NP-vollständigen Probleme sind also die am schwierigsten zu lösenden Probleme in NP.

Wir werden im Laufe der Lektüre dieses Buches von einer Vielzahl von Problemen die NP-Vollständigkeit nachweisen. Diese Probleme sind dann unter der Vorausset-

zung P $\neq$ NP nicht in polynomieller Zeit zu lösen. Darüber hinaus lernen wir aber noch andere Komplexitätsklassen kennen, die uns Verfahren in die Hand geben, von gewissen Problemen zeigen zu können, dass diese sogar schwer zu approximieren sind. Dazu aber später mehr.

Graphen

Eine Vielzahl von Optimierungsproblemen sind graphentheoretischer Natur und damit Gegenstand dieses Buches.

Ein *(endlicher) Graph* ist ein Tupel $G = (V, E)$, wobei $V = V(G)$ eine endliche Menge und $E = E(G)$ eine Teilmenge der zweielementigen Teilmengen von V ist. Die Elemente von V heißen *Knoten* (oder *Ecken*) und die Elemente von E *Kanten*. Wir sagen, ein Knoten $v \in V$ *inzidiert* mit einer Kante $e \in E$, wenn $v \in e$ gilt. Weitere Formulierungen sind: v liegt auf e, e geht durch v oder v ist ein Endknoten von e.

Ein Graph $G = (V, E)$ heißt *knotengewichtet*, wenn zusätzlich eine sogenannte *Gewichtsfunktion*

$$w : V \longrightarrow \mathbb{Q}$$

auf der Menge der Knoten V von G gegeben ist. Der Wert $w(v)$ heißt dann auch Gewicht von $v \in V$. Darüber hinaus kann man auch eine Gewichtsfunktion

$$w : E \longrightarrow \mathbb{Q}$$

auf der Menge der Kanten von G betrachten. In diesem Fall heißt G auch *kantengewichtet*. Wenn klar ist, auf welcher der Mengen von G die Gewichtsfunktion definiert ist, so nennen wir G auch kurz *gewichtet*.

Mit

$$K_n = (V := \{1, \ldots, n\},\ \ E = \{e \subseteq V; |e| = 2\})$$

bezeichnen wir den *vollständigen Graphen* auf n Knoten, siehe Abbildung 1.4; K_3 heißt auch *Dreieck*.

Der *Grad* $\delta(v)$ eines Knotens $v \in V$ ist die Anzahl der mit diesem Knoten inzidenten Kanten. Weiter ist

$$\delta(G) := \min\{\delta(v); v \in V\} \quad \text{bzw.} \quad \Delta(G) := \max\{\Delta(v); v \in V\}$$

der *Minimalgrad* bzw. *Maximalgrad* des Graphen G.

Für einen Graphen $G = (V, E)$ heißt $H = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E \cap \mathfrak{P}(V')$ *Teilgraph* von G. H heißt *induzierter Teilgraph*, falls für alle $e \in E$ mit $\{x, y\} = e$ und $x, y \in V'$ auch $e \in E'$ gilt.

Eine *Clique* $W \subseteq V$ in G ist eine Teilmenge der Knotenmenge so, dass je zwei Knoten aus W durch eine Kante verbunden sind (d. h., der von W induzierte Teilgraph

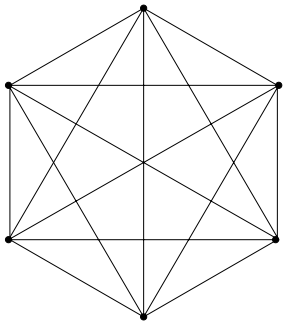

Abbildung 1.4. Der vollständige Graph auf sechs Knoten.

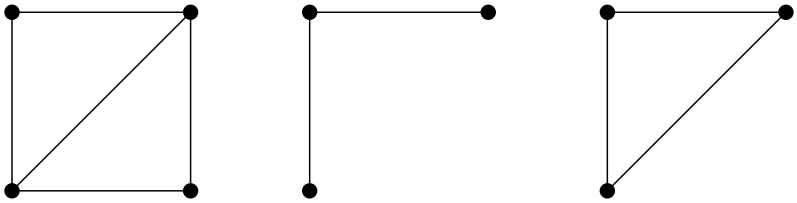

Abbildung 1.5. Ein Graph G, ein Teilgraph von G und ein induzierter Teilgraph von G.

ist vollständig). Umgekehrt heißt eine Teilmenge $W \subseteq V$ *unabhängige Menge*, wenn keine zwei Knoten aus W durch eine Kante verbunden sind.

Ein Graph $G = (V, E)$ heißt *zusammenhängend*, wenn je zwei Knoten v, w aus G durch einen Kantenzug miteinander verbindbar sind, wenn es also eine Menge von Knoten $v_1, \ldots, v_n \in V$ so gibt, dass $\{v_i, v_{i+1}\} \in E$ für alle $i < n$ und $v = v_1$ und $w = v_n$.

Für jeden Knoten v von G bezeichnen wir mit $G - v$ den Graphen, der aus G durch Löschen von v entsteht. Damit werden dann auch alle Kanten, die durch v gehen, gelöscht, also

$$\begin{aligned} V(G - v) &:= V \backslash \{v\} \quad \text{und} \\ E(G - v) &:= E \backslash \{e \in E; v \in e\}. \end{aligned}$$

Analog entsteht $G - e$ aus G durch Löschen der Kante e, also

$$\begin{aligned} V(G - e) &:= V \quad \text{und} \\ E(G - e) &:= E \backslash \{e\}. \end{aligned}$$

Sei $G = (V, E)$ ein Graph mit $V = \{v_1, \ldots, v_n\}$ und $E = \{e_1, \ldots, e_m\}$. Dann

heißt die Matrix

$$I_G = (a_{ij})_{\substack{i\in\{1,\dots,n\}\\ j\in\{1,\dots,m\}}} \text{ mit } a_{ij} = \begin{cases} 1, & \text{falls } v_i \in e_j, \\ 0, & \text{sonst} \end{cases}$$

Inzidenzmatrix von G. Beispielsweise erhalten wir für den in Abbildung 1.6 betrachteten Graphen die Inzidenzmatrix aus Tabelle 1.1.

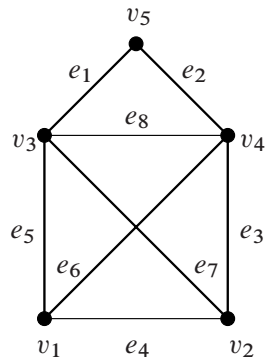

Abbildung 1.6. Der Graph zur Inzidenzmatrix in Tabelle 1.1.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
v_1	0	0	0	1	1	1	0	0
v_2	0	0	1	1	0	0	1	0
v_3	1	0	0	0	1	0	1	1
v_4	0	1	1	0	0	1	0	1
v_5	1	1	0	0	0	0	0	0

Tabelle 1.1. Die Inzidenzmatrix zum Graphen aus Abbildung 1.6.

In der Darstellung von Graphen mit Hilfe der Inzidenzmatrix haben wir eine schöne Kodierung von Graphen über Elemente von $\{0,1\}^*$. In diesem Fall ist die Länge der Kodierung eines Graphen $G = (V, E)$ also $|V| \cdot |E|$. Allerdings ergibt es keinen Sinn, neben der Information $v_i \in e_j$ (in der Inzidenzmatrix steht dann in der i-ten Zeile der j-ten Spalte eine 1) auch noch die Information $v_i \notin e_j$ zu kodieren. Es reicht, nur dann zu kodieren, wenn zwei Knoten v_i und v_j verbunden sind. Dies führt zur Definition der *Adjazenzmatrix* A_G eines Graphen G, d. h.

$$A_G = (a_{ij})_{\substack{i\in\{1,\dots,n\}\\ j\in\{1,\dots,n\}}} \text{ mit } a_{ij} = \begin{cases} 1, & \text{falls } \{v_i, v_j\} \in E, \\ 0, & \text{sonst} \end{cases}$$

Wie man leicht sieht, gilt außerdem $A_G = {I_G}^{\mathrm{t}} I_G$.

Auch in der Adjazenzmatrix gibt es noch redundante Informationen. So steht in A_G neben der Information, ob $\{u, v\} \in E$ auch, ob $\{v, u\} \in E$. Darüber hinaus wissen wir bereits, dass $a_{ii} = 0$ gilt, diese Werte benötigen wir nicht. Die Kodierungslänge eines Graphen ist damit $(|V|^2 - |V|)/2$.

Di- und Multigraphen

Zusätzlich zu den im letzten Unterabschnitt eingeführten endlichen Graphen gibt es noch Erweiterungen dieses Begriffes, die ebenfalls eine wichtige Rolle spielen.

