Das praktisch Unmögliche - Didaktische Reduktion der polynomiellen Reduktion

Melanie Jacksties
m.jacksties@googlemail.com

Examensarbeit für das Lehramt an Gymnasien

betreut von
Prof. Dr. Jürgen Poloczek

Zweitkorrektor:
Prof. Dr. Reinhard Oldenburg

Juni 2012

Kurzfassung

> „Wohl jeder Nutzer, auch Lehrer und Schüler, wird schon vor dem Computer gesessen und nicht gewusst haben, ob eine fehlerhafte Endlosschleife läuft oder ob es sich noch lohnt, auf ein Ergebnis zu warten. Aber auch eine andere Situation wird bekannt sein: Man weiß genau, dass im Programm kein Fehler steckt, und doch dauert die Abarbeitung unerträglich lange. Ja, es kann so lange dauern, dass man auf das Resultat verzichtet bzw. verzichten muss“ [Ker91].

Solche Probleme werden unteranderem als „praktisch unmögliche“ oder „praktisch unlösbare Probleme“ bezeichnet. Wie der Titel meiner Arbeit verrät, wird auch hier ein Blick auf wirklich schwere Probleme gewagt. Die theoretische Informatik ist reich an wertvollen Konzepten und Ideen. In der vorliegenden Arbeit wird sich mit dem Teilgebiet der Komplexitätstheorie intensiv beschäftigt. Wir betrachten die berühmte $\mathcal{P}$-$\mathcal{NP}$-Frage, sowie die hervorragende Entwicklung der Turingmaschine. Zentrale Themen werden der Satz von Cook und das Konzept der polynomiellen Reduktion sein. Der Satz von Cook liefert ein bekanntes schweres (genauer $\mathcal{NP}$-vollständiges) Problem, nämlich das Erfüllbarkeitsproblem (engl. Satisfiability-Problem oder abgekürzt: SAT). Die polynomielle Reduktion wird wie ein Lawinenauslöser dafür sorgen, dass es möglich ist, weitere $\mathcal{NP}$-vollständige Probleme zu klassifizieren.
Auch die theoretische Informatik und die Komplexitätstheorie im Speziellen, hat ihre Berechtigung in Bezug auf die Schulinformatik. Leider gibt es gerade zu den eben aufgeführten Themen nur sehr wenig Material zur Umsetzung im Schulunterricht. Die Aussage des berühmten Satz von Cook lautet, dass das Erfüllbarkeitsproblem $\mathcal{NP}$-vollständig ist. Das Konzept der polynomiellen Reduktion ermöglicht es nun, ähnliche Probleme mithilfe des schon bekannten $\mathcal{NP}$-vollständigen Erfüllbarkeitsproblems zu klassifizieren. In der Fachliteratur wird nur äußerst selten eine Reduktion auf das Erfüllbarkeitsproblem angeboten. Zumeist lautet der Gang „3-SAT“ auf „SAT“, danach „3-COLOR“ auf „3-SAT“. Ab dann gerät das Erfüllbarkeitsproblem meist etwas in Vergessenheit, da man eine lange Kette von Reduktionen auf andere Probleme heranzieht.
Mein Ziel ist es nun, auf der einen Seite durch das Verwenden von alltagsnahen Beispielen aufzuzeigen, wie auch „harte“ Theorie vermittelbar sein kann. Auf der anderen Seite, möchte ich mit meinen Beispielreduktionen auf das Erfüllbarkeitsproblem die oben beschriebene Lücke stopfen, und den Satz von Cook anwendbar machen. Auch wenn die benötigten Grundlagen umfangreich und teilweise sehr mathematisch geprägt sind, wird in den erarbeiteten Beispielen versucht, die Konzepte möglichst informell, intuitiv und anschaulich zu vermitteln.

Eidesstattliche Erklärung

Hiermit erkläre ich, Melanie Jacksties, geboren am 17.11.1981 in Kusel, ehrenwörtlich, dass ich meine Examensarbeit mit dem Titel

„Das praktisch Unmögliche - Didaktische Reduktion der polynomiellen Reduktion“

selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe und die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Frankfurt, den 12.06.2012

_____

(Melanie Jacksties)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I.

Grundlagen

1. Die wichtigsten Grundlagen auf einen Blick

Meine Arbeit beginnt mit einem sehr umfangreichen Grundlagenkapitel. Diese Grundlagen sind wichtig und unumgänglich, möchte man die Thematik komplett durchdringen. Um die polynomielle Reduktion und das dahinterliegende Konzept zu verstehen, ist es nicht zwingend notwendig, das gesamte Grundlagenkapitel lange zu studieren. Auf keinen Fall sind die hier aufgeführten Grundlagen als Lernvoraussetzungen an Schüler und Schülerinnen zu verstehen! Notwendige Voraussetzungen für die erläuterten Beispiele in Kapitel 6 werden vor jedem Beispiel kurz zusammengefasst präsentiert.
Vielmehr soll das Grundlagenkapitel als Nachschlagemöglichkeit dienen und interessierten Lesern ein fundiertes Nachlesen ermöglichen, ohne viel zusätzliche Literatur hinzuziehen zu müssen.

Ich möchte an dieser Stelle keinesfalls die Themen der Grundlagen allgemein nach Wichtigkeit bewerten. Jedes Thema hat seine eigene Berechtigung und ist grundlegend in der Theorie der Infomatik. Mit Wichtigkeit ist hier die Wichtigkeit hinsichtlich dieser Arbeit gemeint.

Das Grundlagenkapitel beginnt mit einer Einführung in die Graphentheorie. Graphen liefern eine sehr anschaulische Darstellung aller möglicher zweistelliger Relationen. Später werden wir sehen, dass viele schwere Probleme der Komplexitätstheorie Graphenprobleme behandeln. Um die Darstellungen in den Beispielen und den Umgang der Graphen zu verstehen, lohnt sich ein Blick in Kapitel 2. Im darauf folgenden Kapitel 3 erfolgt dann eine Einführung in die Theorie der Booleschen Algebra. In meiner Arbeit wird es zentral um den berühmten Satz von Cook gehen. Dieser sagt aus, dass das Erfüllbarkeitsproblem (SAT (vgl. Definition 3.7)) $\mathcal{NP}$-vollständig ist. Um dies zu beweisen, aber noch wichtiger, um die später folgenden Reduktionen auf SAT in Kapitel 6 nachvollziehen zu können, benötigt man grundlegendes Wissen von booleschen Formeln, deren Umformung, sowie Wissen über die konjunktive Normalform.

Kapitel 4 liefert den Kern der Grundlagen zu dieser Arbeit. Hier wird zuerst die historische Entwicklung und wichtige an dieser Entwicklung beteiligter Personen beleuchtet. Das Modell der Turingmaschine, welches wir im späteren Beweis des Satz von Cook als auch zum Verständnis der Definitionen der Komplexitätsklassen $\mathcal{N}$ und $\mathcal{NP}$ benötigen, wird vorgestellt. Kapitel 4.5 widmet sich der polynomiellen Reduktion und der „schwierigsten“ Probleme der Klasse

$\mathcal{NP}$. Das Konzept der polynomiellen Reduktion bildet mit dem Satz von Cook (vgl. Satz 4.6) den Kern der Beispiele im späteren Anwendungsteil der Arbeit (vgl. Kapitel 6).

2. Einführung in die grundlegendsten Begriffe der Graphentheorie

Der Schweizer Mathematiker Leonhard Euler (1707 - 1783) gilt als der Vater der Graphentheorie. Sein Aufsatz „Solutio Problematis ad Geometriam Situs Pertinentis“ wird als die erste Publikation über die Graphentheorie anerkannt [Che12]. Im Jahre 1736 löste Euler das sogenannte Königsberger Brückenproblem, welches ein berühmtes ungelöstes Problem seiner Zeit darstellte.

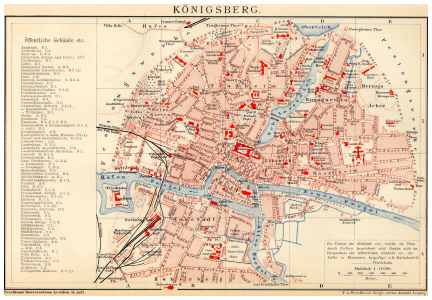


Abbildung 2.1.: Alter Stadtplan von Königsberg [kön12]

In Königsberg gab es zwei miteinander und mit den Ufern des Flusses Pregel durch sieben Brücken verbundene Inseln. In Abbildung 2.1 ist ein alter Stadtplan von Königsberg[1] zu sehen. Es ging nun um die Frage, ob es möglich ist, einen Rundgang durch die Stadt zu machen, der jede Brücke über den Fluss Pregel genau einmal benutzt und wieder zum Ausgangspunkt (zum Startpunkt) zurückkehrt. „Beim Beweis der Unlösbarkeit des Problems ersetzte Euler jedes Landgebiet durch einen Punkt und jede Brücke durch eine Linie, die die entsprechenden Punkte verbindet, und kam dadurch zu einem Graphen“ [Har74]. Der aus dem Stadtplan resultierende Graph ist in Abbildung 2.2 zu sehen.

Das Königsberger Brückenproblem gehört im Übrigen in die Komplexitätsklasse $\mathcal{P}$ (vgl. Kapitel 4.3). Nimmt man eine kleine Änderung in der Fragestellung vor, sucht man nämlich nach einem Weg, der jeden Knoten genau einmal enthält, gelangt man zu dem sogenannten „Hamiltonkreis-Problem“, welches schon zu den Problemen der Klasse $\mathcal{NP}$ gehört.
Mit seinem Artikel „On the theory of the analytical forms called trees“, der in der Zeitschrift Philosophical Magazine erschien, „entdeckte Caley 1857 eine wichtige Klasse von Graphen, die sogenannten Bäume, in ganz natürlichem Zusammenhang mit der organischen Chemie.“ [Har74]
Im Jahre 1936 veröffentlichte der ungarischen Mathematiker Dénes König das

[1] Heute heißt das alte Königsberg Kaliningrad und der Fluß Pregel ist unter dem Namen Pregolya bekannt, [Che12].

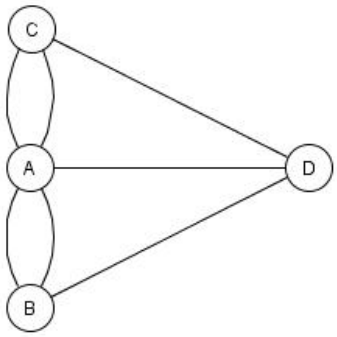


Abbildung 2.2.: Der Graph des Königsberger Brückenproblems

Buch „Theorie der endlichen und unendlichen Graphen“, welches als erstes Lehrbuch zur Graphentheorie angesehen wird. [Che12]

In wie vielen Gebieten die Graphentheorie heute Anwendung findet, soll durch folgendes Zitat verdeutlicht werden:

> „Es gibt mehrere Gründe für das schnelle Interesse an der Graphentheorie. Es ist Mode geworden zu erwähnen, dass Graphentheorie Anwendung findet in einigen Gebieten der Physik und der Chemie, im Verkehrs- und Nachrichtewesen, in der Computer- und Bautechnik, im Operations Research, in Genetik, Soziologie, Ökonomie, Anthropologie und Linguistik. Die Theorie steht auch in enger Beziehung zu vielen Zweigen der Mathematik, wie etwa der Gruppentheorie, der Matrizenrechnung, der numerischen Mathematik, der Wahrscheinlichkeitsrechnung, der Topologie und der Kombinatorik. Tatsächlich dient die Graphentheorie als mathematisches Modell für jedes System mit einer zweistelligen Relation.“ [Har74]

Einen beonderen Reiz bekommt die Graphentheorie insbesondere durch ihre Anschaulichkeit und die Vielfalt der verwendbaren Beweistechniken. Nachdem ich eben den geschichtlichen Ursprung und einige Anwendungsgebiete erläutert habe, soll nun der zentrale Begriff der Graphen näher betrachtet werden. Leider muss man zunächst eine schreckliche Anzahl von Definitionen erarbeiten, um die Grundbegriffe der Graphentheorie zugänglich zu machen. Es folgt also nun eine zwar langweilige, aber notwendige Folge verschiedener Definitionen[2]. Zuerst benötigen wir eine Konkretisierung des intuitiven Begriffs eines Graphen.

Definition 2.1. Ein ungerichteter Graph G=(V, E) besteht aus einer Menge V, die Knotenmenge von G genannt wird, und einer Menge

$$E \subseteq \{\{i,j\} : i \in V,\ j \in V, i \neq j\},$$

[2] Alle nun folgenden Definitionen basieren auf dem Skript zur Vorlesung „Diskrete Modellierung“ [Sch09], die im Wintersemester 09/10 gehalten wurde.

die Kantenmenge von G genannt wird. Die Elemente aus V heißen Knoten von G; die Elemente aus E heißen Kanten von G.

Definition 2.2. Sei G=(V, E) ein ungerichteter Graph.

- Ein Knoten $v \in V$ heißt inzident mit einer Kante $e \in E$, falls $v \in e$.
- Die beiden mit einer Kante $e \in E$ inzidenten Knoten werden Endknoten genannt.
- Zwei Knoten $v, v' \in V$ heißen adjazent, falls es eine Kante $e \in E$ gibt, deren Endknoten v und v' sind (d.h. $e = \{v, v'\}$).

Kanten eines Graphen können auch eine Richtung haben.

Definition 2.3. Ein gerichteter Graph G=(V, E) besteht aus einer Menge V, die Knotenmenge von G genannt wird, und einer Menge

$$E \subseteq \{(i, j) : i \in V,\ j \in V\}$$

die Kantenmenge von G genannt wird. Die Elemente aus V heißen Knoten von G; die Elemente aus E heißen Kanten von G.

Definition 2.4. Sei G=(V, E) ein gerichteter Graph.

- Ist $e = (i, j) \in E$, so heißt i der Ausgangsknoten von e und j der Endknoten von e.
- Ein Knoten $v \in V$ heißt inzident mit einer Kante $e \in E$, falls v entweder Ausgangs- oder Endknoten von e ist.
- Zwei Knoten $v, v' \in V$ heißen adjazent, falls $(v, v') \in E$ oder $(v', v) \in E$.
- Eine Kante der Form (v, v) wird Schleife, manchmal auch Schlinge, genannt.

Neben der abstrakten oder der grafischen Darstellung von Graphen sind zwei weitere Möglichkeiten der Graphrepräsentation wichtig. Zum einen kann man durch Angabe einer Adjazenzliste den Graph vollständig spezifizieren. Eine Adjazensliste gibt zu jedem Knoten i eine Liste aller Knoten, zu denen eine von i ausgehende Kante führt, an. Graphen können aber auch durch eine Adjazenzmatrix angegeben werden. Eine Adjazenzmatrix ist eine Tabelle, deren Zeilen und Spalten mit Knoten beschriftet sind, und die in der mit Knoten i beschrifteten Zeile und der mit Knoten j beschrifteten Spalte den Eintrag 1 hat, falls es eine Kante von Knoten i nach Knoten j gibt - und den Eintrag 0 sonst.

Beispiel 2.1. G=(V, E) mit

$$V := \{1, 2, 3\}$$
$$E := \{(1, 2), (2, 2), (2, 3), (3, 1), (1, 3), (3, 3)\}$$

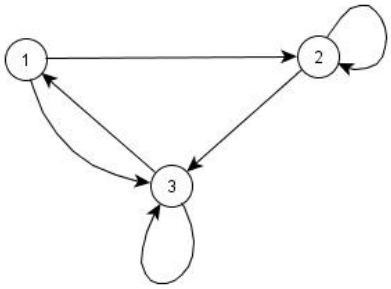


Abbildung 2.3.: Graph zu Beispiel 2.1

ist ein gericheteter Graph. In Abbildung 2.3 ist der Graph zu sehen.

Die Darstellung des Graphen aus Abbildung 2.3 als Adjazenzliste und als Adjazenzmatrix lautet:

Adjazenzliste:

Knoten	Nachfolger
1	(2, 3)
2	(2, 3)
3	(1, 2)

Adjazenzmatrix:

	1	2	3
1	0	1	1
2	0	1	1
3	1	0	1

bzw.

$$A_G = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Wir wollen nun Wege in Graphen betrachten und danach unter anderem den Begriff der Zusammenhangskomponente einführen.

Definition 2.5. Sei G =(V, E) ein (gerichteter oder ungerichteter) Graph.

1. Ein Weg in G ist ein Tupel

$$(v_0, \cdots, v_l) \in V^{l+1}$$

für ein $l \in \mathbb{N}$, so dass für alle i mit $0 \leq i \leq l$ gilt:

- falls G ein gerichteter Graph ist, so ist $(v_i, v_{i+1}) \in E$
- falls G ein ungerichteter Graph ist, so ist $\{v_i, v_{i+1}\} \in E$

Das Tupel $(v_0, \cdots, v_l)$ wird dann ein Weg von v_0 nach v_l genannt. Die Länge des Weges gibt an, wie viele Kanten auf dem Weg durchlaufen werden, d.h. hier ist die Länge des Weges l.

2. Ein Weg heißt einfach, wenn kein Knoten mehr als einmal in dem Weg vorkommt.
3. Ein Weg $(v_0, \cdots, v_l)$ heißt Kreis, wenn die Länge $l \geq 1$ und $v_l = v_0$ ist.
4. Ein Kreis $(v_0, \cdots, v_l)$ heißt einfach, wenn keine Kante mehrfach durchlaufen wird und - abgesehen vom Start- und Endknoten - kein Knoten mehrfach besucht wird.

Beispiel 2.2. Wir betrachten den Graphen[3] in Abbildung 2.4

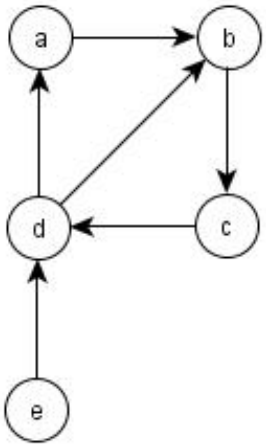


Abbildung 2.4.: Graph zu Beispiel 2.2

- (e,d,b,c,d) ist ein Weg der Länge 4, aber kein einfacher Weg.
- (d,b,c,d) ist ein einfacher Kreis.
- (e,d,a,b) ist ein einfacher Weg.
- (b,d,a) ist kein Weg.
- (a,b,c,d,b,c,d,a) ist ein Kreis, aber kein einfacher Kreis.

Definition 2.6. Ein Graph heißt azyklisch, falls er keinen Kreis enthält.

Definition 2.7. Ein ungerichteter Graph G=(V, E) heißt zusammenhängend, wenn für alle Knoten $v, w \in V$ gilt: Es gibt in G einen Weg von v nach w.
Ein gerichteter Graph G=(V, E) heißt stark zusammenhängend, wenn für alle Knoten $v, w \in V$ gilt: Es gibt in G einen Weg von v nach w.
Sei $v \in V$ ein Knoten von G. Die Menge aller Knoten, die mit v zusammenhängen, nennt man die (starke) Zusammenhangskomponente von v.

[3] Dieses Beispiel wurde entnommen aus [Sch09].

Beispiel 2.3. Der ungerichtete Graph in Abbildung 2.5 ist zusammenhängend und der in Abbildung 2.6 ist nicht zusammenhängend.

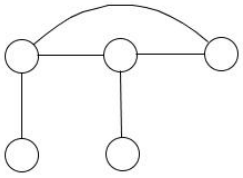


Abbildung 2.5.: Zusammenhängender ungerichteter Graph

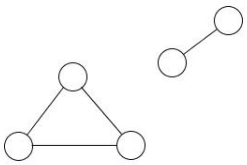


Abbildung 2.6.: Nicht zusammenhängender ungerichteter Graph

In Abbildung 2.7 ist ein gerichteter stark zusammenhängender Graph zu sehen. Der gerichtete Graph in Abbildung 2.8 ist nicht stark zusammenhängend, da es zum Beispiel keinen Weg vom Knoten links oben zum Knoten links unten gibt.

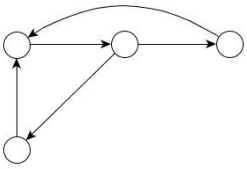


Abbildung 2.7.: Stark zusammenhängender gerichteter Graph

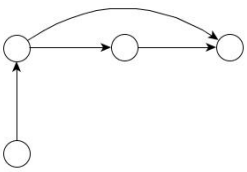


Abbildung 2.8.: Nicht stark zusammenhängender gerichteter Graph

3. Einführung in die Boolesche Algebra

Unter der Aussagenlogik versteht man die „Lehre des vernünftigen Schlußfolgerns“ [Wik12f]. Die Aussagenlogik ist ein Teilbereich der (klassischen) Logik, der sich mit Aussagen und ihren logischen Verknüpfungen (Junktoren, Konnektoren) befasst [Rei12]. In der Informatik lassen sich zahlreiche Anwendungsgebiete der Logik finden. So benutzt man die Logik zum Beispiel in Programmiersprachen, um Bedingungen zu formulieren. Im Bereich der künstlichen Intelligenz wird die Logik dazu verwendet, statisches Wissen zu repräsentieren. Ein weiteres großes Anwendungsfeld ist die Verifikation von Schaltkreisen, Programmen und Protokollen. Auch Datenbankanfragen werden auf der Grundlage der Logik formuliert [Sch09].

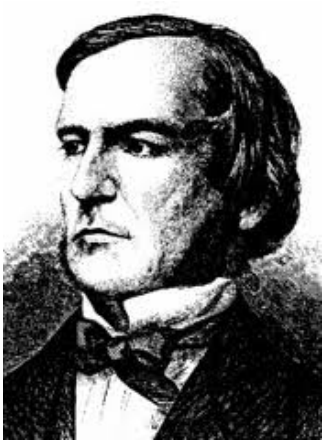


Abbildung 3.1.: George Boole

Ein bekanntes Anwendungsgebiet der Aussagenlogik, die Theorie Boolscher Funktionen, möchte ich im Folgenden einführen. In Kapitel 4.6 greife ich dann auf die vorgestellten Inhalte zurück.

Alle Definitionen und Lemmata in diesem Kapitel basieren auf [Hed12].

3.1. Formeln, Formeln, Formeln

Der Begriff der booleschen Algebra, den ich in diesem Kapitel erläutern möchte, ist nach George Boole benannt. George Boole „[...]war ein englischer Mathematiker, der sich Mitte des 19. Jahrhunderts mit der formalen Sicht digitaler Strukturen beschäftigte. Er war zuerst Autodidakt, später bekam er einen Lehrstuhl in Kork (Ireland)“ [Beh12].

> „Boole schuf in seiner Schrift „The Mathematical Analysis of Logic“ von 1847 den ersten algebraischen Logikkalkül und begründete da-

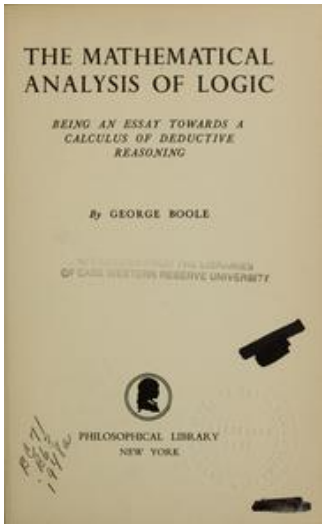


Abbildung 3.2.: The Mathematical Analysis of Logic von 1847 [ope12]

mit die moderne mathematische Logik, die sich von der traditionellen philosophischen Logik durch eine konsequente Formalisierung abhebt.“ [Wik12a]

Definition 3.1. Eine Verknüpfungsstruktur $\mathbb{B} = \{T; \wedge; \vee; \}$, auf deren Trägermenge T zwei zweistellige Verknüpfungen und eine einstellige Verknüpfung (Negation) definiert sind, heißt genau dann Boolesche Algebra, wenn die folgenden Axiome erfüllt sind:

1. **Abgeschlossenheit:** $\wedge$ und $\vee$ sind abgeschlossen auf T
2. **Kommutativgesetz der Konjunktion:** $a \wedge b = b \wedge a$
3. **Kommutativgesetz der Disjunktion:** $a \vee b = b \vee a$
4. **Assoziativgesetz der Konjunktion:** $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
5. **Assoziativgesetz der Disjunktion:** $(a \vee b) \vee c = a \vee (b \vee c)$
6. **Erstes Distributivgesetz:** $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
7. **Zweites Distributivgesetz:** $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
8. **Identitätselement der Konjunktion:** $1 \wedge a = a$
9. **Identitätselement der Disjunktion:** $0 \vee a = a$
10. **Inverses Element der Konjunktion:** $a \wedge \overline{a} = 0$
11. **Inverses Element der Disjunktion:** $a \vee \overline{a} = 1$

Definition 3.2. Ich werde im Folgenden die verkürzenden Schreibweisen

1. $(\overline{a} \vee b) = (a \rightarrow b)$ (Implikation) und
2. $((a \rightarrow b) \wedge (b \rightarrow a)) = (a \leftrightarrow b)$ (Äquivalenz)

benutzen. Die Bindungsstärke dieser neu eingeführten Junktoren ist gleich, aber geringer als die von der Negation ($\overline{a}$), der Konjunktion ($\wedge$) und der Disjunktion ($\vee$).

Lemma 3.1 (Bindungsregeln)**.** *Unter allen Operatoren bindet die Negation ($\overline{x}$) am Stärksten. Die Konjunktion[1] ($\wedge$) bindet wiederum stärker als die Disjunktion ($\vee$). Nach [Bün94] werden binäre Operatoren gleicher Stärke als linksgeklammert angesehen.*

Lemma 3.2 (Idempotenzgesetze)**.** *Für die Konjunktion gilt:*

$$a \wedge a = a$$

und entsprechend gilt für die Disjunktion:

$$a \vee a = a$$

Beweis:
Idempotenzgesetz der Konjunktion:

$$\begin{aligned} a &= 1 \wedge a = a \wedge 1 = a \wedge (a \vee \overline{a}) \\ &= (a \wedge a) \vee (a \wedge \overline{a}) = (a \wedge a) \vee 0 = 0 \vee (a \wedge a) = a \wedge a \end{aligned}$$

□

Idempotenzgesetz der Disjunktion:

$$\begin{aligned} a &= 0 \vee a = a \vee 0 = a \vee (a \wedge \overline{a}) \\ &= (a \vee a) \wedge (a \vee \overline{a}) = (a \vee a) \wedge 1 = 1 \wedge (a \vee a) = a \vee a \end{aligned}$$

□

Lemma 3.3 (Reduktionsgesetze)**.** *Für die Konjunktion gilt:*

$$a \wedge 0 = 0$$

und für die Disjunktion gilt entsprechend:

$$a \vee 1 = 1$$

[1] In der gängigen Literatur wird der Konjugationsoperator häufig, ähnlich wie der Multiplikationsoperator, weggelassen: $a \wedge b = ab$.

Beweis:
Reduktionsgesetz der Konjunktion:

$$\begin{aligned} 0 &= a \wedge \overline{a} = a \wedge (0 \vee \overline{a}) = (a \wedge 0) \vee (a \wedge \overline{a}) \\ &= (a \wedge 0) \vee 0 = 0 \vee (a \wedge 0) = a \wedge 0 \end{aligned}$$

□

Reduktionsgesetz der Disjunktion:

$$\begin{aligned} a &= a \vee \overline{a} = a \vee (1 \wedge \overline{a}) = (a \vee 1) \wedge (a \vee \overline{a}) \\ &= (a \vee 1) \wedge 1 = 1 \wedge (a \vee 1) = a \vee 1 \end{aligned}$$

□

Lemma 3.4 (Eindeutigkeit des inversen Elements)**.** *Zu jedem $a \in T$ existiert genau ein inverses Element $\overline{a} \in T$, so dass gilt:*

$$a \wedge \overline{a} = 0 \quad \textit{und} \quad a \vee \overline{a} = 1$$

Beweis:
Seien $\overline{a}_1$ und $\overline{a}_2$ invers zu a:

$$\begin{aligned} \overline{a}_1 &= 1 \wedge \overline{a}_1 = \overline{a}_1 \wedge 1 = \overline{a}_1 \wedge (a \vee \overline{a}_2) \\ &= (\overline{a}_1 \wedge a) \vee (\overline{a}_1 \wedge \overline{a}_2) = (a \wedge \overline{a}_1) \vee (\overline{a}_1 \wedge \overline{a}_2) \\ &= \overline{a}_1 \wedge \overline{a}_2 = \overline{a}_2 \wedge \overline{a}_1 = 0 \vee (\overline{a}_2 \wedge \overline{a}_1) \\ &= (a \wedge \overline{a}_2) \vee (\overline{a}_2 \wedge \overline{a}_1) = (\overline{a}_2 \wedge a) \vee (\overline{a}_2 \wedge \overline{a}_1) = \overline{a}_2 \wedge (a \vee \overline{a}_1) \\ &= \overline{a}_2 \wedge 1 = 1 \wedge \overline{a}_2 = \overline{a}_2 \end{aligned}$$

□

Durch die Eindeutigkeit des inversen Elements folgt unmittelbar, dass die doppelte Negation keine Auswirkung hat: $\overline{\overline{a}} = a$.
Im folgenden Lemma folgen einige weitere Rechenregeln, die ich an dieser Stelle nicht beweisen möchte. Die Beweise können analog zu den bisherigen Beweisen mit den Axiomen der Booleschen Algebra erfolgen.

Lemma 3.5 (Absorptionsregeln)**.**

1. $a \vee (a \wedge b) = a$
2. $a \wedge (a \vee b) = a$
3. $(a \wedge b) \vee (a \wedge \overline{b}) = a$
4. $(a \vee \overline{b}) \wedge b = a \wedge b$
5. $(a \wedge \overline{b}) \vee b = a \vee b$
6. $(a \vee b) \wedge (a \vee \overline{b}) = a$

Lemma 3.6 (Gesetze von DeMorgan)**.** *Die Umformungsregeln von deMorgan für zwei Boolesche Variablen lauten:*

1. $\overline{a \vee b} = \overline{a} \wedge \overline{b}$
2. $\overline{a \wedge b} = \overline{a} \vee \overline{b}$

Und verallgemeinert auf eine beliebige Anzahl von Variablen:

1. $\overline{\bigvee_{i=0}^{n-1} x_i} = \bigwedge_{i=0}^{n-1} \overline{x}_i$
2. $\overline{\bigwedge_{i=0}^{n-1} x_i} = \bigvee_{i=0}^{n-1} \overline{x}_i$

Beweis:
Ich werde hier nur den ersten Beweis für die Formel mit zwei Variablen angeben. Die anderen Beweise erfolgen analog.
Zu 1.: Die Behauptung sagt aus, dass $\overline{a} \wedge \overline{b}$ das Komplement von $a \vee b$ ist. Um das zu beweisen, muss man die beiden Eigenschaften des inversen Elementes nachweisen. Zu zeigen ist also:

1. $(a \vee b) \wedge (\overline{a} \wedge \overline{b}) = 0$ und
2. $(a \vee b) \vee (\overline{a} \wedge \overline{b}) = 1$

Beweis:

1. $(a \vee b) \wedge (\overline{a} \wedge \overline{b}) = (a \wedge \overline{a} \wedge \overline{b}) \vee (b \wedge \overline{a} \wedge \overline{b}) = (0 \wedge \overline{b}) \vee (0 \wedge \overline{a}) = 0 \vee 0 = 0$ und
2. $(a \vee b) \vee (\overline{a} \wedge \overline{b}) = (a \vee b \vee \overline{a}) \wedge (a \vee b \vee \overline{b}) = (1 \vee b) \wedge (1 \vee a) = 1 \wedge 1 = 1$

□

Abschließend bleiben noch die wichtigsten Normalformen einzuführen. In dieser Arbeit beschränke ich mich dabei auf die sogenannte konjunktive Normalform.

Definition 3.3. Unter einem Literal wird eine Variable oder eine negierte Variable verstanden. Es gilt also für jedes Literal $l \in \{a, \overline{a}\}$.
Eine Konjunktion von Literalen heißt Konjunktionsterm:

$$T^K = \bigwedge_{i \in \{0,\dots,n-1\}} l_i \quad \text{mit} \quad l_i \in \{a, \overline{a}\}$$

Entsprechend nennt man eine Disjunktion von Literalen einen Disjunktionsterm:

$$T^D = \bigvee_{i \in \{0,\dots,n-1\}} l_i \quad \text{mit} \quad l_i \in \{a, \overline{a}\}$$

In beiden Fällen tritt jede Variable höchstens einmal auf.

Definition 3.4. Disjunktionen von Literalen heißen Klauseln [Sch11].

Definition 3.5. Nach [Bün94] liegt eine Formel in konjunktiver Normalform (abgekürzt: KNF) vor, genau dann, wenn sie eine Konjunktion von Klauseln ist:

$$\alpha = \alpha_1 \wedge \cdots \wedge \alpha_n \quad \text{mit Klauseln } \alpha_i \text{ mit } (1 \leq i \leq n)\,.$$

Eine Formel α ist in k-KNF genau dann, wenn α eine Konjunktion von k-Klauseln ist. Die entsprechenden Formelklassen werden mit KNF und k-KNF bezeichnet.

Das folgende Lemma sagt aus, dass jede Formel in eine äquivalente Formel in KNF umgeschrieben werden kann:

Lemma 3.7. *Zu jeder Formel α gibt es eine Formel α_{knf} in konjunktiver Normalform mit $\alpha \equiv \alpha_{knf}$.*

In [Bün94] ist ein Algorithmus, der zu jeder Formel in Negationsnormalform[2] eine äquivalente Formel in konjunktiver Normalform berechnet, zu finden.

Hat man keinen solchen Algorithmus zur Verfügung, so kann man für nicht zu umfangreiche Formeln auch leicht eine KNF durch die zugehörige Wahrheitstabelle entwickeln. Dazu geht man wie folgt vor: Im ersten Schritt, muss eine Wahrheitstabelle zu der gegebenen Funktion aufgestellt werden. Wer das nicht per Hand tun möchte, kann zum Beispiel unter [Böh12] den von André Böhm entwickelten „Formelchecker“ benutzen. Hat man erst einmal die Wahrheitstabelle zur Hand, gelingt die Entwicklung einer KNF leicht. „Dazu genügt

[2] Eine Formel α ist in Negationsnormalform (NNF) genau dann, wenn jedes Negationszeichen direkt vor einem Atom steht und keine zwei Negationszeichen direkt hintereinander stehen [Bün94]. Unter Atomen versteht man Elementaraussagen.

a	b	c	d	f	Klausel
0	0	0	0	0	$a \vee b \vee c \vee d$
0	0	0	1	0	$a \vee b \vee c \vee \bar{d}$
0	0	1	0	0	$a \vee b \vee \bar{c} \vee d$
0	0	1	1	0	$a \vee b \vee \bar{c} \vee \bar{d}$
0	1	0	0	0	$a \vee \bar{b} \vee c \vee d$
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	1	
1	0	0	0	0	$\bar{a} \vee b \vee c \vee d$
1	0	0	1	0	$\bar{a} \vee b \vee c \vee \bar{d}$
1	0	1	0	0	$\bar{a} \vee b \vee \bar{c} \vee d$
1	0	1	1	0	$\bar{a} \vee b \vee \bar{c} \vee \bar{d}$
1	1	0	0	1	
1	1	0	1	1	
1	1	1	0	0	$\bar{a} \vee \bar{b} \vee \bar{c} \vee d$
1	1	1	1	1	

Tabelle 3.1.: Wahrheitstafel zu der Funktion aus Bsp. 3.1

es, die Zeilen der Wahrheitstabelle abzulesen. Für jede Zeile, die als Resultat eine 0 liefert, wird eine Klausel gebildet, die alle Variablen der Funktion disjunktiv mit der invertierten Belegung verknüpft. Die entstehenden Terme sind Maxterme. Deren konjunktive Verknüpfung liefert die kanonische konjunktive Normalform“ [Wik12d].

Beispiel 3.1. Sei $f = bd \vee \bar{a}bc \vee ab\bar{c}$. Tabelle 3.1 zeigt die zugehörige Wahrheitstabelle.

$$
\begin{aligned}
f = {} & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \bar{d}) \wedge (a \vee b \vee \bar{c} \vee d) \\
& \wedge (a \vee b \vee \bar{c} \vee \bar{d}) \wedge (a \vee \bar{b} \vee c \vee d) \wedge (\bar{a} \vee b \vee c \vee d) \\
& \wedge (\bar{a} \vee b \vee c \vee \bar{d}) \wedge (\bar{a} \vee b \vee \bar{c} \vee d) \wedge (\bar{a} \vee b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee \bar{b} \vee \bar{c} \vee d)
\end{aligned}
$$

Diese Formel lässt sich mit den Axiomen aus Definition 3.1 noch weiter vereinfachen. Die Vereinfachungen führen dann zu:

$$f = d \wedge (\bar{a} \vee \bar{c} \vee d) \wedge (a \vee c \vee d)$$

Allgemein ist eine Formel erfüllbar, wenn es eine Belegung gibt, so dass die Formel den booleschen Wert „true“ (1) zurückliefert.

Definition 3.6. α heißt erfüllbar, wenn es (mindestens) eine erfüllende Belegung für α gibt, d.h. wenn es (mindestens) eine zu α passende Belegung B mit

$\alpha(B)^3 = 1$ gibt.

Beispiel 3.2. $\alpha = ((a \vee b) \wedge (\overline{a} \vee b))$ ist erfüllbar, da z.B. die Belegung B mit $B(a) = 0$ und $B(b) = 1$ die Formel erfüllt:

$$((0 \vee 1) \wedge (1 \vee 1)) = (1 \wedge 1) = 1$$

An dieser Stelle haben wir bereits alle Kenntnisse, um das für die Komplexitätstheorie sehr bedeutende SAT-Problem zu definieren. Wir werden später darauf zurückgreifen.

