Skript

Elementare mathematische Begriffe, Probleme und Schreibweisen

Wintersemester 2019/2020

Prof. Dr. Steffen Reith
Steffen.Reith@hs-rm.de

Hochschule RheinMain
Fachbereich Design Informatik Medien

Erstellt von:	Steffen Reith
Zuletzt bearbeitet von:	Steffen Reith
Email:	steffen.reith@hs-rm.de
Erste Version vollendet:	August 2006
Version:	77ab5fc
Letzte Änderung:	2021-01-29 23:47:55 +0100

Wenn Leute nicht glauben, dass Mathematik
einfach ist, dann nur deshalb, weil sie nicht begreifen,
wie kompliziert das Leben ist.

John von Neumann

Wenn es eine gute Idee ist, dann mach es
einfach. Es ist viel einfacher sich hinterher zu
entschuldigen, als vorher dafür eine
Genehmigung zu bekommen.

Grace Hopper

Wake up! Time to die!

Leon in Blade Runner

Dieses Skript ist aus den Fragen und Bemerkungen von Studenten des Diplom, Bachelor und Master-Studiengangs Informatik an der Hochschule RheinMain (ehemals Fachhochschule Wiesbaden) hervorgegangen. Ich danke allen meinen Studenten für konstruktive Anmerkungen und Verbesserungen. Dabei möchte ich besonders Frau Carola Henzel nennen, die sehr viele Tippfehler (Mengen, Summen und Beweistechniken) berichtigte. Herr Kim Stebel hat die Bemerkungen zu bipartiten Graphen verbessert. Herr Norbert Wesp berichtete über sprachliche Fehler in Abschnitt 4 und Tippfehler in der Einleitung. Eine größere Anzahl von Tipp- und Flüchtigkeitsfehlern wurden von Herrn Marcell Dietl entdeckt und gemeldet. Frau Sandra Peruzzi steuerte ein sehr sinniges Zitat von Meister Yoda bei. Danke!

Naturgemäß ist ein Skript nie fehlerfrei (ganz im Gegenteil!) und es ändert (mit Sicherheit!) sich im Laufe der Zeit. Es sollte Ihnen klar sein, dass Fehler, Ungenauigkeiten und Unklarheiten natürlich nur aus didaktischen Gründen und zu Ihrer Belustigung eingebaut wurden. Finden Sie diese Fehler und verbessern Sie mich!

Inhaltsverzeichnis

1 Vorwort 1

2 Grundlagen und Schreibweisen 7

2.1 Mengen . . . 7
2.1.1 Die Elementbeziehung und die Enthaltenseinsrelation . . . 7
2.1.2 Definition spezieller Mengen . . . 7
2.1.3 Operationen auf Mengen . . . 8
2.1.4 Gesetze für Mengenoperationen . . . 9
2.1.5 Tupel (Vektoren) und das Kreuzprodukt . . . 9
2.1.6 Die Anzahl von Elementen in Mengen . . . 10
2.2 Relationen und Funktionen . . . 10
2.2.1 Eigenschaften von Relationen . . . 10
2.2.2 Eigenschaften von Funktionen . . . 11
2.2.3 Hüllenoperatoren . . . 12
2.2.4 Permutationen . . . 12
2.3 Summen und Produkte . . . 13
2.3.1 Summen . . . 13
2.3.2 Produkte . . . 14
2.4 Logarithmieren, Potenzieren und Radizieren . . . 15
2.5 Gebräuchliche griechische Buchstaben . . . 16

3 Einige (wenige) Grundlagen der elementaren Logik 17

4 Einige formale Grundlagen von Beweistechniken 19

4.1 Direkte Beweise . . . 19
4.1.1 Die Kontraposition . . . 20
4.2 Der Ringschluss . . . 21
4.3 Widerspruchsbeweise . . . 22
4.4 Der Schubfachschluss . . . 22
4.5 Gegenbeispiele . . . 22
4.6 Induktionsbeweise und das Induktionsprinzip . . . 23
4.6.1 Die vollständige Induktion . . . 23
4.6.2 Induktive Definitionen . . . 25
4.6.3 Die strukturelle Induktion . . . 25

5 Graphen und Graphenalgorithmen 27

5.1 Einführung . . . 27
5.2 Grundlagen . . . 27
5.3 Einige Eigenschaften von Graphen . . . 28
5.4 Wege, Kreise, Wälder und Bäume . . . 31
5.5 Die Repräsentation von Graphen und einige Algorithmen . . . 31

6 Komplexität 35

6.1 Effizient lösbare Probleme: die Klasse **P** . . . 35
6.1.1 Das Problem der 2-Färbbarkeit . . . 37
6.2 Effizient überprüfbare Probleme: die Klasse **NP** . . . 40
6.3 Schwierigste Probleme in **NP**: der Begriff der **NP**-Vollständigkeit . . . 43
6.3.1 Traveling Salesperson ist **NP**-vollständig . . . 45
6.4 Die Auswirkungen der **NP**-Vollständigkeit . . . 48

6.5 Der Umgang mit **NP**-vollständigen Problemen in der Praxis 49

Stichwortverzeichnis **53**

Literatur **57**

Tu es oder tu es nicht - es gibt kein Versuchen!
Yoda[1] zu Luke Skywalker

1 Vorwort

Viele Studierende haben, besonders in den ersten Semestern, oft erhebliche Probleme mit mathematischen Notationen und Denkweisen, die gerade in der (praktischen) Informatik eine extrem große und wichtige Rolle spielen. Es ist auch zu beobachten, dass dieser Mangel von den Studierenden gar nicht bemerkt wird, da noch keine tiefere Einsichten in das Gebiet der Informatik (und Mathematik) gewonnen werden konnten. Als Folge empfinden die Studierenden Mathematik deshalb als schwierig *und* nutzlos. Dieses Vorurteil wird dann erfahrungsgemäß von vielen nicht (mehr) hinterfragt, wodurch sich diese Studierenden die Chance nehmen, tiefliegende Erkenntnisse auf dem Gebiet der Informatik zu sammeln. Weiterhin scheint es auch einen ausgeprägten Widerwillen bzgl. des Lesens von (linearen) Texten zu geben (Ja, wir sind immer noch im ersten Absatz!), was das Lernen ebenfalls massiv behindert. Deshalb würde ich Ihnen (ja, genau *Sie*) empfehlen, gegen beide Probleme gleich am Anfang Ihres Studiums etwas zu unternehmen. Es lohnt sich!

Alle diese Probleme lassen sich nur durch stetiges, selbstständiges, regelmäßiges und ausdauerndes Üben, „selber machen“ und durch selbstständiges Anfertigen von Vorlesungsmitschriften überwinden. Lassen Sie sich nicht entmutigen, wenn das Lösen einer einzigen Aufgabe manchmal mehrere Stunden benötigt. Dies ist völlig normal! Auch der Autor dieses Skripts sitzt an seinen „Übungsaufgaben“ (Forschung) sehr oft, braucht meist sehr viel Zeit (6 Monate sind da keine Seltenheit) und ist bei Misserfolgen frustriert. Kurz: Das muss so sein, damit sich ein signifikanter Lernerfolg einstellt!