Ein *gerichteter Graph* $G = (V, E)$, auch Digraph genannt, besteht aus einer endlichen Menge V (den Knoten) und einer Menge $E \subseteq V \times V$ (den gerichteten Kanten). Ist $e = (v, w) \in E$ eine Kante, so nennen wir v den *Anfangsknoten* und w den *Endknoten* von e. G heißt *gewichtet*, wenn zusätzlich eine Funktion $w : E \to \mathbb{N} \cup \{\infty\}$ gegeben ist, die jeder Kante $e \in E$ eine Länge bzw. ein Gewicht $w(e)$ zuordnet.

Beispiel 1.9. Wir betrachten als Beispiel einen Graphen $G = (V, E)$

$$\begin{aligned} V &= \{u, v, w\}, \\ E &= \{(u, v), (v, u), (u, w), (w, v)\} \end{aligned}$$

in Abbildung 1.7.

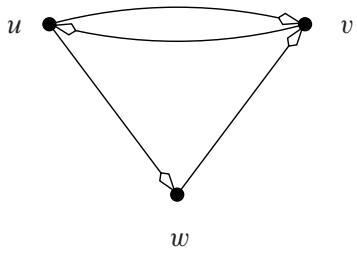

Abbildung 1.7. Der Graph aus Beispiel 1.9.

Obwohl die obige Definition häufig in der Literatur zu finden ist, hat sie den Nachteil, dass man damit nicht beschreiben kann, dass mehrere gerichtete Kanten von einem Knoten zu einem zweiten führen können. Wir wollen daher die folgende Verallgemeinerung einführen: Ein *gerichteter Graph* G ist ein Paar (V, E) von disjunkten Mengen zusammen mit zwei Abbildungen

$$\text{init} : E \longrightarrow V \quad \text{und} \quad \text{ter} : E \longrightarrow V.$$

Ist $e \in E$ eine Kante, so heißt $\text{init}(e)$ Anfangsknoten (*initial vertex*) und $\text{ter}(e)$ Endknoten (*terminal vertex*). Man beachte, dass hier zwei Knoten durch mehrere Kanten verbunden sein können, in diesem Fall heißt der Graph *Multigraph*. Zwei Kanten e_1

und e_2 heißen *parallel*, wenn $\text{init}(e_1) = \text{init}(e_2)$ und $\text{ter}(e_1) = \text{ter}(e_2)$. Eine Kante $e \in E$ mit $\text{init}(e) = \text{ter}(e)$ nennen wir auch *Loop*.

Ein *Multigraph* ist ein Paar $G = (V, E)$ von disjunkten Mengen und einer Abbildung

$$\text{inz} : E \longrightarrow V \cup \{\{u, v\}; u \neq v \in V\}.$$

Zwei Knoten u, v sind durch eine Kante $e \in E$ verbunden, wenn $\text{inz}(e) = \{u, v\}$. Eine Kante $e \in E$ mit $|\text{inz}(e)| = 1$ heißt auch Loop und zwei Kanten $e_1, e_2 \in E$ mit $\text{inz}(e_1) = \text{inz}(e_2)$ parallel, siehe Abbildung 1.8. Man kann sich also einen Multigraphen auch als Digraphen vorstellen, bei dem die Orientierung der Kanten gelöscht wurde.

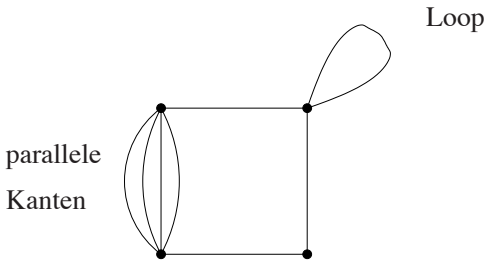

Abbildung 1.8. Beispiel eines Graphen mit Loops und parallelen Kanten.

1.3 Übungsaufgaben

Übung 1.10. Gegeben sei folgendes Problem:

Problem 1.11 (SORTING).
Eingabe: n natürliche Zahlen $m_1, \ldots, m_n \in \mathbb{N}$.
Ausgabe: Eine Folge $i_1, \ldots, i_n$ so, dass

$$m_{i_1} \leq m_{i_2} \leq \cdots \leq m_{i_n}.$$

Zeigen Sie, dass dieses Problem in polynomieller Zeit, genauer in Zeit $\mathcal{O}(n \log n)$, gelöst werden kann.

Übung 1.12. Geben Sie eine Instanz I des Problems MIN JOB SCHEDULING an, für die der Algorithmus LIST SCHEDULE eine Güte von $(2 - 1/m)$ garantiert, wobei m die Anzahl der Maschinen in I ist.

Übung 1.13. Zeigen Sie, dass der in diesem Kapitel kennen gelernte Algorithmus LOCAL IMPROVEMENT für das Problem MAXCUT polynomielle Laufzeit hat.

Übung 1.14. Überlegen Sie sich, ob die Analyse des Algorithmus LOCAL IMPROVEMENT für das Problem MAXCUT bestmöglich war, d. h., beantworten Sie die Frage, ob ein Graph $G = (V, E)$ so existiert, dass der Algorithmus LOCAL IMPROVEMENT eine zulässige Lösung $S \subseteq V$ mit $w(S) = \frac{1}{2} \cdot \text{OPT}(G)$ findet.

Übung 1.15. Wir haben in diesem Abschnitt bereits das Problem MAXCUT kennen gelernt. Betrachten wir anstelle von Graphen gewichtete Graphen, so kann man das Problem wie folgt umformulieren.

Problem 1.16 (WEIGHTED MAXCUT).
Eingabe: Ein gewichteter Graph $G = (V, E)$ mit einer Gewichtsfunktion $w : E \longrightarrow \mathbb{Q}_+$ auf E.
Ausgabe: Eine Partition $(S, V \setminus S)$ der Knotenmenge so, dass die Größe $w(S)$ des *Schnittes*, also die Summe der Gewichte der Kanten zwischen S und $V \setminus S$, maximiert wird.

Versuchen Sie, einen polynomiellen Approximationsalgorithmus für dieses Problem zu konstruieren.

Kapitel 2
Die Komplexitätsklassen P und NP

Ziel dieses Kapitels ist es, eine Entscheidungsgrundlage dafür in die Hand zu bekommen, wann ein Optimierungsproblem nicht polynomiell lösbar ist, d. h., wann es keinen polynomiellen Algorithmus gibt, der optimale Lösungen konstruiert.

Offensichtlich ist ein Optimierungsproblem mindestens so schwer wie das dazugehörige Entscheidungsproblem. Haben wir nämlich einen polynomiellen Algorithmus für das Optimierungsproblem, so liefert uns der Algorithmus unmittelbar einen polynomiellen Algorithmus für die Entscheidungsvariante (vergleiche Satz 2.24). Um also zu zeigen, dass ein Optimierungsproblem schwer ist, genügt es zu zeigen, dass dies für das zugehörige Entscheidungsproblem gilt.

Wir werden uns also bei der Definition der Klassen P und NP auf Entscheidungsprobleme beschränken. Damit wir die Probleme noch ein wenig besser handhaben können, schränken wir noch etwas ein und betrachten zunächst im ersten Abschnitt dieses Kapitels die zu Entscheidungsproblemen assoziierten Wortprobleme, die wie folgt definiert sind: Sei Π ein Entscheidungsproblem, $\mathcal{I}$ die Menge der Instanzen von Π und $Y_\Pi \subseteq \mathcal{I}$ die Menge der JA-Instanzen, kodiert über einem Alphabet Σ. Dabei ist eine Kodierung eine injektive Abbildung

$$e : \mathcal{I} \longrightarrow \Sigma^*$$

von der Menge der Instanzen in die Menge der Wörter

$$\Sigma^* := \{(a_1, \ldots, a_n); n \in \mathbb{N} \text{ und } a_i \in \Sigma \text{ für alle } i \leq n\}$$

mit Buchstaben aus Σ. Im letzten Abschnitt haben wir gesehen, dass es für die Entscheidung, ob ein Algorithmus polynomielle Laufzeit hat, nicht darauf ankommt, über welches Alphabet wir kodieren. Wir können daher o.B.d.A. $\Sigma = \{0, 1\}$ annehmen.

Das Entscheidungsproblem Π liefert uns dann mit dieser Formulierung eine Partition der Menge Σ^* in drei Teile:

(i) $\Sigma^* \setminus \mathcal{I}$: Strings aus Σ^*, die keine Kodierungen von Instanzen sind.

(ii) $\mathcal{I} \setminus Y_\Pi$: Strings aus Σ^*, die Kodierungen von NEIN-Instanzen sind.

(iii) Y_Π: Strings aus Σ^*, die Kodierungen von JA-Instanzen sind.