Definition 3.7. SAT=$\{\ \alpha|$ die aussagenlogische Formel α ist erfüllbar$\}$

Nach Lemma 3.7 kann jede beliebige Formel in eine äquivalente Formel in konjunktiver Normalform umgewandelt werden. Das entsprechende Problem lautet dann:

Definition 3.8. KNF-SAT=$\{\ \alpha|\ \alpha$ ist eine erfüllbare Formel in konjunktiver Normalform[4]$\}$

Es gibt noch eine weitere Abschwächung des Erfüllbarkeitsproblems SAT, in dem man die Anzahl der Literale pro Klausel einschränkt:

Definition 3.9. k-KNF-SAT=$\{\ \alpha \in KNF - SAT|\ \alpha$ besitzt höchstens k Literale pro Klausel$\}$

Wir werden im nächsten Kapitel sehen, dass alle drei[5] eben definierten Probleme $\mathcal{NP}$-vollständig sind.

[3] Hier ist die Notation so zu verstehen, dass die Werte der Variablen einer Belegung in die Formel eingesetzt werden.

[4] vgl. auch Definition 3.5

[5] k-KNF-SAT ist ab k≥3 $\mathcal{NP}$-vollständig, für kleinere k ist es effizient lösbar.

4. Die Komplexitätstheorie

Nach [Hro01] stellte man bereits in den sechziger Jahren fest, dass „alleine die Existenz eines Algorithmus zur Lösung eines Problems noch keine Garantie für eine erfolgreiche rechnerunterstützte Lösung des Problems ist“. Immer wieder wurden praxisrelevante Probleme gefunden, die zwar algorithmisch lösbar waren, bei deren Lösungsversuch aber alle entworfenen Programme scheiterten und selbst nach tagelanger Berechnung kein Ergebnis lieferten. Waren die Programmierer einfach unfähig? Oder war der Misserfolg in den Problemen selbst begründet? Diese Problematik führte dazu, dass man anfing Probleme nach ihrer Schwierigkeit in Bezug auf den benötigten Rechenaufwand zu klassifizieren. Die Komplexitätstheorie als „Theorie der quantitativen Gesetze und Grenzen der algorithmischen Informationsverarbeitung“ [Hro01] entstand. Heute ist die Komplexitätstheorie ein sehr wichtiges Gebiet in der theoretischen Informatik, die die Klasse effizient (d.h. in polynomieller Zeit) lösbarer Probleme von der Klasse der nicht effizient lösbaren Problemen trennt.
Das berühmteste Problem in der Komplexitätstheorie ist das sogenannte $\mathcal{P}$-$\mathcal{NP}$-Problem. Das $\mathcal{P}$-$\mathcal{NP}$-Problem behandelt die Frage, ob $\mathcal{P} \neq \mathcal{NP}$ gilt. Anders formuliert: „Ist es für bestimmte Probleme einfacher die Korrektheit einer Lösung zu finden als eine solche Lösung überhaupt zu finden?“ Nach [Gol10] wurde die $\mathcal{P} = \mathcal{NP}$-Frage erstmals 1971 von Stephen Cook und Leonid Levin gestellt. Heute geht man zwar davon aus, dass $\mathcal{P}$ von $\mathcal{NP}$ verschieden ist, aber ein Beweis ist bisher nicht gelungen und das Problem blieb ungelöst. Die Frage, ob $\mathcal{P} \neq \mathcal{NP}$ gilt, zählt zu den Millennium-Problemen[1], die im Jahr 2000 vom Clay Mathematics Institute in Cambridge zusammengestellt wurden. Für die Lösung eines Millenium-Problems ist jeweils ein Preisgeld von 1 Million Dollar ausgesetzt. Wer sich an der $\mathcal{P}$-$\mathcal{NP}$-Frage versuchen möchte, findet unter [Coo12] die offizielle Problembeschreibung von Stephen Cook.
Es ist jedoch nicht zu erwarten, dass $\mathcal{P} = \mathcal{NP}$ gilt, denn das würde, wie wir später sehen werden, bedeuten, dass unsere Rechner schon heute die Fähigkeit zum Raten besitzen.

> „Ein praktischer Grund für die Annahme $\mathcal{P} \subsetneq \mathcal{NP}$ basiert auf der 40-jährigen Erfahrung in der Algorithmik. Wir kennen mehr als 3000 Probleme in $\mathcal{NP}$, viele davon sehr intensiv untersucht, für die die besten bekannten deterministischen Algorithmen eine expo-

[1] „The seven Millennium Prize Problems were chosen by the founding Scientific Advisory Board of CMI, which conferred with leading experts worldwide. The focus of the board was on important classic questions that have resisted solution for many years“ [Ins12].

nentielle Komplexität haben. Die Algorithmiker halten es nicht für sehr wahrscheinlich, dass dieser Zustand nur eine Folge ihrer Unfähigkeit ist, existierende effiziente Algorithmen für diese Probleme zu finden.“ [Hro01]

4.1. Turingmaschinen - ein einfaches Rechnermodell

Das Streben nach möglichst effizienten Lösungen von Problemen ist zentral in der Informatik und grundlegend, da Zeit und auch andere Ressourcen beschränkt sind. Probleme sollen nach Möglichkeit immer schneller gelöst werden. Wie zuvor erläutert gibt es aber auch Probleme, die, da die praktische Ausführung eines Algorithmus zu deren Lösung mehr Energie bräuchte, als es im gesamten Universum gibt (vgl. Tabelle 4.1), als praktisch unlösbar gelten. Wann also gilt ein Problem als schwierig oder nicht effizient lösbar? Wie komplex ist ein bestimmtes Problem? Um über die Komplexität von Problemen reden zu können, benötigt man zuerst ein Maß, um eben diese zu „messen“. Wichtig bei der algorithmischen Realisierung von Problemlösungen ist die benötigte Zeit (Zeitkomplexität) sowie der benötigte Speicher (Speicherkomplexität). In der vorliegenden Arbeit werde ich mich auf die Zeitkomplexität von Problemen beschränken. Möglichst effizient hinsichtlich der Speicherkomplexität zu arbeiten fällt in das Gebiet des Algorithm Engineerings.
Um zu zeigen, wie wichtig effiziente Algorithmen sind, betrachten wir Tabelle[2] 4.1.

Betrachten wir hier etwa die Spalten zu der quadratischen Laufzeit (n^2) im Vergleich zu der kubischen Laufzeit (n^3), so fällt sogar bei einer recht kleinen Eingabelänge von $n = 1024$ ein beträchtlicher Unterschied auf. Legt man der Tabelle einen Rechner zugrunde, der für die Bearbeitung eines Rechenbefehls etwa 10^{-9} Sekunden benötigt, so ist der Sprung von etwa 15 Minuten zu etwa 10 Jahren enorm! Bei einer Laufzeit der Ordnung $n!$ gilt ein Algorithmus schon ab einer Eingabelänge von $n = 32$ als praktisch unmöglich auszuführen. Seine Ausführung dauert länger als das Universum alt ist![3]
Die von einem Algorithmus benötigte Zeit ist abhängig von der Schnelligkeit des ausführenden Rechners, also von dessen Hardware. Die Klassifizierung von Problemen sollte aber allgemeingültig sein und eben nicht von der Hardware und Schnelligkeit eines bestimmten Rechners abhängen. Um Zeitkomplexität unabhängig zu definieren, verständigt man sich in der Komplexitätstheorie darauf, unter der Zeitkomplexität einer Berechnung die Anzahl an elementaren Operationen, die ausgeführt werden müssen, zu verstehen. Weiterhin

[2] Hier wurde ein Rechner, der für die Ausführung eines einzigen Rechenbefehls etwa 10^{-9} Sekunden benötigt, zugrunde gelegt.
[3] Nach [The12] ist das Universum rund 14 Milliarden Jahre alt.

n	n^2	n^3	n^{10}	2^n	$n!$
16	256	4.096	$\geq 10^{12}$	65536	$\geq 10^{13}$
32	1.024	32.768	$\geq 10^{15}$	$\geq 4 \cdot 10^9$	$\geq 10^{31}$
64	4.096	262.144	$\geq 10^{18}$	$\geq 6 \cdot 10^{19}$	
					mehr als
128	16.384	2.097.152	mehr als	mehr als	10^{14} Jahre
256	65.536	16.777.216	10 Jahre	600 Jahre	
512	262.144	134.217.728			
1024	1.048.576	$\geq 10^9$			
10^6	$\geq 10^{12}$	$\geq 10^{18}$			
	mehr als	mehr als			
	15 Minuten	10 Jahre			

Tabelle 4.1.: Übersicht verschiedener Laufzeiten

benötigt man ein allgemeines Rechnermodell, welches Aussagen über alle bekannten und ebenso über alle vorstellbaren Rechner erlaubt. Damit Aussagen möglichst einfach zu formulieren sind und Beweise nicht Unmengen von Details beachten müssen, sollte das Rechnermodell möglichst einfach sein. Turingmaschinen werden wegen eben dieser positiven Eigenschaften als Basismodell der theoretischen Informatik benutzt, weil sie von vielen als das einfachste Rechnermodell angesehen werden. Im folgenden Kapitel soll zuerst ein historischer Überblick über die Entwicklung von Turingmaschinen gegeben werden. Danach wird der Aufbau und die Funktionsweise der Turingmaschine erläutert, um die Basis der Komplexitätstheorie im nächsten Kapitel zu legen.

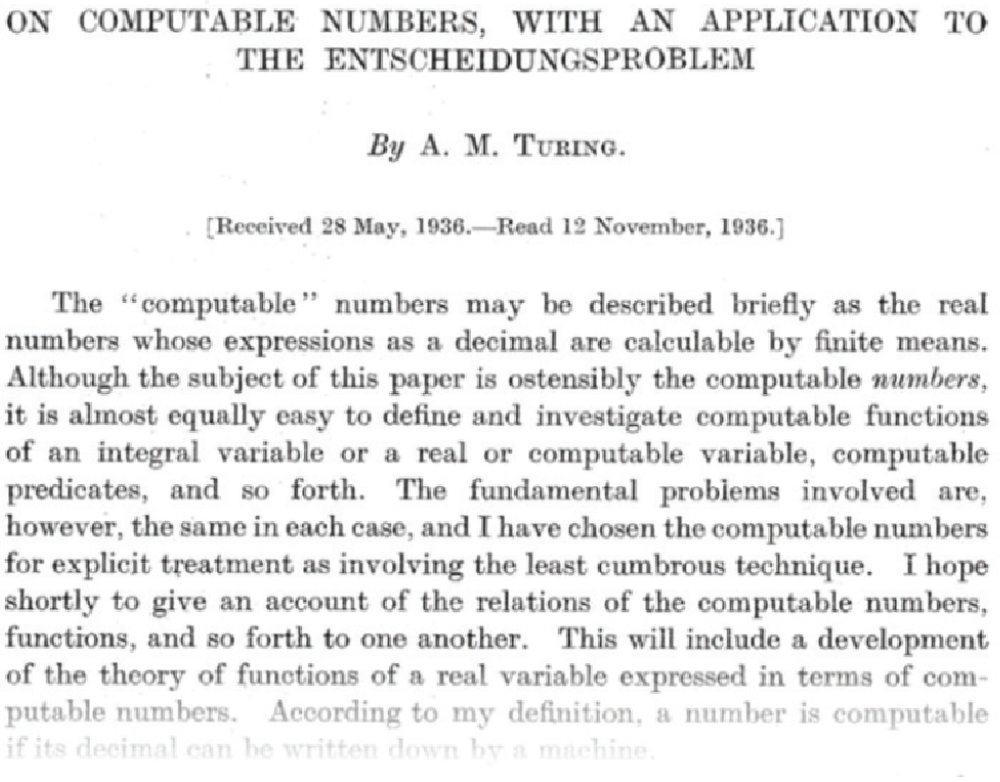
4.1.1. Das Modell der Turingmaschine

Abbildung 4.1.: A. M. Turing

Die Turingmaschine gehört zu den grundlegenden Konzepten der theoretischen Informatik. Im Jahre 1936, lange bevor moderne Computer, wie sie heute zum Standard gehören, gebaut wurden, gelang es dem britischen Mathematiker Alan Mathison Turing das abstrakte Maschinenmodell der Turingmaschine, auf dem große Teile der modernen Berechenbarkeitstheorie und der Komplexitätstheorie basieren, zu entwickeln. Als Antwort auf das Hilbertsche Entscheidungsproblem[4]

[4] Das Hilbertsche Entscheidungsproblem fragt danach, ob es ein Verfahren gibt, welches für jede ausreichend formalisierte Aussage der Mathematik entscheidet, ob diese wahr oder falsch ist. [Mat12]

präsentierte Turing in seiner Arbeit „On computable numbers, with an application to the Entscheidungsproblem“ [Tur36] das Konstrukt der Turingmaschine, um die Menge der berechenbaren Zahlen zu charakterisieren.



ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

Abbildung 4.2.: Titelblatt von Turings Originalarbeit [Hof11]

Für die Entwicklung seiner Turingmaschine überlegte sich Turing, wie ein Mensch eine Berechnung ausführt, und rekonstruierte diese Vorgehensweise.

> „Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetik book.“ [Tur36]

Er begann seine „Überlegungen über die Berechenbarkeit mit dem, was er seit seiner Kindheit zum Rechnen verwendete: einem leeren Stück karierten Papier.“ [Hof11] Da es möglich ist alle Berechnungen, die per Hand auf Papier gemacht werden können, auch auf einem eindimensionalen Band durchzuführen, abstrahierte Turing die zweidimensionale Gestalt. Die erste Komponente der Turingmaschine ist demnach ein lineares, in der Vorstellung nach links und nach rechts unendliches, Band, das in Zellen unterteilt ist. Die Turingmaschine manipuliert ihr Band mithilfe eines Lese-Schreib-Kopfes, der es ermöglicht, das Symbol des aktuellen Feldes zu betrachten und danach die Aufmerksamkeit auf eines der Nachbarfelder zu lenken. Weiterhin ging Turing davon aus, dass es nur eine endliche Menge von Symbolen gibt, die die Felder des Bandes

füllen können. Zu Beginn einer Berechnung ist lediglich die Eingabe auf dem Band abgelegt. Alle anderen Zellen enthalten das Blanksymbol B.

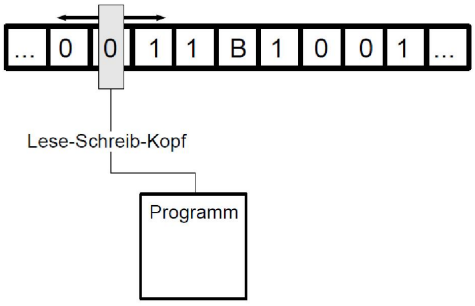


Abbildung 4.3.: Skizze einer Turingmaschine

Während zu Anfang also alle Symbole entweder aus dem Eingabealphabet $\sum$ stammen oder mit dem Blanksymbol B übereinstimmen, dürfen die Zellen im Verlauf mit Symbolen bzw. Buchstaben aus dem Arbeitsalphabet Γ beschriftet werden. Für das Arbeitsalphabet Γ gilt: $\sum \cup \{\mathtt{B}\} \cup \Gamma$, dh. es können von der Turingmaschine Symbole aus dem Eingabealphabet oder wiederum das Blanksymbol B genutzt werden. Turing ging außerdem davon aus, dass „sich das menschliche Gehirn im Zuge einer Berechnung zu jedem Zeitpunkt in einem von endlich vielen Zuständen befindet“ [Hof11]. Technisch realisiert wurde diese Annahme durch die sogenannte endliche Menge von Zuständen Q. Zu Beginn der Rechnung befindet sich die Turingmaschine in einem ausgezeichneten Startzustand $q_0 \in Q$. Wie rechnet eine Turingmaschine? Sie folgt einem festgelegten Programm, welches durch die Zustandsüberführungsfunktion δ mit $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{links, bleib, rechts}\}$ bestimmt ist. Die Menge {links, bleib, rechts} steht hier für die Möglichkeiten der Bewegung des Lese-Schreib-Kopfes. Der Lese-Schreib-Kopf kann auf der aktuellen Zelle verbleiben, die rechte oder die linke Nachbarzelle besuchen. Befindet sich die Maschine im Zustand $q \in Q$ und liest den Zellinhalt $\gamma \in \Gamma$, so wird der Befehl $\delta(q, \gamma)$ ausgeführt. In Abhängigkeit von den Argumenten der Zustandsüberführungsfunktion δ schreibt die Turingmaschine ein Symbol auf die Zelle des Bandes, von dem gerade gelesen wurde[5], ändert den Zustand und bewegt den Lese-Schreib-Kopf gemäß der angegebenen Richtung[6].

Definition 4.1. Man sagt, dass die Turingmaschine hält, wenn sich die Maschine im Zustand $q \in Q$ befindet, den Buchstaben $\gamma \in \Gamma$ liest und wenn $\delta(q, \gamma) = (q, \gamma, \text{bleib})$ gilt. [Sch11]

Hierbei handelt es sich um eine besondere Form einer Endlosschleife. Weder der Zustand, in dem sich die Turingmaschine befindet, noch der Zellinhalt, den der

[5] Dabei kann es sich auch um das gleiche Symbol handeln, so dass sich die Eingabe auf dem Band nicht zwingend ändert.

[6] Der Kopf kann auch auf der aktuellen Zelle stehen bleiben; *bleib* $\in$ {links, bleib, rechts}.

Lese-Schreib-Kopf liest, ändert sich. Zusätzlich verharrt der Lese-Schreib-Kopf auf ein und derselben Position. Die Turingmaschine wird also immer wieder (sozusagen endlos) aufgefordert dasselbe zu tun.
Abschließend zeichnete Turing eine Teilmenge $F \subseteq Q$ als Menge der akzeptierenden Zustände aus.

Definition 4.2. Die Turingmaschine akzeptiert eine bestimmte Eingabe ω, wenn sie in einem Zustand aus F hält.

Zusammenfassend wird eine Turingmaschine M also durch ein 6-Tupel $(Q, \sum, \delta, q_0, \Gamma, F)$ beschrieben. Dabei ist

Q	eine endliche Menge, die Zustandsmenge von M
$\sum$	das Eingabealphabet
δ	die Zustandsüberführungsfunktion
$q_0 \in Q$	der Anfangszustand
Γ	das Arbeitsalphabet
F	die Menge der akzeptierenden Zustände

Definition 4.3. Nach [Sch11] sagt man, „dass die Turingmaschine M= $(Q, \sum, \delta, q_0, \Gamma, F)$ die Sprache L(M)=$\{\omega \mid$ M akzeptiert $\omega\}$ erkennt."

Beispiel 4.1. Gegeben sei die folgende Turingmaschine[7] : $Q = \{q_0, q_1, q_2\}$, Eingabealphabet $\sum = \{1\}$, Startzustand q_0, Bandalphabet $\Gamma = \{1, \mathtt{B}\}$ und $F = q_2$ sei die Menge der akzeptierenden Zustände. Die Zustandsüberführungsfuntion δ sei

$$\begin{aligned}
\delta(q_0, \mathtt{B}) &= (q_1, \mathtt{B}, \text{bleib}) \\
\delta(q_0, 1) &= (q_1, 1, \text{rechts}) \\
\delta(q_1, \mathtt{B}) &= (q_1, \mathtt{B}, \text{bleib}) \\
\delta(q_1, 1) &= (q_2, 1, \text{rechts}) \\
\delta(q_2, \mathtt{B}) &= (q_2, \mathtt{B}, \text{bleib}) \\
\delta(q_2, 1) &= (q_1, 1, \text{rechts})
\end{aligned}$$

Interpretiert man einen Befehl $\delta(q, \gamma) = (q', \gamma', \text{Richtung})$ für $q, q' \in Q,\ \gamma, \gamma' \in \sum$ und Richtung $\in$ {links, bleib, rechts} graphisch wie in Abbildung 4.4, dann lässt sich die konstruierte Turingmaschine M wie in Abbildung 4.5 darstellen. Betrachten wir den Graphen in Abbildung 4.5 genauer, so können wir die Funktionsweise der Turingmaschine ableiten.

[7] Hier ist der Aufbau der Turingmaschine aus einer Übungsaufgabe zu der Veranstaltung „Algorithmentheorie" von Herrn Prof. Dr. Meyer entnommen. Darstellungen und Erklärungen wurden selbst erstellt

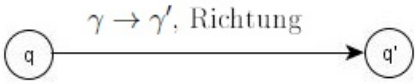


Abbildung 4.4.: Graphische Interpretation eines Befehls einer Turingmaschine

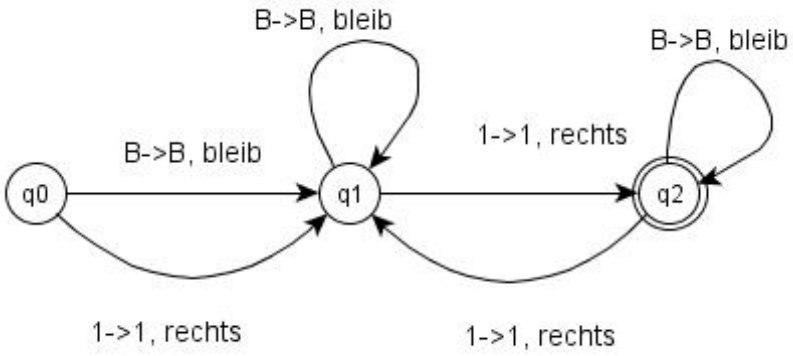


Abbildung 4.5.: Graph der Turingmaschine aus Beispiel 4.1

Wenn sich die Maschine im Zustand q_0, also im Startzustand, befindet und das Blanksymbol `B` liest, so wechselt sie in den Zustand q_1 und „überschreibt“ das Blanksymbol mit einem Blanksymbol. Der Lese-Schreib-Kopf bleibt auf seiner Position. Liest die Maschine im Startzustand q_0 jedoch eine 1, „überschreibt“ sie diese mit einer 1, wechselt in den Zustand q_1 und der Lesekopf bewegt sich eine Zelle nach rechts.
Betrachten wir nun Zustand q_1. Liest die Maschine ein Blanksymbol, überdruckt sie dieses mit einem Blanksymbol. Der Zustand wechselt nicht und auch der Lesekopf bleibt auf seiner Position. An dieser Stelle befindet sich die Turingmaschine in einer Art Endlosschleife wie sie in Definition 4.1 beschrieben wurde. Es wird immer wieder das Blanksymbol `B` eingelesen und weder der Zustand noch die Position des Lese-Schreib-Kopfes ändert sich. Nach Definition 4.1 hält die Turingmaschine. Zu beachten ist hier, dass es sich bei dem Zustand q_1 nicht um einen akzeptierenden Zustand handelt. Nach Definition 4.3 wurde die Eingabe folglich nicht von der Maschine akzeptiert.
Liest die Maschine im Zustang q_1 eine 1, so wechselt sie in Zustand q_2 (einen akzeptierenden Endzustand). Weiterhin wird die 1 mit einer 1 „überschrieben“ und der Lese-Schreib-Kopf wandert eine Zelle nach rechts. Es bleibt den Zustand q_2 genauer zu analysieren.
Befindet sich die Turingmaschine in Zustand q_2, so bewirkt das Lesen eines Blanksymbols, dass die Maschine den Zustand nicht verlässt und immer wieder das Blanksymbol mit einem Blanksymbol überschreibt. Der Lese-Schreib-Kopf behält gleichzeitig seine Position. Die Maschine hält also wieder im Sinne von Definition 4.1. Da es sich bei dem Zustand q_2 aber um einen akzeptierenden Zustand handelt, akzeptiert die Turingmaschine entsprechend Definition 4.3

die Eingabe. Steht in der Zelle, auf der der Lesekopf sich momentan befindet eine 1, so wechselt die Maschine zurück in den Zustand q_1 und schreibt eine 1 in die Zelle. Der Lesekopf wandert in diesem Fall auf dem Band eine Position nach rechts.
Zusammenfassend überprüft die konstruierte Turingmaschine, ob die Eingabe mit mindestens zwei Einsen beginnt und aus einer geraden Anzahl von Symbolen besteht. Nur solche Eingaben werden von der Maschine akzeptiert und führen zu einem Halten in dem akzeptierenden Zustand q_2.

Beispiel 4.2. Im Folgenden möchte ich ein weiteres recht anschauliches Beispiel einer Turingmaschine vorstellen. Sei also hier eine Turingmaschine M wie folgt gegeben:
$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, Eingabealphabet $\sum = \{1\}$, Startzustand q_1, Bandalphabet $\Gamma = \{1, 0\}$ und der Zustand q_5 sei der einzige akzeptierende Zustand. Die Zustandsüberführungsfuntion δ sei

$$\begin{aligned}
\delta(q_0, 1) &= (q_1, 0, \text{rechts}) \\
\delta(q_0, 0) &= (q_5, 0, \text{bleib}) \\
\delta(q_1, 1) &= (q_1, 1, \text{rechts}) \\
\delta(q_1, 0) &= (q_2, 0, \text{rechts}) \\
\delta(q_2, 1) &= (q_2, 1, \text{rechts}) \\
\delta(q_2, 0) &= (q_3, 1, \text{links}) \\
\delta(q_3, 1) &= (q_3, 1, \text{links}) \\
\delta(q_3, 0) &= (q_4, 0, \text{links}) \\
\delta(q_4, 1) &= (q_4, 1, \text{links}) \\
\delta(q_4, 0) &= (q_0, 1, \text{rechts})
\end{aligned}$$

Benutzt man die in Beispiel 4.1 vorgestellte graphische Interpretation, erhält man den resultierende Graph wie in Abbildung 4.6 abgebildet.

Ich möchte die Funktionsweise der Turingmaschine mithilfe des Graphen und zwei beispielhaften Eingaben erläutern.
Sei die Zahl 01101 im Binärsystem[8] die erste Eingabe. Nach der vorgegebenen Konfiguration, befindet sich die Turingmaschine zu Beginn im Anfangszustand q_0. Der Lesekopf steht auf der ersten Zelle der Eingabe und liest demnach eine 0. Die aktuelle Zelle wird mit einer 0 „überschrieben“ und der Zustand wechselt in den akzeptierenden Zustand q_5. Nach Definition 4.1 hält die Turingmaschine schon jetzt nach dem Einlesen der ersten Ziffer ohne die Eingabe weiter zu beachten.
Was passiert mit Eingaben, deren erste Ziffer von Null verschieden ist? Sei also

[8] Das Dualsystem (lat. dualis = zwei enthaltend), auch Zweiersystem oder Binärsystem genannt, ist ein Zahlensystem, das zur Darstellung von Zahlen nur zwei verschiedene Ziffern benutzt. [Wik12i]

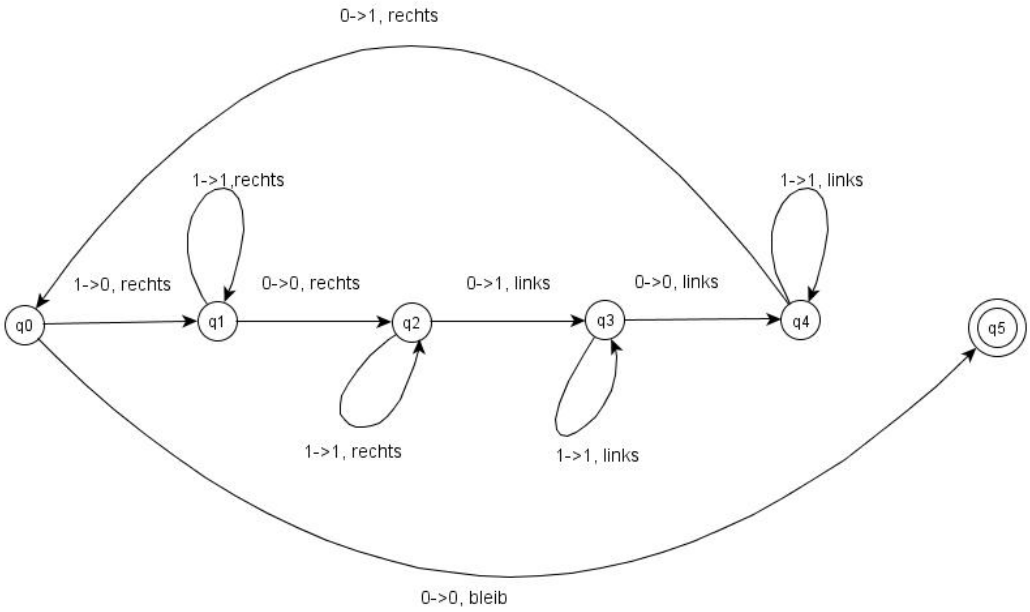


Abbildung 4.6.: Graph der Turingmaschine aus Beispiel 4.2

die Eingabe nun 11000. Wieder befindet sich die Maschine im Anfangszustand q_0 und der Lese-Schreib-Kopf auf der ersten Ziffer der Eingabe. Nun wird eine 1 gelesen. Die Zustandsüberführungsfunktion $\delta(q_0, 1) = (q_1, 0, \text{rechts})$ sorgt dafür, dass die aktuelle Zelle mit einer 0 beschrieben wird und der Zustand der Maschine sich in Zustand q_1 ändert. Der Lesekopf wandert zur rechten Nachbarzelle. In dieser steht wiederum eine 1, die mit einer weiteren 1 überschrieben wird. Die Maschine behält den Zustand und der Lesekopf bewegt sich auf die rechte nächste Ziffer der Eingabe. So wird der Graph in Abbildung 4.6 Ziffer für Ziffer weiter durchlaufen. Da sich die Erklärungen jetzt immer wiederholen und ich denke, dass das Prinzip schon erkennbar ist, möchte ich dazu übergehen, die Schritte der Turingmaschine in abgekürzter Darstellung (wie sie auch in [Wik12h] verwendet wird) zu präsentieren:

Hat man den ganzen Ablauf Schritt für Schritt verfolgt, so erkennt man, dass die Turingmaschine entweder nichts tut (dann, wenn die Eingabe mit einer Null beginnt) oder aber die Anzahl der Einsen (von Einserfolgen) verdoppelt. Eine alternative Beschreibung der „Verdopplung der Einsen“ kann auch in [Wik12h] nachgelesen werden.

Nachdem wir uns die Funktionsweise einer Turingmaschine klar gemacht haben, bleibt die Frage nach der Zeitkomplexität zu beantworten.

Schritt	aktueller Zustand	Bandinhalt
1	q_0	11000
2	q_1	01000
3	q_1	01000
4	q_2	01000
5	q_3	01010
6	q_4	01010
7	q_4	01010
8	q_0	11010
9	q_1	10010
10	q_2	10010
11	q_2	10010
12	q_3	10011
13	q_3	10011
14	q_4	10011
15	q_0	11011
16	q_5	hält

Tabelle 4.2.: Die Schrittfolge der Turingmaschine als Übersicht. Hier ist die jeweilige Position des Lese-Schreib-Kopfes mit roter Farbe markiert.

Definition 4.4. Sei M eine deterministische Turingmaschine auf dem Eingabealphabet $\sum$. Die worst-case Laufzeit $\text{Zeit}_M(n)$ von M ist

$$\text{Zeit}_M(n) = \max\{\text{Schritte}_M(x) | x \in \sum^n\},$$

d.h. die worst-case Laufzeit ergibt sich aus der maximalen Anzahl an Rechenschritten, die M auf Eingaben der Länge n macht.

4.2. Exkurs: Landau Notation

Bei der Bestimmung der Laufzeit ist man in der Regel an der Laufzeit im schlechtesten Fall interessiert. Im Allgemeinen interessiert man sich nicht für den genauen Funktionswert der Laufzeit eines Algorithmus. Wichtig ist hier der qualitative Verlauf einer Funktion. Um verschiedene Algorithmen, oder besser, deren Laufzeitverhalten miteinander vergleichen zu können, bildet man „Klassen“ von Laufzeiten. Um dies zu ermöglichen, bedient sich die Informatik an einem Konzept der Mathematik, den sogenannten Landau-Symbolen. Die Notation der Landau-Symbole geht auf Edmund Landau[9] zurück. Die Landau-Symbole drücken lokale Wachstumseigenschaften einer Funktion f durch entsprechende Eigenschaften einer (einfacheren) Vergleichsfunktion g aus.

[9] 1877-1938, Professor in Göttingen, verlor 1933 aufgrund der Rassengesetze seine Venia legendi, später Gastprofessor in Cambridge. [Wal02]

Ich möchte mich in diesem Kapitel darauf beschränken nur zwei (Groß-Oh und Theta) zu erläutern und zu definieren. Es sei jedoch der Vollständigkeit halber erwähnt, dass es noch weitere Symbole gibt, die mehr oder weniger häufig in der Informatik genutzt werden.
Zunächst führe ich hier eine Schreibweise ein, die für zwei Funktionen f und g beschreibt, dass f für „große“ n ($n \to \infty$) höchstens so schnell wächst wie g.

Definition 4.5. Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ zwei Funktionen, die jeder natürlichen Zahl eine nicht-negative reelle Zahl zuordnen. Es gilt $f = \mathcal{O}(g)$[10] (gesprochen: „f ist von der Größenordnung Groß-Oh von g“) genau dann, wenn es eine positive Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ gilt:

$$f(n) \leq c \cdot g(n)$$

Wie ist Definition 4.5 zu verstehen? Umgangssprachlich sagt die Ungleichung, dass die Funktion f von der Funktion g dominiert („gedeckelt“) wird. Allerdings muss dieser Sachverhalt nicht für alle $n \in \mathbb{N}$ gelten, sondern nur für solche die größer gleich einer „Hausnummer“ n_0 sind. Eine vergleichbare außermathematische Aussage könnte sein: „In der Musterstraße sind alle Häuser ab Hausnummer 34 blau.“. Welche Farbe die Häuser der Straße vor Nummer 34 haben, wissen wir nicht. Wir haben es also mit einer Aussage zu tun, die erst ab einem bestimmten Wert gültig ist. Ab diesem Wert gilt sie aber immer! Im Endeffekt interessiert uns gar nicht, was irgendwo passiert. Wir nutzen die Abschätzung von Funktionen für den asymptotischen Fall, also wenn $n \to \infty$.

Beispiel 4.3. Betrachte die Funktion $f(n) = n^2 + n$ und die Funktion $g(n) = n^3$ mit $n \in \mathbb{N}$.

n	$f(n)$	$g(n)$
1	2	1
2	6	8
3	12	27
4	20	64
5	30	125
6	42	216
7	56	343
8	72	512

Tabelle 4.3.: Wertetabelle zu den gegebenen Funktionen

Wie lassen sich die Größenordnungen der beiden Funktionen vergleichen? Man sieht, dass zu Beginn (vgl. $n = 1$) die Funktion f die Funktion g dominiert. Ab $n = 2$ aber dominiert Funktion g. Was bedeutet dieser Sachverhalt interpretiert

[10] Bitte Anmerkung unter Lemma 4.2 beachten.

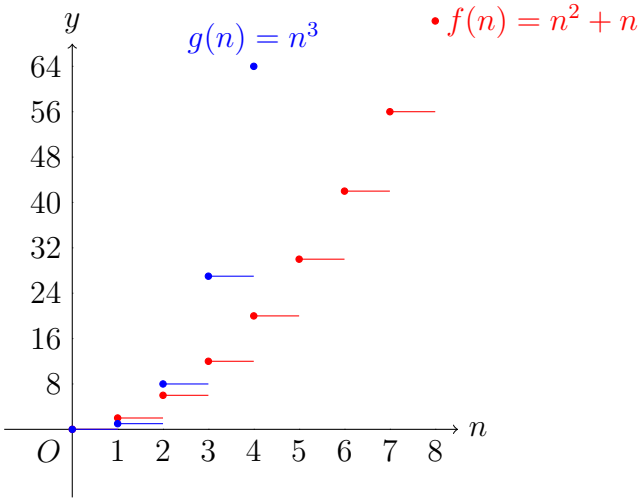


Abbildung 4.7.: Funktionsgraph für $f(n) = n^2 + n$ und $g(n) = n^3$

nach Definition 4.5? Für Konstante $c = 1$ und „Hausnummer“ $n_0 = 2$ gilt: Für alle $n \geq n_0$, d.h. für alle $n \geq 2$ ist die Ungleichung $f(n) \leq c \cdot g(n)$ erfüllt. Nach Definition 4.5 gilt also $n^2 + n = \mathcal{O}(n^3)$. Die Funktion f wächst demnach asymptotisch höchstens so schnell wie die Vergleichsfunktion g. An diesem Beispiel lässt sich auch gut illustrieren, dass hier die Konstante $c > 0$ und die „Hausnummer“ $n_0 \in \mathbb{N}$ ein voneinander abhängiges Paar bilden. Möchte man die Aussage ab $n_0 = 1$ zeigen, so muss die Konstante $c = 2$ gewählt werden.
Die Funktion f lässt sich aber auch schärfer abschätzen: Für $f(n) = n^2 + n$ und $g(n) = n^2$ ist Definition 4.5 für $c = 2$ und $n_0 = 1$ erfüllt. Und es gilt somit $n^2 + n = \mathcal{O}(n^2)$.

Da es bei manchen Funktionen recht unbequem ist, eine Konstante und eine „Hausnummer“ $n_0 \in \mathbb{N}$ zu finden, so dass die Ungleichung in Definition 4.5 erfüllt ist, betrachten wir nun folgendes Lemma, das uns häufig den Größenvergleich zweier Funktionen vereinfacht.

Lemma 4.1. *Wir definieren*

$$c := lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Falls der Grenzwert existiert und $0 \leq c < \infty$*, dann ist* $f(n) = \mathcal{O}(g(n))$ *(oder kürzer:* $f = \mathcal{O}(g)$*).*

Wie entsteht die Eingrenzung der Konstanten c in Lemma 4.1? Eine bekannte Grundvorstellung von Brüchen sind Verhältnisse. Hier setzen wir also zwei Funktionen f und g ins Verhältnis

$$\frac{f(n)}{g(n)}$$

und betrachten den Grenzwert, da wir wissen wollen, wie die Funktionen sich im asymptotischen Wachstum verhalten:

$$lim_{n\to\infty}\frac{f(n)}{g(n)}$$

Welche Fälle sind nun für die Definition 4.5, die aussagt, dass eine Funktion f asymptotisch höchstens so schnell wächst wie eine Funktion g, interessant?