Leider zeigt es sich oft, dass viele Schulen solche Kompetenzen nicht (mehr) vermitteln oder Sie haben einen „nicht klassischen“ Weg an die Hochschule genommen, d.h. Sie werden diese am Anfang Ihres Studiums mühsam, mit Schweiß und oft auch mit Frust erlernen müssen. Das ist leider völlig normal und Sie können sich sicher sein, dass *alle* Ihre Dozenten diese Phase in ihrem Studium genauso wie Sie erlebt (und gemeistert) haben! Glauben Sie keinen „didaktischen Heilsversprechen“ (z.B. eLearning, alternative Lehrformen), die angeblich einen einfachen Weg zur Erkenntnis garantieren. Gäbe es diese Wege, so hätte niemand Probleme mit dem mathematischen Denken, das in der Informatik und der Alltagswelt sehr nützlich ist. Auch die Grundlagen der Naturwissenschaften wären jedem bestens bekannt. Vertrauen Sie lieber auf Ihre *eigenen Fähigkeiten* und akzeptieren Sie die Tatsache, dass Lernen schwer ist und schwer bleibt, zumindest wenn es um nichttriviale Inhalte geht. Glauben Sie auch nicht, dass nur Sie diese Schwierigkeiten haben! Ideen und Einsichten kommen meist „von selbst“, wenn Sie sich (selbst!) nur lange genug mit dem Stoff beschäftigen, d.h. das kompromisslose „Verbeißen“ in ein Thema ist der Schlüssel zum Erfolg. Ein Geheimnis für Ihren persönlichen Studienerfolg ist die Einstellung „ich werde nicht aufgeben“. Eine gewisse Leidenschaft für *Ihr* Fach ist also für die notwendige Motivation unabdingbar. Erfahrungsgemäß sind „nicht locker lassen“ und Motivation[2] fast schon ein Garant für ein erfolgreiches Studium.

Ideen kommen oft zu den komischsten Zeiten, deshalb ist es wichtig sich dafür (viel) Zeit zu nehmen. Schalten Sie alle Ablenkungen (z.B. „Internet“ oder „WhatsApp“) ab! Oft kommen dann Ideen unter der Dusche, beim spazieren gehen, beim Ausdauersport, während Tagträumen[3] oder

[1] Nachdem Luke seinen im Morast versunkenen Raumgleiter nur mit der Macht aus dem Dreck hieven sollte und zu seinem Meister sagte, er könne es ja mal „versuchen“.

[2] Entsprechend der Bemerkung „Sie dringen in unseren Raum ein und wir weichen zurück. Sie assimilieren ganze Welten und wir weichen zurück. Doch jetzt nicht mehr. Hier wird der Schlussstrich gezogen. Bis hierher und nicht weiter. Und ich, ich werde sie bezahlen lassen für ihre Taten!“ würde ich Ihnen empfehlen einfach nicht zurück zu weichen und die Sache einfach durchzuziehen.

[3] Beschäftigen Sie sich einmal mit der Legende von August Kekulé und der Struktur von Benzol.

nach einem Mittagsschlaf. Es gibt Hinweise, dass Schlaf für das Lehren extrem wichtig ist, was mit den persönlichen Erfahrungen des Autors übereinstimmt, deshalb wird das Lernen drei Tage vor der Prüfung meist nicht zu einem brauchbaren Erfolg führen!

Sie erwerben sich durch diese Einstellung und die Übungsaufgaben u.a. eine *Problemlösungskompetenz*, die für spätere Projekte (z.B. Bachelor- und Masterarbeit) extrem hilfreich ist. In der beruflichen Praxis werden Sie sich (hoffentlich) auch regelmäßig mit sehr komplexen Aufgabenstellungen beschäftigen[4], d.h. diese Problemlösungsstrategien sind eine Grundvoraussetzung für das Berufsleben eines erfolgreichen Informatikers. Auch aus diesen Gründen ist es also sehr sinnvoll, die (wöchentlichen) Übungsblätter oder Praktikumsaufgaben der jeweiligen Veranstaltung selbstständig und regelmäßig zu bearbeiten! Es reicht nicht, die in den Übungen präsentierten Lösungen zu konsumieren indem man irgendwelche Musterlösungen oder Lösungen der Kommilitonen abschreibt, denn ein Lerneffekt stellt sich nur ein, wenn Sie die Aufgaben selbstständig vollständig bearbeiten. Es reicht *auch* nicht eine grobe Idee auf einen Schmierzettel zu schreiben! Eine gute Lösung ist sauber ausgearbeitet und durchdacht. Die äußere Form sollte einem Dritten (z.B. einem anderen Kommilitonen) davon überzeugen können, dass die Idee richtig ist. Bleibt nur ein leiser Zweifel, dann ist die Ausarbeitung noch lückenhaft und muss überarbeitet werden.

Alle diese Tipps stellen keinen Widerspruch zu der Arbeit in (kleinen) Gruppen dar, wenn die einzelnen Schritte alleine gelöst und dann mit anderen Studenten zu einer finalen und schönen Version zusammen gesetzt werden. In Ihrer beruflichen Praxis werden Sie später für das selbstständige und vollständige Lösen von Problemen – auch in Teams – bezahlt, deshalb sollte klar sein, dass sich Passivität (im Studium) nicht auszahlt.

Dabei ist es auch hilfreich, wenn Sie eine Aufgabe einmal nicht lösen können, denn auch bei einer ausdauernden aber nicht erfolgreichen Bearbeitung einer Aufgabe kommt es zu einem Lerneffekt! Werden die Übungen nicht selbstständig erarbeitet, so bleibt der Lernerfolg allerdings mit an Sicherheit grenzender Wahrscheinlichkeit aus, was sich sicherlich in der Klausur zum ersten Mal (aber nicht zum letzten Mal) rächen wird. Leider zeigt sich, dass diese Bemerkungen durch die Hörer regelmäßig ignoriert werden, deshalb

Warnung: diese Schwierigkeiten lassen sich auf gar keinen Fall durch „drei Tage Lernen vor der Klausur“ lösen!

Auch eine Woche wird dazu erfahrungsgemäß nicht ausreichen! Dieser Hinweis lässt sich sinngemäß natürlich auch auf (fast) alle anderen Fächer Ihres Studiums übertragen. Deshalb nochmal ein wirklich ernst gemeinter Ratschlag:

Nehmen Sie Ihr Studium selbst in den Hand, denn es ist *Ihres*!