Diese letzte Menge

$$L(\Pi, e) := \{x \in \Sigma^*;\ x \text{ ist die Kodierung einer Instanz } I \in Y_\Pi \text{ bezüglich } e\}$$

heißt die zu Π *assoziierte Sprache*. Ist die Kodierung bekannt, oder spielt keine Rolle, so schreiben wir auch kurz $L(\Pi)$.

Wie bereits beschrieben, werden wir die Komplexitätsklassen P und NP im ersten Abschnitt dieses Kapitels zunächst nur für Sprachen definieren, bevor wir in Abschnitt 2.2 die Definitionen auch für Entscheidungsprobleme einführen. Komplexitätsklassen für Optimierungsprobleme werden wir dann erst in Kapitel 17 behandeln.

Das zentrale Thema in diesem Abschnitt ist die Definition von NP-vollständigen Entscheidungsproblemen. Wir zeigen, dass Optimierungsprobleme, deren Entscheidungsvarianten NP-vollständig sind, nicht in polynomieller Zeit gelöst werden können, jedenfalls unter der Voraussetzung P $\neq$ NP. Dieser Satz ist die zentrale Motivation für die in diesem Buch betrachteten Approximationsalgorithmen.

Das erste Entscheidungsproblem, von dem wir zeigen werden, dass es NP-vollständig ist, ist das sogenannte SAT-Problem (Satisfiability), das in Abschnitt 2.3 behandelt wird. Darauf aufbauend sind wir dann in der Lage, im letzten Abschnitt dieses Kapitels von einer ganzen Reihe von Problemen, die uns im Laufe der Lektüre des Buches immer wieder begegnen werden, die NP-Vollständigkeit nachzuweisen.

2.1 Sprachen (Wortprobleme) und die Klassen P und NP

Einfach gesagt ist eine Sprache $L \subseteq \Sigma^*$ polynomiell lösbar, also aus P, wenn das Wortproblem

Problem 2.1 (WORTPROBLEM L).
Eingabe: Ein Wort $w \in \Sigma^*$.
Frage: Gilt $w \in L$?

in polynomieller Zeit entscheidbar ist, es also einen polynomiellen Algorithmus für dieses Problem gibt. Formal gesehen ist ein Algorithmus eine Turingmaschine. Wir sagen, dass eine Turingmaschine $\mathfrak{M}$ eine Sprache L akzeptiert, wenn L die von $\mathfrak{M}$ erkannte Sprache ist, siehe folgende Definition. Diejenigen Leserinnen und Leser, die noch nicht mit dem Begriff der Turingmaschine vertraut sind, seien auf Anhang A verwiesen, in dem wir eine kurze Einführung in dieses Berechnungsmodell geben.

Definition 2.2. Sei $\mathfrak{M}$ eine Turingmaschine.

(i) Ein Wort $w \in \Sigma^*$ wird von $\mathfrak{M}$ *akzeptiert*, wenn $\mathfrak{M}$ nach Eingabe von w in eine Stoppkonfiguration mit Endzustand läuft (solche Stoppkonfigurationen heißen dann auch *akzeptierende Stoppkonfigurationen*).

(ii) Die von $\mathfrak{M}$ akzeptierten Wörter bilden die Menge der von $\mathfrak{M}$ *erkannten Sprache*.

Die Lauf- bzw. Rechenzeit einer solchen Turingmaschine für eine Eingabe $w \in \Sigma^*$ ist dann die Anzahl der Konfigurationswechsel, die die Maschine für die Berechnung von w benötigt.

Definition 2.3 (Rechenzeit). (i) Sei $\mathfrak{M}$ eine deterministische Turingmaschine. Dann ist $T_{\mathfrak{M}}(w)$ die Anzahl der Konfigurationswechsel, die $\mathfrak{M}$ bei Eingabe von $w \in \Sigma^*$ durchläuft, und

$$T_{\mathfrak{M}} : \mathbb{N} \to \mathbb{N} \cup \{\infty\}; \;\; n \mapsto \max\{T_{\mathfrak{M}}(w); w \in \Sigma^*, |w| = n\}$$

die *Zeitkomplexität* von $\mathfrak{M}$.

(ii) Sei $\mathfrak{M}$ eine nichtdeterministische Turingmaschine. Dann ist $T_{\mathfrak{M}}(w)$ die maximale Anzahl der Konfigurationswechsel, die $\mathfrak{M}$ bei Eingabe von $w \in \Sigma^*$ durchläuft, und

$$T_{\mathfrak{M}} : \mathbb{N} \to \mathbb{N} \cup \{\infty\}; n \mapsto \max\{T_{\mathfrak{M}}(w); w \in \Sigma^*, |w| = n\}$$

die *Zeitkomplexität* von $\mathfrak{M}$.

Wir kommen nun zur Definition der Komplexitätsklassen P und NP.

Definition 2.4. (i) Sei $\phi : \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann ist

$$\mathrm{DTIME}(\phi) = \{L \subseteq \Sigma^*; \text{es ex. eine DTM } \mathfrak{M} \text{ mit } T_{\mathfrak{M}} \in \mathcal{O}(\phi), \text{die } L \text{ erkennt}\}.$$

Weiter ist

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k)$$

die Klasse aller mit einer deterministischen Turingmaschine in polynomieller Zeit lösbaren Wortprobleme.

(ii) Analog ist $\mathrm{NTIME}(\phi)$ die Klasse aller Sprachen $L \in \Sigma^*$, für die es eine nichtdeterministische Turingmaschine $\mathfrak{M}$ mit $T_{\mathfrak{M}} \in \mathcal{O}(\phi)$ gibt, die L erkennt und

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k).$$

Da eine deterministische Turingmaschine insbesondere auch eine nichtdeterministische Turingmaschine ist, folgt unmittelbar

Satz 2.5. *Es gilt*

$$\mathrm{P} \subseteq \mathrm{NP}.$$

Eine der wichtigsten Fragen der algorithmischen Mathematik und der theoretischen Informatik ist, ob auch die Umkehrung gilt, d. h., ob P und NP gleich sind. Trotz vieler Anstrengungen der letzten Jahrzehnte konnte dies bisher nicht beantwortet werden, es wird aber allgemein angenommen, dass P eine echte Teilmenge von NP ist. Wir werden am Ende dieses Kapitels eine Vorstellung davon gewonnen haben, warum dies vermutet wird.

Wir wollen zum besseren Verständnis noch kurz zwei weitere Definitionen der Klasse NP vorstellen, verschieben den Beweis der Äquivalenz jedoch auf Anhang A.

Betrachten wir noch einmal genauer, was eine nichtdeterministische Turingmaschine $\mathfrak{M}$ eigentlich tut. Im Gegensatz zu deterministischen Turingmaschinen ist der Berechnungsweg in nichtdeterministischen Turingmaschinen für eine Eingabe $w \in \Sigma^*$ nicht eindeutig. Genauer, ist $\mathfrak{M}$ eine nichtdeterministische Turingmaschine, so gibt es zu einer Eingabe $w \in \Sigma^*$ verschiedene Berechnungswege, die auch zu verschiedenen Antworten führen können. $\mathfrak{M}$ akzeptiert dann w genau dann als ein Element von $L \subseteq \Sigma^*$, wenn es einen Berechnungsweg in $\mathfrak{M}$ gibt, der als Antwort `wahr` liefert.

Eine Sprache L ist nun aus NP, wenn es eine nichtdeterministische Turingmaschine $\mathfrak{M}$ so gibt, dass $w \in \Sigma^*$ genau dann ein Element von L ist, wenn w in polynomieller Zeit von $\mathfrak{M}$ akzeptiert wird, d. h., wenn es einen Berechnungsweg in $\mathfrak{M}$ gibt, der als Antwort `wahr` ausgibt und nur polynomiell viele Konfigurationswechsel benötigt.

Informell besteht die Klasse NP also aus allen Wortproblemen Π mit der folgenden Eigenschaft:

Ist die Antwort für eine Instanz von Π `wahr`*, dann gibt es dafür einen Beweis von polynomieller Länge.*

Mit anderen Worten, wird ein Wort $w \in L$ von einer nichtdeterministischen Turingmaschine für L akzeptiert, so gibt es eine Berechnung polynomieller Länge der Turingmaschine, die w akzeptiert. Man beachte, dass diese Definition das Finden eines kurzen Beweises (bzw. des Berechnungsweges) nicht verlangt. Zum Beispiel könnte jemand beim Problem MAXCUT einen maximalen Schnitt mit viel Glück, Intuition oder übernatürlicher Kraft finden und diesen aufschreiben. Dies führt dann zu einem kurzen Beweis, dass dieser Schnitt tatsächlich maximal ist.

Die zentrale Frage, ob die beiden Komplexitätsklassen P und NP gleich sind, kann also auch wie folgt formuliert werden: Ist das Finden von Beweisen ähnlich schwierig wie das Verifizieren von Beweisen? Es ist beinahe offensichtlich, dass das Finden eines Beweises bei weitem schwieriger als das Verifizieren ist. Dies ist unter anderem ein Grund dafür, dass P $\neq$ NP angenommen wird.