1. Die Funktion f kann asymptotisch langsamer wachsen wie g
2. Die Funktion f kann asymptotisch gleich schnell wachsen wie g

Im ersten Fall ergibt die Betrachtung des Grenzwertes $lim_{n\to\infty}\frac{f(n)}{g(n)}$, dass der Bruch kleiner als 1 ist und somit der Grenzwert gegen Null geht. Hier gilt also $c = 0$, was die linke Relation in Lemma 4.1 erklärt. Im zweiten Fall sind alle möglichen Zahlen größer gleich 1 für den Bruch und somit auch für den Grenzwert denkbar. Warum aber nicht $c = \infty$? Wann kann der Grenzwert unendlich sein? Dann, wenn im Bruch der Zähler schneller wächst wie der Nenner. Also dann, wenn die Funktion f schneller wächst als die Funktion g. Genau diesen Fall haben wir in Definition 4.5 ausgeschlossen. Die Funktion f darf asymptotisch nur höchstens so schnell wachsen wie g. Somit ist auch die rechte Ungleichung in Lemma 4.1 erklärt.

Beispiel 4.4. Wir wollen die beiden Funktionen $f(n) = 5n + 3$ und $g(n) = n$ vergleichen.

n	$f(n)$	$g(n)$
1	8	1
2	13	2
3	18	3
4	23	4
5	28	5
6	33	6
7	38	7
8	59	8

Tabelle 4.4.: Wertetabelle für $f(n) = 5n + 3$ und $g(n) = n$

Betrachten wir also den Grenzwert nach Lemma 4.1:

$$lim_{n\to\infty}\frac{f(n)}{g(n)} = lim_{n\to\infty}\frac{5n+3}{n} = lim_{n\to\infty}\frac{n\cdot(5+\frac{3}{n})}{n} = lim_{n\to\infty}(5+\frac{3}{n}) = 5$$

Es gilt also $0 \leq c < \infty$ für $c = 5$ und folglich $5n + 3 = \mathcal{O}(n)$ Die Funktion $f(n) = 5n+3$ gehört also zu den Funktionen mit linearem Wachstum. Betrachten wir den Graphen der beiden Funktionen, so fällt auf, dass die Funktion f

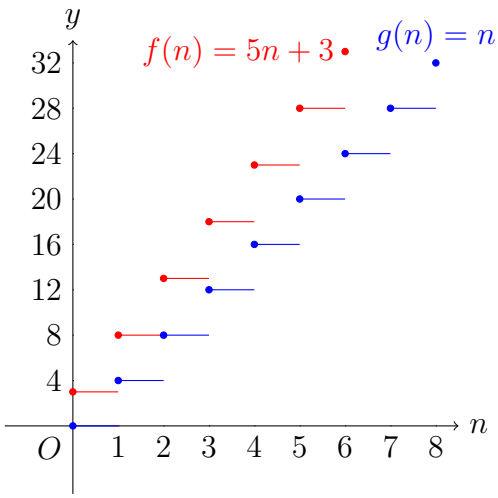


Abbildung 4.8.: Funktionsgraph für $f(n) = 5n + 3$ und $g(n) = n$

die Funktion g dominiert. Dennoch haben wir gerade bewiesen, dass $f = \mathcal{O}(g)$ gilt und die Funktion f höchstens so schnell wächst wie g. Wie kann das sein? Genau dieser Effekt ist erwünscht. Es werden Größenordnungen verglichen! Betrachten wir umgekehrt

$$lim_{n\to\infty}\frac{g(n)}{f(n)} = lim_{n\to\infty}\frac{n}{5n+3} = lim_{n\to\infty}\frac{n}{n\cdot(5+\frac{3}{n})} = lim_{n\to\infty}\frac{1}{(5+\frac{3}{n})} = \frac{1}{5}$$

Das bedeutet es gilt auch $g = \mathcal{O}(f)$ und die Funktionen sind von gleicher asymptotischer Größenordnung. Wir wollen diesen Sachverhalt, der gerade bei Laufzeitanalysen von Algorithmen in der Informatik immer wieder auftritt, in der nächsten Definition präzisieren.

Definition 4.6. Gilt $f = \mathcal{O}(g)$ und gleichzeitig $g = \mathcal{O}(f)$ so sagen wir, dass die beiden Funktionen von asymptotisch gleicher Größenordnung sind.
Notation: $f = \Theta(g)$ („f ist gleich Theta von g“).

Bevor im nächsten Kapitel die Thematik zu Komplexitätsklassen gelenkt wird, möchte ich noch abschließend einige Bemerkungen und Rechenregeln zur Groß-Oh-Notation ergänzen.

Lemma 4.2. *Seien* $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

(a) *Falls* $f = n^k$ *und* $g = n^l$ *mit* $k \leq l$, *dann gilt* $f = \mathcal{O}(g)$.

(b) *Sei* $c \in \mathbb{R}_{\geq 0}$ *eine positive reelle Zahl, dann gilt* $c \cdot f(n) = \mathcal{O}(f(n))$.

(c) *Sei* $f_1 = \mathcal{O}(g_1)$ *und* $f_2 = \mathcal{O}(g_2)$, *dann gilt:* $f_1 + f_2 = \mathcal{O}(g_1 + g_2)$.

(d) *Sei* $f_1 = \mathcal{O}(g_1)$ *und* $f_2 = \mathcal{O}(g_2)$, *dann gilt:* $f_1 \cdot f_2 = \mathcal{O}(g_1 \cdot g_2)$.

Anmerkung: Ich habe im gesamten Kapitel 4.2 die Schreibweise „$f = \mathcal{O}(g)$“ benutzt. Im mathematischen Sinne allerdings ist hier anzumerken, dass es sich in der Definition der Groß-Oh-Notation (vgl. Definition 4.5) strenggenommen um Funktionsklassen handelt. Korrekt müsste es heißen: „f ist in der Klasse der Funktionen der Ordnung X enthalten“, d.h. hier wäre ein „$\in$“ statt eines Gleichheitszeichens zu verwenden. Schaut man genauer hin, so ist die Schreibweise mit „=“ mathematisch sogar falsch: Betrachten wir hier die beiden Funktionen $f(x) = 2x^2 + x$ und $g(x) = x^2 + 1$. Bei beiden Funktionen handelt es sich um quadratische Funktionen, d.h. nach Definition 4.5 gilt:

$$f = \mathcal{O}(x^2) \quad \text{und} \quad g = \mathcal{O}(x^2)$$

Ein Gleichheitszeichen bedeutet, dass man gleichsetzen darf. Es folgt also:

$$f(x) = \mathcal{O}(x^2) = g(x)$$

Diese Aussage bedeutet allerdings, dass es sich bei den Funktionen um ein und diesselbe handelt. Wir haben aber die Funktionen f und g so nicht definiert! Es handelt sich demnach um einen Widerspruch, der durch die Verwendung des Gleichheitszeichens hervorgerufen wird.

Der Grund, weswegen ich dennoch ein Gleichheitszeichen verwendet habe, liegt darin, dass die O-Notation in der Informatik meistens in dieser Form verwendet wird. Bei der Analyse von Laufzeiten zum Beispiel, kann es sein, dass man die durch die Landau-Notation abgeschätzten Laufzeiten, beispielsweise bei der Hintereinanderausführung zweier Algorithmen, addieren möchte. Im Hinblick auf Addition ist der Umgang mit einem Gleichheitszeichen dann leichter. Die „strengen“ Mathematiker mögen mir verzeihen.

4.3. Die Komplexitätsklasse $\mathcal{P}$

In der Komplexitätsklasse $\mathcal{P}$ sollen alle Probleme, die effizient gelöst werden können, enthalten sein. Um welche Probleme handelt es sich also? Intuitiv sind effiziente Algorithmen solche, die so implementiert werden können, dass die Schrittanzahl im Verhältnis zur Eingabelänge nur moderat wächst. Um zu erlauben, dass die Eingabe komplett gelesen werden kann, wird mindestens lineare Laufzeit benötigt. Algorithmen mit exponentieller Laufzeit sollten vermieden werden. Zusätzlich sollte eine gute Definition der Klasse gegenüber Kompositionen von Algorithmen abgeschlossen sein. So sollte die Hintereinanderausführung zweier effizienter Algorithmen immer noch effizient sein. Um diese Bedingungen zu erfüllen, werden Polynome betrachtet. Lässt sich die Zeitkomplexität als Polynom der Form n^k, wobei n die Eingabelänge bezeichnet, angeben, so gehört das Problem zur Klasse $\mathcal{P}$. Im vorherigen Kapitel wurde mit der Konstruktion von Turingmaschinen ein mögliches Rechnermodell vorgestellt, um Probleme zu implementieren. Wir wollen nun unsere bisher erhaltenen Erkenntnisse nutzen, um eine Definition der Klasse $\mathcal{P}$ anzugeben.

Definition 4.7 (Die Klasse $\mathcal{P}$)**.** Die Klasse $\mathcal{P}$ besteht aus allen Sprachen L für die es eine deterministische Turingmaschine M mit L=M(L) gibt, so dass $Zeit_M(n) = \mathcal{O}\left((n+1)^k\right)$ für eine Konstante k gilt. Wenn L$\in \mathcal{P}$ heißt L effizient berechenbar. [Sch11]

Die erste Beobachtung, die einem auffallen sollte, ist, dass die Klasse $\mathcal{P}$ sehr groß ist. Es gehören zum Beispiel auch Probleme mit Laufzeit $\mathcal{O}(n^{1000})$ zu $\mathcal{P}$. Für Probleme, die nicht in der Klasse $\mathcal{P}$ liegen, bedeutet das, dass sie wirklich sehr „schlimme“ Laufzeiten haben. Um die Klasse besser kennen zu lernen, stelle ich hier drei leichte Beispiele vor.

Beispiel 4.5. Wir greifen hier auf die in Kapitel 3 eingeführte Boolesche Algebra zurück und betrachten das sogenannte 2-SAT-Problem:

$$\text{2-SAT} = \{\, \alpha \mid \alpha \text{ ist eine erfüllbare Formel in konjunktiver Normalform mit höchstens 2 Literalen pro Klausel}\}$$

Für eine gegebene aussagenlogische Formel F in konjunktiver Normalform mit höchstens zwei Literalen pro Klausel ist also zu entscheiden, ob sie erfüllbar ist oder nicht. Betrachten wir zum Beispiel die Formel

$$\alpha = (a \vee b) \wedge (\overline{a} \vee b)$$

Nun gibt es verschiedene Vorgehensweisen, das Problem zu lösen. Man könnte versuchen, die Lösung durch Ausprobieren zu erhalten. Ein geübter Betrachter wird ohne große Überlegung beispielsweise die erfüllende Belegung a=1 und b=1 finden. Bei einer so kleinen Formel ist das auch kein Problem. Aber denkt man an Formeln mit beispielsweise 40 Variablen, dann wird das schon weitaus weniger Spaß machen. Wir wollen auf einen Algorithmus hinarbeiten, der auch dann „schnell“ (nämlich in polynomieller Zeit) das 2-SAT-Problem für jede beliebige Formel in KNF mit jeweils genau[11] zwei Literalen löst.
Eine Idee ist es, die gegebene Formel als Graph zu repräsentieren[12]. Dazu geht man wie folgt vor:
Zuerst definiert man einen Graphen G = (V, E), wobei die Knotenmenge V aus allen in der Formel enthaltenen Literalen besteht. In diesem Beispiel ist also V=$\{a, \overline{a}, b\}$. Die Elemente der Kantenmenge E werden folgendermaßen definiert: Enthält die Formel F die Klausel $x_i \vee x_j$, so enthält E die Kante (x_i, x_j) und ebenso die Kante (x_j, x_i)[13]. Abbildung 4.9 zeigt den resultierenden Graphen.

[11] Kommt in einer Klausel nur ein Literal vor, so kann man eine 0 ergänzen, da $a \vee 0 = a$ ist.
[12] Diese Vorgehensweise wird auch in [al.12] betrachtet.
[13] Die Disjunktion $\vee$ ist kommutativ (vgl. Definition 3.1).

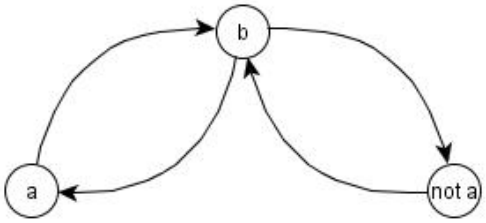


Abbildung 4.9.: Graph, der die Formel $\alpha = (a \vee b) \wedge (\overline{a} \vee b)$ repräsentiert

Laut [Pla79] ist die Formel α erfüllbar (vgl. Definition 3.6) genau dann, wenn es kein Literal x_i gibt mit $\overline{x_i}$ und x_i liegen in derselben starken Zusammenhangskomponente. Basierend auf der Tiefensuche können mit dem Algorithmus aus [Pla79] die starken Zusammenhangskomponenten in einem Graphen bestimmt werden. Hat man die starken Zusammenhangskomponenten gefunden, so muss noch überprüft werden, ob $\overline{x_i}$ und x_i für ein i in derselben Zusammenhangskomponente liegen. Laut den Autoren gelingt es insgesamt mit einer Laufzeit von $\mathcal{O}(n+m)$[14] das angegebene Problem zu lösen. Das Beispiel ist demnach in der Klasse $\mathcal{P}$ enthalten. Und da wir einen allgemeinen Algorithmus mit linearer Laufzeit für das 2-SAT-Problem angeben konnten, ist 2-SAT$\in \mathcal{P}$. Ein formaler Beweis dieser Tatsache kann in [x:K12] nachgelesen werden.

Beispiel 4.6. Denken wir uns hier, wir wollen eine kleine Feier mit 5 Gästen planen. Es ist ein Essen geplant für das zwei Tische zur Verfügung stehen. Leider verträgt sich nicht jeder Gast mit jedem und es soll darauf geachtet werden, dass nur Gäste, die sich mögen, zusammen an einem Tisch sitzen. Unsere Aufgabe ist es also, die Gäste so auf die zwei Tische zu verteilen, dass keine Disharmonie herrscht. Dabei gilt:

1. Paul mag Anne nicht.
2. Anne wiederum möchte nicht an dem gleichen Tisch wie Linda sitzen.
3. Jeder der Gäste mag Thomas.
4. Linda kann Bert nicht leiden.
5. Und da Paul Linda gerne mag, möchte auch er nicht bei Bert sitzen.

Können die fünf Gäste, ohne dass es Streitereien gibt, an die zwei Tische verteilt werden? Modellieren wir die Situation wieder durch einen Graphen G=(V, E): Jeder Gast wird durch einen Knoten dargestellt, dh. wir erhalten die Knotenmenge $V = \{Paul,\ Anne,\ Thomas,\ Linda,\ Bert\}$. Immer dann, wenn ein Gästepaar nicht zusammen an einem Tisch sitzen möchte, repräsentieren wir das durch eine Kante. Die Kantenmenge ergibt sich also als $E = \{(Paul,\ Anne), (Anne,\ Linda), (Paul,\ Bert), (Bert,\ Linda)\}$. Zusammen ergibt sich der Graph, der in Abbildung 4.10 dargestellt ist.

[14] Hier steht n für die Kardinalität von V und m entsprechend für die Anzahl der Kanten.

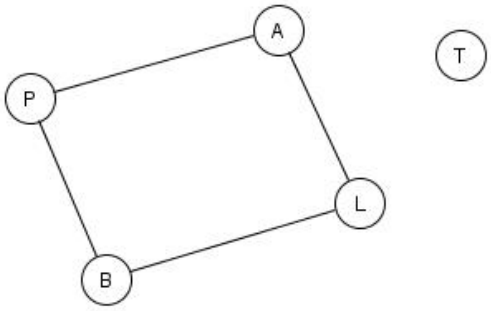


Abbildung 4.10.: Graph zu Beispiel 4.6. Hier wurden die Namen durch die Verwendung des Anfangsbuchstaben abgekürzt.

Hier haben wir es mit einem typischen k-Färbbarkeitsproblem zu tun.

Definition 4.8. Gegeben sei ein ungerichteter Graph G = (V, E). Dieser heißt k-färbbar für $k \in (N)$, wenn es eine Zuordnung von Knoten zu (höchstens) k Farben gibt, so dass keine zwei benachbarten Knoten dieselbe Farbe haben [Sch11].

$$\text{k-Color} = \{f(G) | \text{G ist k-färbbar}\}$$

In unserem Beispiel entsprechen die beiden zur Verfügung stehenden Tische zwei Farben. Wir wollen also ein 2-COLOR-Problem lösen. Intuitiv ist der erste Gedanke zwei verschiedenfarbige Stifte zur Hand zu nehmen und an einem beliebigen Knoten des Graphen mit dem Färben zu beginnen. Wir nehmen hier z.B. die Farben Blau und Rot. Färbt man den ersten Knoten blau, so müssen alle Nachbarn dieses Knotens rot gefärbt werden. Die Nachbarn der roten Knoten wiederum müssen zwingend blau sein. So geht man durch den gesamten Graphen und kann entweder alle Knoten färben oder scheitert eventuell an einer Stelle da keine weitere Farbe mehr zur Verfügung steht. Im Endeffekt ist das die wesentliche Idee zu einem polynomiellen Algorithmus, der das gegebene Problem löst. In [Unb12] kann ein solcher Algorithmus gefunden werden. Die wesentliche Beobachtung ist, dass wir unser Beispiel durch ein Durchlaufen der Adjazenzliste schnell[15] lösen können und somit auch 2-COLOR $\in \mathcal{P}$ ist.

Was aber wird getan, wenn kein polynomieller Algorithmus zur Problemlösung gefunden werden kann? Dieser und vielen weiteren Fragen wird im nächsten Kapitel nachgegangen.

4.4. Die Komplexitätsklasse $\mathcal{NP}$

In diesem Kapitel[16] wird es um Probleme gehen, zu denen kein polynomieller Algorithmus existiert. Abbildung 4.11 zeigt einen Ausschnitt des bekannten

[15] nämlich in $\mathcal{O}(n^2)$

[16] Definitionen und Sätze des Kapitels basieren, wenn nicht anders ausgewiesen, auf [Sch11].

Cartoons aus dem Buch [Joh79]. Was macht ein Informatiker, wenn er für ein Problem keinen polynomiellen Algorithmus findet? Er muss irgendwie sein Dilemma seinem Chef erklären. Wenn man dem Chef beweisen könnte, dass es einfach keinen effizienten Algorithmus gibt, so dürfte er nicht unzufrieden sein. Wie also lässt sich zeigen, dass ein Problem nicht in $\mathcal{P}$ liegt? Mit dieser Frage werden wir uns ab jetzt beschäftigen.

"I can't find an efficient algorithm, but neither can all these famous people."

Abbildung 4.11.: Ausschnitt des Cartoons zur Klasse $\mathcal{NP}$ aus [Joh79]

Nach [Sch11] „[...] müssen wir jetzt den Begriff eines nichtdeterministischen Rechnermodells klären und die Klasse $\mathcal{NP}$ einführen." In Kapitel 4.1 wurde die deterministische Turingmaschine als traditionelles Rechnermodell vorgestellt. Wir bedienen uns nun einer sehr ähnlichen Definition.
Eine nichtdeterministische Turingmaschine wird, wie auch die konventionelle, durch ein 6-Tupel $(Q, \Sigma, \delta, q_0, \Gamma, F)$ beschrieben. Zur Erinnerung liste ich hier noch einmal die einzelnen Tupeleinträge auf:

Q	eine endliche Menge, die Zustandsmenge von M
Σ	das Eingabealphabet
δ	die Zustandsüberführungsfunktion
$q_0 \in Q$	der Anfangszustand
Γ	das Arbeitsalphabet
F	die Menge der akzeptierenden Zustände

Der einzige Unterschied besteht nun darin, dass die Zustandsüberführungsfunktion durch eine Zustandsüberführungsrelation ersetzt wird. Diese wird das Raten der Maschine ermöglichen. Wir definieren die Zustandsüberführungsrelation wie folgt:

$$\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{links, rechts, bleib\}$$

Die Fähigkeit zum Raten wird umgesetzt, indem erlaubt wird, dass der nächste auszuführende Befehl aus einer Menge möglicher Befehle ausgewählt werden

kann. Angenommen, die Turingmaschine befindet sich in dem Zustand q und liest den Zellinhalt γ, dann ist jeder Befehl

$$(q, \gamma) \rightarrow (q', \gamma', Richtung)$$

möglich, solange gilt: (q, γ) und $(q', \gamma', Richtung) \in \delta$.

Genauso wie zuvor definieren wir das Halten der Maschine.

Definition 4.9. Die Turingmaschine akzeptiert eine bestimmte Eingabe ω, wenn sie in einem Zustand aus F hält.

Bleibt noch die Laufzeit der Maschine auf einer bestimmten Eingabe festzulegen. Hier wird die Zeit nur für akzeptierende Eingaben ω gemessen. Gewählt wird die zur kürzesten akzeptierenden Berechnung gehörende Schrittzahl[17]. Wird eine Eingabe ω nicht akzeptiert, so erhalten wir leider keine Information über die Laufzeit.

Definition 4.10. Sei M eine nichtdeterministische Turingmaschine auf dem Eingabealphabet Σ. Die worst-case Laufzeit $\text{Zeit}_M(n)$ von M ist
$\text{Zeit}_M(n) = \max\{\text{Laufzeit von M auf } \omega \mid \text{M akzeptiert } \omega\}$.

Die Definition der Klasse $\mathcal{NP}$ gelingt dann wie folgt:

Definition 4.11. Die Klasse $\mathcal{NP}$ besteht aus allen Sprachen L für die es eine nichtdeterministische Turingmaschine M mit L=M(L) gibt, so dass $Zeit_M(n) = \mathcal{O}\left((n+1)^k\right)$ für eine Konstante k gilt.

Beispiel 4.7. In Kapitel 4.3 habe ich das 2-Färbbarkeitsproblem anhand des Beispiels 4.6 erläutert. Schon eine kleine Änderung des Problems von zwei auf drei Farben sorgt dafür, dass das Problem nicht mehr in polynomieller Zeit lösbar ist! Mit der gleichen Definition wie zuvor betrachten wir jetzt 3-COLOR $= \{G|$ G ist 3-färbbar$\}$.Die Frage lautet: Ist ein gegebener ungerichteter Graph G=(V, E) mit drei Farben färbbar, so dass adjazente Knoten nicht dieselbe Farbe haben? Wir teilen die Gäste zum Beispiel auf drei Tische auf. Hier muss man im Kopf behalten, dass es eigentlich nicht um so kleine Graphen wie in dem Beispiel geht. Für kleine Beispiele ist auch das 3-COLOR-Problem gut lösbar. Aber wenn die Graphen viele Knoten haben nicht mehr. An dieser Stelle möchte ich keinen Beweis zu meiner Behauptung, dass 3-COLOR $\in \mathcal{NP}$ ist, führen. Ich werde diesen später, wenn der Aufbau der Klasse klarer ist, nachreichen. In Abbildung 4.12 ist ein Graph, der mit drei Farben gefärbt wurde, zu sehen.

[17] Ziel ist die Definition einer möglichst großen Klasse $\mathcal{NP}$, „[...] da dann die für $\mathcal{NP}$ schwierigsten Probleme „wirklich“ schwierig sein müssen.“ [Sch11]

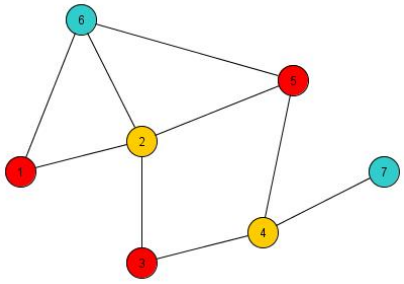


Abbildung 4.12.: Ein mit drei Farben colorierter Graph

4.5. Die polynomielle Reduktion und $\mathcal{NP}$-vollständige Probleme

Nachdem ich in den vorherigen Kapiteln (vgl. Kapitel 4.3 und Kapitel 4.4) die Klasse $\mathcal{P}$ und die Klasse $\mathcal{NP}$ eingeführt habe, möchte ich im Folgenden das Konzept der polynomiellen Reduktion erläutern. In Kapitel 6 wird dann ausgiebig behandelt, wie die polynomielle Reduktion an Beispielen im Schulunterricht motiviert und behandelt werden kann.
In der Beschäftigung mit der Komplexitätstheorie möchte man Probleme hinsichtlich ihrer Komplexität klassifizieren. Man ist also froh, wenn man weiß, ob das betrachtete Problem in einer bestimmten Komplexitätsklasse enthalten ist. Froh ist man aber auch, wenn man weiß, dass es in einer bestimmten Klasse nicht enthalten ist. Zwar sind im Allgemeinen negative Resultate nicht beliebt, wer mag schon schlechte Nachrichten, aber in diesem Fall weiß man dann wenigstens wie man damit umzugehen hat. „Wenn wir aus einem polynomiellen Algorithmus für das eine Problem einen polynomiellen Algorithmus für das andere Problem erhalten und umgekehrt, dann wissen wir, dass beide Probleme zu $\mathcal{P}$ gehören oder nicht.“ [Weg03]
Ziel ist es nun, zu definieren „[...] wann ein Problem A algorithmisch nicht schwieriger zu lösen ist als ein Problem B. Die Probleme A und B sind sich dann algorithmisch ähnlich, wenn A algorithmisch nicht schwieriger als B und B algorithmisch nicht schwieriger als A ist.“ [Weg03] In der Komplexitätstheorie ist der anschauliche Begriff „algorithmisch nicht schwieriger“ leider nicht etabliert. In der folgenden Definition, versuche ich diesen Begriff durch den analogen Begriff der polynomiellen Reduktion zu konkretisieren.

Definition 4.12. Σ_1 und Σ_2 seien zwei Alphabete. Für Sprachen L_1 (über Σ_1) und L_2 (über Σ_2) sagen wir, dass L_1 genau dann auf L_2 polynomiell reduzierbar ist (formal $L_1 \leq_p L_2$), wenn es eine deterministische Turingmaschine M (vgl. Kapitel 4.1) gibt, so dass

$$\omega \in L_1 \Leftrightarrow M(\omega) \in L_2$$

für alle $\omega \in \Sigma_1^*$[18] gilt. M rechnet in polynomieller Zeit und berechnet die „transformierte“ Eingabe $M(\omega)$.
Wir nennen M eine tranformierende Turingmaschine. [Sch11]

In [Sch11] wird angenommen, dass $L_1 \leq_p L_2$ und A ein Algorithmus ist, der in Zeit t(n) entscheidet, ob eine Eingabe ω der Länge n zur Sprache L_2 gehört. Nun wird ein Algorithmus B mit Hilfe des Algorithmus A entworfen, der entscheidet, ob die Eingabe ω zur Sprache L_1 gehört.

Algorithmus B

1. Wende die transformierende Turingmaschine M auf ω an. Wir erhalten die transformierte Eingabe $M(\omega)$.
   /*Anmerkung: $M(\omega)$ kann in polynomieller Zeit in der Länge der Eingabe berechnet werden.
2. Wende Algorithmus A auf die transformierte Eingabe $M(\omega)$ an und akzeptiere ω genau dann, wenn Algorithmus A die transformierte Eingabe $M(\omega)$ akzeptiert.
   /* Da $L_1 \leq_p L_2$ angenommen wurde, folgt $\omega \in L_1 \Leftrightarrow M(\omega) \in L_2$. Algorithmus B ist demnach korrekt.

Die Laufzeit von Algorithmus B ist polynomiell (t(poly(n))). „Wenn L_2 in $\mathcal{P}$ liegt und wenn Algorithmus B einen Algorithmus für L_2 mit polynomieller Laufzeit benutzt, dann ist also auch $L_1 \in \mathcal{P}$“ [Sch11]. Die gleiche Aussage liefert auch der folgende Satz.

Satz 4.1. *Sei $L_1 \leq_p L_2$. Wenn $L_2 \in \mathcal{P}$, dann auch $L_1 \in \mathcal{P}$.*

Satz 4.1 besagt also, dass L_2 mindestens so schwierig ist wie L_1. Ist $L_2 \in \mathcal{P}$, dann ist auch $L_1 \in \mathcal{P}$. Auf der anderen Seite, kann man entsprechend sagen, dass L_2 nicht in $\mathcal{P}$ ist, falls L_1 nicht in $\mathcal{P}$ ist. Aber Vorsicht! Das bedeutet nicht, dass die Sprache L_1 schneller (im Sinnne der Laufzeit) entscheidbar ist als L_2. Nach [Weg03] kann es sein, „dass der beste Algorithmus für L_1 Rechenzeit $\Theta(n^5)$ hat, während es für L_2 Algorithmen mit $\mathcal{O}(n)$ gibt.“ Die Stärke des Konzeptes der polynomiellen Reduktion (manchmal auch Polynomialzeitreduktion genannt) ist es, dass man Probleme unterschiedlichster Art aufeinander reduzieren kann. Dadurch kann man auf gesuchte Komplexitätsklassenzugehörigkeiten durch schon bekannte Zugehörigkeiten schließen. Wir wollen uns nun Gedanken über die schwierigsten Probleme der Klasse $\mathcal{NP}$ machen. Durch unsere Vorüberlegungen aus den vorangegangenen Kapiteln steht fest, wie wir diese definieren müssen.

[18] Die Kleenesche Hülle Σ^* des Alphabets bezeichnet die Menge aller Wörter über dem Alphabet Σ, die durch Symbole aus Σ gebildet werden können. Alphabete stellen somit das Zeicheninventar für Wörter zur Verfügung und bilden damit die Grundlage für formale Sprachen. [uni12]

Definition 4.13. Sei L eine Sprache.

1. L heißt $\mathcal{NP}$-hart[19] genau dann, wenn

$$K \leq_p L \quad \text{für alle Sprachen } K \in \mathcal{NP}$$

2. L heißt $\mathcal{NP}$-vollständig genau dann, wenn $L \in \mathcal{NP}$ und L eine $\mathcal{NP}$-harte Sprache ist. [Sch11]

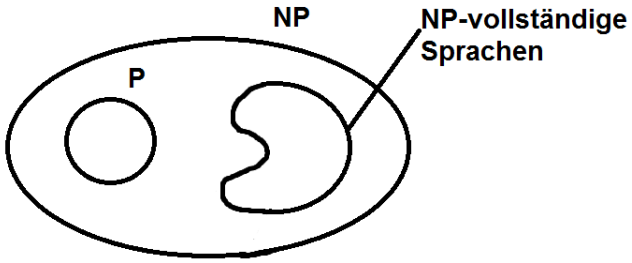


Abbildung 4.13.: Komplexitätsklassen: Größenverhältnisse in der Skizze geben keinen Aufschluss auf die tatsächliche Klassengröße.

Der nachfolgende Satz sagt aus, dass keine $\mathcal{NP}$-vollständige Sprache effizient berechenbar ist, wenn man davon ausgeht, dass $\mathcal{P} \neq \mathcal{NP}$ gilt[20].

Satz 4.2. *Sei die Sprache L $\mathcal{NP}$-vollständig. Gilt $\mathcal{P} \neq \mathcal{NP}$, dann ist $L \notin \mathcal{P}$.*

Beweis
Die Idee des nachfolgenden Beweises ist aus [Sch11] entnommen.
Wir versuchen einen Beweis durch einen Widerspruch zu führen. Nehmen wir also an, es gilt: L $\in \mathcal{P}$. Dann muss gezeigt werden, dass unter dieser Annahme $\mathcal{P} = \mathcal{NP}$ gilt. Wir nehmen des Weiteren an, dass K eine beliebige Sprache aus der Komplexitätsklasse $\mathcal{NP}$ ist. Da es sich nach Voraussetzung bei L um eine $\mathcal{NP}$-vollständige Sprache handelt, muss nach Definition 4.13 $K \leq_p L$ gelten. Aus Satz 4.1 und der Annahme, dass L $\in \mathcal{P}$ ist, folgt K $\in \mathcal{P}$. Die Sprache K war beliebig gewählt. Somit gilt, dass $\mathcal{NP} \subseteq \mathcal{P}$. Wir wissen bereits, dass $\mathcal{P} \subseteq \mathcal{NP}$[21] Zusammen folgt daher $\mathcal{P} = \mathcal{NP}$, was unserer Annahme widerspricht! □

Abschließend ist noch zu sagen, dass die Interpretation von $\leq_p$ als „algorithmisch nicht schwieriger“ intuitiv impliziert, dass $\leq_p$ transitiv sein muss. Wenn

[19] Manchmal wird auch der Begriff $\mathcal{NP}$-schwer benutzt.

[20] Diese Annahme scheint sehr vernüftig zu sein, aber sie ist bis heute nicht bewiesen! (vgl. Kapitel 4)

[21] Eine deterministische Turingmaschine kann als nichtdeterministische TM interpretiert. Die umgekehrte Interpretation gilt aber nicht.

ein Problem A „nicht schwieriger“ ist als ein Problem B, welches wiederum „nicht schwieriger“ ist als ein drittes Problem C, sollte auch Problem A „nicht schwieriger“ sein als Problem C. Im nachfolgenden Lemma beweise ich diesen Sachverhalt. Dadurch ist es möglich eine Kette von Reduktionen aufzubauen.

Lemma 4.3. *Seien L_1, L_2 und L_3 Sprachen. Wenn $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, dann gilt auch $L_1 \leq_p L_3$, dh. $\leq_p$ ist transitiv.*

Beweis
Der nachfolgende Beweis ist dem Skript zur Vorlesung „Algorithmentheorie“ (vgl. [Sch11]) entnommen.
M_1 (bzw. M_2) sei eine transformierende Turingmaschine für die Reduktion $L_1 \leq_p L_2$ (bzw. $L_2 \leq_p L_3$). Dann ist die Turingmaschine M, die zuerst M_1 simuliert und dann M_2 auf der Ausgabe von M_1 simuliert, eine transformierende Turingmaschine für die Reduktion $L_1 \leq_p L_3$, denn aus

$$\forall \quad \omega(\omega \in L_1 \Leftrightarrow M_1(\omega) \in L_2) \text{ und } \forall u \quad (u \in L_2 \Leftrightarrow M_2(u) \in L_3$$

folgt

$$\forall \omega \quad (\omega \in L_1) \Leftrightarrow M_1(\omega) \in L_2 \Leftrightarrow M_2(M_1(\omega)) \in L_3$$

□

Satz 4.3. *Es gelte $L_1 \leq_p L_2$ und sei L_1 $\mathcal{NP}$-vollständig. Dann ist L_2 $\mathcal{NP}$-hart. Wenn zusätzlich noch $L_2 \in \mathcal{NP}$ gilt, ist L_2 $\mathcal{NP}$-vollständig.*

Beweis
Der nachfolgende Beweis ist, wie der Beweis von Lemma 4.3, aus dem Skript zur Vorlesung „Algorithmentheorie“ (vgl. [Sch11]) entnommen.
Sei K eine beliebige Sprache in $\mathcal{NP}$. Dann müssen wir die Reduktion $K \leq_p L_2$ nachweisen. Da L_1 $\mathcal{NP}$-vollständig ist, wissen wir, dass

$$K \leq_p L_1$$

gilt. Aus Lemma 4.3 folgt $K \leq_p L_2$ und $L_1 \leq_p L_2$. □

Beispiel 4.8. Wir werden zwar in Kapitel 6 viele Beispiele der polynomiellen Reduktion sehen, dennoch möchte ich an dieser Stelle ein Beispiel einer formalen polynomiellen Reduktion vorstellen, die uns später nützlich sein wird. Wir betrachten im Folgenden die Probleme

- 3-KNF-SAT=$\{ \alpha \in$ KNF-SAT $\mid \alpha$ besitzt höchstens 3 Literale pro Klausel $\}$ (vgl. Definition 3.9)

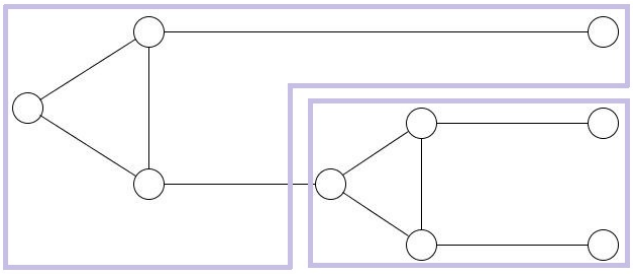


Abbildung 4.14.: Klauselgadget mit Teilgadget

- 3-COLOR = $\{f(G)|$G ist 3-färbbar$\}$ (vgl. Definition 4.8 und Beispiel 4.6)

Wir wollen nun zeigen, dass 3-KNF-SAT polynomiell auf 3-COLOR reduzierbar ist[22]. Die Reduktion gelingt, indem eine Formel α in konjunktiver Normalform mit jeweils genau drei Literalen pro Klausel auf einen Graphen G_α abgebildet wird. Der Graph G_α besteht aus zwei ausgezeichneten Knoten, einem Gadget für jede Variable von α, einem Gadget für jede Klausel, und Kanten die besagte Komponenten wie folgt verbinden:

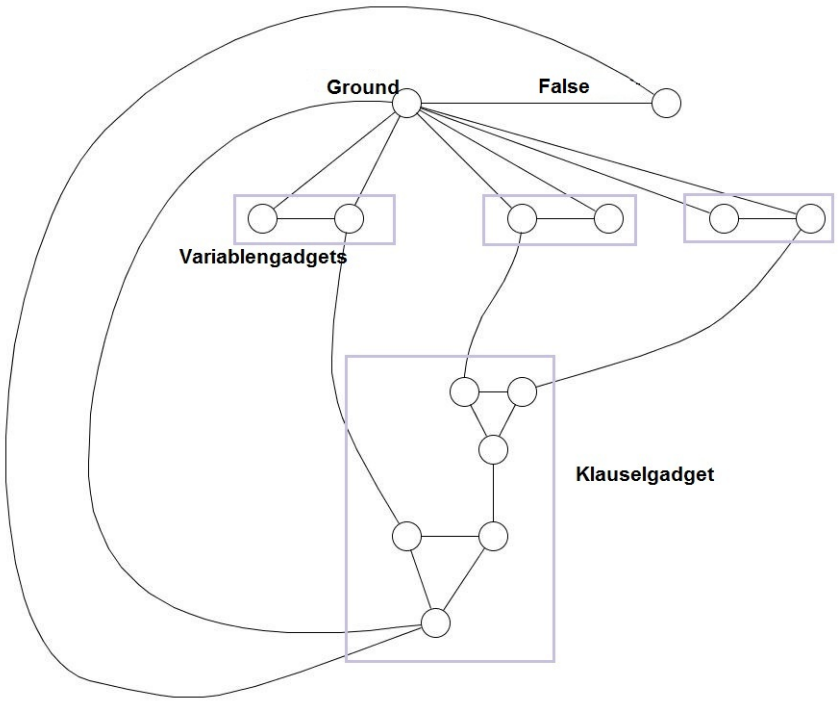


Abbildung 4.15.: Ein Klauselgadget zusammen mit den relevanten Variablengadgets

- Die beiden ausgezeichneten Knoten werden in der Literatur meist „ground“ und „false“ genannt. Diese müssen mit einer Kante verbunden werden,

[22] Die Idee des Beweises stammt aus [Gol10].

so dass sichergestellt ist, dass sie in einer zulässigen 3-Färbung mit zwei verschiedenen Farben gefärbt werden. Der Einfachheit halber nenne ich hier die zugehörigen Farben auch „ground“ und „false“. Natürlich kann man hier beliebige, aber verschiedene Farben einsetzen.