Auf jeden Fall ist es *nicht* das Studium Ihres Dozenten! Beachten Sie auch, dass es *nicht* die „Schuld“ Ihres Dozenten ist, wenn Ihr Studium nicht so erfolgreich verläuft, wie Sie es sich erhofft haben. Sollten Sie mit einem Dozenten fachlich (oder auch menschlich) nicht zurecht kommen, so liegt es in *Ihrer* Verantwortung einen Weg zu finden ohne diesen „lästigen“, „uninspirierten“, „langweiligen“, „unfähigen“ und „unmöglichen“ Menschen erfolgreich zu werden. Auch solche Probleme müssen Sie in Ihrem weiteren beruflichen Leben immer wieder meistern, d.h. es nutzt in einem Studium nichts mit seinem Schicksal zu hadern! Auch macht es keinen Sinn ein Fach als „langweilig“ oder „nicht relevant“ zu betrachten, bloß weil der aktuelle Dozent (menschlich) nicht passt! Ich erinnere nochmal: „Es ist Ihr Studium!“ Bitte beachten Sie, dass man gerade am Anfang eines Studiums noch gar nicht abschätzen kann in welche Richtung[5] es gehen soll, d.h. Sie können noch gar nicht abschätzen welche Fächer (für Sie) wichtig und relevant sind oder werden. Insbesondere

[4] Stellen Sie sich vor wie langweilig und schlecht bezahlt Ihr Leben wäre, wenn Sie immer nur die genau gleiche Tätigkeit ausführen müssten.

[5] Der Autor dieses Textes wollte ursprünglich Waffenschmied werden und hatte dann doch (ganz fest) ein Studium der „Chemie“ geplant. Dieser Wunsch wurde dann durch „Theoretische Physik“ oder „Teilchenphysik“ abgelöst.

werden sich Ihre Interessensgebiete sicherlich verschieben! Deshalb ist es ein guter Tipp sich im Studium breit aufzustellen. Die Spezialisierung kommt im Berufsleben ganz von alleine.

In Ihrer schulischen Laufbahn haben Sie solche Probleme in diesem Ausmaß sicherlich nicht meistern müssen („heile Welt"), deshalb ist es möglicherweise sinnvoll sich, insbesondere am Anfang, helfen zu lassen (z.B. durch (ältere) Mitstudenten, Dozenten oder die Fachschaft).

Als Lehrender bekommt man am Anfang einer (eigentlich jeder) Vorlesung *fast immer* die Frage „Gibt es ein Skript oder müssen wir mitschreiben?" gestellt. Einerseits ist diese Frage wichtig, aber auf der anderen Seite ist sie falsch gestellt, denn Sie brauchen ein Skript *und* eine Mitschrift, um eine Vorlesung wirklich erfolgreich zu meistern. Ein Skript bringt Ihnen in der Regel wenig für den Studienerfolg, auch wenn alle Lerninhalte drin ganz genau beschrieben sind. Ähnlich verhält es sich mit einer Vorlesung. Für das tiefgründige Verständnis des Lerninhalts ist die Vor- und Nachbereitung einer Vorlesung essentiell. Dabei ist es entscheidend die Nachbereitung zeitnah durchzuführen, d.h. möglichst am gleichen Tag oder notfalls zumindest in der gleichen Woche. Lassen Sie sich auch nicht durch Folien blenden. Diese mögen schön aussehen, alle Inhalte genau darstellen und Ihnen die Arbeit des Mitschreibens ersparen (scheinbar!), aber deswegen sind diese noch nicht verstanden, gelernt und verinnerlicht! Es reicht *nicht* diese auszudrucken und ein paar Kommentare an den Rand zu schreiben, sondern es ist empfehlenswert aus den Folien ein eigenes Skript zu erstellen. Sie werden schnell lernen bei welchen Fächern Sie eine unmittelbare Nachbereitung brauchen und bei welchen Fächern Sie die Zügel *etwas* schleifen lassen können.

Für die Nacharbeit ist ein Skript hilfreich, da Sie die Lerninhalte dort (hoffentlich) zuverlässig nachschlagen können. Noch hilfreicher sind aber entsprechende Lehrbücher, die Sie in der Hochschulbibliothek finden können. Überhaupt sollte die Bibliothek einer Ihrer Lieblingsaufenthaltsorte werden, denn die Lehr- und Fachbücher enthalten das für Ihr Studium notwendige Wissen in ordentlich aufbereiteter Form und machen es dadurch leichter zugänglich. Auf keinen Fall reicht es, Suchmaschinen wie „Google" oder „Bing" oder Quellen wie „Wikipedia" zu verwenden! Sie verschwenden *massiv* Zeit, wenn Sie sich die benötigten Informationen im Netz zusammen suchen und Sie bleiben bezüglich deren Qualität weiterhin unsicher. Bitte bedenken Sie, dass eine große Portion von Fachwissen benötigt wird, um gute und schlechte Informationsquellen zu unterscheiden. Bei guten Fachbüchern können Sie recht sicher sein, dass die Autoren ausgewiesene Experten auf ihrem Gebiet sind. Trotzdem sollten Sie verschiedene Autoren „ausprobieren", um das Buch zu finden, das Ihnen am besten weiterhilft, denn jeder Autor hat einen unterschiedlichen Schreib- und Erklärstil.

Das effektivste und wichtigste Mittel für die Nachbereitung ist eine persönlich erstellte *Mitschrift*, die Sie während der Veranstaltung selbst erstellen. Oft wird gerade von Anfängern bezweifelt, dass eine Mitschrift nützlich ist. Dazu werden allerlei Ausreden hervorgebracht. Für manche ist es „zu viel Arbeit" oder sie haben eine „schlechte Handschrift". Auch die Argumente „ich kann nicht mit der Hand schreiben und zuhören" oder „ich kann nicht schnell genug schreiben" finden sich jedes Semester immer wieder. Damit stellt sich die Frage, warum eine handschriftliche Mitschrift so wichtig ist. Der Grund ist einfach: Eine Mitschrift enthält die von Ihnen individualisierte Version des (unvertrauten) Lernstoffs und ist ganz auf Ihr Denken und Ihre Anschauung zugeschnitten. Sie enthält Ihre Gedankenbilder, Fragen und die von Ihnen entdecken Zusammenhänge und vielleicht sogar kleine Beispiele. Weiterhin gibt es wissenschaftliche Hinweise die nahelegen, dass das Schreiben mit der Hand den Wissenserwerb unterstützt (z.B. [MB14]).

Damit ergibt sich die Frage wie eine „gute" Mitschrift aussehen sollte. Die äußere Form ist auf jeden Fall nicht sonderlich wichtig. Lassen Sie sich nicht von schön gesetzten und formatieren Texten[6] blenden! Besser ist *immer* eine Menge von selbst beschriebenen Papier mit Kaffeeflecken,

Informatik war nur ein „Betriebsunfall", der sich als sehr segensreich und spannend erwiesen hat. Inzwischen wäre es kein Informatikstudium mehr, sondern ein Studium der Mathematik mit Schwerpunkt auf Zahlentheorie und algebraische Geometrie.

[6] Wie dieser Text zeigt ist dies recht einfach, wenn man das Textsatzsystem LaTeX verwendet.