Die obige Beschreibung der Klasse NP motiviert die folgende formale Charakterisierung:

Satz 2.6. *Die Klasse* NP *besteht aus allen Sprachen L, für die ein polynomiell zeitbeschränkter Algorithmus* A *und ein Polynom p so existieren, dass für alle $x \in \Sigma^*$ gilt:*

$$x \in L \iff \exists w \in \Sigma^* \textit{mit } |w| \leq p(|x|) \textit{ so, dass } \mathsf{A}(x, w) = \texttt{wahr}.$$

Ist $x \in L$, dann heißt $w \in \Sigma^*$ mit $|w| \leq p(|x|)$ und $\mathsf{A}(x, w) = \texttt{wahr}$ ein *Zeuge* für $x \in L$. Den String w können wir dann als (kurzen) Beweis dafür auffassen, dass das Wort x in der Sprache L enthalten ist, allerdings kann, wie oben schon erwähnt, das Finden dieses Zeugen sehr schwer sein. Gilt umgekehrt $x \notin L$, so werden wir keinen Zeugen finden.

Wir werden in Kapitel 19 noch eine weitere Definition der Klasse NP mit Hilfe sogenannter probabilistischer Verifizierer kennen lernen.

Wie bereits oben beschrieben, wird allgemein vermutet, dass die beiden Komplexitätsklassen P und NP verschieden sind, d. h., dass P eine echte Untermenge von NP ist. Eine Idee, warum dies vermutet wird, wollen wir nun kennen lernen. Der Schlüssel dazu liegt in der folgenden Definition.

Definition 2.7. Seien L_1 und L_2 Sprachen über Σ. Eine *polynomielle Reduktion* von L_1 auf L_2 ist eine Funktion $f : \Sigma^* \to \Sigma^*$ so, dass gilt:

(i) Es gibt eine Turingmaschine $\mathfrak{M}$, die f in polynomieller Zeit berechnet.

(ii) Für alle $x \in \Sigma^*$ gilt $x \in L_1$ genau dann, wenn $f(x) \in L_2$.

Gibt es eine polynomielle Reduktion von L_1 auf L_2, so schreiben wir auch $L_1 \leq L_2$.

Anschaulich bedeutet dies, dass L_2 mindestens so schwer ist wie L_1 (jedenfalls in Bezug auf polynomielle Lösbarkeit). Denn wenn es einen polynomiellen Algorithmus für L_2 gibt, so können wir mit Hilfe der in polynomieller Zeit berechenbaren Reduktion f zunächst in polynomieller Zeit prüfen, ob $f(x) \in L_2$, und haben damit insgesamt einen effizienten Algorithmus für das Wortproblem L_1.

Wir formulieren diese Beobachtung im folgenden Satz.

Satz 2.8. *Seien L_1 und L_2 Sprachen über Σ mit $L_1 \leq L_2$. Dann gilt:*

(i) $L_2 \in \mathrm{P} \Longrightarrow L_1 \in \mathrm{P}$ *und*

(ii) $L_2 \in \mathrm{NP} \Longrightarrow L_1 \in \mathrm{NP}$.

Beweis. Wir zeigen nur (i), (ii) zeigt man analog. Sei also $\mathfrak{M}$ eine deterministische Turingmaschine für L_2 und p ein Polynom mit $T_{\mathfrak{M}} \leq p$. Sei weiter $\mathfrak{N}$ eine deterministische Turingmaschine, die die Reduktionsfunktion $f : \Sigma^* \to \Sigma^*$ in polynomieller Zeit berechnet, und q ein Polynom mit $T_{\mathfrak{N}} \leq q$. Durch Kombination der beiden Turingmaschinen erhalten wir also eine Laufzeit für das Wortproblem über L_1 bei der Eingabe von $x \in \Sigma^*$ der Größenordnung

$$p(|f(x)|) + q(|x|).$$

Da die Turingmaschine $\mathfrak{N}$ für die Berechnung von $f(x)$ höchstens $q(|x|)$ Konfigurationswechsel durchführt, gilt

$$|f(x)| \leq q(|x|) + |x|.$$

Weiter können wir o.B.d.A. annehmen, dass die Koeffizienten von p nichtnegative ganze Zahlen sind, p also insbesondere monoton wachsend ist. Wir erhalten damit

$$p(|f(x)|) \leq p(q(|x|) + |x|),$$

also

$$p(|f(x)|) + q(|x|) \leq p(q(|x|) + |x|) + q(|x|).$$

Damit ist dann auch das Wortproblem L_1 in polynomieller Zeit lösbar. □

Definition 2.9. Eine Sprache L heißt NP-*vollständig*, wenn $L \in$ NP und sich jede Sprache aus NP in polynomieller Zeit auf L reduzieren lässt.

Es gilt also

Satz 2.10. *Liegt ein* NP*-vollständiges Problem in* P*, so folgt* P = NP.

Beweis. Dies folgt sofort aus Satz 2.8 (i). □

NP-vollständige Probleme sind also die am schwierigsten zu lösenden Sprachen in NP. Nun stellt sich natürlich die Frage, wie schwer NP-vollständige Probleme denn wirklich sind. Die Antwort ist, wie oben schon geschrieben, unbekannt. Es gibt allerdings inzwischen über 1000 Probleme aus NP, von denen man weiß, dass sie NP-vollständig sind, und für keines ist ein polynomieller Algorithmus bekannt. Es wird deshalb vermutet, dass P $\neq$ NP gilt. Unter dieser Voraussetzung ergibt sich also die Abbildung 2.1.

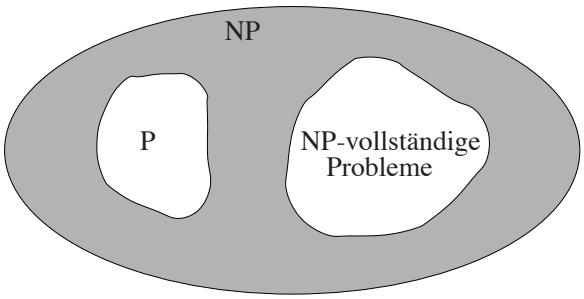

Abbildung 2.1. Die Komplexitätsklassen P und NP unter der Voraussetzung P $\neq$ NP.

Bevor wir gleich im übernächsten Abschnitt das erste NP-vollständige Problem kennen lernen, wollen wir die Klassen P und NP auch für Entscheidungsprobleme definieren.

2.2 Entscheidungsprobleme und die Klassen P und NP

Die meisten Entscheidungsprobleme sind zunächst nicht, im Gegensatz zu den im letzten Abschnitt betrachteten Wortproblemen, in einer Form gegeben, so dass ihre Instanzen von einem Algorithmus bzw. einer Turingmaschine entschieden werden können. Wir müssen also zunächst die Instanzen eines Entscheidungsproblems in eine

für Algorithmen verständliche Sprache übersetzen. Dies wird mit sogenannten Kodierungen gemacht, die wir schon in der Einleitung zu diesem Kapitel definiert haben.

Allerdings stellen wir an Kodierungen eine zentrale Bedingung, nämlich die, dass das Bild der Instanzenmenge unter der Kodierung eine Sprache aus P ist. Solche Kodierungen nennen wir im Folgenden auch zulässig. Dies heißt insbesondere, dass man in polynomieller Zeit entscheiden kann, ob ein Wort tatsächlich eine Instanz des Entscheidungsproblems kodiert. Da wir ja von vornherein wissen, welche Wörter wir von einer Turingmaschine entscheiden lassen wollen (nämlich nur diejenigen Wörter, die auch Instanzen des Problems kodieren), ist diese Bedingung keine wirkliche Einschränkung. Die Definition zulässiger Kodierungen hat zum einen den Vorteil, dass wir bei der Betrachtung von Algorithmen für Entscheidungsprobleme nur noch Eingaben betrachten müssen, die Instanzen des Problems sind. Auf der anderen Seite erhalten wir als Ergebnis, dass für ein polynomiell entscheidbares Problem dann auch die Menge der JA-Instanzen eine Sprache aus P im Sinne der im letzten Abschnitt angegebenen Definition ist, siehe Lemma 2.13.

Definition 2.11. Sei $e : \mathcal{I} \to \Sigma^*$ eine Kodierung eines Optimierungs- oder Entscheidungsproblems Π mit der Instanzmenge $\mathcal{I}$. Dann heißt *e zulässige Kodierung*, wenn für alle $x \in \Sigma^*$ in polynomieller Zeit entschieden werden kann, ob x eine Instanz von Π kodiert, d. h., wenn $e(\mathcal{I}) \in \mathrm{P}$.

Für den Rest dieses Kapitels nehmen wir stets an, dass Probleme zulässig kodiert sind.