- Die dritte Farbe werde ich „true“ nennen.
- Das zu der Variablen x gehörige Gadget besteht aus den zu den beiden Literalen von x gehörigen Knoten (also aus x und $\overline{x}$). Die Knoten werden wieder durch eine Kante verbunden. Zusätzlich wird jeder der beiden Knoten mit dem Knoten „ground“ verbunden. Dies hat zur Folge, dass bei jeder zulässigen 3-Färbung von G_α eine der Variablen mit „true“ und eine mit „false“ gefärbt werden muss.
- Zusätzlich wird jede Klausel mit einem Gadget assoziiert. In Abbildung 4.14 ist das zu C zugehörige Gadget zu sehen. Es besteht aus einem Masterknoten (M) und drei Terminalknoten (T1, T2, T3). Der Masterknoten ist durch eine Kante mit dem Knoten „ground“ sowie mit einer Kante mit „false“ verbunden. Dies erzwingt in einer zulässigen Färbung, dass der Masterknoten mit „true“ gefärbt wird. Das Gadget ermöglicht es, dass die Terminale mit jeder beliebigen Kombination aus „true“ und „false“ der Farbgebung belegt wird. Ausgeschlossen ist nur, dass alle Terminale auf „false“ gesetzt werden. Das heißt, in jeder beliebigen zulässigen 3-Färbung von G_α, ist, wenn kein Terminal eines Klauselgadget mit „ground“ gefärbt ist, mindestens ein Terminal mit „true“ gefärbt.

Die Klauselgadgets[23] sind mit den Variablengadgets verbunden. Natürlich können verschiedene Klauseln die gleichen Terminale benutzen. Dieser Sachverhalt wird auch in Abbildung 4.15 dargestellt. Damit die Konstruktion etwas anschaulicher wird, ist in Abbildung 4.16 der zur Formel $\alpha = (u \vee \overline{v} \vee w) \wedge (v \vee x \vee \overline{y})$ konstruierte Graph[24] abgebildet. Hier steht „T“ für „true“, „F“ für „false“ und „G“ für „ground“. Soweit ist nun die Konstruktion der Reduktion gelungen. Es bleibt ihre Korrektheit zu zeigen.

1. **α ist erfüllbar $\Rightarrow$ G_α ist 3-färbbar:**
   - Ist die Variable $v_i = 1$, dann färbe Knoten v_i mit „true“ und $\overline{v_i}$ mit „false“
   - Für jede Klausel C=(a, b, c) ist mindestens ein Literal „true“, das Klauselgadget (Oder-Gatter) kann mit drei Farben so gefärbt werden, dass die Ausgabe „true“ lautet.
2. **G_α ist 3-färbbar $\Rightarrow$ α ist erfüllbar:**
   - Wenn Variable v_i mit „true“ gefärbt ist, dann setze die Variable auf „true“

[23] Durch die Konstruktion arbeiten die Klauselgadgets wie ODER-Gatter.
[24] Dieses Beispiel habe ich aus [3SA12] entnommen.

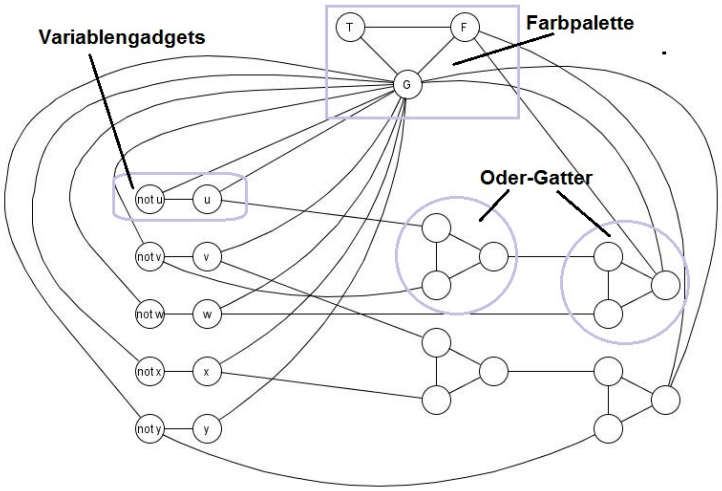


Abbildung 4.16.: Zu $\alpha = (u \vee \overline{v} \vee w) \wedge (v \vee x \vee \overline{y})$ korrespondierender Graph

- Betrachte irgendeine Klausel C=(a, b, c). Es ist nicht möglich, dass a, b und c gleichzeitig „false“ ist, da sonst auch die Ausgabe des Oder-Gatters „false“ sein müsste. Dieser Ausgang ist aber durch eine Kante sowohl mit „false“ als auch mit „ground“ verbunden.

Wir haben damit die Korrektheit der oben präsentierten polynomiellen Reduktion gezeigt und gesehen, dass 3-SAT polynomiell auf 3-COLOR reduzierbar ist.

4.6. SAT ist $\mathcal{NP}$- vollständig - Der Satz von Cook

Im vorherigen Kapitel wurde die Technik der polynomiellen Reduktion vorgestellt. Lemma 4.3 besagt, dass man die $\mathcal{NP}$-Vollständigkeit eines Problems nachweisen kann, indem man zuerst zeigt, dass das Problem in der Klasse $\mathcal{NP}$ ist und man im zweiten Schritt das Problem auf ein schon bekanntes $\mathcal{NP}$-vollständiges Problem polynomiell reduziert. Sei B ein Problem, von dem wir zeigen wollen, dass es $\mathcal{NP}$-vollständig ist. Nach [Weg96] ergibt sich das Grundkonzept eines $\mathcal{NP}$-Vollständigkeitsbeweises aus vier Schritten:

1. Zeige, dass B $\in \mathcal{NP}$, indem eine Lösung nichtdeterministisch geraten wird und dann deterministisch verifiziert wird[25].
2. Nun muss ein „geeignetes“ als $\mathcal{NP}$-vollständig bekanntes Problem A ausgesucht werden, so dass die polynomielle Reduktion $A \leq_p B$ möglichst einfach ist.
3. Angabe einer deterministisch in polynomieller Zeit berechenbaren Transformation f, die Eingaben für A in Eingaben für B überführt.

[25] Was ist die Gemeinsamkeit aller Probleme in $\mathcal{NP}$? Geratene Lösungen sind leicht zu verifizieren.

4. Als letzter Schritt muss die Eigenschaft $x \in A \Leftrightarrow f(x) \in B$ nachgewiesen werden.

Warum reicht dieses Vorgehen aus? Im ersten Schritt wird gezeigt, dass $B \in \mathcal{NP}$ ist. Dieser erste Schritt wird in der Literatur auch oft in eine Rate- und eine Verifikationsphase[26] aufgeteilt (vgl. [Weg96]). Auch häufig in der Literatur zu finden ist der sogenannte „Orakelansatz“. Dabei wird davon ausgegangen, dass ein Orakel nach einem Lösungskandidaten befragt wird.

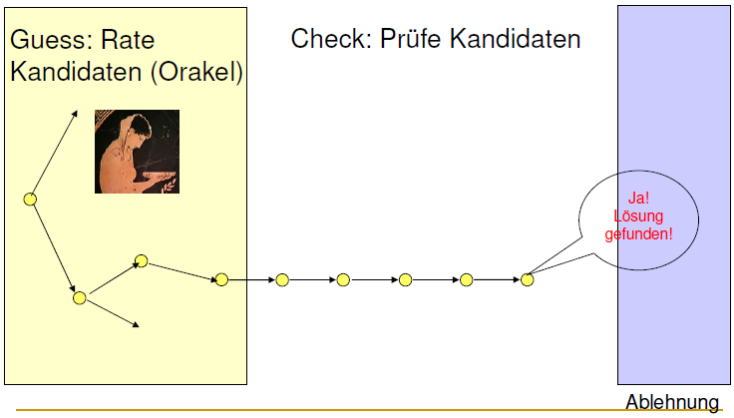


Abbildung 4.17.: Die Guess and Check-Methode (Grafik entnommen aus [Pol12])

Nachdem wir wissen, dass das Problem in $\mathcal{NP}$ enthalten ist, wird gezeigt, dass man A auf B polynomiell reduzierbar ist. Da A selbst $\mathcal{NP}$-vollständig ist, gilt nach Definition 4.13, dass alle Sprachen L $\in \mathcal{NP}$ polynomiell auf A reduziert werden können. In Lemma 4.3 haben wir gezeigt, dass die polynomielle Reduktion transitiv ist. Es folgt unmittelbar, dass $L \leq_p B$ für alle Sprachen $L \in \mathcal{NP}$, und B somit $\mathcal{NP}$-vollständig ist.

Beispiel 4.9. Sei G = (V, E) ein ungerichteter Graph. Eine Teilmenge T$\in V$ heißt eine Überdeckung, wenn alle Kanten einen Endpunkt in T besitzen. Das Vertex Cover Problem (Knotenüberdeckungsproblem) lautet wie folgt:

VC= { (G, k)| G besitzt eine Menge T von k Knoten,
so dass jede Kante einen Endpunkt in der Menge T besitzt.}

Wir wollen zeigen, dass dieses Problem in $\mathcal{NP}$ ist. Dafür „raten“ wir eine Lösung von k Knoten für eine Eingabe (G, k) und verifizieren, dass alle Kanten mindestens einen geratenen Knoten als Endpunkt haben. Um das zu tun reicht es aus, jede Kante einmal „anzufassen“ und zu überprüfen, ob ein Endpunkt in T enthalten ist. Die Anzahl der Kanten des Graphen kann höchstens $|V| = n^2$ sein, nämlich dann, wenn es sich um einen vollständigen[27] Graphen handelt.

[26] manchmal auch: Guess and Check-Methode genannt
[27] Jeder Knoten ist mit jedem verbunden.

Das bedeutet man muss höchstens n^2 Kanten „anfassen“ und dies gelingt in polynomieller Zeit. VC ist also in der Komplexitätsklasse $\mathcal{NP}$ für eine beliebige Eingabe (G, k) enthalten.
Das Knotenüberdeckungsproblem ist übrigens, was ich an dieser Stelle nicht beweisen möchte, auch $\mathcal{NP}$-vollständig.

Bisher habe ich zwar von dem ein oder anderen Problem behauptet, dass es in $\mathcal{NP}$ liegt, aber bei keinem wurde ein Beweis geführt. Das heißt „offiziell“ wissen wir bis jetzt noch von keinem Problem, dass es $\mathcal{NP}$-vollständig ist. Um die Lawine durch die polynomielle Reduktion loszutreten, benötigen wir aber ein $\mathcal{NP}$-vollständiges Problem. Ziel nun ist es „endlich“ ein $\mathcal{NP}$-vollständiges Problem zu finden. Hier kommt der berühmte Satz von Cook ins Spiel. Nach [Weg96] ist der Satz von Cook „eines der wichtigsten Ergebnisse der Informatik“. Er sagt aus, dass das Satisfiability Problem (Erfüllbarkeitsproblem) $\mathcal{NP}$-vollständig ist. Bei dem Satisfiability Problem handelt es sich um das Problem, das danach fragt, ob eine Formel erfüllbar ist (vgl. auch Bsp. 4.5).

$$\text{SAT}=\{\alpha|\ \alpha \text{ ist eine erfüllbare Formel in konjunktiver Normalform.}\}$$

Satz 4.4. *SAT[28] ist $\mathcal{NP}$-vollständig.*

Der vollständige Beweis kann unter [Sch11] nachgelesen werden. Hier möchte ich nur die Idee dahinter skizzieren, da der Beweis recht langatmig und sehr technisch ist.
Beweisidee:
Zuerst zeige ich, dass SAT in $\mathcal{NP}$ liegt. Dazu raten wir eine 0-1-Belegung B=$(x_1, \cdots, x_n)$ der Länge n. Damit eine Klausel erfüllt ist, muss für jede Klausel in α mindestens ein Literal den Wert 1 liefern. Die Eingabe für SAT wird akzeptiert, wenn alle Klauseln erfüllt sind. Das zu kontrollieren klappt offensichtlich in polynomieller Zeit, indem man die einzelnen Variablen einsetzt und den Wert von α berechnet.
Weiter muss für eine beliebige Sprache L $\in \mathcal{NP}$ gezeigt werden, dass $L \neq_p SAT$. Was wissen wir über eine beliebige Sprache L? Wir wissen, dass es eine nichtdeterministische Turingmaschine M_L=$(Q, \Sigma, \delta, q_0, \Gamma, F)$ und ein Polynom p gibt mit

- $L = L(M_L)$ und
- M_L hat die Laufzeit p. Es gibt also für jedes Wort $\omega \in L$ eine akzeptierende Berechnung der Länge höchstens $p(|\omega|)$.

[28] In [Sch11], aus dem große Teile der Beweisidee entnommen wurden, bezieht sich der Satz von Cook auf KNF-SAT, also auf Formeln in konjunktiver Normalform. Dies ist aber eigentlich egal, da wir nach Lemma 3.7 wissen, dass jede Formel in eine äquivalente Formel in konjunktiver Normalform transfomiert werden kann.

Nach Umbenennung von Zuständen und Buchstaben des Bandalphabets Γ kann man annehmen, dass

- M_L die Zustandsmenge $\{0, 1, \cdots, q\}$ besitzt
- sowie das Bandalphabet $\Gamma = \{0, 1, \mathtt{B}\}$
- 0 ist Anfangszustand und 1 ist der einzige akzeptierende Zustand[29]. Weiterhin wird M_L stets halten, wenn Zustand 1 erreicht wird.

Was muss nun gezeigt werden? Es muss zu jeder Eingabe ω eine Formel α_ω in konjunktiver Normalform konstruiert werden, so dass

$$\begin{aligned} \omega \in L \Leftrightarrow & \quad \text{Es gibt eine Berechnung von } M_L, \text{ die } \omega \text{ akzeptiert} \\ \Leftrightarrow & \quad \alpha_\omega \in SAT \end{aligned}$$

Dabei muss die Funktion $\omega \longmapsto \alpha_\omega$ in polynomieller Zeit von einer transformierenden Turingmaschine M entwickelt werden können.
Im weiteren Beweis des Satzes, wird eine Turingmaschine durch das Benutzen der Aussagenlogik „konstruiert“. Dabei wird ausgenutzt, dass das Raten einer akzeptierenden Berechnung als Raten einer erfüllenden Belegung interpretiert werden kann. „Um dies erreichen zu können, muss sich die Aussagenlogik als genügend ausdrucksstark herausstellen, damit wir in der Aussagenlogik über Turingmaschinen „sprechen“ können. Anders ausgedrückt, wir müssen die Aussagenlogik als eine Programmiersprache verstehen und in dieser Programmiersprache ein zu M_L äquivalentes Programm schreiben.“ [Sch11]
Der vollständige Beweis betrachtet jede Konfiguration und Aktion einer Turingmaschine und „übersetzt“ dies in ein Programm der Aussagenlogik. Durch die Variablen

$$Zustand_t(i),\ 0 \leq t \leq T \leq, i \in \{0, \cdots, q\}$$

wird beispielsweise ausgedrückt, dass M_L zum Zeitpunkt den Zustand i besitzt.

$$Kopf_t(wo),\ 0 \leq t \leq T \leq, -T \leq wo \leq T$$

soll dann den Wert 1 haben, wenn sich der Lese-Schreib-Kopf der Turingmaschine M_L zum Zeitpunkt t auf der Zelle i befindet. Die Variablen

$$Zelle_t(wo, was),\ 0 \leq t \leq T \leq, -T \leq wo \leq T, was \in \Gamma$$

geben an, was zu einem bestimmten Zeitpunkt in welcher Zelle (wo) auf dem Band steht.
Wie zuvor gesagt, möchte ich hier nur einen kleinen Einblick in den Beweis geben. Damit der Leser aber eine Vorstellung dazu aufbauen kann, wie es möglich ist mit Hilfe der Aussagenlogik zu „programmieren“ möchte ich vorstellen, wie zum Beispiel eine Startkonfiguration in Klauseln beschrieben werden kann. Wir wollen ausdrücken, dass sich die Turingmaschine M_L

[29] Gibt es mehrere akzeptierende Zustände, dann kann man alle durch die Überführungsfunktion auf den Zustand 1 bringen.

- im Zustand 0 befindet,
- ihr Lese-Schreib-Kopf die Zelle 1 liest,
- die Zellen $1, \cdots, |\omega|$ die Eingabe ω speichern
- und in den restlichen Zellen das Blanksymbol B steht.

Mit Hilfe der zuvor definierten Variablen gelingt die „Übersetzung“ fast allein durch das Nutzen der Konjunktion:

$$\alpha_0 \equiv Zustand_0(0) \ \wedge \ Kopf_0(1) \ \wedge \ \bigwedge_{wo=1}^{|\omega|} Zelle_0(wo, \omega_{wo}) \wedge \bigwedge_{wo \in \{-T, \cdots, T\}(\setminus\{1, \cdots, |\omega|\}} Zelle_0(wo, \mathtt{B}) \quad (4.1)$$

Entsprechend kann ein Update-Schritt der Turingmaschine dargestellt werden. Allerdings muss man dafür etwas mehr Arbeit investieren. Interessierten Lesern empfehle ich eine Blick in [Weg03] oder [Sch11], um den vollständigen Beweis nachzulesen.

Was haben wir dadurch erreicht? Wir haben festgestellt, dass man mit der Aussagenlogik über nichtdeterministische Berechnungen „reden“ kann. Durch den Satz von Cook haben wir unser erstes, auch durch einen Beweis untermauertes, $\mathcal{NP}$-vollständiges Problem gefunden. Nun können wir die Lawine mit der polynomiellen Reduktion lostreten, um weitere $\mathcal{NP}$-vollständige Probleme zu finden (vgl. Beginn Kapitel 4.6: Grundkonzept eines $\mathcal{NP}$-Vollständigkeitsbeweises). An dieser Stelle sollte auch klar geworden sein, warum SAT ein so schwieriges Problem ist. Es ist gerade deswegen so schwierig, weil Aussagen über nichtdeterministische Berechnungen gemacht werden können.

Den berühmten Satz von Cook[30] kann man auch anders ausdrücken: Wenn man zeigen kann, dass SAT $\in \mathcal{P}$ ist, dann bedeutet das, dass $\mathcal{NP} = \mathcal{P}$ gilt. Sollte jemandem das gelingen, so kann er sich eine Million Dollar verdienen.[31]

Nun, da wir gezeigt haben, dass das Erfüllbarkeitsproblem $\mathcal{NP}$-vollständig ist, können wir aus den Erkenntnissen aus Beispiel 4.8 schlussfolgern, dass auch 3-COLOR $\mathcal{NP}$-vollständig ist.

[30] „Einen vergleichbaren Satz veröffentlichte Leonid Levin unabhängig von Cook im Jahre 1973, deswegen spricht man manchmal auch vom Satz von Cook und Levin.“ [Wik12j]

[31] Wie in Kapitel 4 erläutert gehört das $\mathcal{P}$-$\mathcal{NP}$-Problem zu den Milleniumproblemen.

Teil II.

Das praktisch Unmögliche im Informatikunterricht

5. Didaktische Betrachtung

Die vorliegende Arbeit ist durch und durch der theoretischen Informatik gewidmet. Nach [Hro01] gehört die theoretische Informatik statistisch gesehen nicht gerade zu den Lieblingsfächern der Studierenden. Diese Aussage dürfte zu einem großen Teil auch auf Schüler und Schülerinnen der Informatikkurse in der Schule zutreffen. Die Frage ist demnach, warum das so ist. Sicherlich ist ein Grund dafür, dass „[...] die theoretische Informatik von allen Informatikbereichen am stärksten mathematisch geprägt ist, und so einen höheren Schwierigkeitsgrad besitzt“ [Hro01]. Ich möchte im Folgenden einige gute Gründe erläutern, warum es dennoch sehr lohnend ist, sich gerade mit Themen der theoretischen Informatik, hier mit der Komplexitätstheorie im Speziellen, im Informatikunterricht zu befassen.

Die theoretische Informatik ist praxisrelevant und besitzt eine allgemeinbildende Wirkung. „Die Meinung: Mit genügend schnellen und genügend vielen Computern können wir alle unsere Probleme bewältigen, ist ein Irrglaube. Ihm nicht zu verfallen ist ein Moment kritischen Vernunftgebrauchs und damit von Bildung“ [Bau96]. Wissen wir, dass es sich bei unserem Problem um ein $\mathcal{NP}$-vollständiges Problem handelt, so müssen wir die Suche nach einer optimalen Lösung aufgeben, da wir sonst unsere Zeit verschwenden. Schließlich versuchen wir auch nicht immernoch mit Lineal und Zirkel zu einem gegebenen Kreis ein Quadrat gleicher Fläche zu konstruieren. Auch Immo O. Kerner ist der Meinung, dass die Komplexitätstheorie zur Allgemeinbildung gehört: „ Einige Grundaussagen daraus [aus der Komplexitätstheorie] gehören eigentlich zur Allgemeinbildung eines modernen Menschen [...]. Diese Aussagen stehen dem noch immer weit verbreiteten Glauben entgegen, dass mit genügend vielen, genügend schnellen und genügend großen Computern alle Probleme [...] gelöst werden können“ [Ker91]. Ergebnisse aus der Komplexitätstheorie haben eine direkte Auswirkung auf die Entwicklung und den Einsatz von Algorithmen. Wie ist mit Problemen, zu denen es keine Lösung in polynomieller Zeit geben wird, umzugehen? Approximationsalgorithmen, wie sie in Kapitel 8 vorgestellt werden, liefern gute, wenn auch nicht optimale Lösungen. Wie in Kapitel 8 geschildert, kann man sogar den gemachten Fehler abschätzen und erhält so eine Möglichkeit für die Praxis äußerst „passable“ Lösungen zu finden.

Wie in Kapitel 4 geschildert stellt die $\mathcal{P} = \mathcal{NP}$-Frage eine der bedeutensten Fragen der Informatik dar. Es geht um nichts geringeres, als zu beweisen, ob Computer schon heute fähig sind zu raten. Nach [Gol10] wurde die $\mathcal{P} = \mathcal{NP}$-Frage erstmals 1971 von Stephen Cook und Leonid Levin gestellt. Bis heute

hat es niemand, trotz des Ansporns eines sehr hohen Preisgeldes, geschafft, die Aussage zu widerlegen oder zu beweisen. Viele berühmte Mathematiker und Informatiker haben sich darum bemüht, dennoch wird noch viel Zeit vergehen, bis das $\mathcal{P} = \mathcal{NP}$-Problem gelöst wird, sollte dies überhaupt gelingen. Einige Experten sind der Meinung, dass es mit den heute zur Verfügung stehenden Beweismethoden einfach noch nicht gelöst werden kann. Diese Beweistechniken müssen sich wohl zuerst noch weiterentwickeln. Yao[1] hat die momentane Ausgangsposition bei der $\mathcal{P} = \mathcal{NP}$-Frage „[...] mit der Situation derjenigen verglichen, die vor 200 Jahren davon träumten, auf den Mond zu gelangen“ [Weg03].

> „Uns ist kein wissenschaftliches Problem bekannt, das „allumfassender“ und dennoch einfacher zu formulieren wäre als die berühmte $\mathcal{P} = \mathcal{NP}$-Frage. [...] Das Besondere hierbei ist, dass zum einen schon Tausende von Forschern aus völlig verschiedenen Gebieten [...] sich mit dieser Fragestellung beschäftigt haben, andererseits die zugrundeliegende Thematik sehr einfach und intuitiv vermittelbar ist.“ [al.07]

Der berühmte Satz von Cook, einer der Hauptkatalysatoren der Komplexitätstheorie, stellt einen weiteren Meilenstein in der Entwicklung der Informatik, wie wir sie heute kennen, dar. Erst er ermöglichte die Klassifizierung von mittlerweile mehreren tausend $\mathcal{NP}$-vollständigen Problemen. Mit dem einfachen Modell der Turingmaschine und dem Konzept der polynomiellen Reduktion lässt er sich durchaus auch auf Schulniveau verdeutlichen. So ist es möglich Schülern und Schülerinnen einen Einblick in den aktuellen Forschungsstand der theoretischen Infomatik zu geben, trotzdem, oder gerade weil es sich um Wissen handelt, das schon etwa 1970 bearbeitet wurde.

Wie zuvor erläutert, ist das Thema der polynomiellen Reduktion sehr umfassend. Man beschäftigt sich unter anderem mit Grundlagen der Graphentheorie. Graphen sind eine äußerst ausdrucksstarke und anschauliche Struktur um beispielsweise Beziehungsgeflechte darzustellen.

> „Graphen empfehlen sich als Paradigmen für den Informatikunterricht. Denn aus Sicht der Anwendungen sind Graphen weitaus wichtiger (und auch motivierender) als Baumstrukturen - und zwar aufgrund der einfachen Tatsache, dass sich jede Sammlung irgendwelcher Objekte, seien es Personen, Städte, Spielpositionen oder Begriffe - und die Beziehungen zwischen ihnen - als Graphen modellieren lassen.“ [Bau94]

S. Schubert und A. Schwill bezeichnen Graphen als „das zentrale Hilfsmittel zur Bildung ikonischer Modelle“ [SS04]. Selbstverständlich könnte man das Thema Graphen auch an anderen leichter zugänglichen Themen motivieren.

[1] Der Gewinner des Turingawards im Jahr 2000.

Aber die Komplexitätstheorie hat wesentlich mehr zu bieten, auch da sie nach [Sch94] zu den fundamentalen Ideen der Informatik im Zusammenhang mit der Bewertung von Algorithmen gehört. Sehr interessant ist auch der Einblick in das Konzept der Turingmaschine, die als das Rechnernmodell schlechthin gilt. Die Turingmaschine gehört zu den grundlegenden Konzepten der theoretischen Informatik und ist auch im Lehrplan enthalten. Aber dazu später mehr. Die Wichtigkeit der Turingmaschine und ihren Einfluss auf grundlegende Erkenntnisse der Informatik sollte durch Kapitel 4.1 deutlich geworden sein.

> „Insgesamt bildet die Komplexitätstheorie eine intellektuelle Herausforderung, die sich von den Anforderungen anderer Gebiete der Informatik unterscheidet. Sie ordnet sich der Wissenschaftslandschaft in die Reihe der Disziplinen ein, in denen die Grenzen des mit den vorhandenen Ressourcen Machbaren ausgelotet werden.“ [Weg03]

Natürlich ist jedes Thema der Grundlagen für sich sehr interessant und bereichernd. Alle einzeln durchzugehen scheint mir nicht notwendig zu sein. Ich denke, dass die Bedeutung von Themen innerhalb der Komplexitätstheorie mittlerweile deutlich wurde. Aber sind Themen der Komplexitätstheorie nur interessant für Experten der Informatik oder solche, die noch Experte werden wollen? Nach Hubwieser soll Gelerntes einen „[...] möglichst breiten Anwendungsbereich haben. Diese Breite lässt sich in Bezug auf Informatiksysteme in vier Klassen[2] unterteilen“ [Hub01]. Nach Hubwieser werden hier explizit (vgl. Tabelle 5.1) sowohl die Komplexitätstheorie als auch die Grenzen der Berechenbarkeit, die zentrales Thema dieser Arbeit sind, der Klasse 2 zugeordnet. Er führt als weiteres Kriterium zur didaktischen Auswahl von Lerninhalten die Lebensdauer der Thematik an. Die Geschichte der Komplexitätstheorie reicht weit zurück. Grundlagen, wie zum Beispiel das Modell der Turingmaschine (1936), wurden schon ab 1930 entwickelt. „Als erste informelle komplexitätstheoretische Untersuchungen werden Ergebnisse von John Myhill (1960), Raymond Smullyan (1961) und Hisao Yamada (1962) angesehen, die sich mit speziellen raum- und zeitbeschränkten Problemklassen beschäftigt haben, jedoch in ihren Arbeiten noch keinen Ansatz zu einer allgemeinen Theorie entwickelten“ [Wik12c]. Die Entwicklung ist auch heute noch längst nicht abgeschlossen. In den vergangenen zwei Jahrzehnten sind zahlreiche neue Forschungsfelder hinzugekommen: Approximationsalgorithmen, Onlinealgorithmen, randomisierte Algorithmen, Algorithmische Spieltheorie, Model-Checking, Quantum Computing und viele mehr. [Tho12]

Die vorgehende Diskussion über die Thematik der Komplexitätstheorie mündet in die Analyse von Wolfgang Klafki. Ziel seiner fünf Grundfragen, die sich auf die Gegenwartsbedeutung, Zukunftsbedeutung, Struktur des Inhalts, exemplarische Bedeutung und die Zugänglichkeit des Unterrichtsinhaltes beziehen, ist

[2] Die Klassifizierung von Lerninhalten nach Allgemeingültigkeit ist in Tabelle 5.1 dargestellt.

Klasse	Beschreibung	Beispiele
1	Anwendung auch außerhalb des Bereiches der EDV möglich	Modellierungstechniken, Problemlösestrategien
2	charakteristisch für alle elektronischen Informatiksysteme	Komplexitätsklassen, Grenzen der Berechenbarkeit
3	Anwendung beschränkt sich auf eine Klasse von Informatiksystemen	Programmierparadigmen, spezielle Datenstrukturen, Netzwerktopologien, Sortieralgorithmen
4	betrifft nur ein konkretes System	Syntax einer Programmiersprache, Menüstruktur eines Anwendersystems, Architektur eines Mikroprozessors

Tabelle 5.1.: Klassifizierung von Lerninhalten nach Allgemeingültig [Hub01]

die Frage, ob sich besagter Inhalt überhaupt für die Schüler lohnt. [Hub04]
Die Gegenwartsbedeutung und die Zukunftsbedeutung wurden, denke ich, durch die gesamte Arbeit hindurch und vorallem im oberen Abschnitt deutlich. Bei der Struktur des Inhaltes gilt es folgende Fragestellungen zu beachten: Welches sind die einzelnen Momente des Inhalts? Der Kern des Themas liegt in dem Satz von Cook und der polynomiellen Reduktion vor. Erst der Satz von Cook ermöglicht es, dass ein bereits als $\mathcal{NP}$-vollständiges Problem vorliegt, so dass andere Problemstellungen darauf reduziert werden können. Das Konzept der polynomiellen Reduktion, gedacht als „Transformationsmaschine“, lässt Ähnlichkeiten von auf den ersten Blick völlig verschiedenen Problemen entdeckbar werden. Eine wichtige Grundstruktur des Inhaltes liegt darin vor, dass man versucht effizient lösbare Probleme von den praktisch nicht effizient lösbaren zu trennen. Die Turingmaschine als zugrundegelegtes Modell, um die „Transformation“ (eigentlich Reduktion) umzusetzen, bildet eine weitere wichtige Säule. Der große sachliche Zusammenhang in dem Gedanken, Probleme und zugehörige Algorithmen zur Lösung zu klassifizieren. In einem Zeitalter, wo der Gedanke, dass ein Computer jedes berechenbare Problem bewältigen kann, in der breiten Masse vorherrschend ist, werden deutliche Grenzen aufgezeigt.
Die exemplarische Bedeutung, die man mit Themen der Komplexitätttheorie, und der polynomiellen Reduktion im Speziellem, behandeln kann, sind folgende:

- Die Turingmaschine, die als „Nähmaschine“ heutige Rechner mit einem

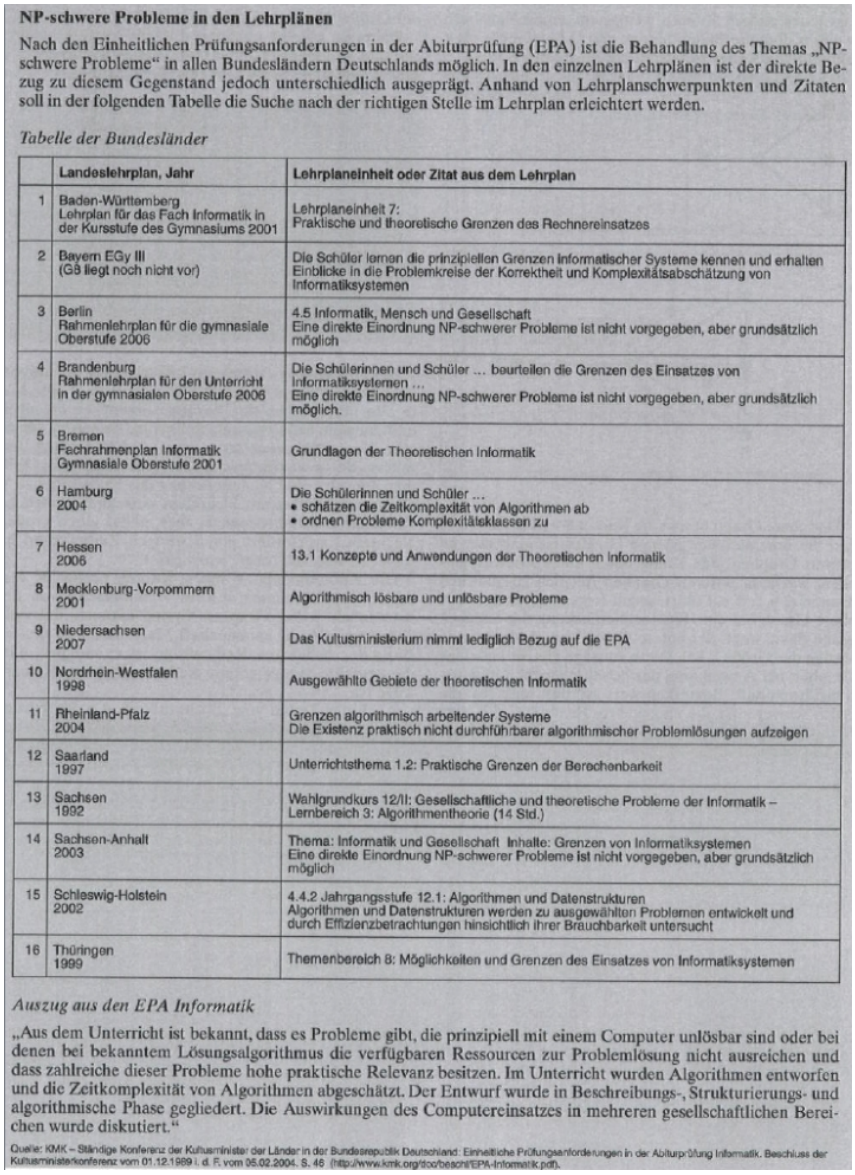
sehr einfachen Aufbau modelliert.

- Eine ganze Theorie wird darauf aufgebaut, dass $\mathcal{P} \neq \mathcal{NP}$ gilt. Wie wahrscheinlich ist die Korrektheit dieser Annahme? Was passiert, wenn diese Annahme doch nicht gültig ist? Auswirkungen von hypothetischen Annahmen auf Konzepte allgemein.
- Grenzen der Berechenbarkeit werden durch die Begriffe $\mathcal{NP}$-hart und $\mathcal{NP}$-vollständig verbalisiert und vorallem überhaupt erfahrbar.

Zur Zugänglichkeit des Themas ist zu sagen, dass durch die Fülle an Grundlagen, die teilweise notwendig sind, etwas erschwerte Bedingungen herrschen. Aber, wenn man die Grundlagen nicht zu formal vermittelt und sich dabei auf das Wesentliche beschränkt, so eröffnet sich den Schülern und Schülerinnen ein sehr spannendes Thema. Die Komplexitätstheorie hat sehr starke aktuelle Bezüge und wirkt sich unmittelbar auf Algorithmen zur Problemlösung aus. Die Konsequenzen, zum Beispiel approximative Lösung etc., dürfte dadurch auch diejenigen Schüler und Schülerinnen interessieren, die sich mehr zu der Praxis (meistens zum Programmieren) hingezogen fühlen.