Eselsohren, das durch den häufigen Gebrauch[7] schon ganz speckig geworden ist, als der neuste Hochglanzausdruck eines aus dem Internet gesaugten PDF-Files! In Ihrer Mitschrift sollte Wichtiges von Unwichtigem klar unterschieden werden. Das benötigt Konzentration, denn Sie müssen dies beim Hören der Vorlesung für sich entscheiden. Mit ein wenig Übung werden Sie hier schnell einen Weg finden. Dabei muss der Tafelanschrieb nicht wörtlich übernommen werden, sondern Sie sollten ihn geeignet verkürzen und umformen. Durch diese Denkleistung behält Ihr Gehirn schon einen Teil der Informationen, ohne das Sie es merken. Aus diesem Grund reicht es nicht, wenn Sie die Mitschrift eines Kommilitonen kopieren. Verwenden Sie, wenn möglich, *eigene* Formulierungen, kurze Sätze, einzelne Wörter, Symbole und Abkürzungen. Die Gliederung und die Überschriften übernehmen Sie einfach aus der Vorlesung. Sehr wichtige zentrale Punkte kann man durch *sparsames* (farbiges) Unterstreichen hervorheben. Schreiben Sie ein Blatt nicht ganz voll. So bleibt Platz für eigene Anmerkungen, Ideen, Querverweise, Fragen und Kommentare. Gerade diese sind extrem wertvoll für Ihren Lernprozess. Deshalb macht es auch Sinn die Mitschrift aus der Vorlesung (zumindest in Teilen) in der Nachbereitungsphase nochmal komplett neu zu schreiben. Ob Sie kompliziertere Fragen gleich oder am Anfang der nächsten Vorlesung stellen ist eigentlich egal, allerdings werden Sie bemerken, dass sich viele Fragen von selbst lösen, wenn Sie sie einfach einmal *richtig aufgeschrieben* haben.

In Ihrem Studium und in jeder Wissenschaft stellt sich sofort die Frage, wie man (neue) Erkenntnisse gewinnen kann. Da Sie sich in der Schule mit solchen Fragestellungen noch nicht beschäftigt haben, sind diese Vorgehensweisen an Hochschulen gerade in der ersten Zeit ungewohnt.

Eine mögliche Methode zur Erzeugung neuen Wissens funktioniert wie folgt: Hat man eine Kette von Aussagen, die *zwingend* und Schritt für Schritt auseinander hervorgehen, so nennt man das eine *Deduktion*. Das Wort Deduktion kommt aus dem Lateinischen und bedeutet „ableiten“ (von Wasser) oder „fortführen“. Die initiale Aussage dieser Kette wird *Hypothese* und die letzte Aussage wird *Konklusion* genannt. Ist die Hypothese richtig, so muss auch die Konklusion richtig sein. Dieses Vorgehen ist als *deduktive Methode* bekannt und wird in der Mathematik rigoros angewendet, d.h. *niemals* werden Aussagen oder Definitionen verwendet, die vorher nicht klar belegt wurden und jeder Schritt in der Kette muss *genauestens* begründet werden. Dieses Vorgehen ist am Anfang ungewohnt, da viele Menschen gewohnt sind einfach (unbelegte) Behauptungen aufzustellen und daraus dann (die gewünschten) Schlüsse mit Hilfe von unpräzisen Argumenten zu ziehen. Im starken Kontrast dazu, stellt die deduktive Methode sicher, dass keine falschen Aussagen gemacht werden können (zumindest so lange die Deduktion fehlerfrei ist). Folgendes Beispiel gibt einen erste Idee für das Vorgehen der Mathematik:

Wir wissen, dass „Sokrates ist ein Mensch“ eine wahre Aussage ist. Dies stellt unsere Hypothese dar. Nun wissen wir auch „alle Menschen sind sterblich“. Wir können daraus unsere Konklusion folgern, dass „Sokrates ist sterblich“, wodurch wir eine neue Erkenntnis gewonnen haben. Weiterhin erhalten wir ganz einfach die Einsicht, dass Sokrates sterben wird.

Das Vorgehen keine mehrdeutigen Definitionen zu verwenden und jeden Argumentationsschritt klar und logisch zu begründen wird als (mathematische) *Strenge* bezeichnet und wird neben der Mathematik auch in der Informatik und anderen auf der Mathematik basierenden Wissenschaften verwendet. So kommentiert der berühmte Mathematiker David Hilbert im Jahr 1900 auf einem Kongress in Paris in seinem Vortrag „Mathematische Probleme“ das mathematische Vorgehen wie folgt:

> „. . . welche berechtigten allgemeinen Forderungen an die Lösung eines mathematischen Problems zu stellen sind: ich meine vor Allem die, daß es gelingt, die Richtigkeit der Antwort durch eine endliche Anzahl von Schlüssen darzuthun und zwar

[7] Der Autor dieses Dokuments verwendet Teile seiner alten Vorlesungsmitschriften immer noch. Diese sind zu einer wichtigen Quelle geworden, da sich dort Hinweise über damalige Verständnisprobleme finden. Tipp: Heben Sie sich Ihre eigenen Ausarbeitungen für später auf, es lohnt sich!

> auf Grund einer endlichen Anzahl von Voraussetzungen, welche in der Problemstellung liegen und die jedesmal genau zu formulieren sind. Diese Forderung der logischen Deduktion mittelst einer endlichen Anzahl von Schlüssen ist nichts anderes als die Forderung der Strenge in der Beweisführung. In der That die Forderung der Strenge, die in der Mathematik bekanntlich von sprichwörtlicher Bedeutung geworden ist, entspricht einem allgemeinen philosophischen Bedürfnis unseres Verstandes und andererseits kommt durch ihre Erfüllung allein erst der gedankliche Inhalt und die Fruchtbarkeit des Problems zur vollen Geltung. . . . "

In der *Erkenntnistheorie* sind auch andere Methoden des Erkenntnisgewinns bekannt, wie z.B. die Induktion. Hier versucht man bestehende Tatsachen zu verallgemeinern, was (auch) zu nicht korrekten Aussagen führen *kann*, die man dann genauer untersuchen muss. Dies könnte beispielsweise so aussehen: „Alle Fahrzeuge auf dem Parkplatz sind Autos" und „Alle Fahrzeuge sind rot". Daraus folgern wir „Alle Autos sind rot". Diese Vorgehensweise spielt hier (erst einmal) keine oder eine eher untergeordnete Rolle.

Weiterhin zeigt sich, dass die mathematischen Beweismethodiken (Deduktionen) oft eng mit Konzepten aus der Softwareerstellung verknüpft sind, d.h. mathematisches Denken hat in der Praxis für praktische Informatiker einen großen Wert. Ein schönes Beispiel hierfür ist die starke Ähnlichkeit von Induktionsbeweisen und rekursiven Algorithmen. Dies zeigt, dass sich eine Einarbeitung in mathematische Denkweisen auch für Informatiker lohnt, auch wenn ein direkter Zusammenhang vielleicht nicht sofort ersichtlich ist.