Die folgende Definition überträgt die im letzten Abschnitt eingeführten Komplexitätsklassen P und NP auf Entscheidungsprobleme.

Definition 2.12. Sei $\Pi = (\mathcal{I}, Y_\Pi)$ ein Entscheidungsproblem.

(i) Das Entscheidungsproblem Π ist aus P, wenn es einen polynomiell zeitbeschränkten Algorithmus A so gibt, dass $\mathsf{A}(I) = \texttt{wahr}$ genau dann gilt, wenn $I \in Y_\mathcal{I}$.

(ii) Das Entscheidungsproblem Π ist aus NP, wenn es einen polynomiell zeitbeschränkten Algorithmus A und ein Polynom p so gibt, dass gilt

$$I \in Y_\Pi \iff \exists y \in \Sigma^* \text{mit } |y| \leq p(|x|) \text{ so, dass } \mathsf{A}(x, y) = \texttt{wahr}.$$

Offensichtlich gilt dann

Lemma 2.13. *Sei Π ein Entscheidungsproblem. Genau dann ist Π aus* P *(bzw.* NP*), wenn die zu Π assoziierte Sprache $L(\Pi) = Y_\Pi$ aus* P *(bzw.* NP*) ist.*

Beweis. Wir zeigen nur die erste Aussage, die zweite zeigt man wieder analog.

Sei also B ein polynomieller Algorithmus für Π. Da Π zulässig kodiert ist, existiert ein polynomieller Algorithmus A, der für alle $x \in \Sigma^*$ entscheidet, ob x eine Instanz

von Π kodiert. Insgesamt können wir unseren Algorithmus C für $L(\Pi)$ also wie folgt formulieren:

Algorithmus $\mathsf{C}(x)$

```
1  if A(x) = falsch then
2    return falsch
3  else
4    if B(x) = falsch then
5      return falsch
6    else
7      return wahr
8    fi
9  fi
```

Es gelte nun umgekehrt $L(\Pi) \in$ P. Da dann für $L(\Pi)$ ein polynomieller Algorithmus existiert, der für alle $x \in \Sigma^*$ entscheidet, ob $x \in L(\Pi)$, entscheidet die Einschränkung dieses Algorithmus auf Instanzen von Π auch Π. □

Man beachte, dass wir die Äquivalenz im obigen Lemma nur mit der Voraussetzung, dass die Probleme zulässig kodiert sind, beweisen konnten. Ohne diese Voraussetzungen können unerwartete Ergebnisse vorkommen, wie das folgende Beispiel zeigt.

Beispiel 2.14. Sei $L \subseteq \Sigma^*$ eine Sprache, die nicht polynomiell lösbar ist, und $w \in L$. Dann ist das folgende Entscheidungsproblem polynomiell lösbar.

Problem 2.15 (L_w)**.**
Eingabe: $x \in L$.
Frage: Ist $x \in L \backslash \{w\}$?

Offensichtlich lässt sich in polynomieller Zeit testen, ob $x = w$ gilt. Damit ist auch das obige Entscheidungsproblem gelöst.

Die zum Problem L_w assoziierte Sprache ist

$$L(L_w) = L \backslash \{w\}.$$

Da L nicht polynomiell lösbar ist, gilt dies natürlich auch für $L \backslash \{w\}$.

Wir wollen nun zeigen, dass das Cliquenproblem aus NP ist.

Problem 2.16 (CLIQUE)**.**
Eingabe: Ein Graph $G = (V, E)$, eine Zahl $k \in \mathbb{N}$.
Frage: Existiert eine Clique $C \subseteq V$ mit $|C| \geq k$?

Lemma 2.17. *Es gilt*

$$\text{CLIQUE} \in \text{NP}.$$

Beweis. Wir betrachten folgenden Algorithmus für das Problem, der als Eingabe einen Graphen $G = (V, E)$, eine Zahl $k \in \mathbb{N}$ und eine Knotenmenge $C \subseteq V$ erwartet:

Algorithmus IsClique$(G = (V, E), k, C)$

```
1  if |C| ≤ k then
2    return falsch
3  fi
4  for v ∈ C do
5    for w ∈ C\{v} do
6      if {v, w} ∉ E then
7        return falsch
8      fi
9    od
10 od
11 return wahr
```

Der Algorithmus testet also zunächst, ob die Kardinalität der Knotenmenge C überhaupt der Minimalforderung $|C| \geq k$ genügt, um danach zu überprüfen, ob C auch eine Clique bildet.

Da der Algorithmus offensichtlich polynomiell zeitbeschränkt ist, folgt die Behauptung. □

Der obige Algorithmus ist ein typisches Beispiel dafür, wie man von einem Entscheidungsproblem zeigen kann, dass es von einer nichtdeterministischen Turingmaschine entschieden werden kann. Man wähle sich zunächst einen Kandidaten, der den Anforderungen genügen könnte (im obigen Beispiel die Knotenmenge C als Kandidat für eine Clique im Graphen der Größe mindestens k), und überprüfe dann in polynomieller Zeit, ob dieser Kandidat den Anforderungen tatsächlich genügt, d. h., man versucht nicht, einen Beweis zu finden, sondern nur, Beweise in kurzer Zeit zu verifizieren.

Ähnlich wie die Klassen P und NP lässt sich auch die Definition der NP-Vollständigkeit auf Entscheidungsprobleme übertragen. Wir beginnen mit der Definition der polynomiellen Reduktion.

Definition 2.18. Seien $\Pi_1 = (\mathcal{I}_1, Y_{\mathcal{I}_1})$ und $\Pi_2 = (\mathcal{I}_2, Y_{\mathcal{I}_2})$ zwei Entscheidungsprobleme. Eine *polynomielle Reduktion* von Π_1 auf Π_2 ist eine Abbildung $f : \mathcal{I}_1 \longrightarrow \mathcal{I}_2$ so, dass gilt:

(i) Es gibt eine Turingmaschine $\mathfrak{M}$, die f in polynomieller Zeit berechnet.

(ii) Für alle $I \in \mathcal{I}_1$ gilt $I \in Y_{\mathcal{I}_1}$ genau dann, wenn $f(I) \in Y_{\mathcal{I}_2}$.

Gibt es eine polynomielle Reduktion von Π_1 auf Π_2, so schreiben wir auch $\Pi_1 \leq \Pi_2$.

Man beachte, dass sich ein Entscheidungsproblem Π_1 genau dann polynomiell auf ein zweites Entscheidungsproblem Π_2 reduzieren lässt, wenn dies für die zu den Problemen assoziierten Sprachen $L(\Pi_1)$ und $L(\Pi_2)$ gilt, siehe Übungsaufgabe 2.48. Die im letzten Abschnitt gezeigten Sätze für Sprachen gelten also auch für Entscheidungsprobleme, und wir erhalten

Satz 2.19. *Seien* Π_1 *und* Π_2 *Entscheidungsprobleme mit* $\Pi_1 \leq \Pi_2$. *Dann gilt:*

(i) $\Pi_2 \in \mathrm{P} \Longrightarrow \Pi_1 \in \mathrm{P}$ *und*

(ii) $\Pi_2 \in \mathrm{NP} \Longrightarrow \Pi_1 \in \mathrm{NP}$.

Schließlich ist die Definition der NP-Vollständigkeit für Entscheidungsprobleme ganz analog zu der für Sprachen.

Definition 2.20. (i) Ein Entscheidungsproblem Π heißt NP-*vollständig*, wenn $\Pi \in \mathrm{NP}$ und sich jedes Entscheidungsproblem aus NP in polynomieller Zeit auf Π reduzieren lässt.

(ii) Ein Entscheidungsproblem Π heißt NP-*schwer*, wenn sich jedes Entscheidungsproblem aus NP in polynomieller Zeit auf Π reduzieren lässt.

Analog zu Satz 2.10 gilt nun:

Satz 2.21. *Liegt ein* NP-*vollständiges Entscheidungsproblem in* P*, so folgt* $\mathrm{P} = \mathrm{NP}$.

Erinnern wir uns an die Einleitung zu diesem Kapitel, so war das Ziel der Untersuchung der Klassen P und NP, eine Möglichkeit dafür in die Hand zu bekommen, von Optimierungsproblemen zu zeigen, dass diese nicht in polynomieller Zeit optimal gelöst werden können, es sei denn $\mathrm{P} = \mathrm{NP}$. Die Schwere eines Optimierungsproblems, d. h. die Frage, ob ein Optimierungsproblem in polynomieller Zeit gelöst werden kann, lässt sich an der Schwere des zugehörigen Entscheidungsproblems ablesen. Wir werden zunächst, obwohl in Kapitel 1 bereits beschrieben, daran erinnern, wie wir das zu einem Optimierungsproblem zugehörige Entscheidungsproblem definiert haben.