Nachdem gezeigt wurde, dass es sich um ein durchaus sehr lohnendes Thema für Schüler und Schülerinnen handelt, wird im Folgenden ein Blick in den Lehrplan des hessischen gymnasialen Bildungsgangs des Kultusministeriums (vgl. [Kul12]) gewagt. Die Komplexität des Themas und die vorausgesetzten Grundlagen sind selbstverständlich nicht für Schüler und Schülerinnen der Sekundarstufe I geeignet. In der Sekundarstufe II jedoch finden sich gleich mehrere Bezüge, die das Thema für den Schulunterricht legitimieren.
Unter den Lern- und Prüfungsbereichen, die „unbeschadet der didaktischen und methodischen Freiheit bei der Schwerpunktsetzung und Intensität der Behandlung einzelner Teilgebiete“ [Kul12] zur Abiturprüfung zur Verfügung stehen sollen, wird unter den anwendungsbezogenen Qualifikationen die Behandlung von Grenzen und Möglichkeiten, Chancen und Risiken des Einsatzes der Informations- und Kommunikationstechniken angesprochen. Die $\mathcal{NP}$-Vollständigkeitstheorie weist sicherlich Grenzen bei der Problemlösung auf. Chancen bestehen darin, trotz der $\mathcal{NP}$-Vollständigkeit zahlreicher Probleme, diese durch Abschwächung der Problemstellung oder durch fallen lassen der Optimalitätsforderung[3] dennoch adäquat, und für die Praxis ausreichend, zu lösen.
Zu Beginn der Qualifikationsphase (Q1 bis Q4) wird in Q1 die objektorientierte Modellierung gelehrt. Die in der Einführungsphase (E1 und E2) begonnene Einführung in die Grundlagen der Programmierung wird vertieft und zum ersten Mal wird auf die Komplexität von Algorithmen hingewiesen. Es soll hier, zumindest im Leistungskurs verbindlich, polynomielle und exponentielle Zeitkomplexität betrachtet werden. Für den Grundkurs Informatik ist dieser Lerninhalt fakultativ. Zu dem können Graphen als fakultative Unterrichtsin-

[3] Approximationsalgorithmen



NP-schwere Probleme in den Lehrplänen

Nach den Einheitlichen Prüfungsanforderungen in der Abiturprüfung (EPA) ist die Behandlung des Themas „NP-schwere Probleme" in allen Bundesländern Deutschlands möglich. In den einzelnen Lehrplänen ist der direkte Bezug zu diesem Gegenstand jedoch unterschiedlich ausgeprägt. Anhand von Lehrplanschwerpunkten und Zitaten soll in der folgenden Tabelle die Suche nach der richtigen Stelle im Lehrplan erleichtert werden.

Tabelle der Bundesländer

	Landeslehrplan, Jahr	Lehrplaneinheit oder Zitat aus dem Lehrplan
1	Baden-Württemberg Lehrplan für das Fach Informatik in der Kursstufe des Gymnasiums 2001	Lehrplaneinheit 7: Praktische und theoretische Grenzen des Rechnereinsatzes
2	Bayern EGy III (G8 liegt noch nicht vor)	Die Schüler lernen die prinzipiellen Grenzen informatischer Systeme kennen und erhalten Einblicke in die Problemkreise der Korrektheit und Komplexitätsabschätzung von Informatiksystemen
3	Berlin Rahmenlehrplan für die gymnasiale Oberstufe 2006	4.5 Informatik, Mensch und Gesellschaft Eine direkte Einordnung NP-schwerer Probleme ist nicht vorgegeben, aber grundsätzlich möglich
4	Brandenburg Rahmenlehrplan für den Unterricht in der gymnasialen Oberstufe 2006	Die Schülerinnen und Schüler ... beurteilen die Grenzen des Einsatzes von Informatiksystemen ... Eine direkte Einordnung NP-schwerer Probleme ist nicht vorgegeben, aber grundsätzlich möglich.
5	Bremen Fechrahmenplan Informatik Gymnasiale Oberstufe 2001	Grundlagen der Theoretischen Informatik
6	Hamburg 2004	Die Schülerinnen und Schüler ... • schätzen die Zeitkomplexität von Algorithmen ab • ordnen Probleme Komplexitätsklassen zu
7	Hessen 2006	13.1 Konzepte und Anwendungen der Theoretischen Informatik
8	Mecklenburg-Vorpommern 2001	Algorithmisch lösbare und unlösbare Probleme
9	Niedersachsen 2007	Das Kultusministerium nimmt lediglich Bezug auf die EPA
10	Nordrhein-Westfalen 1998	Ausgewählte Gebiete der theoretischen Informatik
11	Rheinland-Pfalz 2004	Grenzen algorithmisch arbeitender Systeme Die Existenz praktisch nicht durchführbarer algorithmischer Problemlösungen aufzeigen
12	Saarland 1997	Unterrichtsthema 1.2: Praktische Grenzen der Berechenbarkeit
13	Sachsen 1992	Wahlgrundkurs 12/I: Gesellschaftliche und theoretische Probleme der Informatik – Lernbereich 3: Algorithmentheorie (14 Std.)
14	Sachsen-Anhalt 2003	Thema: Informatik und Gesellschaft Inhalte: Grenzen von Informatiksystemen Eine direkte Einordnung NP-schwerer Probleme ist nicht vorgegeben, aber grundsätzlich möglich
15	Schleswig-Holstein 2002	4.4.2 Jahrgangsstufe 12.1: Algorithmen und Datenstrukturen Algorithmen und Datenstrukturen werden zu ausgewählten Problemen entwickelt und durch Effizienzbetrachtungen hinsichtlich ihrer Brauchbarkeit untersucht
16	Thüringen 1999	Themenbereich 8: Möglichkeiten und Grenzen des Einsatzes von Informatiksystemen

Auszug aus den EPA Informatik

„Aus dem Unterricht ist bekannt, dass es Probleme gibt, die prinzipiell mit einem Computer unlösbar sind oder bei denen bei bekanntem Lösungsalgorithmus die verfügbaren Ressourcen zur Problemlösung nicht ausreichen und dass zahlreiche dieser Probleme hohe praktische Relevanz besitzen. Im Unterricht wurden Algorithmen entworfen und die Zeitkomplexität von Algorithmen abgeschätzt. Der Entwurf wurde in Beschreibungs-, Strukturierungs- und algorithmische Phase gegliedert. Die Auswirkungen des Computereinsatzes in mehreren gesellschaftlichen Bereichen wurde diskutiert."

Quelle: KMK – Ständige Konferenz der Kultusminister der Länder in der Bundesrepublik Deutschland: Einheitliche Prüfungsanforderungen in der Abiturprüfung Informatik. Beschluss der Kultusministerkonferenz vom 01.12.1989 i. d. F. vom 05.02.2004. S. 46 (http://www.kmk.org/doc/beschl/EPA-Informatik.pdf).

Abbildung 5.1.: $\mathcal{NP}$-harte Probleme in den Lehrplänen [al.07]

halte in beiden Kursformen unterrichtet werden. Meiner Meinung nach sollten Graphen gerade wegen ihrer Anschaulichkeit und der Möglichkeit zum vielfältigen Einsatz bei der Modellierung obligatorisch unterrichtet werden.
In der dritten Qualifikationsphase ist dann das Thema „Konzepte und Anwendungen der Teoretischen Informatik" für das gesamte Halbjahr vorgesehen. Unter den verbindlichen Unterrichtsinhalten ist hier die Behandlung der Turingmaschine oder der Registermaschine aufgeführt. Wie zuvor soll im Leistungskurs eine der beiden Maschinen obligatorisch, im Grundkurs fakultativ behandelt werden.

> „Die Turingmaschine als universelle symbolverarbeitende Maschine repräsentiert nicht nur eine Präzisierung des Algorithmenbegriffs, sondern dient auch zur Veranschaulichung der prinzipiellen Gren-

> zen maschineller Informationsverarbeitung. [...] Die Turingmaschine hat eine einfachere Struktur als eine Registermaschine.“ [Kul12]

Bei der Behandlung endlicher Automaten wird im Leistungskurs der Begriff des Nichtdeterminismus erarbeitet. Dieser ist für die Bildung der Klasse $\mathcal{NP}$ unumgänglich. Nach [Bau96] „[...] liegt die Einsicht nahe, dass Nichtdeterminismus auch ein wichtiges didaktisches Prinzip ist. Wir dürfen ihn ohne Zögern zu den fundamentalen Ideen der Informatik rechnen!“. Die Übertragung der hier gewonnen Erkenntnisse auf Nichtdeterministische Turingmaschinen gelingt, da eine Turingmaschine als nichts anderes wie eine präzisere Modellierung eines Automaten aufgefasst werden kann. Der Fokus zum Thema Berechenbarkeit liegt bei den verbindlichen Unterrichtsinhalten allerdings auf der Entscheidbarkeit, dem Halteproblem und prinzipieller Grenzen algorithmischer Verfahren. Die Komplexitätstheorie als Verfeinerung der Berechenbarkeitstheorie taucht unter den fakultativen Unterrichtsinhalten auf. Fakultativ können die Klassen $\mathcal{P}$ und $\mathcal{NP}$, die $\mathcal{P} = \mathcal{NP}$-Problematik und $\mathcal{NP}$-vollständige Probleme behandelt werden. Etwas verwunderlich ist, dass hier der Satz von Cook, der eine Klassifizierung von $\mathcal{NP}$-vollständigen Problemen überhaupt erst möglich macht, nicht explizit erwähnt wird.
In Q4 ist ein Wahlthema vorgesehen. Entscheidet man sich als Lehrkraft für das Wahlthema technische Informatik, so kann man beispielsweise die logischen Grundschaltungen AND, OR, NOT, NAND, NOR, XOR behandeln. Die boolesche Algebra mit den Gesetzen zur Vereinfachung, wird im Hinblick auf die Reduktion von Schaltnetzen erwähnt. Ebenso können die disjunktive und konjunktive Normalform behandelt werden.

Zusammenfassend ist zu sagen, dass die in dieser Arbeit vorgestellte Thematik zwar durchaus im Lehrplan zu finden ist, aber fast immer im fakultativen Bereich und wenn ein Teilgebiet obligatorisch erwähnt wird, dann nur für den Leistungskurs. Auch der Sachverhalt, dass die boolesche Algebra nur wahlweise in Q4 eingeführt wird, kann ein Problem darstellen, da man diese um den Satz von Cook zu verstehen braucht.

In Anlehnung an die von der Gesellschaft der Informatik vorgestellten Bildungsstandards für die Sekundarstufe I wird im Folgenden eine weitere Betrachtung der Thematik um die polynomielle Reduktion und der $\mathcal{NP}$- Vollständigkeit erfolgen. Die von der GI entwickelten Bildungsstandards sind der Sek I gewidmet, eine Empfehlung für die Sek II ist bisher nicht erfolgt. Ich bin aber der Meinung, dass viele der angesprochenen Prozess- und Inhaltsbereiche (vgl. Abbildung 5.2) auf die gymnasiale Oberstufe übertragbar sind.
Von den Inhaltsbereichen können zwei der genannten, nämlich Algorithmen, sowie die Thematik Informatik, Mensch und Gesellschaft durch das vorgestellte Thema besonders gut abgedeckt werden. Die Komplexitätstheorie, die Existenz von $\mathcal{NP}$-vollständigen Problemen im Speziellen, weist klare Schranken von Algorithmen auf. Selbst wenn ein Problem durchaus algorithmisch lösbar (also berechenbar) ist, gibt es Probleme, die trotzdem nicht praktisch

Abbildung 5.2.: Die Prozess- und Inhaltsbereiche der Bildungsstandards Informatik sind untrennbar miteinander verzahnt.

lösbar sind. Es geht hier um die Bewertung von Algorithmen. Laufzeiten werden betrachtet und so die Effizienz abgeschätzt. Die Resultate, Ergebnisse und Erfahrungen der Komplexitätstheorie stehen in sehr engem Zusammenhang mit der Praxis. Erkenntnisse wirken sich unmittelbar aus. Es lohnt nicht bei einem $\mathcal{NP}$-vollständigen Problem nach dem Optimum zu suchen. Eine Abschwächung macht das Lösen wahrscheinlich doch möglich. Im heutigen Zeitalter, in dem immer mehr Aufgaben auf den Computer übertragen werden, und man dem Glauben verfallen ist, der Computer könne alle erdenklichen berechenbaren Probleme (schnell) lösen, klären Erkenntnisse aus der Komplexitätstheorie den Misstand diesen Irrglaubens auf.
Meiner Meinung nach kann die in dieser Arbeit vorgestellte Thematik je nach Auslegung und Stundenkonzeption in alle Prozessbereiche eingebunden werden, weswegen ich an dieser Stelle versuche, mich auf die wesentlichen Beobachtungen zu beschränken. Die Schüler und Schülerinnen werden mit Sicherheit viel begründen und bewerten, denn das ist genau das, was man bei der Klassifizierung der Probleme macht. Gibt es einen effizienten (also in polynomieller Zeit) Algorithmus? Gehört das Problem in die Komplexitätsklasse $\mathcal{P}$? Oder gibt es keine praktisch durchführbare Lösungsmöglichkeit? Wenn nein, warum nicht und wie ist dieser Sachverhalt beweisbar? Vorallem die Begründungen warum eine Reduktion in polynomieller Zeit durchführbar ist, sind an dieser Stelle nennenswert. Denn für diese Begründungen muss man einen Algorithmus und dessen Struktur und auch den genauen Aufbau eines Problems vollständig durchdrungen haben.
Noch Hervorzuheben ist der Bezug zu dem Prozessbereich Modellieren. Jedes der vorgestellten $\mathcal{NP}$-vollständigen Probleme lässt sich auf verschiedene Art und Weise modellieren. Das Konzept der polynomiellen Reduktion baut auf dieser Basis auf. Nur durch unterschiedliche Repräsentationen und Modellie-

rungen ist es erst möglich, zwischen (auf den ersten Blick) sehr unterschiedlichen Problemen Ähnlichkeiten zu erkennen und zu nutzen. Im nächsten Kapitel wird schnell deutlich, dass man ohne Werkzeuge der Modellbildung in dieser Thematik nicht sehr weit kommt. Jedes der vorgestellten Beispiele hat eine große praktische Relevanz und alle Probleme müssen als ersten Schritt mathematisch oder informatisch modelliert werden, um der Lösung der Probleme und deren Einordnung einen Schritt näher zu kommen. Nach Schubert und Schwill

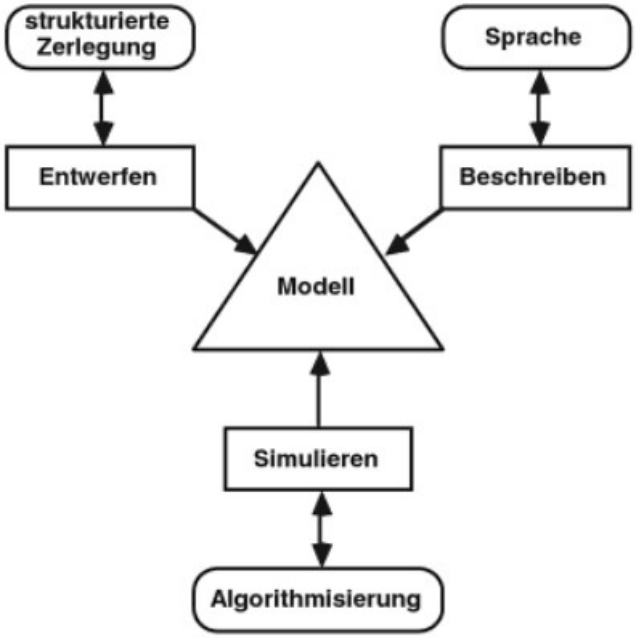


Abbildung 5.3.: Phasen der Modellbildung [SS04]

in [SS04] wird die Informatik gelegentlich auch „[...] als Wissenschaft der Modellbildung bezeichnet und damit der Modellbildung ein hoher Stellenwert im Infomatikunterricht zugesprochen.“ Akzeptiert man diese Charakterisierung der Informatik, so lassen sich nach den Autoren drei sogenannte Masterideen als tragende Säulen der Modellbildung (vgl. Abbildung 5.3) auffassen:

- Mit der strukturierten Zerlegung sind Ideen verbunden, mit deren Hilfe man ein reales System analysiert und die modellrelevanten Eigenschaften ableitet.
- Das Modell wird anschließend auf der Basis einer Beschreibungssprache[4] präzisiert und öffnet sich so weiteren syntaktischen und vorallem semantischen Analysen und Transformationen[5].
- Der dynamische Aspekt von Modellen, die Möglichkeit, sie zu simulieren, wird durch die Algorithmisierung erfasst. Die zugehörigen Ideen dienen

[4] In den Beispielen in Kapitel 6 meist durch Verwendung der Boolesche Algebra

[5] Zum Beispiel durch polynomieller Reduktion oder der „Transformation“ in eine äquivalente Formel in konjunktiver Normalform

dem Entwurf und dem Ablauf von Simulationsprogrammen, wobei die Simulation im weitesten Sinne zu verstehen ist.

Alle der eben vorgestellten Masterideen lassen sich durch die Verwendung von praxisnahen Beispielen für die Veranschaulichung der polynomiellen Reduktion und die damit verbundenen Auswirkungen der $\mathcal{NP}$-Vollständigkeit gut erlernen und verdeutlichen.

6. Das Konzept der polynomiellen Reduktion an Anwendungsbeispielen oder „Was man alles färben kann“

Nachdem nun alle Grundlagen bekannt sind, möchte ich zu dem Kern meiner Arbeit kommen. Wie kann das Konzept der polynomiellen Reduktion an Beispielen motiviert und erarbeitet werden?
Alle hier vorgestellten Beispiele können, da sie nicht aufeinander aufbauen, unabhängig voneinander behandelt werden. Die Schwierigkeit der Beispiele variiert. Die Beispiele werden, soweit eine Differenzierung möglich war, nach aufsteigender Schwierigkeit präsentiert. Das meiner Meinung nach leichteste Beispiel der Modellierung von sicheren Ampelschaltungen an Kreuzungen wird daher zuerst vorgestellt. Das anspruchsvollste Beispiel ist gewiss das Sudoku-Problem am Ende.

Die wichtigste Voraussetzung für die Behandlung von allen Beispielen ist das Verständnis davon, wie ein $\mathcal{NP}$-Vollständigkeits-Beweis geführt wird (vgl. Kapitel 4.6). Natürlich sollen auch die Definitionen der Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ verstanden sein. Aber es kann dennoch eine Entwarnung geben. Um die Beispiele im Unterricht sinnvoll einzusetzen, müssen nicht alle Grundlagen aus den Grundlagenkapiteln (vgl. ab Kapitel 1) notwendigerweise vollständig behandelt und bekannt sein. Um einen Überblick der notwendigen Lernvoraussetzungen zu geben, werde ich diese vor jedem Beispiel nennen.
Allen Beispielen in diesem Kapitel ist gemeinsam, dass die Formulierung als Boolsche Formel intuitiv leicht nachvollziehbar und Schülern und Schülerinnen nach präsentierten Beispielen auch selbst gelingen sollte. Zudem wird der Zusammenhang der Beweise der $\mathcal{NP}$-Vollständigkeit durch die polynomielle Reduktion auf ein bekanntes $\mathcal{NP}$-vollständiges Problem versucht zu verdeutlichen. Um das so anschaulich wie möglich zu gestalten, habe ich mich entschieden immer auch eine Interpretation als Färbbarkeits-Problem anzugeben.

6.1. Modellierung von „sicheren“ Ampelschaltungen

Lernvoraussetzungen:

- Grundlagen der booleschen Algebra: Darstellung als Wahrheitstabelle, grundlegende Rechenregeln (vgl. Kapitel 3), konjunktive Normalform
- Aufbau eines Graphen: Ein Graph besteht aus Knoten und Kanten. Durch Kanten können Zusammenhänge (Relationen) modelliert werden, benachbarte Knoten (vgl. Kapitel 2)
- Graphfärbung (vgl. Definition 4.8)

Sollten Grundlagen der booleschen Algebra noch nicht behandelt worden sein, so kann dies auch mit dem Programm „Infotraffic“ (vgl. Kapitel A.1) anhand der Beispiele erfolgen. Die Boolesche Algebra muss also nicht als zwingende Voraussetzung gesehen werden. Wichtig ist, dass man das Konzept zur Bearbeitung des Beispiels benötigt. Zu Beginn möchte ich eine leicht zu motivierende und gut vermittelbare „Problemklasse“ vorstellen. Wir begeben uns in den Straßenverkehr. Aufgabe wird es sein, Kreuzungen möglichst sicher[1] zu

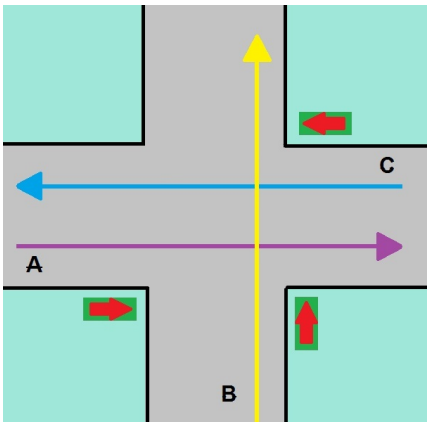


Abbildung 6.1.: Verkehrssituation: Kreuzung mit drei Spuren

gestalten. Betrachten wir beispielsweise Abbildung 6.1. In dieser Verkehrssituation treffen drei Spuren aufeinander und es stehen drei Ampeln zur Verfügung. Wie müssen die Ampeln geschaltet sein, damit es nicht zu Kollisionen kommen kann? Ich werde im Folgenden zwei mögliche Modellierungen der Situation vorstellen.
Als erstes betrachten wir die Darstellung des Problems als Erfüllbarkeitsproblem (vgl. SAT in 4.6). Im ersten Schritt modellieren wir die Situation durch eine Wahrheitstabelle (vlg. Tabelle 6.1). Dabei gehen wir wie folgt vor: Angenommen eine 1 steht dafür, dass eine Ampel grün und eine 0 entsprechend

[1] Unter „sicher“ werden wir im Folgenden Kollisionsfreiheit verstehen.

A	B	C	sicher	Klausel
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	0	$A \vee \bar{B} \vee \bar{C}$
1	0	0	1	
1	0	1	1	
1	1	0	0	$\bar{A} \vee \bar{B} \vee C$
1	1	1	0	$\bar{A} \vee \bar{B} \vee \bar{C}$

Tabelle 6.1.: Wahrheitstabelle zu der Verkehrssituation aus Abbildung 6.1

dafür, dass eine Ampel rot ist. Wir betrachten alle möglichen Kombinationen von Ampelschaltungen. Kann es während einer bestimmten Schaltung nicht zu Kollisionen kommen, dann gilt diese als sicher und wir tragen in der entsprechenden Spalte eine 1 ein. Ist die Sicherheit der Kombination nicht gewährt trägt man eine 0 ein. Dieses Vorgehen resultiert in Tabelle 6.1.

In Kapitel 3 haben wir festgestellt, dass man eine KNF-Formel ganz einfach aus einer Wahrheitstabelle der Funktion entwickeln kann. Wir benutzen das in Beispiel 3.1 beschriebene Verfahren auch hier und erhalten die konjunktive Normalform der durch die Wahrheitstabelle in Tabelle 6.1 beschriebenen Funktion als

$$f = (A \vee \bar{B} \vee \bar{C}) \wedge (\bar{A} \vee \bar{B} \vee C) \wedge (\bar{A} \vee \bar{B} \vee \bar{C})$$

Die Frage, ob eine sichere Ampelschaltung möglich ist, kann jetzt durch das Beantworten der Frage, ob es für die erhaltene KNF eine erfüllende Belegung gibt, beantwortet werden. Wir wollen demnach das in Kapitel 4.6 beschriebene Satisfiability Problem („Erfüllbarkeitsproblem“) lösen. In unserem Beispiel können wir durch geübtes Hinsehen eine erfüllende Belegung mit C=0 und B=0 finden. Bei größeren Problemen ist das, wie wir bereits in Kapitel 4.6 gesehen haben, wesentlich schwieriger. Festzuhalten ist hier als wesentlicher Punkt, dass die Modellierung der sicheren Verkehrssituation als allgemeines KNF-SAT-Problem[2] beschrieben werden kann.

Wie ich oben erwähnt habe, gibt es aber noch eine weitere Modellierungsmöglichkeit. Wir wollen im Folgenden das eben erzielte SAT-PROBLEM auf ein Färbbarkeitsproblem (vgl. Definition 4.8) polynomiell reduzieren. Betrachten wir also Abbildung 6.1 erneut.
Zuerst übersetzen wir die Situation der Kreuzung in Abbildung 6.2 in einen Graphen G=(V, E). Diesen konstruiert man wie folgt:

- Die Knotenmenge V erhalten wir durch die gegebenen Ampeln, d.h. $V = \{A, B, C\}$.

[2] in diesem Beispiel als 3-SAT-Problem

- Die Kantenmenge wird so gewählt, dass wir genau wissen, wenn es für zwei Ampeln zu Kollisionen kommen kann. Wir fügen eine Kante $e = \{u, v\}$ zu E, wenn es zwischen Knoten u und v einen Konflikt gibt, d.h. die Kantenmenge ergibt sich aus:

  $E = \{u, v|$ Sind Ampel u und v zeitgleich grün, kann es Kollisionen geben$\}$

Der resultierende Graph ist in Abbildung 6.2 abgebildet. Nun ist unser Pro-

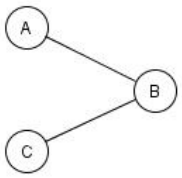


Abbildung 6.2.: Graph zur Situation aus Abbildung 6.1

blem direkt in ein Färbbarkeitsproblem transformierbar. Die Frage, wie viele Ampelphasen notwendig sind, kann in die Frage: „Wie viele Farben benötigt man mindestens, um eine konfliktfreie Knotenfärbung des Graphen zu erhalten?“ übersetzt werden. In Abbildung 6.3 ist eine konfliktfreie Knotenfärbung des Graphen zu sehen.

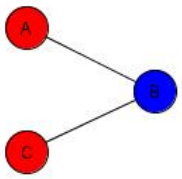


Abbildung 6.3.: Gefärbter Graph zur Situation aus Abbildung 6.1

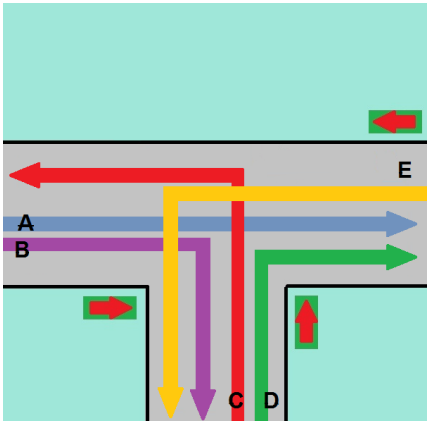


Abbildung 6.4.: Verkehrsituation: Kreuzung mit fünf Spuren

Betrachten wir als weiteres Beispiel die Verkehrssituation in Abbildung 6.4. Wir können genauso vorgehen wie eben bei der etwas übersichtlicheren Variante mit nur drei Spuren. Schauen wir uns zuerst die Modellierung als Satisfiability-Problem an. In Tabelle 6.2 ist die zu Abbildung 6.4 gehörige Wahrheitstabelle zu finden. Allein durch die Größe der Wahrheitstabelle (vgl. Tabelle 6.2) erkennt man, dass das Aufstellen einer zugehörigen KNF mit dem im ersten Beispiel benutzten Verfahren mühselig ist und eine recht lange unübersichtliche Formel liefert. Liest man die KNF aus Tabelle 6.2 aus, so erhält man folgende Formel:

$$\begin{aligned} f = &(\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee E) \wedge (\bar{A} \vee \bar{B} \vee C \vee D \vee E) \wedge (A \vee \bar{B} \vee C \vee D \vee \bar{E}) \wedge \\ &(A \vee \bar{B} \vee C \vee \bar{D} \vee \bar{E}) \wedge (A \vee \bar{B} \vee \bar{C} \vee D \vee \bar{E}) \wedge (A \vee \bar{B} \vee \bar{C} \vee \bar{D} \vee \bar{E}) \wedge \\ &(\bar{A} \vee B \vee C \vee D \vee \bar{E}) \wedge (\bar{A} \vee B \vee C \vee \bar{D} \vee E) \wedge (\bar{A} \vee B \vee C \vee \bar{D} \vee \bar{E}) \wedge \\ &(\bar{A} \vee B \vee \bar{C} \vee D \vee E) \wedge (\bar{A} \vee B \vee \bar{C} \vee D \vee \bar{E}) \wedge (\bar{A} \vee B \vee \bar{C} \vee \bar{D} \vee E) \wedge \\ &(\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee E) \wedge (\bar{A} \vee \bar{B} \vee C \vee D \vee \bar{E}) \wedge (\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee E) \wedge \\ &(\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee \bar{E}) \wedge (\bar{A} \vee \bar{B} \vee \bar{C} \vee D \vee E) \wedge (\bar{A} \vee \bar{B} \vee \bar{C} \vee D \vee \bar{E}) \wedge \\ &(\bar{A} \vee \bar{B} \vee \bar{C} \vee \bar{D} \vee E) \wedge (\bar{A} \vee \bar{B} \vee \bar{C} \vee \bar{D} \vee \bar{E}) \end{aligned}$$

Die ausgelesene Formel ist unnötig lang und nicht gut lesbar. Nun kann man sich der Techniken zur Reduzierung einer Formel bedienen, aber auch das ist recht mühsam und im Unterricht meiner Meinung nach unpraktikabel. An dieser Stelle drängt sich der Computereinsatz förmlich auf. Es gibt zahlreiche Programme, die es dem Nutzer ermöglichen zu einer Formel verschiedene Normalformen berechnen zu lassen[3].
Um die geschilderte Situation hier komfortabel in ein überschaubares SAT-Problem zu übertragen, bediene ich mich eines recht schönen Programms mit dem Namen „Infotraffic“[4]. Das von R. Arnold in der Schweiz entwickelte Programm, ist kostenfrei zugänglich[5] und wird in Kapitel A.1 vorgestellt.
An dieser Stelle nutzen wir das Programm, um sehr bequem eine KNF zu dem oben beschriebenen Problem zu finden. Man erhält folgende Formel in konjunktiver Normalform:

$$(\bar{C} \vee \bar{B}) \wedge (\bar{B} \vee \bar{E}) \wedge (\bar{A} \vee \bar{E}) \wedge (\bar{A} \vee \bar{D}) \wedge (\bar{A} \vee \bar{C})$$

Wie zuvor, wenn auch etwas arbeitsintensiver, erhalten wir also eine Fragestellung des Satisfiability-Problems, welches $\mathcal{NP}$-vollständig ist (vgl. Kapitel 4.6). Betrachten wir im zweiten Schritt wieder den von der Situation in Abbildung 6.4 induzierten Graph[6] (vgl. Abbildung 6.5). Wie zuvor handelt es sich um ein Graphfärbungsproblem. Eine Möglichkeit den Graphen zu färben ist in Abbildung 6.6 zu sehen. Wichtig hier ist zu Beobachten, dass jede konfliktfreie Färbung nicht mit weniger als drei Farben auskommen kann[7]! Übertragen auf

[3] Zum Beispiel den KNF-DNF-Konverter von [KNF12]
[4] Die beiden in diesem Kapitel vorgestellten Beispiele entstammen dem Programm „Infotraffic“
[5] Der Download des Programms kann unter [RA12b] erfolgen.
[6] Analog zu vorherigem Beispiel konstruiert
[7] Betrachte Teilgraph, der aus V=$\{A, E, C\}$ besteht

A	B	C	D	E	sicher	Klausel
0	0	0	0	0	1	
0	0	0	0	1	1	
0	0	0	1	0	1	
0	0	0	1	1	1	
0	0	1	0	0	1	
0	0	1	0	1	0	$\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee E$
0	0	1	1	0	1	
0	0	1	1	1	0	$\bar{A} \vee \bar{B} \vee C \vee D \vee E$
0	1	0	0	0	1	
0	1	0	0	1	0	$A \vee \bar{B} \vee C \vee D \vee \bar{E}$
0	1	0	1	0	1	
0	1	0	1	1	0	$A \vee \bar{B} \vee C \vee \bar{D} \vee \bar{E}$
0	1	1	0	0	1	
0	1	1	0	1	0	$A \vee \bar{B} \vee \bar{C} \vee D \vee \bar{E}$
0	1	1	1	0	1	
0	1	1	1	1	0	$A \vee \bar{B} \vee \bar{C} \vee \bar{D} \vee \bar{E}$
1	0	0	0	0	1	
1	0	0	0	1	0	$\bar{A} \vee B \vee C \vee D \vee \bar{E}$
1	0	0	1	0	0	$\bar{A} \vee B \vee C \vee \bar{D} \vee E$
1	0	0	1	1	0	$\bar{A} \vee B \vee C \vee \bar{D} \vee \bar{E}$
1	0	1	0	0	0	$\bar{A} \vee B \vee \bar{C} \vee D \vee E$
1	0	1	0	1	0	$\bar{A} \vee B \vee \bar{C} \vee D \vee \bar{E}$
1	0	1	1	0	0	$\bar{A} \vee B \vee \bar{C} \vee \bar{D} \vee E$
1	0	1	1	1	0	$\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee E$
1	1	0	0	0	1	
1	1	0	0	1	0	$\bar{A} \vee \bar{B} \vee C \vee D \vee \bar{E}$
1	1	0	1	0	0	$\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee E$
1	1	0	1	1	0	$\bar{A} \vee \bar{B} \vee C \vee \bar{D} \vee \bar{E}$
1	1	1	0	0	0	$\bar{A} \vee \bar{B} \vee \bar{C} \vee D \vee E$
1	1	1	0	1	0	$\bar{A} \vee \bar{B} \vee \bar{C} \vee D \vee \bar{E}$
1	1	1	1	0	0	$\bar{A} \vee \bar{B} \vee \bar{C} \vee \bar{D} \vee E$
1	1	1	1	1	0	$\bar{A} \vee \bar{B} \vee \bar{C} \vee \bar{D} \vee \bar{E}$

Tabelle 6.2.: Wahrheitstabelle zu der Verkehrssituation aus Kapitel 6.1

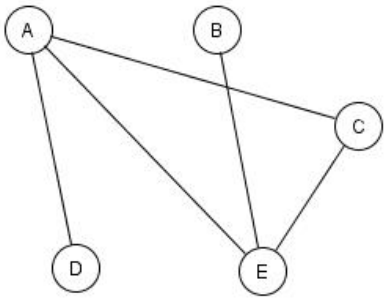


Abbildung 6.5.: Graph zur Situation aus Abbildung 6.4

das gestellte Problem bedeutet das, dass wir mindestens drei Ampelphasen benötigen, um Kollisionsfreiheit zu garantieren. Hier natürlich immer vorausgesetzt, dass die Autofahrer die Ampelschaltung beachten und nicht bei rot fahren. Der ein oder andere aufmerksame Leser hat sich bestimmt schon ge-

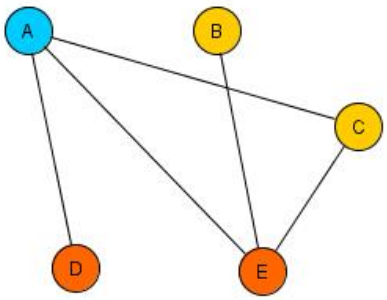


Abbildung 6.6.: Gefärbter Graph zur Situation aus Abbildung 6.4

fragt, wo denn hier ein Beispiel zum Konzept der polynomiellen Reduktion zu finden ist. Fakt ist, dass die durchzuführende polynomielle Reduktion an den vorgestellten Beispielem so intuitiv ist, dass man kaum bemerkt, dass man die Hürde allein durch die zwei verschiedenen Modellierung des Problems schon genommen hat. Dahinter steckt, dass COLOR allgemein auf SAT polynomiell reduzierbar ist (vgl. Kapitel 4.6).

6.2. Von Parties und Experimenten und davon was Abgeordnete damit zu tun haben

Lernvoraussetzungen:

- Grundlagen der booleschen Algebra: Darstellung als Wahrheitstabelle, grundlegende Rechenregeln (vgl. Kapitel 3), Biimplikation, konjunktive Normalform

- Es werden Graphprobleme behandelt, demnach werden Grundlagen der Graphentheorie benötigt: Aufbau eines Graphen, Adjazenz, Komplementärgraph

In diesem Kapitel wird sich um drei bekannte Probleme gekümmert, die immer wieder, unter anderem bei [al.07], in der Literatur auftauchen. Zuvor möchte ich an ein Problem erinnern, dass schon in Beispiel 4.9 vorgestellt wurde: Sei G = (V, E) ein ungerichteter Graph. Eine Teilmenge T$\in V$ heißt eine Überdeckung, wenn alle Kanten einen Endpunkt in T besitzen. Das Vertex Cover Problem (Knotenüberdeckungsproblem) lautet dann wie folgt:

$$\text{VC}=\{\,(G,k)|\ \text{G besitzt eine Menge T von k Knoten,}\\ \text{so dass jede Kante einen Endpunkt in T besitzt.}\}$$

Beispiel 6.1. Sei der Graph G=(V, E) mit V=$\{1,2,3,4,5\}$ und $E = \{\{1,3\},\{1,4\},\{1,5\},\{2,3\},\{3,4\},\{3,5\},\{4,5\}\}$ (vgl. Abbildung 6.7) gegeben. Die Teilmenge $VC_G = \{1,3,4\}$ ist ein minimales Vertex Cover des Graphen G.

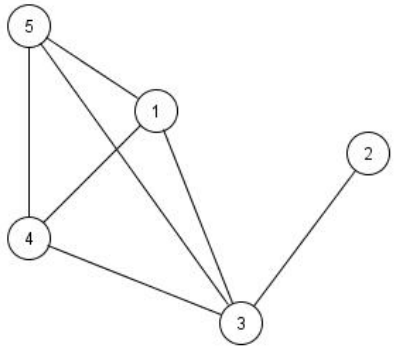


Abbildung 6.7.: Graph zu den Beispielen 6.1 und 6.2

Ein verwandtes Problem ist das sogenannte Cliquen-Problem[8]:

$$\text{k-CLIQUE}=\{(G,k)|\ \text{G hat eine Clique der Größe k}\}$$

Unter einer Clique versteht man eine Teilmenge T der Knoten von G, so dass je zwei Knoten durch eine Kante verbunden sind.

Beispiel 6.2. Betrachte erneut den in Abbildung 6.7 dargestellten Graphen. Die Menge $C_{G4} \in V$ mit $C_{G4} = \{1,3,4,5\}$ bildet eine Clique der Größe 4. Eine Clique der Größe 3 wäre beispielsweise durch $C_{G3} = \{1,3,4\}$ gegeben.