Das hier vorliegende Dokument ist ein (sehr) kurzer Crashkurs mit einigen Beispielen, der evtl. Defizite aus der Schule ausgleichen helfen soll. Um Hinweise zur Ergänzung dieses Skriptes wird dringend gebeten, denn Vorlesungsskripte werden für Sie (und nicht für den/die Dozenten) erstellt, d.h. es ist in Ihrem eigenen Interesse, dass Verbesserungen und Erweiterungen eingebaut werden. Beachten Sie auch, dass die Inhalte der Vorlesungen von den Inhalten der jeweiligen Skripten abweichen können! Es kann insbesondere Inhalte in einem Skript geben, die nicht prüfungsrelevant sind, und prüfungsrelevante Inhalte, die nicht im Skript zur Vorlesung enthalten sind.

2 Grundlagen und Schreibweisen

2.1 Mengen

Es ist sehr schwer den fundamentalen Begriff der Menge mathematisch exakt zu definieren. Aus diesem Grund soll uns hier die von Cantor im Jahr 1895 gegebene Erklärung genügen, da sie für unsere Zwecke völlig ausreichend ist:

Definition 1 (Georg Cantor ([Can95])): *Unter einer ‚*Menge*' verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objecten m unsrer Anschauung oder unseres Denkens (welche die ‚*Elemente*' von M genannt werden) zu einem Ganzen*[8].

Für die Formulierung „genau dann wenn" verwenden wir im Folgenden die Abkürzung gdw. um Schreibarbeit zu sparen.

2.1.1 Die Elementbeziehung und die Enthaltenseinsrelation

Sehr oft werden einfache große lateinische Buchstaben wie N, M, A, B oder C als Symbole für Mengen verwendet und kleine Buchstaben für die Elemente einer Menge. Mengen von Mengen notiert man gerne mit kalligraphischen Buchstaben wie $\mathcal{A}$, $\mathcal{B}$ oder $\mathcal{M}$.

Definition 2: *Sei M eine beliebige Menge, dann ist*

- $a \in M$ *gdw. a ist ein Element der Menge M,*
- $a \notin M$ *gdw. a ist kein Element der Menge M,*
- $M \subseteq N$ *gdw. aus $a \in M$ folgt $a \in N$ (M ist* Teilmenge *von N),*
- $M \not\subseteq N$ *gdw. es gilt nicht $M \subseteq N$. Gleichwertig: es gibt ein $a \in M$ mit $a \notin N$ (M ist keine Teilmenge von N) und*
- $M \subset N$ *gdw. es gilt $M \subseteq N$ und $M \neq N$ (M ist echte Teilmenge von N).*

Statt $a \in M$ schreibt man auch $M \ni a$, was in einigen Fällen zu einer deutlichen Vereinfachung der Notation führt.

2.1.2 Definition spezieller Mengen

Spezielle Mengen können auf verschiedene Art und Weise definiert werden, wie z.B.

- durch Angabe von Elementen: So ist $\{a_1, \ldots, a_n\}$ die Menge, die aus den Elementen a_1, $\ldots$, a_n besteht, oder
- durch eine Eigenschaft E: Dabei ist $\{a \mid E(a)\}$ die Menge aller Elemente a, die die Eigenschaft[9] E besitzen.

Alternativ zu der Schreibweise $\{a \mid E(a)\}$ wird auch oft $\{a\colon E(a)\}$ verwendet.

[8] Diese Zitat entspricht der originalen Schreibweise von Cantor.
[9] Die Eigenschaft E kann man dann auch als *Prädikat* bezeichnen.

Beispiel 3: *Mengen, die durch die Angabe von Elementen definiert sind:*

- $\mathbb{B} =_{\text{def}} \{0, 1\}$
- $\mathbb{N} =_{\text{def}} \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$ *(Menge der* natürlichen Zahlen*)*
- $\mathbb{Z} =_{\text{def}} \{\dots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \dots\}$ *(Menge der* ganzen Zahlen*)*
- $2\mathbb{Z} =_{\text{def}} \{0, \pm 2, \pm 4, \pm 6, \pm 8, \dots\}$ *(Menge der geraden ganzen Zahlen)*
- $\mathbb{P} =_{\text{def}} \{2, 3, 5, 7, 11, 13, 17, 19, \dots\}$ *(Menge der* Primzahlen*)*

Beispiel 4: *Mengen, die durch eine Eigenschaft E definiert sind:*

- $\{n \mid n \in \mathbb{N}$ *und* n *ist durch* 3 *teilbar*$\}$
- $\{n \mid n \in \mathbb{N}$ *und* n *ist Primzahl und* $n \leq 40\}$
- $\emptyset =_{\text{def}} \{a \mid a \neq a\}$ *(die leere Menge)*

Aus Definition 2 ergibt sich, dass die leere Menge (Schreibweise: $\emptyset$) Teilmenge jeder Menge ist. Dabei ist zu beachten, dass $\{\emptyset\} \neq \emptyset$ gilt, denn $\{\emptyset\}$ enthält *ein* Element (die leere Menge) und $\emptyset$ enthält *kein* Element.

2.1.3 Operationen auf Mengen

Definition 5: *Seien A und B beliebige Mengen, dann ist*

- $A \cap B =_{\text{def}} \{a \mid a \in A$ *und* $a \in B\}$ *(*Schnitt *von A und B),*
- $A \cup B =_{\text{def}} \{a \mid a \in A$ *oder* $a \in B\}$ *(*Vereinigung *von A und B),*
- $A \setminus B =_{\text{def}} \{a \mid a \in A$ *und* $a \notin B\}$ *(*Differenz *von A und B),*
- $\overline{A} =_{\text{def}} M \setminus A$ *(*Komplement *von A bezüglich einer festen Grundmenge M) und*
- $\mathcal{P}(A) =_{\text{def}} \{B \mid B \subseteq A\}$ *(*Potenzmenge *von A).*

Zwei Mengen A und B mit $A \cap B = \emptyset$ nennt man disjunkt.

Beispiel 6: *Sei $A = \{2, 3, 5, 7\}$ und $B = \{1, 2, 4, 6\}$, dann ist $A \cap B = \{2\}$, $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$ und $A \setminus B = \{3, 5, 7\}$. Wählen wir als Grundmenge die natürlichen Zahlen, also $M = \mathbb{N}$, dann ist $\overline{A} = \{n \in \mathbb{N} \mid n \neq 2$ und $n \neq 3$ und $n \neq 5$ und $n \neq 7\} = \{1, 4, 6, 8, 9, 10, 11, \dots\}$.*

Als Potenzmenge der Menge A ergibt sich die folgende Menge von Mengen von natürlichen Zahlen $\mathcal{P}(A) = \{\emptyset, \{2\}, \{3\}, \{5\}, \{7\}, \{2, 3\}, \{2, 5\}, \{2, 7\}, \{3, 5\}, \{3, 7\}, \{5, 7\}, \{2, 3, 5\}, \{2, 3, 7\}, \{2, 5, 7\}, \{3, 5, 7\}, \{2, 3, 5, 7\}\}$.

Offensichtlich ist die Menge $\{0, 2, 4, 6, 8, \dots\}$ der geraden natürlichen Zahlen und die Menge $\{1, 3, 5, 7, 9, \dots\}$ der ungeraden natürlichen Zahlen disjunkt.