Definition 2.22. Sei $\Pi = (\mathcal{I}, F, w)$ ein Optimierungsproblem. Dann heißt $\Pi' = (\mathcal{I}', Y_{\mathcal{I}'})$ mit $\mathcal{I}' = \mathcal{I} \times \mathbb{Q}$ und

$$Y_{\mathcal{I}'} = \{(I, x) \in \mathcal{I}'; \text{es existiert } t \in F(I) \text{ mit } w(t) \geq x\}$$

bzw.

$$Y_{\mathcal{I}'} = \{(I, x) \in \mathcal{I}'; \text{es existiert } t \in F(I) \text{ mit } w(t) \leq x\}$$

das zu Π *zugehörige Entscheidungsproblem*, wenn Π ein Maximierungs- bzw. ein Minimierungsproblem ist.

Die nächste Definition charakterisiert, wie Satz 2.24 zeigt, diejenigen Optimierungsprobleme, die unter der Voraussetzung $P \neq NP$ nicht in polynomieller Laufzeit gelöst werden können.

Definition 2.23. Sei Π ein Optimierungsproblem und Π' das zugehörige Entscheidungsproblem. Dann heißt Π NP-*schwer*, wenn Π' NP-schwer ist.

Der folgende Satz ist die zentrale Motivation für die Betrachtung von Approximationsalgorithmen.

Satz 2.24. *Unter der Voraussetzung* $P \neq NP$ *existieren für* NP*-schwere Optimierungsprobleme keine optimalen polynomiellen Algorithmen.*

Beweis. Sei $\Pi = (\mathcal{I}, F, w)$ NP-schwer. Angenommen, es existiert ein optimaler polynomieller Algorithmus A für Π. Dann gibt es offensichtlich auch einen polynomiellen Algorithmus für das zugehörige Entscheidungsproblem Π' (man berechne erst den optimalen Wert und vergleiche diesen dann mit der Zahl in der Eingabe). Nach Satz 2.21 gilt dann $P = NP$, ein Widerspruch. □

2.3 Das Problem SAT und der Satz von Cook

Bisher wissen wir noch nicht, ob NP-vollständige Probleme überhaupt existieren. Wir werden deshalb in diesem Abschnitt einige kennen lernen. Zunächst starten wir mit dem berühmten Satz von Cook, der zeigt, dass das Problem SAT (Satisfiability), d. h. die Frage, ob eine aussagenlogische Formel in konjuktiver Normalform erfüllbar ist, NP-vollständig ist. Dieser Satz wurde von Cook 1971 (siehe [44]) und unabhängig davon von Levin 1973 (siehe [143]) bewiesen.

Aussagenlogische Formeln bestehen aus booleschen Variablen $x_1, x_2, \ldots$ und Operatoren $\wedge, \vee, \neg$. Die Variablen x_i und $\neg x_i$ heißen auch *Literale*. $(y_1 \vee \cdots \vee y_k)$ mit Literalen $y_1, \ldots, y_k$ heißt *Klausel* und $\alpha = c_1 \wedge \cdots \wedge c_m$ mit Klauseln $c_1, \ldots, c_m$ auch Ausdruck in *konjunktiver Normalform*. Wir sagen, dass ein Ausdruck α *erfüllbar* ist, wenn es eine *Belegung* der Variablen in α mit den Werten `wahr` und `falsch` so gibt, dass α den Wert `wahr` bekommt. Ist α eine Formel über den Variablen $x_1, \ldots, x_n$, so ist also eine Belegung nichts anderes als eine Abbildung

$$\mu : \{x_1, \ldots, x_n\} \longrightarrow \{\texttt{wahr}, \texttt{falsch}\}.$$

Beispiel 2.25. (i) Wir betrachten die Formel

$$\alpha = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3).$$

Dann ist μ mit $\mu(x_1) = \mu(x_2) = \mu(x_3) = \texttt{wahr}$ eine erfüllende Belegung.

(ii) Für die Formel

$$\begin{aligned}\alpha \;=\; & (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \\ & (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge \\ & (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)\end{aligned}$$

gibt es offensichtlich keine erfüllende Belegung.

Damit können wir das in diesem Abschnitt zu behandelnde Problem wie folgt formulieren.

Problem 2.26 (SAT).
Eingabe: Eine aussagenlogische Formel α in konjunktiver Normalform.
Frage: Ist α erfüllbar?

Satz 2.27 (Satz von Cook (1971)). SAT *ist* NP*-vollständig*.

Beweis. Ähnlich wie schon bei dem Problem CLIQUE lässt sich zeigen, dass auch das Problem SAT aus NP ist. Man kann ja leicht von einer gegebenen Belegung in polynomieller Zeit prüfen, ob diese alle Klauseln erfüllt.

Wir müssen nun nachweisen, dass jedes Problem aus NP in polynomieller Zeit auf SAT reduzierbar ist. Allerdings wollen wir zunächst zum besseren Verständnis die Hauptideen vorstellen, ohne uns mit zu genauen Betrachtungen der zugrunde liegenden Formalien den Blick für das Wesentliche zu verstellen.

Sei also Π ein Entscheidungsproblem aus NP und A ein polynomieller Algorithmus so, dass für alle `wahr`-Instanzen $I = (I_1, \ldots, I_n)$ von Π ein Zeuge $y_I = (y_1^I, \ldots, y_{p(n)}^I)$ so existiert, dass A für die Eingabe (I, y_I) `wahr` liefert, und für alle `falsch`-Instanzen I von Π gilt $\mathsf{A}(I, y) = \texttt{falsch}$ für alle y mit $|y| \leq p(|I|)$ für ein Polynom p. Wir müssen nun zeigen, wie sich Π in polynomieller Zeit auf SAT reduzieren lässt.

Der Algorithmus A lässt sich in eine boolesche Formel umwandeln. Dies ist nicht weiter verwunderlich, da Computer, in denen Algorithmen ablaufen, ja nur boolesche Operationen ausführen können. Allerdings benötigt diese Aussage, wie wir gleich sehen werden, eine genaue Analyse der den Algorithmen zugrundeliegenden Theorie der Turingmaschinen.

Diese dem Algorithmus A zugeordnete boolesche Formel lässt sich in eine Formel in konjunktiver Normalform umwandeln. Wir bezeichnen die so erhaltene Formel mit

$$\alpha = c_1 \wedge \cdots \wedge c_m$$

mit Klauseln $c_1, \ldots, c_m$ über der Variablenmenge

$$X = \{x_1, \ldots, x_n, y_1, \ldots, y_{p(n)}\}.$$

Damit erhalten wir $\mathsf{A}(I, y_I) = \texttt{wahr}$ genau dann, wenn α mit der Variablenbelegung $x_i = I_i$ für alle $i \leq |I|$ und $y_i = y_i^I$ für alle $i \leq p(|I|)$ erfüllt ist.

Für jede Eingabe I von Π betrachtet man die Formel α_I, die aus α entsteht, indem die Variablen $x_1, \ldots, x_n$ mit den Werten $I_1, \ldots, I_n$ belegt sind. Es lässt sich zeigen, dass α_I in polynomieller Zeit aus I berechnet werden kann. Offensichtlich gilt nun: I ist genau dann eine `wahr`-Instanz von Π, wenn es eine erfüllende Belegung für α_I gibt. Damit haben wir aber Π auf SAT reduziert. Da Π beliebig gewählt wurde, folgt die Behauptung.

Wir kommen nun zum formalen Beweis. Sei also im Folgenden L eine Sprache, die von einer nichtdeterministischen Turingmaschine $\mathfrak{M}$ in polynomieller Zeit berechnet werden kann. Wir werden $L \leq \text{SAT}$ zeigen, genauer konstruieren wir eine polynomielle Transformation

$$f : \Sigma^* \longrightarrow \Sigma^*$$

so, dass $f(u)$ für alle $u \in \Sigma^*$ eine Instanz von SAT ist. Weiter muss f folgende Eigenschaft besitzen: Ein Wort $u \in \Sigma^*$ ist genau dann ein Element von L, wenn $f(u)$ eine erfüllende Belegung besitzt.

Sei $\mathfrak{M} = (Q, \Gamma, q_0, \delta, F)$ eine nichtdeterministische Turingmaschine für L mit Zustandsmenge $Q = \{q_0, \ldots, q_s\}$, Endzuständen $F = \{q_r, \ldots, q_s\}$, Anfangszustand q_0, Arbeitsalphabet $\Gamma = \{a_0, \ldots, a_m\}$, Blanksymbol $a_0 = \mathbf{b}$ und der Übergangsfunktion $\delta : (Q \backslash F) \times \Gamma \longrightarrow 2^{\Gamma \times \{l,r\} \times Q}$, die für alle $u \in \Sigma^*$ nach höchstens $T(|u|)$ Schritten eine Stoppkonfiguration erreicht.