[8] Die Definition des Cliquen-Problems stammt aus [Sch11]

In Beispiel 4.7 haben wir uns schon einmal mit einem sogenannten Partyproblem beschäftigt. Statt die Gäste so auf Tische zu verteilen, dass es nicht zu Disharmonie kommt, werden wir nun nur Gäste einladen, die miteinander harmonieren. Wir betrachten in diesem Kapitel das von M. Fothe in [al.07] vorgestellte Partyproblem.
Nehmen wir an, eine Person X möchte eine harmonische Party veranstalten. Am schönsten wäre es, acht Gäste einzuladen. Bezeichnen wir die Gäste mit den Namen A bis H. Leider harmonisieren nicht alle Gäste miteinander, manche mögen sich einfach nicht. Ziel von Person X ist es nun, möglichst viele der acht Gäste einzuladen. Betrachten wir dazu den Graphen in Abbildung 6.8. Hier besteht Disharmonie zwischen folgenden Personenpaaren: (A, D),(B, D),(B, E), (C, E), (D, E), (D, F), (E, G), (F, G) und (F, H). Disharmonien werden in dem Graphen durch eine Kante ausgedrückt. Eine Möglichkeit

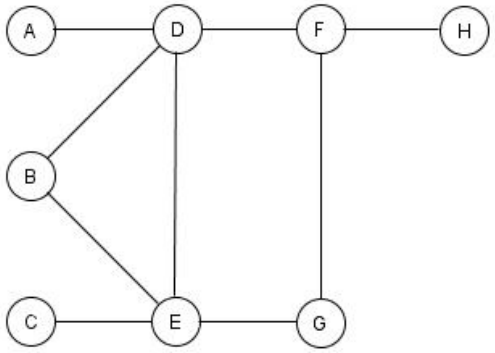


Abbildung 6.8.: Beziehungsgraph zum beschriebenen Partyproblem [al.07]

zu entscheiden, welche Gäste keine Einladung bekommen sollen, ist es nach einem Vertex Cover zu suchen. Entsprechend erhalten wir die „geladenen Gäste“ durch „Subtraktion“ des gefundenen Vertex Covers von der Knotenmenge. Wie aber erhält man möglichst geschickt ein Vertex Cover[9]? Betrachten wir dazu den Komplementärgraphen $\overline{G}$ zu G. Das Komplement[10] eines Graphen G=(V, E) ist wie folgt definiert:

$$\overline{G} = (\overline{V}, \overline{E}) \text{ mit } \overline{V} = V \text{ und } \overline{E} = \{\{x, y\} \mid x, y \in V, x \neq y, \{x, y\} \notin E\}$$

Der zu Abbildung 6.8 gehörige Komplementärgraph ist in Abbildung 6.9 zu sehen. Die Frage nach einem minimalen Vertex Cover kann dadurch gelöst werden, dass man eine maximale Clique C in dem Komplementärgraphen sucht. Das gesuchte Vertex Cover ergibt sich dann als V\C. Bei einem relativ „kleinen“ Graphen, wie wir ihn hier vorliegen haben, ist das noch gut zu bewältigen. Wir können alle Knoten der Knotenmenge nacheinander besuchen und schauen, ob der besuchte Knoten Kanten zu allen anderen Knoten besitzt. Erfüllt

[9] Das Vertex-Cover-Problem ist ein $\mathcal{NP}$-vollständiges Problem.
[10] Die Definition wurde aus [Sch09] übernommen.

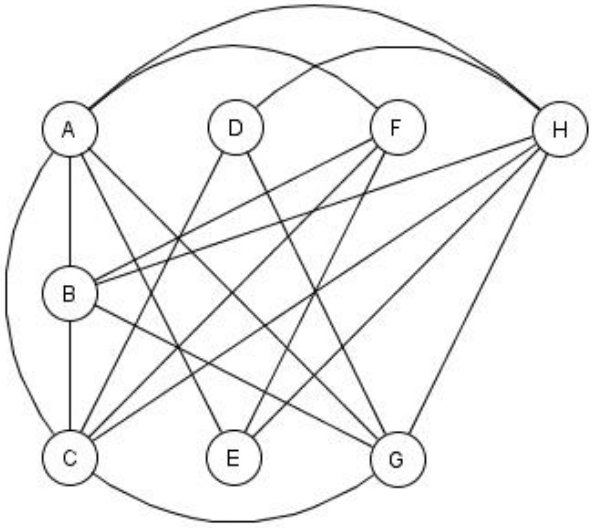


Abbildung 6.9.: Komplementärgraph zu dem Graphen aus Abbildung 6.8

er diese Bedingung nicht, so verwerfen wir ihn erst einmal. So geht man sukzessive vor, bis eine Clique gefunden wurde[11]. Allerdings muss man bedenken, wie viele verschiedene Teilmengen bei einer solchen sogenannten „vollständigen Durchmusterung“ zu testen sind. Eine maximale Clique in unserem Beispielgraph kann noch gut gefunden werden. Die Teilmenge $\{A, B, C, G, H\}$ bildet eine Clique der Größe 5. Die Suche nach einer Clique ist im Allgemeinen aber ein $\mathcal{NP}$-vollständiges Problem. „Die Knotenmenge V mit n Knoten besitzt 2^n verschiedene Teilmengen, d.h. in Abhängigkeit von der Knotenzahl n des Graphen G sind exponentiell viele Teilmengen (d.h. exponentiell viele Fälle) zu testen“ [al.07].
Für unseren Gastgeber bedeutet das gefundene Ergebnis $\{A, B, C, G, H\}$ für eine Clique, dass er am besten die Personen D, E und F nicht einlädt.

Nun möchte ich meine Behauptung, dass es sich bei dem Cliquen-Problem um ein $\mathcal{NP}$-vollständiges Problem handelt, beweisen. Wie zuvor wird die polynomielle Reduktion (vgl. Kapitel 4.5) und der Satz von Cook (vgl. Kapitel 4.6) zentral sein. An dieser Stelle genügt es, zu zeigen, dass man das Problem als Erfüllbarkeitsproblem auffassen kann. Dieses ist nach dem Satz von Cook $\mathcal{NP}$-vollständig. Ziel ist es also, das Partyproblem durch eine boolesche Formel (in konjunktiver Normalform) zu formulieren.
Eine Kante $e = \{X, Y\}$ im Graphen (vgl. Abbildung 6.8) drückt aus, dass sich Personen X und Y nicht gut verstehen. Am besten lädt man also nur eine der beiden Personen ein. Die Aussagenlogische Biimplikation hilft uns weiter: Wenn Person X eingeladen wird, dann wird Person Y nicht geladen und umgekehrt. Als boolescher Ausdruck folgt demnach $(X \leftrightarrow \overline{Y})$. Natürlich kann

[11] Wenn die Kantenmenge des Graphen nicht leer ist, werden wir immer fündig, da die Endknoten einer Kante eine Clique der Größe 2 definiert. Selbst der leere Graph, der keine Kanten enthält besitzt Cliquen der Größe 1.

man diesen Sachverhalt auch analog durch zwei Implikationen ausdrücken: $(X \rightarrow \overline{Y}) \wedge (Y \rightarrow \overline{X})$. Wir gehen also so vor, dass wir jede Disharmonie zwischen den Personen mithilfe einer Biimpliaktion ausdrücken. Daraus ergibt sich folgende Formel:

$$\begin{aligned} f = &(A \rightarrow \overline{D}) \wedge (B \rightarrow \overline{D}) \wedge (B \rightarrow \overline{E}) \wedge (C \rightarrow \overline{E}) \wedge \\ &(D \rightarrow \overline{E}) \wedge (D \rightarrow \overline{F}) \wedge (E \rightarrow \overline{G}) \wedge (F \rightarrow \overline{G}) \wedge \\ &(F \rightarrow \overline{H}) \end{aligned}$$

Nach Lemma 3.7 können wir die Formel f auch in eine äquivalente Formel in konjunktiver Normalform transformieren. Das Partyproblem kann demnach auf das Erfüllbarkeitsproblem reduziert werden. Wie viel Zeit wird für die Reduktion benötig? Sicher schaffen wir die Reduktion in polynomieller Zeit. Wir durchlaufen die Adjazenzliste und assozieren mit jeder Kante eine Biimplikation, d.h. wir bewegen uns in der Größenordnung $\mathcal{O}(n^2)$.

Ein sehr ähnliches Problem ist das Experimentproblem, dass ebenfalls in M. Fothes Beitrag [al.07] betrachtet wird.

> „Forscherin A arbeitet in einer experimentellen Wissenschaft auf Basis teurer, nicht immer fehlerfreier Experimentdurchführungen an einer neuen wissenschaftlichen Hypothese. Leider widersprechen sich einige der Experimentergebnisse, was aber allein an der Fehlerhäufigkeit der Ergebnisdaten liegen kann. In den empirischen Wissenschaften ist es daher nicht unüblich, folgende Plausibilitätsbetrachtung anzustellen: Die aus den Experimentdaten abzuleitende Hypothese ergibt sich am besten bzw. am wahrscheinlichsten so, indem man eine möglichst kleine Zahl von (vermutlich) fehlerhaften Experimentergebnissen eliminiert, so dass der verbleibende Rest in sich widerspruchsfrei ist und somit zur Hypothesenbildung taugt.“ [al.07]

Die Aufgabenstellung lautet wie folgt: Es soll versucht werden, eine möglichst große Hypothesenbasis zu finden, wenn sich beispielsweise folgende Hypothesen widersprechen:
(1, 2), (2, 3), (2, 4), (3, 4), (4, 5), (4, 6), (5, 6), (6, 7), (6, 8) und (6, 9).
Abbildung 6.10 zeigt den zugehörigen Konfliktgraphen, der entsteht, wenn man Widersprüche zwischen den Experimentergebnissen durch eine Kante repräsentiert. Die Knotenmenge ergibt sich durch die verschiedenen Ergebnisse der Experimente. Spätestens hier ist zu erkennen, dass es sich um die gleiche Fragestellung handelt, wie in dem Beispiel zuvor. Statt möglichst wenige Personen zu einer Party nicht einzuladen, streichen wir nun möglichst wenige abzuleitende Hypothesen. Ich möchte daher dieses Beispiel nicht wieder von Null an komplett durchgehen, sondern direkt eine boolesche Formel angeben, die das entsprechende Problem modelliert. Wiederum setzen wir die durch Kanten

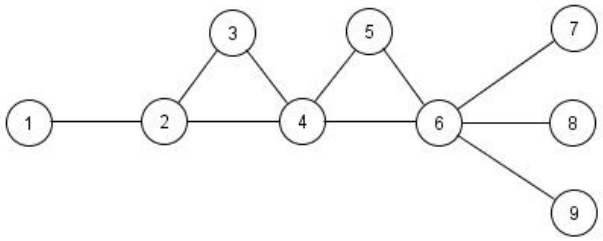


Abbildung 6.10.: Konfliktgraph zu dem Experimentproblem [al.07]

angezeigten Konflikte durch Biimplikationen um:

$$\begin{aligned} f = &(1 \to \overline{2}) \wedge (2 \to \overline{3}) \wedge (2 \to \overline{4}) \wedge (3 \to \overline{4}) \wedge \\ &(4 \to \overline{5}) \wedge (4 \to \overline{6}) \wedge (6 \to \overline{7}) \wedge (6 \to \overline{8}) \wedge \\ &(6 \to \overline{9}) \end{aligned}$$

Auch diese Formel kann nach Aussage von Lemma 3.7 in eine äquivalente Formel in konjunktiver Normalform überführt werden. Ich möchte an dieser Stelle den Ansatz der Umformung[12] vorstellen. Aus Gründen der Übersicht forme ich nur die erste Klausel um, die Umformung der restlichen Klauseln erfolgt analog.

$$\begin{aligned} f &= (1 \Leftrightarrow \overline{2}) \wedge \cdots \\ &= ((1 \wedge \overline{2}) \vee (\overline{1} \wedge 2)) \wedge \cdots \\ &= \left[(1 \vee \overline{1}) \wedge (1 \vee 2) \wedge (\overline{1} \vee \overline{2}) \wedge (\overline{2} \vee 2)\right] \wedge \cdots \\ &= \left[1 \wedge (1 \vee 2) \wedge (\overline{2} \vee \overline{1}) \wedge 1\right] \cdots \\ &= (1 \vee 2) \wedge (\overline{2} \vee \overline{1}) \wedge \cdots \end{aligned}$$

Eine erfüllende Belegung[13] liefert uns die Experimentergebnisse, die am Besten berücksichtigt werden sollten. In diesem Beispiel ist beispielsweise die Belegung: 1=1, 2=0, 3=1, 4=0, 5=1, 6=0, 7=1, 8=1 und 9=1 eine erfüllende Belegung. Es sollten demnach die Experimente 1, 3, 5, 7, 8 und 9 für die Hypothesenbildung herangezogen werden.

Was für den Studenten eine Party ist, ist für einen Abgeordneten eine Komiteesitzung. Naja, zumindest so ähnlich. Das bekannte zugehörige Problem wurde schon 1992 von Simon in [Sim92] betrachtet:

> „Die Abgeordneten eines Parlaments gehören n verschiedenen Ausschüssen an. Jeder Ausschuß tagt jede Woche genau einmal. Ist ein Abgeordneter Mitglied in zwei verschiedenen Ausschüssen, so dürfen diese nicht zur gleichen Zeit stattfinden. Wir möchten wissen, ob wir mit k verschiedenen Sitzungsterminen auskommen.“

[12] Für die Umformung werden die in Kapitel 3 vorgestellten Axiome und Rechenregeln der booleschen Algebra benutzt.

[13] Variablen die auf „true“ bzw. „1“ gesetzt sind entsprechen dabei den Experimenten, deren Ergebnis als nicht fehlerhaft gilt.

Der Graph in Abbildung 6.11 zeigt ein mögliches Szenario. Die Knoten entsprechen dabei den Ausschüssen. Zwei Knoten sind genau dann durch eine Kante verbunden, wenn es einen Abgeordneten gibt, der in beiden Ausschüssen tätig ist. Um die notwendige Anzahl an Sitzungsterminen zu ermitteln, muss ein Färbungsproblem gelöst werden (vgl. Definition 4.8). Analog zu dem Beispiel

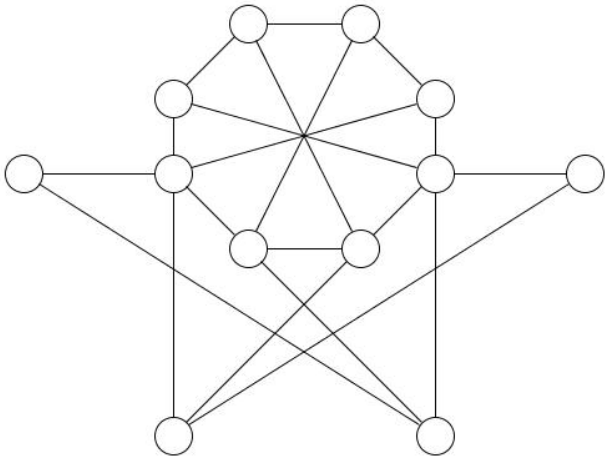


Abbildung 6.11.: Konfliktgraph zu dem Abgeordnetenproblem [Sch94]

aus Kapitel 6.1 kann man dieses Problem durch eine boolesche Funktion ausdrücken, allerdings wird diese dann sehr umfangreich und unübersichtlich. Wir können aber auch naiv vorgehen. Wir wählen eine Farbe aus und beginnen bei einem beliebigen Knoten und färben diesen. Alle Knoten, die nicht mit dem eben gefärbten Knoten verbunden sind, dürfen mit derselben Farbe gefärbt werden. Dann nehmen wir die nächste Farbe und einen beliebigen Knoten der noch übrigen und fahren so fort. In unserem Beispiel können wir so eine Färbung erreichen. Eine mögliche Färbung des Graphen aus Abbildung 6.11 zeigt Abbildung 6.12. Wieder handelt es sich um ein Problem, das durch ein Färbbarkeitsproblem modelliert werden kann. Nach Beispiel 4.8 wissen wir, dass COLOR auf SAT polynomiell reduzierbar ist. Der Satz von Cook (vgl. Satz 4.6) sagt aus, dass es sich bei SAT um ein $\mathcal{NP}$-vollständiges Problem handelt. Zusammen mit Lemma 4.3 folgt, dass auch das Abgeordnetenproblem ein $\mathcal{NP}$-vollständiges Problem ist.
In der Arbeit von Andreas Schwill (vgl. [Sch94]) wird ein naiver Algorithmus zur Lösung des Abgeordnetenproblems diskutiert. Nehmen wir an, wir hätten k Farben zur Verfügung. Diese k Farben können auf k^n verschiedene Arten den n Knoten zugeordnet werden. Wir haben es also leider mit einer gewaltigen Anzahl von Lösungskandidaten zu tun. Nachdem wir die Farben zugeordnet haben, überprüfen wir für jede erhaltene Zuordnung, ob sie die Bedingung erfüllt, dass im Graphen mit einer Kante verbundene Knoten mit unterschiedlichen Farben gefärbt wurden. Wie lange werden wir asymptotisch für diese Überprüfung brauchen? „Hierzu untersuchen wir jedes der $\frac{1}{2} \cdot n(n-1) = \mathcal{O}(n^2)$ möglichen Paare von Knoten und testen ihre beiden Farben, sofern das Paar

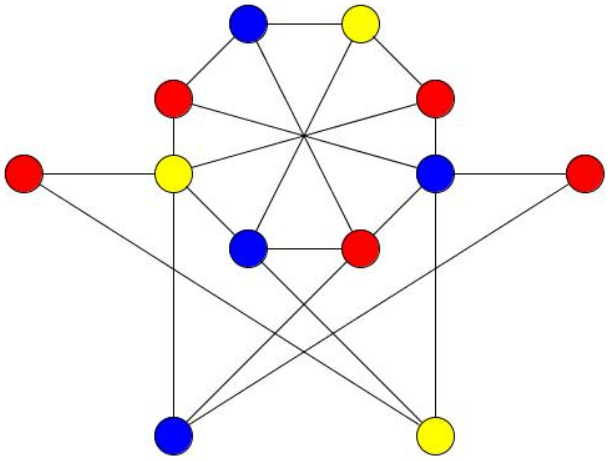


Abbildung 6.12.: Mit drei Farben gefärbter Konfliktgraph zu dem Abgeordnetenproblem [Sch94]

durch eine Kante verbunden ist“ [Sch94]. Die gesamte Laufzeit dieses Algorithmus setzt sich zusammen aus dem Färben der Knoten und der Überprüfung danach. Insgesamt hat der Algorithmus also asymptotisch eine Laufzeit von $\mathcal{O}(n^2 \cdot k^n)$.

6.3. Ausflug in die Wirtschaft - Maschinenbelegungen und Produktionspläne

Lernvoraussetzungen:

- Grundlagen von Graphen: Aufbau, Färbbarkeitsproblem (vgl. Definition 4.8)
- Grundlagen im Aufbau von booleschen Ausdrücken

Stellen wir uns vor, wir leiten eine Produktionsfirma und bekommen von einem Kunden den Auftrag 6 verschiedene Produkte herzustellen. Natürlich wollen wir den Kunden so zufrieden wie nur möglich stellen und machen uns Gedanken über den optimalen Ablauf der Produktion. Für bestimmte Fertigungen stehen uns entsprechende Maschinen zur Verfügung. Leider haben wir von jedem Maschinentyp immer nur eine Maschine. Verschiedene Produkte durchlaufen während ihrer Produktion mehrere Maschinen. Teilweise benötigen die Produkte gleiche Maschinen. Eine Maschine kann immer nur ein bestimmtes Produkt herstellen, so dass jeweils nur Produkte gleichzeitig hergestellt werden können, wenn sie nicht die gleiche Maschine benötigen. Für das Beispiel nehmen wir an, dass der Konfliktgraph wie in Abbildung 6.13 aussieht. Wie im Beispiel aus Kapitel 6.1 entsteht der Graph G=(V, E) wie folgt:

- Die verschiedenen Produkte bilden die Knotenmenge V von G, d.h. $V = \{1, 2, 3, 4, 5, 6\}$.
- Immer dann, wenn bei der Herstellung zweier unterschiedlicher Produkte auf dieselbe Maschine zurückgegriffen werden muss, wird dem Graphen eine Kante hinzugefügt. Die Kantenmege ergibt sich somit aus:

$$E = \{u, v | u \text{ und } v \text{ benutzen die gleiche Maschine}\}$$

In dem vorliegenden Beispiel sollen sechs (Kardinalität von V) verschiedene Produkte hergestellt werden. Dabei kommt es zu |E| vielen Konflikten. Versucht man nun die in dem Graph enthaltene Information in eine Wahr-

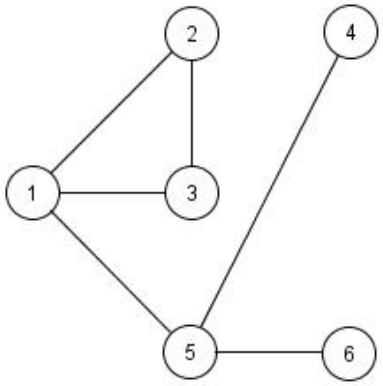


Abbildung 6.13.: Konfliktgraph für sechs verschiedene Produkte

heitstabelle einzuarbeiten, so erhält man eine Tabelle mit 64 (2^6) Zeilen. Dieser Weg ist recht mühselig. Im Unterricht kann dadurch exemplarisch gezeigt werden, wie umfangreich schon ein einfaches Beispiel werden kann. Es geht aber auch eleganter, wenn auch dennoch die Situation ausgiebig betrachtet werden muss. Interpretieren wir unser Problem als ein Färbbarkeitsproblem (COLOR vgl. Definition 4.8). Wie viele Maschinen werden benötigt, wenn alle Produkte zeitgleich hergestellt werden sollen? Oder anders ausgedrückt: Wie viele Farben werden benötigt, um den Graph aus Abbildung 6.13 konfliktfrei zu färben? Um den Umfang des vorgestellten Beispiels zu reduzieren beantworten wir jedoch zuerst folgende Frage: Ist der Graph mit drei Farben konfliktfrei färbbar[14]? Wir wählen hier die Farben Rot, Grün und Blau. Ein denkbares Vorgehen ist es, bei einem beliebigen Knoten zu beginnen und ihn rot zu färben. Alle Nachbarn des eben gefärbten Knotens müssen dann eine andere Farbe erhalten. Färben wir die Nachbarn grün. Nun betrachtet man alle Nachbarn der Nachbarn und so fort. In unserem Beispiel mag das noch praktikabel sein, aber dieses „Ausprobieren“ ist für größere Graphen impraktikabel[15]. Wir wollen anders vorgehen und übersetzen den Graphen in eine boolesche Formel. Dafür

[14] Handelt es sich also um einen 3-färbbaren Graph?

[15] Wie in Kapitel 4.5 festgestellt ist das Färbbarkeitsproblem ab drei Farben ein $\mathcal{NP}$-vollständiges Problem

benötigt man eine Reihe von Variablen, nämlich für jeden Knoten des Graphen drei[16]. Im Folgenden wird die Notation f_i für $f \in R(\text{Rot}), G(\text{Grün}), B(\text{Blau})$ und $i \in \{1, 2, \cdots, 6\}$ benutzt. Wie sollen die Variablen f_i definiert sein?

$$f_i = \begin{cases} 1 \text{ Wenn Knoten i mit Farbe f gefärbt wurde und} \\ 0 \quad \text{sonst} \end{cases}$$

Beispiel 6.3. $R_1 = 1$ bedeutet Knoten 1 ist rot gefärbt.
$G_5 = 0$ bedeutet Knoten 5 ist nicht grün. Mit welcher Farbe der Knoten gefärbt ist, wissen wir (noch) nicht.

Das Einfügen der oben beschriebenen Variablen kann man sich anschaulich als „Aufblähen“ des Graphen vorstellen. Der „aufgeblähte“ Graph ist in Abbildung 6.14 zu sehen. Natürlich soll jeder Knoten am Ende eine Farbe besitzen, dh.

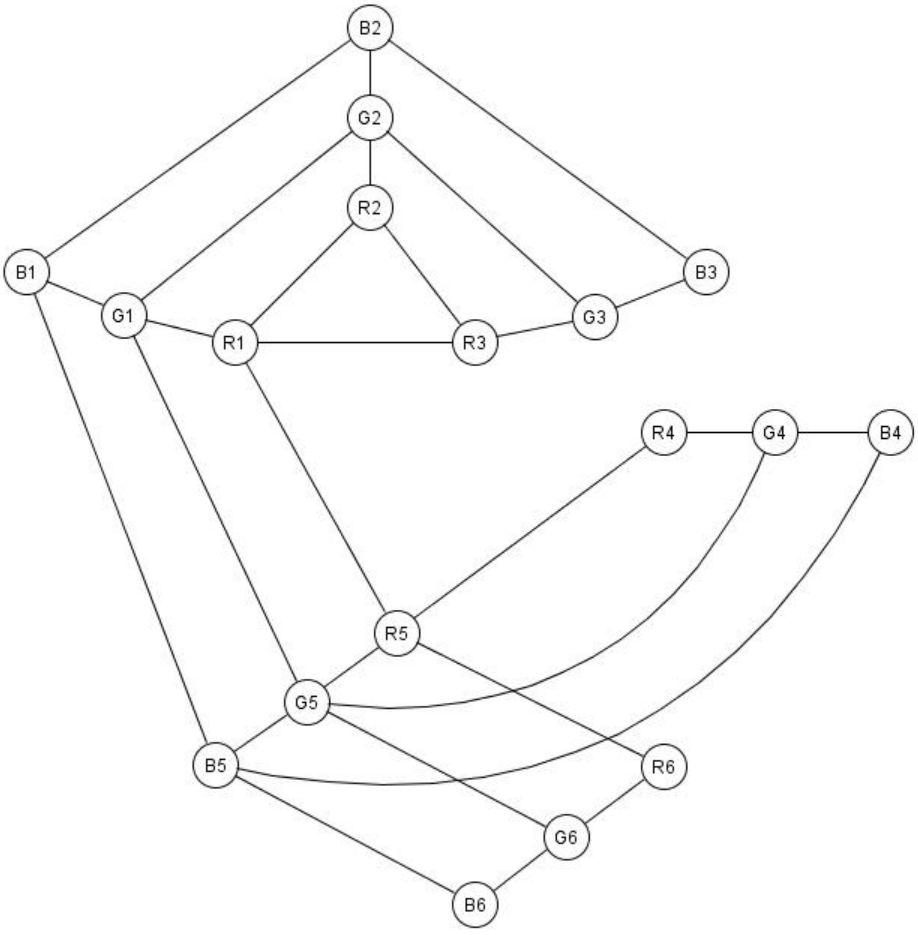


Abbildung 6.14.: „Aufgeblähter“ Graph zu dem Graphen aus Abbildung 6.13

für jeden Knoten i muss mindestens eine der Variablen R_i, G_i oder B_i gleich

[16] eine Variable für jede mögliche Farbe

1 sein. Allerdings darf der Knoten wirklich nur mit einer Farbe gefärbt sein. Das bedeutet für jeden Knoten muss genau eine Variable R_i, G_i oder B_i gleich 1 sein. Wir modellieren den Sachverhalt mithilfe boolescher Ausdrücke:

$$(R_i \vee G_i \vee B_i) \wedge (\bar{R}_i \vee \bar{G}_i) \wedge (\bar{R}_i \vee \bar{B}_i) \wedge (\bar{G}_i \vee \bar{B}_i) \quad \forall\, i \in \{1, 2, \cdots, 6\}$$

Zusätzlich dürfen Knoten, die durch eine Kante verbunden sind, nicht die gleiche Farbe haben. Diese Bedingung gilt sowohl für den ursprünglichen Graphen in Abbildung 6.13 als auch für den „aufgeblähten“ Graphen in Abbildung 6.14. Demnach muss für alle Knotenpaare $\{i, j\} \in E$ gelten, dass $(R_i \neq R_j) \wedge (G_i \neq G_j) \wedge (B_i \neq B_j)$. Auch diese Bedingung kann mithilfe des Konzeptes der booleschen Algebra (vgl. Kapitel 3) umgesetzt werden, indem man sicherstellt, dass jede Farbe nur von höchstens einem der beiden Knoten benutzt wird.

$$(\bar{R}_i \vee \bar{R}_j) \wedge (\bar{G}_i \vee \bar{G}_j) \wedge (\bar{B}_i \vee \bar{B}_j) \quad \forall\, \{i, j\} \in E \text{ mit } i \neq j$$

Setzt man die beiden eben erarbeiteten Bedingungen zusammen, so ergibt sich das gestellte Problem als Satisfiability-Problem wie folgt:

$$\begin{aligned} f = &(R_1 \vee G_1 \vee B_1) \wedge (\bar{R}_1 \vee \bar{G}_1) \wedge (\bar{R}_1 \vee \bar{B}_1) \wedge (\bar{G}_1 \vee \bar{B}_1) \wedge \\ &(R_2 \vee G_2 \vee B_2) \wedge (\bar{R}_2 \vee \bar{G}_2) \wedge (\bar{R}_2 \vee \bar{B}_2) \wedge (\bar{G}_2 \vee \bar{B}_2) \wedge \\ &(R_3 \vee G_3 \vee B_3) \wedge (\bar{R}_3 \vee \bar{G}_3) \wedge (\bar{R}_3 \vee \bar{B}_3) \wedge (\bar{G}_3 \vee \bar{B}_3) \wedge \\ &(R_4 \vee G_4 \vee B_4) \wedge (\bar{R}_4 \vee \bar{G}_4) \wedge (\bar{R}_4 \vee \bar{B}_4) \wedge (\bar{G}_4 \vee \bar{B}_4) \wedge \\ &(R_5 \vee G_5 \vee B_5) \wedge (\bar{R}_5 \vee \bar{G}_5) \wedge (\bar{R}_5 \vee \bar{B}_5) \wedge (\bar{G}_5 \vee \bar{B}_5) \wedge \\ &(R_6 \vee G_6 \vee B_6) \wedge (\bar{R}_6 \vee \bar{G}_6) \wedge (\bar{R}_6 \vee \bar{B}_6) \wedge (\bar{G}_6 \vee \bar{B}_6) \wedge \\ &(\bar{R}_1 \vee \bar{R}_2) \wedge (\bar{G}_1 \vee \bar{G}_2) \wedge (\bar{B}_1 \vee \bar{B}_2) \wedge \\ &(\bar{R}_1 \vee \bar{R}_3) \wedge (\bar{G}_1 \vee \bar{G}_3) \wedge (\bar{B}_1 \vee \bar{B}_3) \wedge \\ &(\bar{R}_1 \vee \bar{R}_5) \wedge (\bar{G}_1 \vee \bar{G}_5) \wedge (\bar{B}_1 \vee \bar{B}_5) \wedge \\ &(\bar{R}_2 \vee \bar{R}_3) \wedge (\bar{G}_2 \vee \bar{G}_3) \wedge (\bar{B}_2 \vee \bar{B}_3) \wedge \\ &(\bar{R}_4 \vee \bar{R}_5) \wedge (\bar{G}_4 \vee \bar{G}_5) \wedge (\bar{B}_4 \vee \bar{B}_5) \wedge \\ &(\bar{R}_5 \vee \bar{R}_6) \wedge (\bar{G}_5 \vee \bar{G}_6) \wedge (\bar{B}_5 \vee \bar{B}_6) \wedge \end{aligned}$$

Finden wir eine erfüllende Belegung für die Formel, wissen wir, dass es eine Möglichkeit gibt, den Graphen mit 3 Farben zu färben. Eine erfüllende Belegung ist zum Beispiel durch

$$\begin{aligned} &R_1 = 1,\ G_1 = 0,\ B_1 = 0 \text{ für Knoten 1} \\ &R_2 = 0,\ G_2 = 1,\ B_2 = 0 \text{ für Knoten 2} \\ &R_3 = 0,\ G_3 = 0,\ B_3 = 1 \text{ für Knoten 3} \\ &R_4 = 1,\ G_4 = 0,\ B_4 = 0 \text{ für Knoten 4} \\ &R_5 = 0,\ G_5 = 1,\ B_5 = 0 \text{ für Knoten 5 und} \\ &R_6 = 1,\ G_6 = 0,\ B_6 = 0 \text{ für Knoten 6} \end{aligned}$$

gegeben.
In Abbildung 6.15 ist der eingefärbte Graph abgebildet. Entsprechend kann

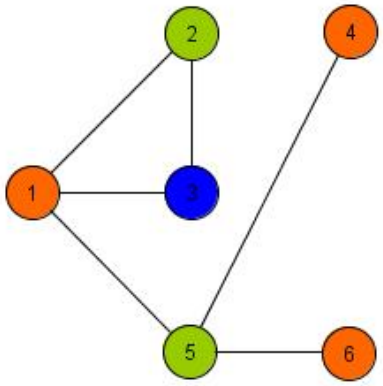


Abbildung 6.15.: Beispiel einer Färbung des Graphen aus Abbildung 6.13

man auch den „aufgeblähten“ Graphen einfärben. Das Ergebnis zeigt Abbildung 6.16. Man benötigt also mindestens drei Maschinen, um alle sechs Produkte ohne Konflikte herstellen zu können. Wären auch zwei Maschinen ausreichend? Ist der Graph 2-färbbar? Dazu betrachten wir den Teilgraphen, der aus den Knoten 1, 2 und 3 besteht. Der Teilgraph ist vollständig, dh. jeder Knoten ist mit jedem Knoten verbunden. An dieser Stelle erkennt man, dass zwei Farben nicht ausreichen können.
Zusammenfassend haben wir das gestellte 3-Färbbarkeits-Problem dadurch gelöst, dass wir es auf ein entsprechendes Satisfiability-Problem reduziert haben. Die verwendete Reduktion ist mit Sicherheit in polynomieller Zeit zu bewältigen, da nur „Hilfsknoten“ (nämlich $3 \cdot |V|$ viele) eingefügt wurden.