2.1.4 Gesetze für Mengenoperationen

Für die klassischen Mengenoperationen gelten die folgenden Beziehungen:

$A \cap B$	$=$	$B \cap A$	Kommutativgesetz für den Schnitt
$A \cup B$	$=$	$B \cup A$	Kommutativgesetz für die Vereinigung
$A \cap (B \cap C)$	$=$	$(A \cap B) \cap C$	Assoziativgesetz für den Schnitt
$A \cup (B \cup C)$	$=$	$(A \cup B) \cup C$	Assoziativgesetz für die Vereinigung
$A \cap (B \cup C)$	$=$	$(A \cap B) \cup (A \cap C)$	Distributivgesetz
$A \cup (B \cap C)$	$=$	$(A \cup B) \cap (A \cup C)$	Distributivgesetz
$A \cap A$	$=$	A	Duplizitätsgesetz für den Schnitt
$A \cup A$	$=$	A	Duplizitätsgesetz für die Vereinigung
$A \cap (A \cup B)$	$=$	A	Absorptionsgesetz
$A \cup (A \cap B)$	$=$	A	Absorptionsgesetz
$\overline{A \cap B}$	$=$	$(\overline{A} \cup \overline{B})$	de-Morgansche Regel
$\overline{A \cup B}$	$=$	$(\overline{A} \cap \overline{B})$	de-Morgansche Regel
$\overline{\overline{A}}$	$=$	A	Gesetz des doppelten Komplements

Die „de-Morganschen Regeln“ wurden nach dem englischen Mathematiker Augustus De Morgan[10] benannt.

Als Abkürzung schreibt man statt $X_1 \cup X_2 \cup \cdots \cup X_n$ (bzw. $X_1 \cap X_2 \cap \cdots \cap X_n$) einfach $\bigcup_{i=1}^{n} X_i$ (bzw. $\bigcap_{i=1}^{n} X_i$). Möchte man alle Mengen X_i mit $i \in \mathbb{N}$ schneiden (bzw. vereinigen), so schreibt man kurz $\bigcap_{i \in \mathbb{N}} X_i$ (bzw. $\bigcup_{i \in \mathbb{N}} X_i$).

Oft benötigt man eine Verknüpfung von zwei Mengen, eine solche Verknüpfung wird allgemein wie folgt definiert:

Definition 7 („Verknüpfung von Mengen“): *Seien A und B zwei Mengen und „$\odot$“ eine beliebige Verknüpfung zwischen den Elementen dieser Mengen, dann definieren wir*

$$A \odot B =_{\text{def}} \{a \odot b \mid a \in A \text{ und } b \in B\}.$$

Beispiel 8: *Die Menge $3\mathbb{Z} = \{0, \pm 3, \pm 6, \pm 9, \dots\}$ enthält alle Vielfachen[11] von 3, damit ist $3\mathbb{Z} + \{1\} = \{1, 4, -2, 7, -5, 10, -8, \dots\}$. Die Menge $3\mathbb{Z} + \{1\}$ schreibt man kurz oft auch als $3\mathbb{Z} + 1$, wenn klar ist, was mit dieser Abkürzung gemeint ist.*

2.1.5 Tupel (Vektoren) und das Kreuzprodukt

Seien $A, A_1, \dots, A_n$ im folgenden Mengen, dann bezeichnet

- $(a_1, \dots, a_n) =_{\text{def}}$ die Elemente $a_1, \dots, a_n$ in genau dieser festgelegten *Reihenfolge* und z.B. $(3, 2) \neq (2, 3)$. Wir sprechen von einem n-Tupel.
- $A_1 \times A_2 \times \cdots \times A_n =_{\text{def}} \{(a_1, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}$ (Kreuzprodukt der Mengen $A_1, A_2, \dots, A_n$),
- $A^n =_{\text{def}} \underbrace{A \times A \times \cdots \times A}_{n\text{-mal}}$ (n-faches Kreuzprodukt der Menge A) und
- speziell gilt $A^1 = \{(a) \mid a \in A\}$.

[10] *1806 in Madurai, Tamil Nadu, Indien - †1871 in London, England

[11] Eigentlich müsste man statt $3\mathbb{Z}$ die Notation $\{3\}\mathbb{Z}$ verwenden. Dies ist allerdings unüblich.

Wir nennen 2-Tupel auch *Paare*, 3-Tupel auch *Tripel*, 4-Tupel auch *Quadrupel* und 5-Tupel *Quintupel*. Bei n-Tupeln ist, im Gegensatz zu Mengen, eine Reihenfolge vorgegeben, d.h. es gilt z.B. immer $\{a, b\} = \{b, a\}$, aber im Allgemeinen $(a, b) \neq (b, a)$.

Beispiel 9: *Sei $A = \{1, 2, 3\}$ und $B = \{a, b, c\}$, dann bezeichnet das Kreuzprodukt von A und B die Menge von Paaren $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$.*

2.1.6 Die Anzahl von Elementen in Mengen

Sei A eine Menge, die endlich viele Elemente[12] enthält, dann ist

$$\#A =_{\text{def}} \text{Anzahl der Elemente in der Menge } A.$$

Beispielsweise ist $\#\{4, 7, 9\} = 3$. Mit dieser Definition gilt

- $\#(A^n) = (\#A)^n$,
- $\#\mathcal{P}(A) = 2^{\#A}$,
- $\#A + \#B = \#(A \cup B) + \#(A \cap B)$ und
- $\#A = \#(A \setminus B) + \#(A \cap B)$.

2.2 Relationen und Funktionen

2.2.1 Eigenschaften von Relationen

Seien $A_1, \ldots, A_n$ beliebige Mengen, dann ist R eine *n-stellige Relation* gdw. $R \subseteq A_1 \times A_2 \times \cdots \times A_n$. Eine zweistellige Relation nennt man auch *binäre Relation*. Oft werden auch Relationen $R \subseteq A^n$ betrachtet, diese bezeichnet man dann als n-stellige Relation über der Menge A.

Definition 10: *Sei R eine zweistellige Relation über A, dann ist R*

- reflexiv *gdw. $(a, a) \in R$ für alle $a \in A$,*
- symmetrisch *gdw. aus $(a, b) \in R$ folgt $(b, a) \in R$,*
- antisymmetrisch *gdw. aus $(a, b) \in R$ und $(b, a) \in R$ folgt $a = b$,*
- transitiv *gdw. aus $(a, b) \in R$ und $(b, c) \in R$ folgt $(a, c) \in R$ und*
- linear *gdw. es gilt immer $(a, b) \in R$ oder $(b, a) \in R$.*

Definition 11: *Sei R eine zweistellige Relation, dann*

- *heißt R* Halbordnung *gdw. R ist reflexiv, antisymmetrisch und transitiv,*
- Ordnung *gdw. R ist eine lineare Halbordnung und*
- Äquivalenzrelation *gdw. R reflexiv, transitiv und symmetrisch ist.*

Beispiel 12: *Die Teilmengenrelation „$\subseteq$“ auf allen Teilmengen von $\mathbb{Z}$ ist eine Halbordnung, aber keine Ordnung.*

[12] Solche Mengen werden als *endliche Mengen* bezeichnet.