Weiter gelte o.B.d.A.: Akzeptiert $\mathfrak{M}$ die Eingabe $u = a_{j_1} \cdots a_{j_{|u|}}$ mit einer Berechnung der Länge höchstens $T(|u|)$, so akzeptiert $\mathfrak{M}$ die Eingabe u mit einer Berechnung der Länge genau $T(|u|)$ (man kann immer künstlich Konfigurationswechsel einführen, bis die Anzahl der Wechsel den Wert $T(|u|)$ erreicht).

Dabei stellen wir, wie üblich, δ durch eine Folge von Zeilen (Quintupeln)

$$z^i = (z_1^i, \ldots, z_5^i) \in (Q \backslash F) \times \Gamma \times \Gamma \times \{l, r\} \times Q$$

für alle $i \leq \varrho$ dar, wobei ϱ die Anzahl der Zeilen von $\mathfrak{M}$ sei. Dabei bedeute $qaa'\beta q'$, dass $(a', \beta, q') \in \delta(q, a)$. Wir werden im Folgenden die Menge $\{l, r\}$, was ja „bewege den Schreib-/Lesekopf nach links/rechts“ bedeutet, mit $\{-1, 1\}$ identifizieren. Weiter betrachten wir nur Turingmaschinen, die ein von links beschränktes Feld haben.

Wir beginnen nun, die boolesche Formel α zu konstruieren. Dabei soll $\alpha := f(u)$ die Berechnung, die $\mathfrak{M}$ bei Eingabe von u durchführt, simulieren.

Sei dazu die Variablenmenge X wie folgt definiert:

$$X := \{z_{tk}, s_{ti}, b_{tl}, a_{tij}; 0 \leq t, i \leq T(|u|), 0 \leq k \leq s, 1 \leq l \leq \varrho, 0 \leq j \leq m\}.$$

Zur besseren Lesbarkeit des folgenden Beweises wollen wir die Variablen in geeig-

neter Weise interpretieren:

$$
\begin{aligned}
z_{tk} &\mathrel{\hat{=}} \text{Nach } t \text{ Schritten wird Zustand } q_k \text{ erreicht.}\\
s_{ti} &\mathrel{\hat{=}} \text{Nach } t \text{ Schritten steht der Schreib-/Lesekopf auf Arbeitsfeld } i.\\
b_{tl} &\mathrel{\hat{=}} \text{Nach } t \text{ Schritten wird Zeile } z^l \text{ ausgeführt.}\\
a_{tij} &\mathrel{\hat{=}} \text{Nach } t \text{ Schritten steht auf Arbeitsfeld } i \text{ der Buchstabe } a_j.
\end{aligned}
$$

Die Anzahl der Variablen ist offensichtlich nach oben durch $c \cdot T(|u|)^2$ beschränkt, wobei c eine von $\mathfrak{M}$ abhängige Konstante ist.

Die gesuchte Formel α besteht nun aus acht Teilformeln

$$
\alpha := \underbrace{\alpha_1}_{\text{Anfang}} \wedge \underbrace{\alpha_2 \wedge \alpha_3 \wedge \alpha_4}_{\text{Eindeutigkeit}} \wedge \underbrace{\alpha_5 \wedge \alpha_6 \wedge \alpha_7}_{\text{Übergänge}} \wedge \underbrace{\alpha_8}_{\text{Ende}},
$$

die im Einzelnen die folgende Gestalt haben:

$$
\begin{aligned}
\alpha_1 &\mathrel{\hat{=}} \text{Startkonfiguration (auf dem Arbeitsband steht } u\mathbf{b}\cdots\mathbf{b}\text{, } \mathfrak{M} \text{ befindet sich im Zustand } q_0 \text{ und das Arbeitsfeld steht auf Nummer 0)}\\
&= \textstyle\bigwedge_{i\le|u|} a_{0ij_i} \wedge \bigwedge_{|u|+1\le i\le T(|u|)} a_{0i0} \wedge z_{00} \wedge s_{00},\\
\alpha_2 &\mathrel{\hat{=}} \mathfrak{M} \text{ befindet sich zu jedem Zeitpunkt in genau einem Zustand}\\
&= \textstyle\bigwedge_{0\le t\le T(|u|)}(\bigvee_{0\le i\le s} z_{ti} \wedge \bigwedge_{i\ne j} \neg(z_{ti} \wedge z_{tj})),\\
\alpha_3 &\mathrel{\hat{=}} \text{der Schreib-/Lesekopf befindet sich zu jedem Zeitpunkt auf genau einer Bandposition}\\
&= \textstyle\bigwedge_{0\le t\le T(|u|)}(\bigvee_{0\le i\le T(|u|)} s_{ti} \wedge \bigwedge_{i\ne j} \neg(s_{ti} \wedge s_{tj})),\\
\alpha_4 &\mathrel{\hat{=}} \text{zu jedem Zeitpunkt befindet sich auf jeder Bandposition genau ein Buchstabe aus dem Arbeitsalphabet } \Gamma\\
&= \textstyle\bigwedge_{0\le t\le T(|u|)} \bigwedge_{0\le r\le T(|u|)}(\bigvee_{0\le i\le m} a_{tri} \wedge \bigwedge_{i\ne j} \neg(a_{tri} \wedge a_{trj})),\\
\alpha_5 &\mathrel{\hat{=}} \text{ausgeführte Züge bewirken die gewünschte Änderung } (z^l = (q_{k_l}, a_i, a_{i_j}, \beta_l, q_{\tilde{k}_l}))\\
&= \textstyle\bigwedge_{0\le t\le T(|u|)} \bigwedge_{0\le i\le T(|u|)} \bigwedge_{1<l\le\varrho} \big((s_{ti} \wedge b_{tl}) \to (z_{tk_l} \wedge a_{tij_l} \wedge z_{(t+1)\tilde{k}_l} \wedge a_{(t+1)i\tilde{j}_l} \wedge s_{(t+1)(i+\beta_l)})\big),\\
\alpha_6 &\mathrel{\hat{=}} \text{wo der Schreib-/Lesekopf nicht steht, bleibt das Arbeitsfeld unverändert}\\
&= \textstyle\bigwedge_{0\le t\le T(|u|)} \bigwedge_{0\le i\le T(|u|)} \bigwedge_{0\le j\le m} \big((\neg s_{ti}) \wedge a_{tij}\big) \to a_{(t+1)ij}\big),\\
\alpha_7 &\mathrel{\hat{=}} \mathfrak{M} \text{ führt zu jedem Zeitpunkt } t \ne T(|u|) \text{ genau einen Schritt aus}\\
&= \bigwedge_{0\le t\le T(|u|)} \big(\textstyle\bigvee_{1\le l\le\varrho} b_{tl} \wedge \bigwedge_{l\ne l'}(b_{tl} \to \neg b_{tl'})\big),\\
\alpha_8 &\mathrel{\hat{=}} \text{die Endkonfiguration ist akzeptiert}\\
&= \textstyle\bigvee_{r\le i\le s} z_{T(|u|)i}.
\end{aligned}
$$

Wie man sieht, folgen wir mit der obigen Konstruktion genau der zu Beginn dieses Beweises erläuterten Idee, die Arbeitsweise der Turingmaschine durch eine Formel zu simulieren.

Da man $\alpha = \alpha_{\text{Anfang}} \wedge \alpha_{\text{Ende}} \wedge \alpha_{\text{Eindeutig}} \wedge \alpha_{\text{Übergang}}$ in konjunktiver Normalform darstellen kann, müssen wir jetzt nur noch zeigen, dass

(a) die Transformation $u \to \alpha := f(u)$ polynomiell ist, und

(b) eine akzeptierende Berechnung von $\mathfrak{M}$ zu u der Länge $T(|u|)$ genau dann existiert, wenn $\alpha = f(u)$ erfüllbar ist.

Zu (a): Wir bestimmen die Anzahl beziehungsweise Häufigkeit $|\alpha|$ der Variablen in $\alpha = T(u)$, wobei im Folgenden $n := |u|$. Da

$$\alpha = \alpha_1 \wedge \cdots \wedge \alpha_8,$$

reicht es also, die Anzahl der Variablen in den Teilformeln abzuschätzen. Wir erhalten

$$\begin{aligned}
|\alpha_1| &= T(n),\\
|\alpha_2| &= T(n) \cdot (s + s(s-1)),\\
|\alpha_3| &= T(n) \cdot (T(n) + T(n)(T(n)-1)),\\
|\alpha_4| &= T(n)^2 \cdot (m + 1 + (m+1)m),\\
|\alpha_5| &= T(n)^2 \cdot \varrho \cdot 7,\\
|\alpha_6| &= T(n)^2 \cdot m \cdot 3,\\
|\alpha_7| &= T(n) \cdot (\varrho + \varrho(\varrho - 1)) \quad \text{und}\\
|\alpha_8| &= s.
\end{aligned}$$

Insgesamt ergibt sich damit eine in $T(|u|)$ polynomielle Anzahl der Variablen (man beachte, dass die Werte s, m und ϱ Konstanten sind, die von der Turingmaschine $\mathfrak{M}$ abhängen).