6.4. Mobilfunknetze

Lernvoraussetzungen:

- Graphengrundlagen (vgl. Kapitel 2), Aufbau und der Begriff der Adjazenz genügt
- Modellierung von Bedingungen mithilfe der Booleschen Algebra (vgl. Kapitel 3)

In diesem Kapitel wird ein Beispiel aus [Sie12] vorgestellt. Es geht dabei um die Abdeckung von Mobilfunktnetzen. Hier soll ein Gebiet durch 7 Basisstationen abgedeckt werden. In Abbildung 6.17 ist die Situation skizziert dargestellt. Jede Basisstation deckt einen gewissen Bereich ab. Mehrere Funkbereiche können sich überlappen, aber dann dürfen die Basisstationen nicht die gleiche Frequenz besitzen, da es sonst zu Störungen kommen kann. Wieviele verschiedene Frequenzen müssen in dem Gebiet für die Basisstationen zur Verfügung stehen, damit es keine Störungen gibt? Eine Idee, die mir recht intuitiv erscheint, ist folgende Herangehensweise: Man benutzt die Skizze in Abbildung

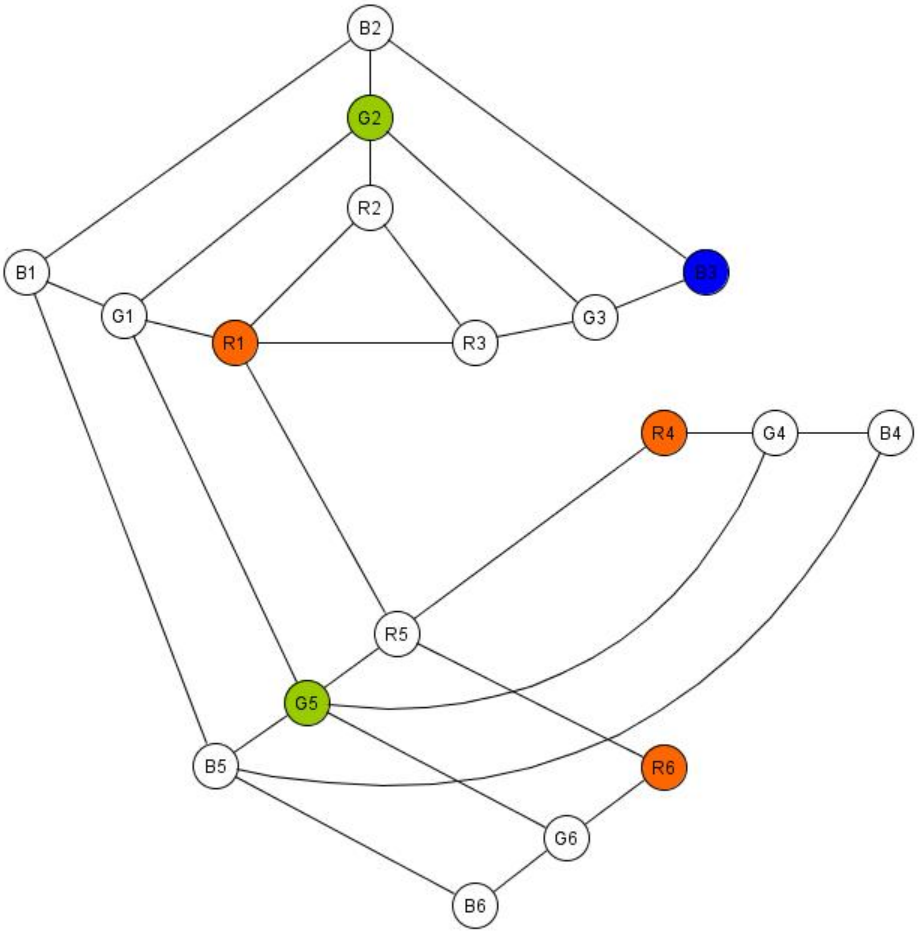


Abbildung 6.16.: Gefärbter Graph zum Graphen aus Abbildung 6.14

6.17 und nimmt einen Stift in die Hand. Nun fängt man bei einem beliebigen Knoten an, die Frequenz festzulegen, sagen wir mit f_1. Alle Basisstationen, deren Frequenzbereiche (gestrichelte Kreise in Abbildung 6.17) sich mit dem Frequenzbereich der eben markierten Basisstation überschneiden, müssen eine andere Frequenz benutzen. Gibt es zwischen den Nachbarn sonst keine weiteren Überschneidungen, können wir allen Nachbarn eine Frequenz f_2 zuordnen. Bestehen allerdings weitere Konflikte, so müssen auch diese beachtet werden. So kann man sich dann von Basistation zu Basisstation „hangeln“ bis allen eine Frequenz zugeteilt wurde.
Etwas systematischer überlegt man sich, wie die Situation in Abbildung 6.17 mathematischer modelliert werden kann. Eine Darstellung als Graph (vgl. Kapitel 2) bietet sich an. Konstruieren wir einen Graphen G=(V, E). Die Knotenmenge V soll alle Basisstationen enthalten, dh. V=$\{1, 2, 3, 4, 5, 6, 7\}$. Die Kantenmenge des Graphen soll alle Überschneidungen der Frequenzbereiche modellieren. Dazu sei die Kantenmenge E wie folgt definiert:

$$E = \{\{u, v\} \mid \text{Frequenzbereiche der Stationen } u \text{ und } v \text{ überschneiden sich}\}$$

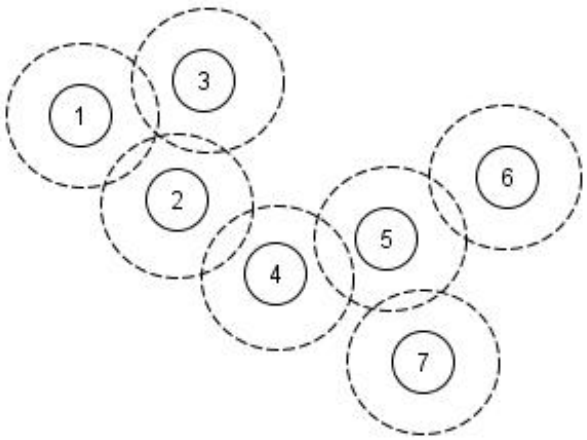


Abbildung 6.17.: Basisstationen des Mobilfunkanbieters

Die Idee hier ist die gleiche wie schon zuvor in Kapitel 6.1, wo Konflikte im Straßenverkehr vermieden wurden. Der so konstruierte Graph ist in Abbildung 6.18 zu sehen. Statt von verschiedenen Frequenzen $f_1, f_2, \cdots$ zu sprechen, können

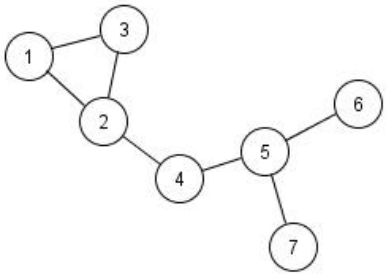


Abbildung 6.18.: Graph zur Situation aus Abbildung 6.17

wir auch Farben verwenden. Erinnern wir uns an Definition 4.8, so ist erkennbar, dass das Problem der Mobilfunknetzmodellierung als Färbbarkeitsproblem interpretiert werden kann. Wie viele Farben (bzw. Frequenzen) werden benötigt? Um die Anschauung zu erhöhen und die Modellierung für den Unterricht zu vereinfachen, versuchen wir zuerst die Problemlösung mit 4 Farben/Frequenzen zu erreichen. Die zugehörige mathematische Problemstellung lautet: Ist der Graph in Abbildung 6.18 4-färbbar? Analog zum Beispiel aus Kapitel 6.3 fügen wir eine Reihe von Variablen ein und „blähen“ den Graphen dadurch auf. Statt für jeden Knoten drei Variablen zu benutzen, benötigen wir hier vier, da wir herausfinden wollen, ob vier Frequenzen ausreichend sind. Wie zuvor benutze ich die Notation f_i für $f \in R(\text{Rot}), G(\text{Grün}), B(\text{Blau}), O(\text{Orange})$ und $i \in \{1, 2, \cdots, 7\}$. Auch die Definition der Variablen kann übernommen

werden:

$$f_i = \begin{cases} 1 \text{ Wenn Knoten i mit Farbe f gefärbt wurde und} \\ 0 \quad \text{sonst} \end{cases}$$

Der so entstandene Graph ist in Abbildung 6.19 zu finden. Jeder Basisstation

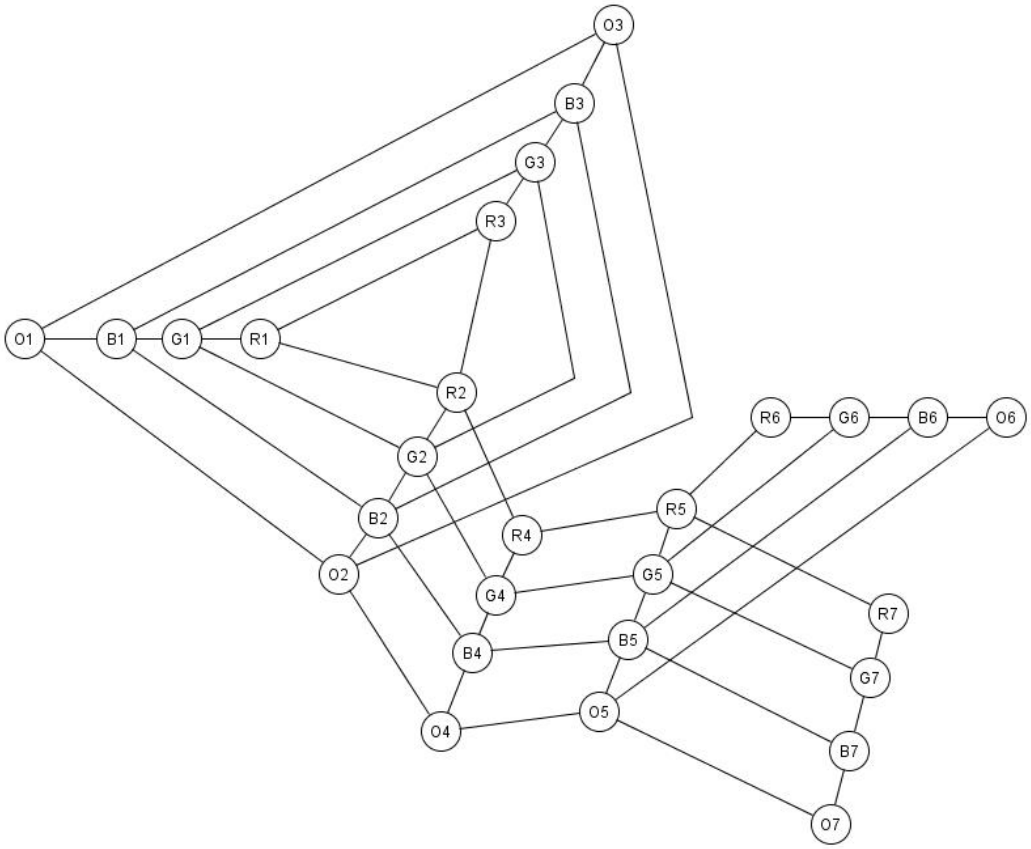


Abbildung 6.19.: „Aufgeblähter“ Graph zu Abbildung 6.18

muss eine Frequenz zur Verfügung stehen, dh. jedem Knoten in Abbildung 6.18 muss mindestens eine Farbe zugeordnet werden. Gehen wir davon aus, dass es auch in Ordnung ist, wenn eine Basisstation theoretisch mehrere Frequenzen nutzen kann (zum Beispiel wäre auch ein Wechsel zwischen Frequenzen denkbar), dann muss nicht genau eine Zuordnung verlangt werden. Es reicht aus, wenn mindestens eine der Variablen R_i, G_i, B_i oder O_i für jeden Knoten i $\in$ V gleich 1 ist. Mit Hilfe der in Kapitel 3 vorgestellten Booleschen Algebra ausgedrückt, erhält man:

$$R_i \vee G_i \vee B_i \vee O_i \quad \forall i \in \{1, 2, \cdots, 7\}$$

Natürlich möchte ein Mobilfunknutzer ohne Störungen zum Beispiel mit seinem Handy telefonieren. Dies kann dadurch umgesetzt werden, indem wir sicherstellen, dass Knoten (Basisstationen), die durch eine Kante verbunden

sind, nicht die gleiche Frequenz (respektive Farbe) verwenden. Wiederum gilt diese Voraussetzung sowohl für den Graphen in Abbildung 6.18, sowie für den „aufgeblähten“ Graphen in Abbildung 6.19. Für alle Knotenpaare $\{i,j\} \in E$ muss gelten, dass $(R_i \neq R_j) \wedge (G_i \neq G_j) \wedge (B_i \neq B_j) \wedge (O_i \neq O_j)$. Die Bedingung kann mithilfe des Konzeptes der booleschen Algebra umgesetzt werden, indem man sicherstellt, dass jede Farbe nur von höchstens einem der beiden Knoten benutzt wird.
$(\bar{R}_i \vee \bar{R}_j) \wedge (\bar{G}_i \vee \bar{G}_j) \wedge (\bar{B}_i \vee \bar{B}_j \wedge (\bar{O}_i \vee \bar{O}_j) \quad \forall \{i,j\} \in E$ mit $i \neq j$
Zu einer Formel zusammengesetzt, erhält man folgende Formel

$$
\begin{aligned}
f = &(R_1 \vee G_1 \vee B_1 \vee O_1) \wedge \\
&(R_2 \vee G_2 \vee B_2 \vee O_2) \wedge \\
&(R_3 \vee G_3 \vee B_3 \vee O_3) \wedge \\
&(R_4 \vee G_4 \vee B_4 \vee O_4) \wedge \\
&(R_4 \vee G_5 \vee B_5 \vee O_5) \wedge \\
&(R_5 \vee G_6 \vee B_6 \vee O_6) \wedge \\
&(R_6 \vee G_7 \vee B_7 \vee O_7) \wedge \\
&(\bar{R}_1 \vee \bar{R}_2) \wedge (\bar{G}_1 \vee \bar{G}_2) \wedge (\bar{B}_1 \vee \bar{B}_2) \wedge (\bar{O}_1 \vee \bar{O}_2) \wedge \\
&(\bar{R}_1 \vee \bar{R}_3) \wedge (\bar{G}_1 \vee \bar{G}_3) \wedge (\bar{B}_1 \vee \bar{B}_3) \wedge (\bar{O}_1 \vee \bar{O}_3) \wedge \\
&(\bar{R}_2 \vee \bar{R}_3) \wedge (\bar{G}_2 \vee \bar{G}_3) \wedge (\bar{B}_2 \vee \bar{B}_3) \wedge (\bar{O}_2 \vee \bar{O}_3) \wedge \\
&(\bar{R}_2 \vee \bar{R}_4) \wedge (\bar{G}_2 \vee \bar{G}_4) \wedge (\bar{B}_2 \vee \bar{B}_4) \wedge (\bar{O}_2 \vee \bar{O}_4) \wedge \\
&(\bar{R}_4 \vee \bar{R}_5) \wedge (\bar{G}_4 \vee \bar{G}_5) \wedge (\bar{B}_4 \vee \bar{B}_5) \wedge (\bar{O}_4 \vee \bar{O}_5) \wedge \\
&(\bar{R}_5 \vee \bar{R}_6) \wedge (\bar{G}_5 \vee \bar{G}_6) \wedge (\bar{B}_5 \vee \bar{B}_6) \wedge (\bar{O}_5 \vee \bar{O}_6) \wedge \\
&(\bar{R}_5 \vee \bar{R}_7) \wedge (\bar{G}_5 \vee \bar{G}_7) \wedge (\bar{B}_5 \vee \bar{B}_7) \wedge (\bar{O}_5 \vee \bar{O}_7)
\end{aligned}
$$

die alle Bedingungen umsetzt.
„Gibt es eine erfüllende Belegung für die obige Formel?“[17] entspricht also der Frage, ob es mit vier verschiedenen Frequenzen möglich ist, Störungen zu vermeiden.
Eine mögliche Färbung ist in Abbildung 6.20 zu sehen, die zugehörige Belegung lautet:

$$
\begin{aligned}
&R_1 = 1,\ G_1 = 0,\ B_1 = 0\ O_1 = 0 \text{ für Knoten 1} \\
&R_2 = 0,\ G_2 = 1,\ B_2 = 0\ O_2 = 0 \text{ für Knoten 2} \\
&R_3 = 0,\ G_3 = 0,\ B_3 = 1\ O_3 = 0 \text{ für Knoten 3} \\
&R_4 = 1,\ G_4 = 0,\ B_4 = 0\ O_4 = 0 \text{ für Knoten 4} \\
&R_5 = 0,\ G_5 = 1,\ B_5 = 0\ O_5 = 0 \text{ für Knoten 5} \\
&R_6 = 1,\ G_6 = 0,\ B_6 = 0\ O_6 = 0 \text{ für Knoten 6 und} \\
&R_5 = 1,\ G_5 = 1,\ B_5 = 0\ O_7 = 0 \text{ für Knoten 7}
\end{aligned}
$$

Betrachtet man die gefundene Belegung, so fällt auf, dass alle zur vierten Farbe

[17] sprich: Ist die Formel erfüllbar? (vgl. SAT-Problem in Kapitel 4.6)

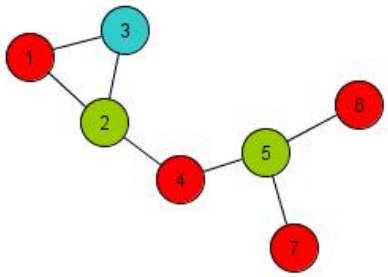


Abbildung 6.20.: Beispiel einer Färbung des Graphen aus Abbildung 6.18

korrespondierenden Variablen 0 sind. Die vierte Farbe „Orange“ wird demnach gar nicht benötigt. Drei Farben bzw. Frequenzen sind völlig ausreichend. Kann man das Problem auch mit nur zwei Frequenzen lösen? Ein schneller erneuter Blick auf Abbildung 6.18 hilft uns, diese Möglichkeit direkt zu verwerfen. Betrachten wir den Teilgraphen, der nur Knoten 1, 2 und 3 enthält. Das „Dreieck“ ist ein vollständiger Graph mit drei Knoten. Zwei Farben können nicht ausreichend sein.

> „Rückblickend auf die Versteigerung der UMTS-Mobilfunklizenzen im Jahre 2000 ist einzusehen, dass das Nutzungsrecht an möglichst vielen Frequenzen einen sehr hohen Wert besitzt, da man in gewissem Umfang die Errichtung von Sendetürmen einsparen sowie eine größere Zahl von Kunden versorgen kann.“ [SS04]

6.5. „Der Rubik's Cube des 21sten Jahrhunderts“ - Sudoku

Lernvoraussetzungen:

- Aufbau eines ganzzahligen linearen Optimierungsproblems
- Grundlagen der Graphentheorie (vgl. Kapitel 2)
- Grundlagen der Booleschen Algebra, vorallem die Entwicklung von booleschen Ausdrücken aus Aussagen / Bedingungen, konjunktive Normalform

Ein Sudoku ist ein logisches Puzzle, das erstmals in den USA im Jahre 1979 erschien[18]. Der Architekt Howard Garns beschäftigte sich, nachdem er in Ren-

[18] Schon der berühmte Mathematiker Leonhard Euler (geb. 15. April 1707 in Basel, verstorben 7. September 1783 in Sankt Petersburg) beschäftigte sich intensiv mit lateinischen Quadraten. Bei einem lateinischen Quadrat handelt es sich um „ein quadratisches Schema mit n Reihen und n Spalten, wobei jedes Feld mit einem von n verschiedenen Symbolen belegt ist, so dass jedes Symbol in jeder Zeile und in jeder Spalte jeweils genau einmal auftritt“ [Wik12e]. Fügt man einem lateinischen Quadrat die Zusatzbedingung, dass in den neun 3×3-Quadrate in jedem dieser Quadrate alle Symbole jeweils genau einmal auftreten müssen, hinzu, erhält man ein Sudoku.

te gegangen war, mit dem Entwickeln von Puzzeln und Rätseln und entwarf dabei das heute populäre Sudoku. Die erste große Aufmerksamkeit genoss das Puzzlespiel in den 1980er Jahren in Japan. Die japanische Sprache prägte auch den Namen „Sudoku“, der ein Kürzel des japanischen Satzes „suji wa dokushin ni kagiru“, der sich mit „the digits must remain single“ ins Englische übersetzen lässt, darstellt. 1997 entdeckte dann Wayne Gould, ein 1945 in Neuseeland geborener Unternehmer, dass man mit dieser Idee viel Geld machen kann. Er versuchte einen Algorithmus zu entwickeln, der schnell Sudokus mit verschiedenen Schwierigkeitsgraden herstellt. Zuvor ist das gleiche Puzzlespiel erstmals 1979 unter dem Namen „NumberPlace“ in einer Rätselzeitschrift veröffentlicht worden [Wik12g]. Nach etwa 6 Jahren krönte Howard Garns der Erfolg. Seine

	8		2		6			
1	6							
6	1				4			
	9					3		
						7		5
							1	
				7			8	
		7		3				

Abbildung 6.21.: Beispiel eines Sudoku, welches mit der derzeit kleinsten bekannten Anzahl von 17 vorgegebenen Zahlen auskommt. [VK12]

Firma Pappocom begann den Verkauf von Sudokus an zahlreiche Publikationsabnehmer. 2005 breitete sich das Sudoku-Fieber dann auch in den USA und vielen anderen Ländern aus. [ACB12]
Heutzutage ist es eine fast schon $\mathcal{NP}$-schwere Aufgabe Sudokus zu meiden. Man findet sie in allen erdenklichen Zeitschriften und Zeitungen abgedruckt. Seit Ende 2005 gibt es auch tragbare elektronische Sudoku-Geräte. Man kann einfache Sudoku-Brettspiele kaufen oder den Rätselspass im Internet online oder als Computerspiel genießen.
Obwohl der Bekanntheitsgrad von Sudokus so groß ist, möchte ich im Folgenden ihren Aufbau kurz erläutern, um dann dahin überzugehen, diesen mathematisch zu beschreiben.
Meistens handelt es sich bei einem Sudoku um ein 9×9 Quadrat. Die Regeln das Sudoku auszufüllen sind sehr einfach: Der Spieler soll das Quadrat so ausfüllen, dass in jeder Zeile und Spalte, sowie in jedem 3×3 Teilquadrat die Zahlen 1 bis 9 genau einmal enthalten sind. In jedem Puzzle sind verschiedene „feste“ Zahlen schon vorgegeben, deren Anzahl und Platz die Schwierigkeit des

Puzzles bestimmt. Neben der Form eines 9×9 Quadrates gibt es noch zahlreiche andere Versionen, bei denen jeweils die Größe variiert. Auch der Faktor der Größe des zu lösenden Sudoku beeinflusst dessen Schwierigkeitsgrad. Ein Sudoku der Größe 25×25 wird, da das Lösen in diesem Fall viel anspruchsvoller ist, manchmal auch „Samurai Sudoku“ genannt. [ACB12]

Wie können Sudokus mathematisch exakt beschrieben werden?

> „Mathematisch kann man Sudokus als lineare diophantische Gleichungssysteme[19] mit Nichtnegativitätsbedingungen formulieren. Solche ganzzahligen linearen Programme sind die wichtigsten Modellierungswerkzeuge in zahlreichen Anwendungsgebieten wie z.B. der Optimierung von Telekommunikations- und Verkehrsnetzen.“ [VK12]

Das Spannende daran ist, das eben diese sogenannten ganzzahligen linearen Programme nach Karp zu den $\mathcal{NP}$-vollständigen Problemen gehören. [Wik12b]

Es soll nun die Formulierung eines Sudoku als 0-1-ganzzahliges Programm erarbeitet werden. Anschließend präsentiere ich eine Möglichkeit das Lösen von Sudokus als SAT-Problem zu formulieren. Gleichzeitig, sozusagen durch den Satz von Cook (vgl. Kapitel 4.6) geschenkt, erhält man die $\mathcal{NP}$-Vollständigkeit.

Zuerst überlegt man sich, intuitiv sollte das klar sein, dass das Sudoku-Problem in der Klasse $\mathcal{NP}$ liegt. Für ein 9×9-Sudoku „rät“ man die Belegungen von 729 (errechnet aus 9^3 für die Kombinationsmöglichkeiten) Variablen und überprüft in polynomieller Zeit, dass tatsächlich in jeder Spalte, Zeile und in jedem Teilquadrat jede der neun möglichen Zahlen von 1 bis 9 genau einmal vorkommt.

In [ACB12] ist eine Darstellung als ganzzahliges lineares Programm für Sudokus allgemeiner Größe, also der Größe $n \times n$ zu finden. Die grundlegenden Ideen bei den mathematischen Formulierung zu unterschiedlichen Größen sind immer dieselben. Im vorliegenden Kapitel möchte ich mich auf die am häufigsten auftretende Größe 9×9 und somit auch Schülern und Schülerinnen bekannteste Version beschränken.

Wir benötigen eine Variable, die angibt, ob eine bestimmte Zahl $k \in 1, ..., 8, 9$ (im Folgenden mit [9] abgekürzt) auftritt. Wir definieren dafür die Variable x_{ijk} wie folgt

[19] Eine diophantische Gleichung, benannt nach Diophantos von Alexandria (um 250 v.Chr.), ist eine Polynomfunktion in x, y, z, $\cdots$ bei der als Lösungen nur ganze Zahlen erlaubt sind:

$$ax^{n_1} \cdot y^{m_1} \cdot z^{p_1} + ax^{n_2} \cdot y^{m_2} \cdot z^{p_2} + \cdots = d$$

$$\text{mit nicht-negativen} \quad n_i, m_i, p_i \in \mathbb{N} \quad \text{und ganzzahligen Koeffizienten} \quad a, b, d \in \mathbb{Z}.$$

Eine lineare diophantische Gleichung enthält in jedem Term nur eine der Variablen in der ersten (linearen) Potenz: $ax + by + cz + \cdots = d$ [Kon12]

$$x_{ijk} := \begin{cases} 1 & \text{wenn in Feld (i, j) die Zahl k steht} \\ 0 & \text{sonst} \end{cases}$$

Im 9×9-Fall definieren wir also 729 (9^3) Variablen. Würden wir ein Sudoku der Größe 25×25 betrachten, so benötigt man schon 15625 Variablen! Volker Kaibel und Thorsten Koch veranschaulichen in ihrer Arbeit [VK12] das Lösen des Sudoku aus Abbildung 6.21 als Würfel. Diese Darstellung ist in Abbildung 6.22 zu finden. Die Autoren repräsentieren die verschiedenen Zahlen von 1 bis 9 mit Farben. Ein Würfel markiert eine durch das Sudoku festgelegte, schon gegebene Zahl. Bei den Punkten handelt es sich um „errätselte“ Positionen der einzelnen, zu Beginn noch zu ergänzenden, Zahlen. Sowohl die Farbe als auch die Höhe jeder Kugel und jedes Würfels repräsentiert einen Eintrag im darunter liegenden Feld.

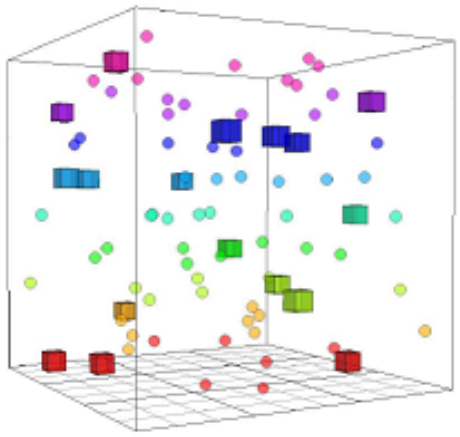


Abbildung 6.22.: Lösung des Sudokus aus Abbildung 6.21

Wir müssen nun noch die drei Regeln des Sudokus umsetzen. Die erste Regel fordert, dass die Zahl k in jeder Zeile genau einmal vorkommt. Es muss also

$$\sum_{j=1}^{9} x_{ijk} = 1, \quad \text{für jede Zeile} \quad i \in [9] \tag{6.1}$$

gelten.
Analog darf jede Zahl k in jeder Spalte nur einmal auftreten

$$\sum_{i=1}^{9} x_{ijk} = 1, \quad \text{für jede Spalte} \quad j \in [9] \tag{6.2}$$

Bleibt die dritte Regel eines Sudokus umzusetzen. Jede Zahl k darf in jedem 3×3-Quadrat nur genau einmal vorkommen.

$$\sum_{(i,j) \in Q(a,b)} x_{ijk} = 1 \tag{6.3}$$

$$\begin{aligned} &\text{mit} \quad Q(a,b) = (i,j) : 3(a-1) < i \leq 3a, 3(b-1) < j \leq 3b \\ &\text{und} \quad a, b \in 1, 2, 3 \end{aligned}$$

> „Ist nun $S \subseteq [9] \times [9]$ die Menge der Felder, deren Einträge vorgegeben sind, und ist $k(i,j) \in [9]$ für $(i,j) \in S$ die im Feld (i, j) vorgegebene Zahl, so ist die Lösung des gegebenen Sudokurätsels also die (eindeutige) nicht-negative ganzzahlige Lösung des um die Gleichungen $x_{ijk(i,j)} = 1$ (für alle $(i,j) \in S$) erweiterten Systems.“ [VK12]

Zusammenfassend können wir also das Lösen eines Sudokus mithilfe der sogenannten ganzzahligen linearen Programmierung modellieren. In der Regel werden mit der ganzzahligen linearen Programmierung Optimierungsprobleme, bei denen es darum geht eine sogenannte Zielfunktion zu minimieren oder zu maximieren, formalisiert und gelöst. Im Fall des Sudokus ist uns keine solche Zielfunktion gegeben. Der Vollständigkeit halber sei hier erwähnt, dass man als Zielfunktion

$$\max \ / \ \min \qquad 0^T x \tag{6.4}$$

wählen kann, so dass wir eine Funktion maximieren (minimieren), deren Wert konstant gleich 0 ist. Im Endeffekt wählen wir also die Zielfunktion so, dass sie wie gewünscht keine Rolle spielt.

Es ist uns also eine mathematische Formulierung des Problems, ein Sudoku der Größe 9×9 zu lösen, gelungen. Wir haben festgestellt, dass es sich dabei um ein $\mathcal{NP}$-vollständiges Problem (vgl. [Wik12b]) handelt. Wir wollen nun diesen Sachverhalt mithilfe der in Kapitel 4.5 vorgestellten Methode der polynomiellen Reduktion beweisen, indem wir das Sudoku-Problem auf KNF-SAT reduzieren.
Wie bereits in Kapitel 4.6 festgehalten, besteht das SAT-Problem aus n Variablen $x_1, x_2, \cdots, x_n$, welche den Wert 0 („false“) oder 1 („true“) annehmen können. Ein Literal l ist entweder die Variable x_i oder ihre Negation $\overline{x_i}$. Eine Klausel besteht aus Disjunktionen von Literalen und eine KNF-Formel ϕ ist eine Konjunktion von Klauseln.
Ein Sudoku-Puzzle lässt sich, wenngleich man dafür eine beträchtliche Zahl an Variablen benötigt, leicht als SAT Problem repräsentieren. Wie die Autoren das in [IL12] umsetzen möchte ich im Folgenden erläutern.

Wie zuvor definieren wir Variablen x_{ijk}. Diesen Variablen soll der Wert 1 genau dann, wenn in der Zelle in Zeile j und Spalte i die Zahl k steht, zugewiesen werden und 0 sonst. Wir definieren also

$$x_{ijk} := \begin{cases} 1 & \text{wenn in Feld (i, j) die Zahl k steht} \\ 0 & \text{sonst} \end{cases}$$

Ist also beispielsweise $x_{845} = 1$ so bedeutet das, dass in der Zelle (8,4) die Zahl 5 steht.
Zuerst stellen wir sicher, dass bei einem gelösten Sudoku jedes Feld ausgefüllt ist

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{9} x_{ijk} \tag{6.5}$$

Versuchen wir nun die Regeln analog zu denen des 0-1-ganzzahligen linearen Programmes umzusetzen.
Jede Zahl kommt in jeder Zeile höchstens einmal vor

$$\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{9}\bigwedge_{i=1}^{8}\bigwedge_{l=i+1}^{9} (\neg x_{ijk} \vee \neg x_{ljk}) \tag{6.6}$$

Sowie höchstens einmal in jeder Spalte

$$\bigwedge_{i=1}^{9}\bigwedge_{k=1}^{9}\bigwedge_{j=1}^{8}\bigwedge_{l=j+1}^{9} (\neg x_{ijk} \vee \neg x_{ilk}) \tag{6.7}$$

Und jede Zahl soll in jedem 3×3-Quadrat auch nur einmal auftreten

$$\bigwedge_{k=1}^{9}\bigwedge_{l=0}^{2}\bigwedge_{m=0}^{2}\bigwedge_{i=1}^{3}\bigwedge_{j=1}^{3}\bigwedge_{n=j+1}^{3} (\neg x_{(3l+i)(3m+j)k} \vee \neg x_{(3l+i)(3m+n)k}) \tag{6.8}$$

$$\bigwedge_{k=1}^{9}\bigwedge_{l=0}^{2}\bigwedge_{m=0}^{2}\bigwedge_{i=1}^{3}\bigwedge_{j=1}^{3}\bigwedge_{n=j+1}^{3}\bigwedge_{o=1}^{3} (\neg x_{(3l+i)(3m+j)k} \vee \neg x_{(3l+n)(3m+o)k}) \tag{6.9}$$

Da es für jede Zelle genau eine passende Zahl geben soll, damit die Sudokuregeln erfüllt sind, benötigen wir noch weitere Bedingungen.
Es gibt also für jede Zelle höchstens eine Zahl

$$\bigwedge_{i=1}^{9}\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{8}\bigwedge_{l=k+1}^{9} (\neg x_{ijk} \vee \neg x_{ijl}) \tag{6.10}$$

Jede Zahl kommt in jeder Zeile mindestens einmal vor

$$\bigwedge_{j=1}^{9}\bigwedge_{k=1}^{9}\bigvee_{i=1}^{9} x_{ijk} \tag{6.11}$$

Analog in mindestens jeder Spalte

$$\bigwedge_{i=1}^{9}\bigwedge_{k=1}^{9}\bigvee_{j=1}^{9} x_{ijk} \tag{6.12}$$

Genauso kommt jede Zahl mindestens einmal in jedem 3×3-Teilquadrat vor

$$\bigwedge_{k=1}^{9}\bigwedge_{l=0}^{2}\bigwedge_{m=0}^{2}\bigwedge_{i=1}^{3}\bigwedge_{j=1}^{3} x_{(3l+i)(3m+j)k} \tag{6.13}$$

Die Anforderung, dass für jede Zelle eine Zahl eindeutig bestimmt ist, wird also durch die Kombination von „höchtens“ und „mindestens“ Klauseln gesichert. Laut dem Paper [IL12] hat diese Darstellung des Sudoku-Problems als SAT-Problem letztendlich 11988 Klauseln ohne die Klauseln, die noch aufgrund der vorgegebenen Zahlen des Sudoku hinzukommen. Da nach dem Satz von Cook (Vgl. Kapitel 4.6) das SAT-Problem $\mathcal{NP}$-vollständig ist, folgt nach Lemma 4.3 die NP-Vollständigkeit des Sudoku-Problems.

Bevor ich die Betrachtung des Sudoku-Problems abschließe, möchte ich eine weitere mögliche Interpretation des Problems vorstellen. In Kapitel 4.5 haben wir gesehen, dass COLOR auf SAT reduzierbar ist. Ist es auch möglich das Sudoku-Problem als Färbbarkeitsproblem auszudrücken? Dazu betrachten wir erneut Abbildung 6.22. Schaut man von oben auf den Würfel, so kann man erkennen, dass das Lösen eines Sudoku auch als Graphfärbungsproblem (vgl. COLOR in Kapitel 4.8) aufgefasst werden kann. „Ist G der Graph mit Knotenmenge $[9] \times [9]$, in dem zwei Knoten v,w $\in [9] \times [9]$ genau dann mit einer Kante verbunden sind, wenn die Felder v und w in der gleichen Zeile oder Spalte oder im gleichen 3×3-Quadrat liegen, so ist die Aufgabe, eine Färbung der Knoten mit neun Farben zu finden (wobei einige Knoten schon vorgefärbt sind), bei der keine adjazenten (vgl. Definition 2.2) Knoten gleich gefärbt werden“ [VK12]. In Abbildung 6.24 ist ein so konstruierter Graph der Größe 4×4[20] abgebildet.

In dieser Arbeit liegt der Schwerpunkt auf dem Beweis der NP-Vollständigkeit durch die Technik der polynomiellen Reduktion. Lesern, die sich für das Lösen des Sudoku-Problems weitergehend interessieren, empfehle ich die Arbeiten [IL12] und [ACB12] sowie [Sim05].

[20] Hier wurde eine 4×4 zugrunde gelegt, da ein Graph eines 9×9-Sudoku zu unübersichtlich ist.

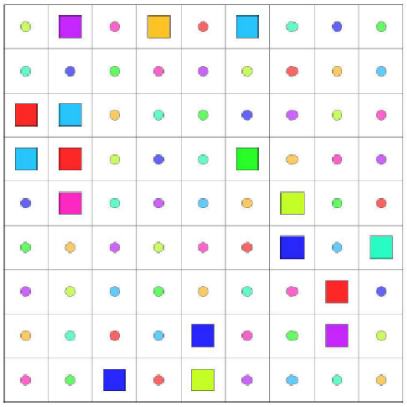


Abbildung 6.23.: Sudoku-Würfel aus Abb. 6.22: Blick von Oben [VK12]

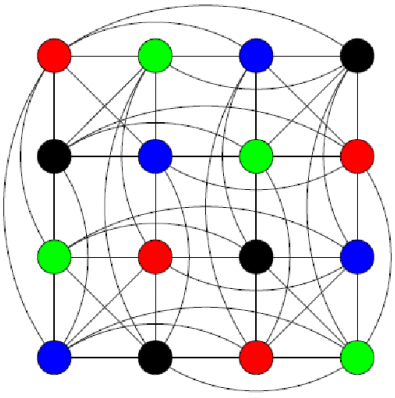


Abbildung 6.24.: Gefärbter Graph zum 4×4-Sudoku [VK12]

7. Vorschläge zur Umsetzung im Unterricht

In Kapitel 6 wurden viele Beispiele zur Veranschauung der polynomiellen Reduktion vorgeschlagen. Auch der Satz von Cook sollte in jedem der voliegenden Beipiele mit seiner Auswirkung bewusst werden. Wie aber können die Beispiele nun im Unterricht sinnvoll eingesetzt werden? In den Beispielen wurde versucht mit so wenig Voraussetzungen und Formalismus wie möglich auszukommen. Dennoch ist manches an Voraussetzung unumgänglich. Bevor genauer darauf eingegangen wird, wie die Beispiele Einzug in den Unterricht erhalten, werden zuerst eine Reihe von Vorschlägen zur Erarbeitung der notwendigen Grundlagen gemacht.
Die Dissertation von Rainer Eidmann mit dem Titel „Algorithmische Graphentheorie im Unterricht unter Verwendung objektorientierter Datenstrukturen“ (vgl. [Eid]) beschäftigt sich mit der Behandlung der Graphentheorie im Zusammenhang mit der objektorientierten Programmierung. Nach Eidmann bietet es sich „[...] hier an, den Aufbau eines Graphen als Programmierungsprojekt im Informatikunterricht als ein Musterbeispiel für die Verwendung und Handhabung von Objektdatentypen und als didaktisch-methodisches Konzept eines Einführungskurses in die objektorientierte Programmierung zu wählen“ [Eid]. In seiner Arbeit werden viele Einstiegsbeispiele genau erläutert und didaktisch und methodisch analysiert. Auch die Arbeit von Thomas Wassong „Graphentheoretische Konzepte in der Gymnasialen Oberstufe“ (vgl. [Was12]) kann ich sehr empfehlen. Hier können Ideen und Umsetzungen zum Unterricht der Graphentheorie basierend auf der Nutzung von neuen Medien nachgelesen werden. Auch zur Thematik der Booleschen Algebra gibt es bestimmt zahlreiche gute Arbeiten, die ich an dieser Stelle empfehlen könnte. Allerdings bietet das in Kapitel 6.1 benutzte und in Kapitel A.1 vorgestellte Programm „Infotraffic“ von Ruedi Arnold meiner Meinung nach einen wirklich gelungenen Einstieg in die Boolesche Algebra. Benutzt man „Infotraffic“ als Einstieg in die Boolesche Algebra, so hat man zudem den Vorteil, dass man in einem fließenden Übergang zu den Beispielen der sicheren Ampelschaltungen übergehen kann. Wichtig ist hier zu beachten, dass es nicht notwendig ist, alle Rechenregeln und Gesetze formal zu lehren. Für das Lösen der Probleme in Kapitel 6 ist es völlig ausreichend, wenn die Schülerinnen und Schüler die Fähigkeit zur Modellierung von Sachverhalten und Bedingungen durch die Boolesche Algebra haben. Da der Satz von Cook eine Aussage über das allgemeine Erfüllbarkeitsproblem

macht, ist strenggenommen noch nicht einmal die Überführung in eine konjunktive Normalform notwendig. Der Schwerpunkt liegt auf der Modellierung der Beispiele. Natürlich schadet zusätzliches Vorwissen nicht. Im Hinblick der Graphentheorie gilt das Gleiche. Es genügt Graphen als Werkzeug der Modellbildung zu betrachten und zu diesem Zweck mit Ihnen umgehen zu können. Die angesprochenen Probleme können dann immernoch an kleinen Beispielen verdeutlicht werden. Die meisten Probleme sind intuitiv gut lösbar. Jede Schülerin und auch jeder Schüler wird verstehen was zu tun ist, wenn man sie/ihn auffordert Knoten eines Graphen so zu färben, dass miteinander verbundene Knoten nicht die gleiche Farbe haben, auch ohne alle möglichen Varianten von Graphfärbungen[1] und Dige wie den chromatischen Index zu kennen. Ich denke, dass gerade darin die Stärke der Auswahl der Beispiele liegt. Sie sind intuitiv gut lösbar und das obwohl es sich „eigentlich“ um $\mathcal{NP}$-vollständige Probleme handelt. Leider kann das gleichzeitig auch ein Hindernis für die Erkenntnis sein, dass es sich um sehr schwere Probleme handelt. Die Schülerinnen und Schüler werden sich fragen, warum es denn so leicht ist, einfach eine Knotenfärbung vorzunehmen und dennoch gelehrt wird, dass es sich um die schwierigsten bekannten Probleme handelt und diese nicht effizint lösbar sind. Hier muss die Lehrkraft immer wieder darauf hinweisen, wodurch die Schwierigkeit der Probleme hervorgehoben wird. Ein Computer hat keine Intuition, er kann eben noch nicht Raten[2], und die sehr große Menge an Lösungskanditaten, die alle Probleme aufweisen, macht es dadurch schwer, die Probleme zu lösen.
Zu den bisher erwähnten Lernvoraussetzungen benötigen Schüler und Schülerinnen Verständnis dafür, wann eine Laufzeit polynomiell oder nicht polynomiell ist. Eine ausgezeichnete didaktische Arbeit zu diesem Zusammenhang wurde von Markus Steinert erstellt. In seinem Artikel „Grenzen der Berechenbarkeit“ (vgl. [Ste11]) wird ein Weg erarbeitet wie Inhalte der Komplexitätstheorie handlungsorientiert erfahren werden können. Im ersten Teil seiner Unterrichtssequenz werden Laufzeiten von Algorithmen ausführlich behandelt.