Beispiel 13: *Wir schreiben $a \equiv b \bmod n$, falls es eine ganze Zahl q gibt, für die $a - b = qn$ gilt. Für $n \geq 2$ ist die Relation $R_n(a,b) =_{\text{def}} \{(a,b) \mid a \equiv b \bmod n\} \subseteq \mathbb{Z}^2$ eine Äquivalenzrelation.*

2.2.2 Eigenschaften von Funktionen

Seien A und B beliebige Mengen. f ist eine *Funktion* von A nach B (Schreibweise: $f\colon A \to B$) gdw. $f \subseteq A \times B$ und für jedes $a \in A$ gibt es *höchstens* ein $b \in B$ mit $(a,b) \in f$. Ist also $(a,b) \in f$, so schreibt man $f(a) = b$. Ebenfalls gebräuchlich ist die Notation $a \mapsto b$.

Bemerkung 14: *Unsere Definition von Funktion umfasst auch mehrstellige Funktionen. Seien C und B Mengen und $A = C^n$ das n-fache Kreuzprodukt von C. Die Funktion $f\colon A \to B$ ist dann eine n-stellige Funktion, denn sie bildet n-Tupel aus C^n auf Elemente aus B ab.*

Definition 15: *Sei f eine n-stellige Funktion. Möchte man die Funktion f benutzen, aber keine Namen für die Argumente vergeben, so schreibt man auch*

$$f(\underbrace{\cdot, \cdot, \ldots, \cdot}_{n\text{-mal}})$$

Ist also der Namen des Arguments einer einstelligen Funktion $g(x)$ für eine Betrachtung unwichtig, so kann man $g(\cdot)$ schreiben, um anzudeuten, dass g einstellig ist, ohne dies weiter zu erwähnen.

Definition 16: *Sei nun $R \subseteq A_1 \times A_2 \times \cdots \times A_n$ eine n-stellige Relation, dann definieren wir eine Funktion $P_R^n\colon A_1 \times A_2 \times \cdots \times A_n \to \{0,1\}$ wie folgt:*

$$P_R^n(x_1, \ldots, x_n) =_{\text{def}} \begin{cases} 1, & \text{falls } (x_1, \ldots, x_n) \in R \\ 0, & \text{sonst} \end{cases}$$

Eine solche n-stellige Funktion, die „anzeigt", ob ein Element aus $A_1 \times A_2 \times \cdots \times A_n$ entweder zu R gehört oder nicht, nennt man (n-stelliges) Prädikat.

Beispiel 17: *Sei $\mathbb{P} =_{\text{def}} \{n \in \mathbb{N} \mid n \text{ ist Primzahl}\}$, dann ist $\mathbb{P}$ eine 1-stellige Relation über den natürlichen Zahlen. Das Prädikat $P_{\mathbb{P}}^1(n)$ liefert für eine natürliche Zahl n genau dann 1, wenn n eine Primzahl ist.*

Ist für ein Prädikat P_R^n sowohl die Relation R als auch die Stelligkeit n aus dem Kontext klar, dann schreibt man auch kurz P oder verwendet das Relationensymbol R als Notation für das Prädikat P_R^n.

Nun legen wir zwei spezielle Funktionen fest, die oft sehr hilfreich sind:

Definition 18: *Sei $\alpha \in \mathbb{R}$ eine beliebige reelle Zahl, dann gilt*

- $\lceil \alpha \rceil =_{\text{def}}$ *die kleinste ganze Zahl, die größer oder gleich α ist ($\triangleq$ „Aufrunden")*
- $\lfloor \alpha \rfloor =_{\text{def}}$ *die größte ganze Zahl, die kleiner oder gleich α ist ($\triangleq$ „Abrunden")*

Definition 19: *Für eine beliebige Funktion f legen wir fest:*

- *Der* Definitionsbereich *von f ist $D_f =_{\text{def}} \{a \mid$ es gibt ein b mit $f(a) = b\}$.*
- *Der* Wertebereich *von f ist $W_f =_{\text{def}} \{b \mid$ es gibt ein a mit $f(a) = b\}$.*
- *Die Funktion $f\colon A \to B$ ist* total *gdw. $D_f = A$.*
- *Die Funktion $f\colon A \to B$ heißt* surjektiv *gdw. $W_f = B$.*

- *Die Funktion f heißt* injektiv *(oder eineindeutig[13]) gdw. immer wenn $f(a_1) = f(a_2)$ gilt auch $a_1 = a_2$.*
- *Die Funktion f heißt* bijektiv *gdw. f ist injektiv und surjektiv.*

Mit Hilfe der Kontraposition (siehe Abschnitt 4.1.1) kann man für die Injektivität alternativ auch zeigen, dass immer wenn $a_1 \neq a_2$, dann muss auch $f(a_1) \neq f(a_2)$ gelten.

Beispiel 20: *Sei die Funktion $f : \mathbb{N} \to \mathbb{Z}$ durch $f(n) = (-1)^n \lceil \frac{n}{2} \rceil$ gegeben. Die Funktion f ist surjektiv, denn $f(0) = 0, f(1) = -1, f(2) = 1, f(3) = -2, f(4) = 2, \ldots$, d.h. die ungeraden natürlichen Zahlen werden auf die negativen ganzen Zahlen abgebildet, die geraden Zahlen aus $\mathbb{N}$ werden auf die positiven ganzen Zahlen abgebildet und deshalb ist $W_f = \mathbb{Z}$.*

Weiterhin ist f auch injektiv, denn aus[14] $(-1)^{a_1} \lceil \frac{a_1}{2} \rceil = (-1)^{a_2} \lceil \frac{a_2}{2} \rceil$ folgt, dass entweder a_1 und a_2 gerade oder a_1 und a_2 ungerade, denn sonst würden auf der linken und rechten Seite der Gleichung unterschiedliche Vorzeichen auftreten. Ist a_1 gerade und a_2 gerade, dann gilt $\lceil \frac{a_1}{2} \rceil = \lceil \frac{a_2}{2} \rceil$ und auch $a_1 = a_2$. Sind a_1 und a_2 ungerade, dann gilt $-\lceil \frac{a_1}{2} \rceil = -\lceil \frac{a_2}{2} \rceil$, woraus auch folgt, dass $a_1 = a_2$. Damit ist die Funktion f bijektiv. Weiterhin ist f auch total, d.h. $D_f = \mathbb{N}$.