Zu (b): „$\Longrightarrow$“ Sei also $u \in L$. Dann existiert eine akzeptierende Berechnung von $\mathfrak{M}$ der Länge $T(|u|)$, d. h. eine Konfigurationsfolge

$$\varepsilon q_0 u_0 u_1 \cdots u_n \kappa_0 \vdash_{\mathfrak{M}} \kappa_1 \vdash_{\mathfrak{M}} \cdots \vdash_{\mathfrak{M}} \kappa_{T(|u|)} = \alpha q a \beta$$

mit $q \in F$. Wir übersetzen diese Berechnung in eine Belegung der Variablen aus X wie folgt:

$$\begin{aligned}
\mu(a_{tij}) := \texttt{wahr} &\iff \text{nach Schritt } t \text{ steht } a_j \text{ auf Arbeitsfeld } i,\\
\mu(b_{tl}) := \texttt{wahr} &\iff \text{nach Schritt } t \text{ wird Zeile } l \text{ ausgeführt},\\
\mu(s_{ti}) := \texttt{wahr} &\iff \text{nach Schritt } t \text{ steht der Kopf auf Arbeitsfeld } i,\\
\mu(z_{tl}) := \texttt{wahr} &\iff \text{nach Schritt } t \text{ ist } \mathfrak{M} \text{ im Zustand } q_k.
\end{aligned}$$

Offensichtlich gilt dann $\mu(f(u)) = \texttt{wahr}$.

„$\Longleftarrow$“ Sei nun umgekehrt μ eine erfüllende Belegung für $\alpha = f(u)$. Wegen

$$\mu(\alpha_{\text{Eindeutig}}) = \texttt{wahr}$$

existiert zu jedem Zeitpunkt t genau ein k mit $z_{tk} = \texttt{wahr}$. Wir bezeichnen dieses mit $k(t)$. Analog definieren wir $i(t)$, $l(t)$ und $j(t,i)$. Für alle Zeitpunkte t wird eine Konfiguration κ_t eindeutig definiert durch den Zustand $q_{k(t)}$, das Arbeitsfeld Nr. $i(t)$ und die Bandinschrift $a_{j(t,0)}, \ldots, a_{j(t,T(|u|))}$.

Wir zeigen nun, dass $\kappa_0, \ldots, \kappa_{T(|u|)}$ eine akzeptierende Berechnung ist. Wegen $\alpha_{\text{Anfang}} = \texttt{wahr}$ ist κ_0 die Startkonfiguration zu u. Wegen $\alpha_{\text{Ende}} = \texttt{wahr}$ ist $\kappa_{T(|u|)}$ eine akzeptierende Konfiguration. Wegen $\alpha_{\text{Übergang}} = \texttt{wahr}$ geschieht der Übergang $\kappa_t \vdash_{\mathfrak{M}} \kappa_{t+1}$ gemäß Zeile $z^{l(t)}$. Insgesamt erhalten wir also eine akzeptierende Berechnung von $\mathfrak{M}$ auf u. □

Nachdem wir nun gesehen haben, dass es zumindest ein NP-vollständiges Problem gibt, liefert uns der nächste Satz eine Möglichkeit, von weiteren Entscheidungsproblemen zeigen zu können, dass sie NP-vollständig sind.

Satz 2.28. *Seien* Π_1 *und* Π_2 *zwei Entscheidungsprobleme aus* NP *so, dass*

$$L(\Pi_1) \leq L(\Pi_2).$$

Dann gilt: Ist Π_1 NP*-vollständig, so auch* Π_2.

Beweis. Der Beweis verläuft analog zu dem von Satz 2.8. □

Beweise für den Nachweis der NP-Vollständigkeit eines Entscheidungsproblems Π laufen dann in der Regel nach dem folgenden Schema ab:

(i) Zeige zunächst $\Pi \in$ NP.

(ii) Wähle ein geeignetes NP-vollständiges Entscheidungsproblem Π'.

(iii) Konstruiere eine polynomielle Transformation f von Π' auf Π.

Die Hauptschwierigkeit ist natürlich das Finden einer polynomiellen Transformation von Π' auf Π. Es gibt allerdings einige Standardtechniken, wie man diese Transformationen konstruiert. Wir werden im letzten Abschnitt dieses Kapitels noch einmal genauer darauf eingehen.

Zunächst wollen wir aber im nächsten Abschnitt von einer ganzen Reihe von Entscheidungsproblemen die NP-Vollständigkeit nachweisen und benötigen dafür, dass schon eine Einschränkung von SAT, das sogenannte 3SAT-Problem, NP-vollständig ist.

Eine aussagenlogische Formel in konjunktiver Normalform liegt in *k-konjunktiver Normalform*, $k \geq 2$, vor, wenn zusätzlich jede Klausel genau k Literale enthält. Diese Ausdrücke haben also die Form

$$\alpha = (y_1 \vee y_2 \vee \cdots \vee y_k) \wedge \cdots \wedge (y_{n-(k+1} \vee y_{n-(k+2)} \vee \cdots \vee y_n).$$

Es stellt sich heraus, dass das Problem 2SAT in polynomieller Zeit sogar linear lösbar ist, siehe [45]. Der Fall $k = 1$ ist offensichtlich trivial. Für $k > 2$ liegen uns aber NP-vollständige Probleme vor. Wir wollen dies für den Fall $k = 3$ exemplarisch beweisen.

Wir behandeln damit das folgende Problem.

Problem 2.29 (3SAT).
Eingabe: Eine aussagenlogische Formel in 3-konjunktiver Normalform.
Frage: Ist α erfüllbar?

Satz 2.30. 3SAT *ist* NP-*vollständig*.

Beweis. Da jede Instanz von 3SAT auch eine Instanz von SAT ist und SAT $\in$ NP, folgt dies auch für das obige Problem.

Wir müssen also nur noch SAT $\leq$ 3SAT zeigen. Dazu werden wir zu jeder Formel $\alpha = c_1 \wedge \cdots \wedge c_m$ in konjunktiver Normalform über den Variablen $X = \{x_1, \ldots, x_n\}$ eine Formel α' in 3-konjunktiver Normalform so konstruieren, dass α genau dann erfüllbar ist, wenn α' erfüllbar ist. Genauer konstruieren wir zu jeder Klausel c_i einen Ausdruck α_i' in 3-konjunktiver Normalform so, dass c_i genau dann erfüllbar ist, wenn dies für α_i' gilt. Mit $\alpha' = \alpha_1' \wedge \cdots \wedge \alpha_m'$ folgt dann die Behauptung.

Sei nun $i \leq m$ und $c_i = (y_1 \vee \cdots \vee y_k)$ mit Literalen $y_1, \ldots, y_k \in \{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$. Die Form von α_i' hängt von der Anzahl der in c_i enthaltenen Literale ab. Man beachte, dass die zusätzlich definierten Variablen aus Z_i nur in der Formel α_i' benutzt werden. Wir erhalten die folgende Fallunterscheidung:

$$
\begin{array}{lll}
\textit{Fall } k = 1: & Z_i := \{z_i^1, z_i^2\}, & \\
& \alpha_i' := (y_1 \vee z_i^1 \vee z_i^2) \wedge (y_1 \vee z_i^1 \vee \neg z_i^2) & \\
& \quad \wedge (y_1 \vee \neg z_i^1 \vee z_i^2) \wedge (y_1 \vee \neg z_i^1 \vee \neg z_i^2). & \\
\textit{Fall } k = 2: & Z_i := \{z_i\}, & \\
& \alpha_i' := (y_1 \vee y_2 \vee z_i) \wedge (y_1 \vee y_2 \vee \neg z_i). & \\
\textit{Fall } k = 3: & Z_i := \varnothing, & \\
& \alpha_i' := c_i. & \\
\textit{Fall } k = 4: & Z_i := \{z_i^j ; 1 \leq j \leq k-3\}, & \\
& \alpha_i' := (y_1 \vee y_2 \vee z_i^1) \wedge \bigwedge_{1 \leq j \leq k-4} (\neg z_i^j \vee y_{j+2} \vee z_i^{j+1}) & \\
& \quad \wedge (\neg z_i^{k-3} \vee y_{k-1} \vee y_k). &
\end{array}
$$

Sei nun $\mu : \{x_1, \ldots, x_n\} \longrightarrow \{\texttt{wahr}, \texttt{falsch}\}$ eine erfüllende Belegung für c_i. Wir zeigen, dass sich μ zu einer erfüllenden Belegung für α_i' auf ganz $X' := X \cup Z_i$ erweitern lässt.