> „In diesem Teil machen sich Schülerinnen und Schüler mit den wichtigsten Fragen und Lösungsansätzen zum Laufzeitverhalten von Algorithmen vertraut. Zentrales Lernziel ist es dabei, ihnen zu vermitteln, wie sie polynomiales und exponentielles Laufzeitverhalten experimentell verifizieren oder falsifizieren können.“ [Ste11]

Auch zu der Vermittlung von Turingmaschinen lassen sich einige Unterrichtsentwürfe finden. Nachdem der grundlegende Aufbau und die Idee von Turing dahinter vermittelt wurde, empfiehlt es sich mit Simulationen zu arbeiten. Dadurch können Schülerinnen und Schüler ihre Erkenntnisse testen und weiter vertiefen. Eine mögliche Simulationsumgebung und weitere Erklärungen sind auf der Homepage „Matheprisma“ (vgl. [Mat12]) zu finden. Weitere Unterstützung zur Planung einer Unterrichtseinheit kann man dem Artikel von „Der pri-

[1] Zum Beispiel: Knotenfärbungen, Kantenfärbungen, Listenfärbungen,
[2] Zumindest nicht, wenn man davon ausgeht, dass $\mathcal{NP} \neq \mathcal{P}$ gilt.

mitivste Computer der Welt“ (vgl. [Rap87a] und [Rap87b]) entnehmen. Auch hier wird eine Simulationsumgebung für Turingmaschinen verwendet. „[...] Die Schüler haben [dadurch] die Möglichkeit, auch bei diesem von Natur aus eher theoretischen Thema aktiv am Gerät zu arbeiten; vorallem aber ergibt sich ein beträchtlicher Gewinn an Anschaulichkeit“ [Rap87a].

Nach dem nun viele Tipps zur Einführung der Themen Graphentheorie, Boolesche Algebra, Laufzeitverhalten und Turingmaschine erfolgten, wird nun der konkrete Umgang mit den Beispielen dieser Arbeit diskutiert werden. Bevor mit der Behandlung der Beispiele begonnen wird, sollten die Strukturen der Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ schon verstanden sein. Zusätzlich sollte auch

Abbildung 7.1.: Wie kann das sein? Wenn man etwas kaum glauben möchte...dann betrachtet man es genauer.

der Begriff der schwierigsten Probleme in $\mathcal{NP}$, die $\mathcal{NP}$-vollständigen Probleme, schon thematisiert worden sein. Hat man auch den Satz von Cook und dessen Auswirkungen schon bearbeitet, kann es los gehen. Das Motto ist nun, durch die polynomielle Reduktion weitere $\mathcal{NP}$-vollständige Probleme zu finden. Auch soll durch die Beispiele deutlich werden, dass selbst Probleme, die man im ersten Augenblick sicher als effizient lösbar betrachten würde, richtig schwer sein können. Es gilt Neugier durch Verblüffung zu erwecken. Auch das Beweisbedürfnis bei Schülerinnen und Schülern kann durch das „Kaumglauben-wollen“ gestärkt oder erst hervorgebracht werden.
Wie schon in Kapitel 6 gesagt, ist es nicht notwendig die Beispiele der Reihe nach abzuarbeiten. Ich empfehle mit dem ersten Beispiel „Modellierung von „sicheren“ Ampelschaltungen“ (vgl. Kapitel 6.1) zu beginnen. So ist es möglich gemeinsam den Begriff der Färbbarkeit im Zusammenhang mit dem Erfüllbarkeitsproblem an einem einfachen Beispiel zu erarbeiten. Durch die Unterstützung durch „Infotraffic“ (vgl. Kapitel A.1) sollte auch die Hürde des Aufstellens von booleschen Ausdrücken zur Modellierung von Bedingungen leicht gelingen. Es empfiehlt sich ein komplettes Beispiel unterstützend zu erarbeiten. So können die Schülerinnen und Schüler auf die Zusammenhänge leichter aufmerksam gemacht werden. Einmal gesehen ist das Konzept der polynomiellen Reduktion

leicht übertragbar. Den Schülerinnen und Schülern stehen zu diesem Zeitpunkt noch nicht viele $\mathcal{NP}$-vollständige Probleme zur Reduktion zur Verfügung, so dass zu erwarten ist, dass sie bei einem anschließenden Beispiel selbst versuchen, Färbbarkeit mithilfe der Erfüllbarkeit von booleschen Funktionen zu reduzieren. Bevor die Beispiele zu Clique und Vertex Cover (vgl. Kapitel 6.2) betrachtet werden können, müssen den Schülerinnen und Schülern die Probleme erläutert werden. Dazu stehen unter anderem die Beispiele 6.1 und 6.2 zur Verfügung. Weitere Beispiele lassen sich leicht konstruieren.
Neben der Unterstüzung bei der Modellierung ist es sehr wichtig, dass die Zusammenhänge der Beispiele zu der $\mathcal{NP}$-Vollständigkeitstheorie deutlich hervorgehoben werden. Warum reicht es aus das Problem als boolesche Formel auszudrücken, um zu sagen, dass es $\mathcal{NP}$-vollständig sein muss? Woran sieht man, dass ein einfaches Ausprobieren im allgemeinen Fall nicht zum Ziel führt, wenn es doch bei den Beispielen (und das schnell) klappt? Der Irrglaube daran, dass die Probleme leicht zu lösen sind, darf durch die Verwendung der Beispiele nicht noch gestärkt werden. Die Lehrkraft sollte daher immer anstreben, die Menge der Lösungskanditaten abschätzen zu lassen, um so zu zeigen, dass dieses „Ausprobieren“ für einen Computer ohne Intuition und Ratefähigkeit nicht praktikabel ist. Hier lohnt es sich auch exemplarisch einen Algorithmus zu einem „größeren“ Graphen vorzuführen. Dafür stehen verschiedene Applets und auch Videos im Internet zur Verfügung.
Dadurch, dass es sich um ein recht umfangreiches Thema handelt, wäre es am schönsten, die Erarbeitung im Rahmen eines Projektes anzustreben. Auch ist ein Arbeiten an verschiedenen Stationen denkbar. Im Rahmen dieser Staatsexamensarbeit war es mir leider nicht möglich eine genauere Planung zu verwirklichen. Weiterführende Ideen zur Realisierung einer Unterrichtsreihe und ein Ausblick ist in Kapitel 8 zu finden.

Teil III.

Schlussbetrachtung

8. Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde versucht, der theoretischen Informatik durch einen an Beispielen orientierten Einstieg in die Komplexitätstheorie mehr Raum im Schulunterricht zu schaffen. Wie in Kapitel 5 erläutert, gibt es viele gute Gründe Schülerinnen und Schülern einen Einblick in eine beständige Theorie, hier in das Konzept der polynomiellen Reduktion und der $\mathcal{NP}$-Vollständigkeitstheorie, anzubieten.

> „Die Ergebnisse der Theorie sind von beständigem Wert und nicht den schnellen Entwicklungszyklen und kurzen Halbwertszeiten der hektischen Computerwelt unterworfen. Die Inhalte der theoretischen Informatik sind nicht immer so direkt anwendbar wie viele Inhalte anderer informatischer Themenbereiche. Dafür sind die Erkenntnisse der theoretischen Informatik oft allgemeiner, umfassender und weitreichender als in anderen Gebieten der Informatik." [Kul12]

Mit dem Konzept der polynomiellen Reduktion und der Ergebnisse von $\mathcal{NP}$-Vollständigkeitsbeweisen erfahren und erleben Schülerinnen und Schüler die Grenzen des praktisch Möglichen. Viele Meilensteine in der Entwicklung der theoretischen Informatik werden durch die gewählte Thematik abgedeckt. Anwendungsbezogene Beispiele erleichtern den informellen Zugang zu „harter" Theorie. Aber welche Auswege gibt es, wenn man es mit einem $\mathcal{NP}$-vollständigen Problem zu tun hat? Häufig behilft man sich mit Heuristiken und Näherungsalgorithmen. Diese liefern dann zwar keine exakte, aber eine in gewissem Sinne brauchbare Lösung. Meist wird hier „Brauchbarkeit" durch festgelegte Fehlertoleranzen ermittelt. In anderen Fällen wird die Abweichung von der Optimalität abgeschätzt. Für alle in dieser Arbeit vorgestellten Probleme existieren verschieden gute Approximationsalgorithmen, die in der Fachliteratur nachgelesen werden können. Michael Fothe betrachtet für das Knotenüberdeckungsproblem (Vertex-Cover-Problem oder abgekürzt: VC) zwei Lösungsstrategien (vgl. [al.07]), die ich hier als Exempel skizzieren möchte. Zur Erinnerung hier noch einmal die Definition des Vertex-Cover-Problems:

$$\text{VC} = \{\, (G,k) \mid \text{G besitzt eine Menge T von k Knoten, so dass jede Kante einen Endpunkt in T besitzt.}\}$$

Es muss also nach der Definition jede Kante durch einen ihrer beiden Endknoten abgedeckt werden. „Das Problem ist, dass im Allgemeinen Fall nicht schnell

zu sehen ist, welche Wahl die günstigere ist“ [al.07]. Warum nicht einfach beide dem Vertex-Cover hinzufügen? Mindestens einer der Endknoten muss in einer optimalen Lösung enthalten sein. Man wählt eine beliebige Kante, fügt beide Endknoten dem Cover hinzu und löscht die abgedeckte Kante aus dem Graph. Dieses Vorgehen wird solange wiederholt, bis keine Kante mehr übrig ist. Man bekommt so eine Lösung, die höchstens doppelt so groß ist wie eine optimale Lösung. In der Fachliteratur spricht man hier von einem 2-approximativen-Algorithmus.
Eine weitere Strategie zum Erhalt einer Näherungslösung ist ein sogenanntes „gieriges“ (engl. greedy)[1] Verfahren. „Je mehr Kanten an einem Knoten anliegen, desto mehr Kanten werden bei Wahl dieses Knotens abgedeckt“ [al.07]. Intuitiv spricht nichts dagegen, jeweils denjenigen Knoten zu wählen, der aktuell die meisten Kanten abdeckt[2]. Nach Fothe führt diese Strategie überraschenderweise zu einer Näherungslösung, die im schlimmsten Fall etwas schlechter wie die zuvor beschriebene Vorgehensweise ist.
Viele Experten werden auch in Zukunft weiter an besseren Näherungsverfahren arbeiten. Und vielleicht wird es irgendwann, mit neuen, verbesserten Beweistechniken, auch möglich sein, dem $\mathcal{P} = \mathcal{NP}$-Problem auf den Grund zu gehen.

Leider war es mir im, zeitlich recht strengen, Rahmen dieser Arbeit nicht möglich, eine konkrete Unterrichtssequenz zu erstellen. Meine Arbeit lässt didaktisch und auch methodisch noch viel Spielraum zur Verfeinerung und Konkretisierung. Denkbar wäre zum Beispiel auch die Entwicklung einer Lernumgebung, die es Schülern ermöglicht gefundene Färbungen eines Graphen zu den konkreten Beispielen am Computer zu verifizieren. So wäre es Schülerinnen und Schülern auch möglich ihr Modell zu überprüfen. Man könnte sogar soweit gehen einen eigenen Verifizierer für die erstellten Booleschen Formeln zu entwickeln. Die Technik der SAT-SOLVER ist für die relativ „kleinen“ Probleme schon weit genug entwickelt. Durch Integration eines SAT-SOLVERS wäre es dann auch möglich, gefundene Belegungen zu verifizieren oder falsifizieren. Der nächste Schritt aber sollte ein konkreter Unterrichtsentwurf sein. Die ersten Schritte sind getan und ich werde weiter an einem ausgefeilteren Konzept arbeiten.

[1] Daher spricht man auch von Greedy-Algorithmen
[2] Hier wird demnach der Grad des Knotens betrachtet

Literaturverzeichnis

[3SA12] *3-Coloring is NP-Complete.* http://www.csc.liv.ac.uk/~igor/COMP309/3CP.pdf, Mai 2012

[ACB12] Andrew C. Bartlett, Amy N. L.: *An Integer Programming Model for the Sudoku Problem.* http://langvillea.people.cofc.edu/Sudoku/sudoku2.pdf, April 2012

[al.07] al., M. F.: Das Knotenüberdeckungsproblem. In: *LOG IN* 146 (2007), S. 53–59

[al.12] al., Myriam E.: *Ausarbeitung über das 2-Sat Problem.* https://www.ads.tuwien.ac.at/teaching/ws03/AlgoGraph/ue1Auf1.pdf, April 2012

[Arn07] Arnold, Ruedi: *Interactive Learning Environments for Mathematical Topics.* Zürich : Dissertation ETH ZURICH, 2007

[Bau94] Baumann, Rüdeger: Graphen und Algorithmen. In: *LOG IN* 14 (1994), S. 25–37

[Bau96] Baumann, Rüdeger: *Didaktik der Informatik.* 2^{st} edition. Stuttgart : Klett Verlag, 1996. – ISBN 3–12–985010–4

[Beh12] Behrens, Grit: *Boolesche Algebra.* http://www.cs.hs-rm.de/~behrens/EinfInf/EinfInf_WS0809_6_Boolesche_Algebra.pdf, April 2012

[Böh12] Böhm, André: *Formelchecker.* http://www.tks.informatik.uni-frankfurt.de/formelchecker/, April 2012

[Bün94] Büning, Lettmann: *Aussagenlogik: Deduktion und Algorithmen.* Stuttgart : Teubner Verlag, 1994. – ISBN 3–519–02133–1

[Che12] Chen, Jiehua: *Graphen als Modelle der Wirklichkeit.* http://www.user.tu-berlin.de/jchen/GraphenAlsModelle.pdf, Mai 2012

[Coo12] Cook, Steven: *The P Versus NP Problem.* http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf, April 2012

[Eid] Eidmann, Rainer: *Algorithmische Graphentheorie im Unterricht unter Verwendung objektorientierter Datenstrukturen.* Wuppertal : Dissertation an der Bergischen Universität/Gesamthochschule Wuppertal

[Gol10] Goldreich, Oded: *P,NP and NP-Completeness.* 1^{st} Auflage. Cambridge : University Press, 2010. – ISBN 978–0–521–19248–4

[Har74] Harary, Frank: *Graphentheorie.* München : Oldenbourg Verlag, 1974. – ISBN 3–486–34191–X

[Hed12] Hedrich, Prof. Dr.-Ing. L.: *Boolesche Funktionen und Boolesche Algebra.* http://www.em.informatik.uni-frankfurt.de/fileadmin/em_files/lehre/ss12/hwr/folien/hwr_kap03_vers53.pdf, April 2012

[Hof11] Hoffmann, Dirk W.: *Grenzen der Mathematik - Eine Reise Durch Die Kerngebiete Der Mathematischen Logik.* 1^{st} edition. Heidelberg : Spektrum Verlag, 2011. – ISBN 3–827–42559–X

[Hro01] Hromkovic, Juraj: *Algorithmische Konzepte der Informatik.* 1^{st} edition. Stuttgart : Teubner, 2001. – ISBN 3–519–00332–5

[Hub01] Hubwieser, Peter: *Didaktik der Informatik.* 1^{st} Auflage. Berlin : Springer, 2001. – ISBN 3–540–65564–6

[Hub04] Hubwieser, Peter: *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele.* 2^{st} Auflage. Berlin : Springer, 2004. – ISBN 3–540–43510–7

[IL12] Inês Lynce, Joël O.: *Sudoku as a SAT Problem.* http://anytime.cs.umass.edu/aimath06/proceedings/P34.pdf, April 2012

[Inf12] Infomatiktools.de: *InfoTraffic.* http://informatiktools.de/index.php?title=InfoTraffic, Mai 2012

[Ins12] Institute, Clay M.: *Dedicated to increasing and disseminating mathematical knowledge.* http://www.claymath.org/millennium/P_vs_NP/, April 2012

[Joh79] Johnson, Michael R. Garey ; David S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* 1^{st} edition. New York : Freeman, 1979. – ISBN 0–716–71045–5

[Ker91] Kerner, Immo O.: Harte Nüsse. In: *LOG IN* 3 (1991), S. 9–17

[kön12] *Königsberg.* http://www.koenigsberg-stadtplan.de.vu/, Mai 2012

[KNF12] *KNF-DNF-Konverter.* http://web.student.tuwien.ac.at/~e0425426/praktikum/, April 2012

[Kon12] KONEN, Prof. Dr. W.: *Diophantische Gleichungen*. http://www.gm.fh-koeln.de/~konen/DisMa/DisMa-Kap2.pdf, April 2012

[Kul12] KULTUSMINISTERIUM, Hessisches: *Lehrplan Informatik Gymnasialer Bildungsgang*. http://www.kultusministerium.hessen.de/irj/HKM_Internet?uid=3b43019a-8cc6-1811-f3ef-ef91921321b2, Mai 2012

[Mat12] MATHEPRISMA: *Turingmaschine*. http://www.matheprisma.uni-wuppertal.de/Module/Turing/index.htm, April 2012

[ope12] OPENLIBRARY: *The mathematical analysis of logic*. http://openlibrary.org/works/OL16072250W/The_mathematical_analysis_of_logic, Mai 2012

[Pla79] PLASS, Robert Tarjan; Bengt Aspvall; M.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. In: *Information processing letters* 8 (1979), S. 121–123

[Pol12] POLOCZEK, Dr. J.: *Färbungen auf Graphen*. http://www-stud.informatik.uni-frankfurt.de/~poloczek/, April 2012

[RA12a] R. ARNOLD, W. H.: *InfoTraffic - Teaching Important Concepts of Computer Science and Math through RealWorld Examples*. http://www.vs.inf.ethz.ch/publ/papers/infotraffic_ACM_SIGCSE_2007.pdf, Mai 2012

[RA12b] R. ARNOLD, W. H.: *InfoTraffic: Informatikkonzepte im Alltag (Verkehrssteuerung)*. http://www.swisseduc.ch/informatik/infotraffic/, Mai 2012

[Rap87a] RAPP, Andreas: Der primitivste Computer der Welt (Teil I). In: *LOG IN* 1 (1987), S. 38–40

[Rap87b] RAPP, Andreas: Der primitivste Computer der Welt (Teil II). In: *LOG IN* 2 (1987), S. 36–39

[Rei12] REISCHER, Jürgen: *Boolesche Algebra*. http://www.gruenderboom.de/Mathematik/Aussagenlogik.pdf, April 2012

[Sch94] SCHWILL, Andreas: Praktisch unlösbare Probleme. In: *LOG IN* 14 (1994), S. 16–21

[Sch09] SCHWEIKARDT, Prof. Dr. N.: *Diskrete Modellierung: Eine Einführung in grundlegende Begriffe und Methoden der Theoretischen Informatik*. Frankfurt : Goethe Universität, 2009

[Sch11] SCHNITGER, Prof. Dr. G.: *Skript zur Vorlesung Algorithmentheorie*. Frankfurt : Goethe Universität, 2011

[Sie12] Siegfried, Robert: *Färbungen auf Graphen.* http://www.rsiegfried.de/files/fhw/2003_Faerbungen_auf_Graphen_Slides.pdf, Mai 2012

[Sim92] Simon, K.: *Effiziente Algorithmen für perfekte Graphen.* Stuttgart : Teubner Verlag, 1992

[Sim05] Simonis, H.: Sudoku as a constraint problem. In: *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems* 12 (2005), S. 13–27

[SS04] S. Schubert, A. S.: *Didaktik der Informatik.* 1^{st} edition. Heidelberg : Spektrum Akademischer Verlag, 2004. – ISBN 3–8274–1382–6

[Ste11] Steinert, Markus: Grenzen der Berechnbarkeit. In: *LOG IN* 168 (2011), S. 42–49

[The12] Then, Dr. H.: *Woher wissen wir das Alter unseres Universums.* http://www.physik.uni-oldenburg.de/36997.html, Mai 2012

[Tho12] Thomas, Wolfgang: *Theoretische Informatik: ein Kurzprofil.* http://www.automata.rwth-aachen.de/download/papers/thomas/tho10c.pdf, Mai 2012

[Tur36] Turing, Alan M.: On computable numbers, with an application to the Entscheidungsproblem. In: *Proceedings of the London Mathematical Society* 42 (1936), S. 230–265

[Unb12] Unbekannt: *Vorlesungsmitschrift Graphentheorie.* http://avlinux.et.fh-stralsund.de/~wkampow/downloads/Script4.pdf, April 2012

[uni12] *Definitionen.* http://www.ims.uni-stuttgart.de/lehre/teaching/2004-WS/Formale-Sprachen/pdf/session2.pdf, Mai 2012

[VK12] Volker Kaibel, Thorsten K.: *Mathematik für den Volkssport.* http://www.math.uni-magdeburg.de/~kaibel/Downloads/mdmv-sudoku.pdf, April 2012

[Wal02] Walter, Wolfgang: *Analysis 2.* 5^{st} Auflage. Heidelberg : Springer Verlag, 2002. – ISBN 3–540–42953–0

[Was12] Wassong, Thomas: *Graphentheoretische Konzepte in der Gymnasialen Oberstufe: Ein Unterrichtsentwurf unter Berücksichtigung der Neuen Medien.* http://lama.uni-paderborn.de/fileadmin/Mathematik/People/wassong/examen.pdf, Mai 2012

[Weg96] Wegener, Ingo: *Kompendium Theoretische Informatik - eine Ideensammlung*. 3^{st} Auflage. Stuttgart : Teubner, 1996. – ISBN 3–519–02145–5

[Weg03] Wegener, Ingo: *Komplexitätstheorie*. 1^{st} Auflage. Berlin : Springer, 2003. – ISBN 3–540–00161–1

[Wik12a] Wikipedia: *George Boole*. http://de.wikipedia.org/wiki/George_Boole, April 2012

[Wik12b] Wikipedia: *Karps 21 NP-vollständige Probleme*. http://de.wikipedia.org/wiki/Karps21NP-vollständigeProbleme, April 2012

[Wik12c] Wikipedia: *Komplexitätstheorie*. http://de.wikipedia.org/wiki/Komplexitätstheorie, April 2012

[Wik12d] Wikipedia: *Konjunktive Normalform*. http://de.wikipedia.org/wiki/Konjunktive_Normalform, April 2012

[Wik12e] Wikipedia: *Lateinische Quadrate*. http://de.wikipedia.org/wiki/Lateinisches_Quadrat, April 2012

[Wik12f] Wikipedia: *Logik*. http://de.wikipedia.org/wiki/Logik#Geschichte_der_Logik, April 2012

[Wik12g] Wikipedia: *Sudoku*. http://de.wikipedia.org/wiki/Sudoku, April 2012

[Wik12h] Wikipedia: *Turingmaschine*. http://de.wikipedia.org/wiki/Turingmaschine, April 2012

[Wik12i] Wikipedia: *Dualsystem*. http://de.wikipedia.org/wiki/Dualsystem, Mai 2012

[Wik12j] Wikipedia: *Satz von Cook*. http://de.wikipedia.org/wiki/Satz_von_Cook, Mai 2012

[x:K12] *Komplexität*. http://isgwww.cs.uni-magdeburg.de/ag/lehre/SS2003/ThInf/slides/Ch6.pdf, April 2012

Anhang

A. Verwendete Software

A.1. Infotraffic

Bei „Infotraffic“ handelt es sich um eine Lernumgebung zu wichtigen Konzepten aus der Informatik und der Mathematik. Die Lernumgebung wurde von Ruedi Arnold (und Werner Hartmann) im Zusammenhang mit seiner Dissertation im Jahr 2007 entwickelt. Die Lernsoftware beschäftigt sich rund um Themen der Verkehrssteuerung.

> „InfoTraffic umfasst drei originäre interaktive Lernumgebungen (vgl. Abbildung A.1) zu Aussagenlogik, Warteschlangentheorie und dynamischen Systemen. Die drei Programme sind eingebettet in das gemeinsame Szenario „Verkehrssteuerung“ und nutzen konsequent einen virtuell-enaktiven, das heißt handlungsorientierten, und alltagsbezogenen Ansatz zur Einführung abstrakter Themen im Unterricht.“ [Arn07]

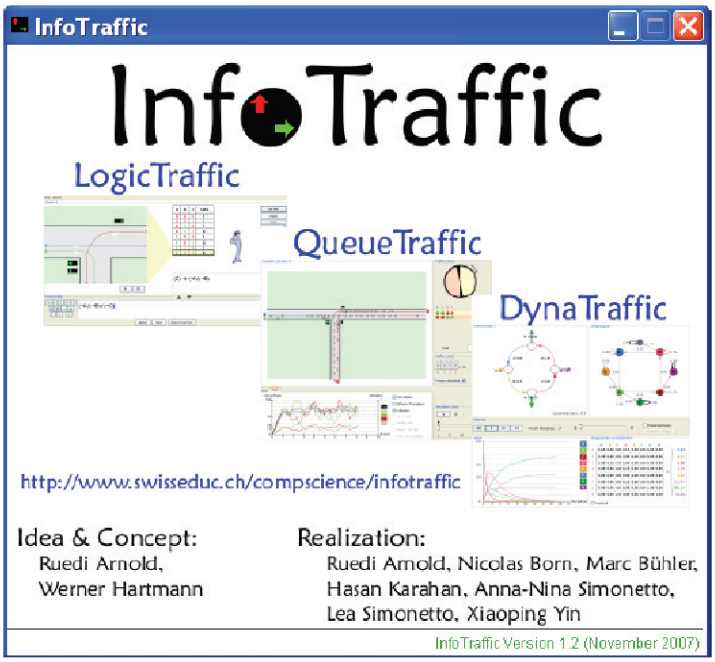


Abbildung A.1.: Infotraffic Startfenster [Inf12]

Die drei hier erwähnten interaktiven Lernumgebungen sind:

- LogicTraffic, welches einen Einstieg in die Aussagenlogik bietet.
- QueueTraffic beschäftigt sich mit Grundbegriffen der Warteschlangentheorie.
- DynaTraffic bietet einen Einstieg in die Theorie der Markovketten.

In der vorliegenden Arbeit wurde nur eine der drei Komponenten von Infotraffic benutzt. Auch wenn die beiden anderen großes Potential besitzen, beschränke ich mich hier auf das Vorstellen von LogicTraffic. Eine schöne Beschreibung und Einführung der beiden anderen Komponenten QueueTraffic und DynaTraffic kann in [Arn07] oder [RA12a] nachgelesen werden. Viele weitere Informationen finden sich auch auf der zugehörigen Homepage [RA12b].

LogicTraffic nutzt das Beispiel der Ampelschaltungen an Kreuzungen, um die Grundlagen der Aussagenlogik zu verdeutlichen. Hauptziel ist es in dem Programm, dass Schüler und Schülerinnen versuchen, Ampeln einer vorgegebenen Kreuzungssituation so zu schalten, dass es nicht zu Kollisionen kommen kann. Jede Variable korrespondiert zu einer vorgegebenen Autospur. Die booleschen Variablen „true“ (1) und „false“ (0) entsprechen einem grünen bzw. roten Ampelsignal der entsprechenden Spur.

> „The learning targets of LogicTraffic include the basics of propositional logic (PL), and cover concepts such as variables, truth values, logical operators, formulas, equivalence of formulas, and normal forms.“ [RA12a]

In Abbildung A.2 ist die Benutzeroberfläche zu sehen.

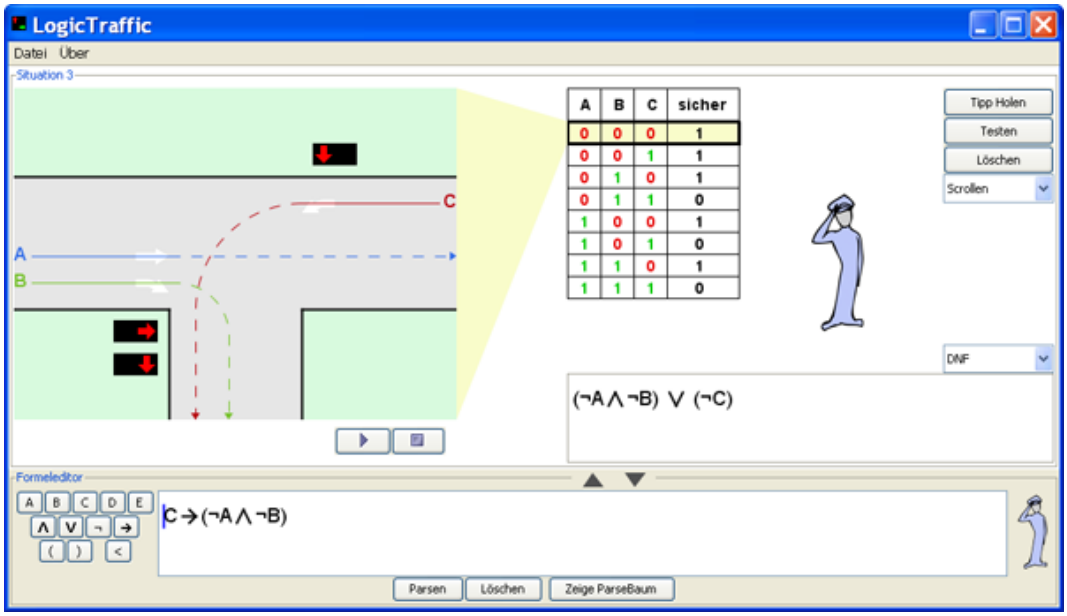


Abbildung A.2.: Benutzeroberfläche von LogicTraffic [Inf12]

Die Benutzeroberfläche von InfoTraffic besteht aus fünf Bereichen (vgl. Abbildung A.2). Zum einen wird links eine bestimmte Verkehrssituation mit sich

schneidenden Autospuren und zugehörigen Ampeln anhand eines Bildes gezeigt. Hier stehen dem Benutzer mehrere Situationen, die durch das Menu auswählbar sind, zur Verfügung. Rechts daneben findet man eine Wahrheitstabelle, die alle möglichen Ampelsignalbeschaltungen widergibt. Die Wahrheitstabelle bietet den Schülern und Schülerinnen die Möglichkeit verschiedene Signalschaltungen als sicher, durch setzen einer „1“ („true“) (bzw. unsicher entsprechend durch setzen einer „0“ für „false“), zu markieren. Unter der Wahrheitstabelle wird eine Boolesche Formel, die zu den ausgewählten sicheren und unsicheren Schaltungen der Wahrheitstabelle korrespondiert, angezeigt. Statt von Hand die einzelnen „sicher“ und „unsicher“ Markierungen zu setzen und sich dann die zugehörige Boolesche Formel anzeigen zu lassen, ist es auch möglich (in der Benutzeroberfläche zentral am unteren Rand) direkt eine Boolesche Formel einzugeben. Außerdem erlauben mehrere Bedienelemente dem Benutzer jede einzelne Zeile der Wahrheitstabelle in dem Bild der Verkehrssituation zu simulieren. So ist es möglich, gewählte Belegungen zu testen und zu verifizieren. In Abbildung A.4 kann eine solche Simulation angeschaut werden.

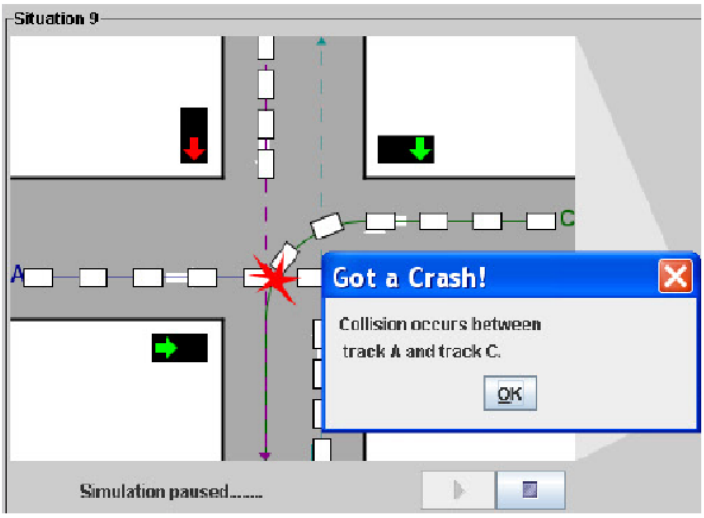


Abbildung A.3.: Simulierter Crash in LogicTraffic [RA12a]

Das Programm stellt mehrere Darstellungsformen der booleschen Formeln zur Verfügung, so dass Schüler und Schülerinnen sich zum Beispiel direkt eine konjunktive Normalform zu der von Ihnen hergestellten Wahrheitstabelle anzeigen lassen können. Neben der konjunktiven Normalformen bietet das Programm den Wechsel in die disjunktive Normalform an. Die beiden Normalformen können jeweils auch in ihrer kanonischen Form dargestellt werde. Außerdem ist es möglich sich eine äquivalente Formel, die Implikationen nutzt, und eine „einfachste“ Darstellung[1] der Formel anzusehen. Eine weitere schöne Option ist das Anzeigen des zugehörigen Syntaxbaumes. Ziel ist es dadurch die verschie-

[1] Unter [Arn07] ist ausführlich erklärt, was die Autoren unter „einfach“ verstehen.

denen Darstellungsformeln miteinander und auch mit der Wahrheitstabelle zu verknüpfen, so dass Schüler und Schülerinnen die Zusammenhänge erlernen können.

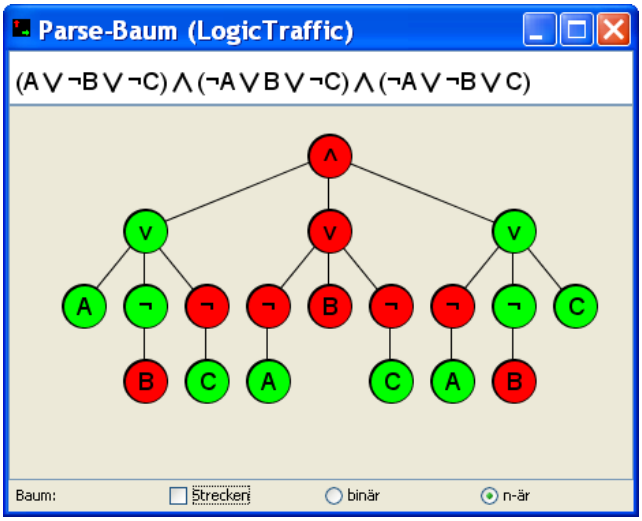


Abbildung A.4.: Darstellung einer booleschen Formel als Syntaxbaum [RA12b]

Das System ist inklusive Unterrichtsmaterialien im Netz (siehe [RA12b]) frei verfügbar und wird an Gymnasien, Hochschulen und auch in der Lehrerausbildung eingesetzt.

Danksagungen

„Leider lässt sich eine wahrhafte Dankbarkeit mit Worten nicht ausdrücken."
[Johann Wolfgang von Goethe, 1749-1832]

Wie in Abschlussarbeiten üblich, möchte ich an dieser Stelle die Gelegenheit nutzen einigen Personen zu danken:

An erster Stelle danke ich meinem Betreuer Herrn Dr. Jürgen Poloczek des Fachbereiches „Didaktik der Informatik" von der Goethe Universität Frankfurt am Main für die professionelle Betreuung bei meiner Examensarbeit. Mit seiner Hilfe und seinen Ratschlägen stand er mir jederzeit zur Seite.

Außerdem danke ich Herrn Prof. Dr. Reinhard Oldenburg des Fachbereiches „Didaktik der Informatik und Didaktik der Mathematik" von der Goethe Universität Frankfurt am Main dafür, dass er sich als Zweitgutachter zur Verfügung gestellt hat.

Meinen Eltern dient der allergrößte Dank. Ohne deren Unterstützung wäre mein Studium und diese Arbeit letztendlich nicht möglich gewesen. Vielen Dank!

Weiterhin danke ich meiner Tochter. Süße, danke dafür, dass du es auch in der stressigen Zeit jeden Tag geschafft hast mir ein Lächeln auf mein Gesicht zu zaubern!

Ich bedanke mich bei meinem Lebensgefährten Daniel Cattarius. Ohne ihn hätte ich diese Zeit wohl kaum geschafft. Danke für seine Unterstützung in allen Lagen und natürlich auch danke für all die Anregungen und Tipps beim Erstellen dieser Arbeit.

Ich möchte mich auch bei der Familie meines Lebensgefährten bedanken. Danke für die Bereitschaft meine Arbeit Korrektur zu lesen und danke auch für all die anderen „Kleinigkeiten".