Definition 21: *Unter einem n-stelligen* Operator *f (auf der Menge Y) versteht man in der Mathematik eine Funktion der Form $f : Y^n \to Y$. Einfache Beispiele für zweistellige Operatoren sind der Additions- oder Multiplikationsoperator.*

2.2.3 Hüllenoperatoren

Definition 22: *Sei X eine Menge. Ein einstelliger Operator $\Psi : \mathcal{P}(X) \to \mathcal{P}(X)$ heißt* Hüllenoperator*, wenn er die folgenden drei Eigenschaften erfüllt:*

Einbettung: *für alle $A \in \mathcal{P}(X)$ gilt $A \subseteq \Psi(A)$*

Monotonie: *für alle $A, B \in \mathcal{P}(X)$ mit $A \subseteq B$ folgt $\Psi(A) \subseteq \Psi(B)$*

Abgeschlossenheit: *für alle $A \in \mathcal{P}(X)$ gilt $\Psi(\Psi(A)) = \Psi(A)$*

Aufgrund der Monotonieeigenschaft eines Hüllenoperators kann man bei der Abgeschlossenheit die Eigenschaft $\Psi(\Psi(A)) = \Psi(A)$ auch durch $\Psi(\Psi(A)) \subseteq \Psi(A)$ ersetzen. In der Informatik spielen Hüllenoperatoren eine große Rolle. Gute Beispiele hierfür sind z.B. die *transitive Hülle* (vgl. Computergraphik), die Kleene-Hülle (vgl. Formale Sprachen) oder der Abschluss einer Komplexitätsklasse unter Schnitt oder Vereinigung.

2.2.4 Permutationen

Sei S eine beliebige endliche Menge, dann heißt eine bijektive Funktion π der Form $\pi : S \to S$ *Permutation*. Das bedeutet, dass die Funktion π Elemente aus S wieder auf Elemente aus S abbildet, wobei für jedes $b \in S$ ein $a \in S$ mit $f(a) = b$ existiert (Surjektivität) und falls $f(a_1) = f(a_2)$ gilt, dann ist $a_1 = a_2$ (Injektivität).

Bemerkung 23: *Man kann den Permutationsbegriff auch auf unendliche Mengen erweitern, aber besonders häufig werden in der Informatik* Permutationen von endlichen Mengen *benötigt. Aus diesem Grund sollen hier nur endliche Mengen S betrachtet werden.*

[13] Achtung: Dieser Begriff wird manchmal unterschiedlich, je nach Autor, in den Bedeutungen „bijektiv“ oder „injektiv“ verwendet.

[14] Für die Definition der Funktion $\lceil \cdot \rceil$ siehe Definition 18.

Sei nun $S = \{1, \ldots, n\}$ (eine endliche Menge) und $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Permutation. Permutationen dieser Art kann man sehr anschaulich mit Hilfe einer Matrix aufschreiben:

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

Durch diese Notation wird klar, dass das Element 1 der Menge S durch das Element $\pi(1)$ ersetzt wird, das Element 2 wird mit $\pi(2)$ vertauscht und allgemein das Element i durch $\pi(i)$ für $1 \leq i \leq n$. In der zweiten Zeile dieser Matrixnotation findet sich also *jedes* (Surjektivität) Element der Menge S genau *einmal* (Injektivität).

Beispiel 24: *Sei $S = \{1, \ldots, 3\}$ eine Menge mit drei Elementen. Dann gibt es, wie man ausprobieren kann, genau 6 Permutationen von S:*

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad \pi_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \quad \pi_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$
$$\pi_4 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \quad \pi_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \quad \pi_6 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Satz 25: *Sei S eine endliche Menge mit $n = |S|$, dann gibt es genau $n!$ (Fakultät) verschiedene Permutationen von S.*

Beweis: Jede Permutation π der Menge S von n Elementen kann als Matrix der Form

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

aufgeschrieben werden. Damit ergibt sich die Anzahl der Permutationen von S durch die Anzahl der verschiedenen zweiten Zeilen solcher Matrizen. In jeder solchen Zeile muss jedes der n Elemente von S genau einmal vorkommen, da π eine bijektive Abbildung ist, d.h. wir haben für die erste Position der zweiten Zeile der Matrixdarstellung genau n verschiedene Möglichkeiten, für die zweite Position noch $n-1$ und für die dritte noch $n-2$. Für die n-te Position bleibt nur noch 1 mögliches Element aus S übrig[15]. Zusammengenommen haben wir also $n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1 = n!$ verschiedene mögliche Permutationen der Menge S. #

2.3 Summen und Produkte

2.3.1 Summen

Zur abkürzenden Schreibweise verwendet man für Summen das Summenzeichen $\sum$. Dabei ist

$$\sum_{i=1}^{n} a_i =_{\text{def}} a_1 + a_2 + \cdots + a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\sum_{i=1}^{n} a_i = n \cdot a$ (Summe gleicher Summanden).
- $\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten einer Summe).

[15] Dies kann man sich auch als die Anzahl der verschiedenen Möglichkeiten vorstellen, die bestehen, wenn man aus einer Urne mit n numerierten Kugeln alle Kugeln *ohne* Zurücklegen nacheinander zieht.

- $\sum_{i=1}^{n}(a_i + b_i + c_i + \dots) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i + \sum_{i=1}^{n} c_i + \dots$ (Addition von Summen).
- $\sum_{i=1}^{n} a_i = \sum_{i=l}^{n+l-1} a_{i-l+1}$ und $\sum_{i=l}^{n} a_i = \sum_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Summen).
- $\sum_{i=1}^{n}\sum_{j=1}^{m} a_{i,j} = \sum_{j=1}^{m}\sum_{i=1}^{n} a_{i,j}$ (Vertauschen der Summationsfolge).

Manchmal verwendet man keine Laufindizes an ein Summenzeichen, sondern man beschreibt (durch ein Prädikat) welche Zahlen aufsummiert werden sollen. So kann man eine Funktion definieren, die die Summe aller Teiler einer natürlichen Zahl liefert:

$$\sigma(n) =_{\text{def}} \sum_{\substack{t \le n \\ t \text{ teilt } n}} t$$

2.3.2 Produkte

Zur abkürzenden Schreibweise verwendet man für Produkte das Produktzeichen $\prod$. Dabei ist

$$\prod_{i=1}^{n} a_i =_{\text{def}} a_1 \cdot a_2 \cdot \ldots \cdot a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \le i \le n$, dann gilt $\prod_{i=1}^{n} a_i = a^n$ (Produkt gleicher Faktoren).
- $\prod_{i=1}^{n}(ca_i) = c^n \prod_{i=1}^{n} a_i$ (Vorziehen von konstanten Faktoren)
- $\prod_{i=1}^{n} a_i = \prod_{i=1}^{m} a_i \cdot \prod_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten in Teilprodukte).
- $\prod_{i=1}^{n}(a_i \cdot b_i \cdot c_i \cdot \ldots) = \prod_{i=1}^{n} a_i \cdot \prod_{i=1}^{n} b_i \cdot \prod_{i=1}^{n} c_i \cdot \ldots$ (Das Produkt von Produkten).
- $\prod_{i=1}^{n} a_i = \prod_{i=l}^{n+l-1} a_{i-l+1}$ und $\prod_{i=l}^{n} a_i = \prod_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Produkten).
- $\prod_{i=1}^{n}\prod_{j=1}^{m} a_{i,j} = \prod_{j=1}^{m}\prod_{i=1}^{n} a_{i,j}$ (Vertauschen der Reihenfolge bei Doppelprodukten).

Ähnlich wie bei Summen kann man bei Produkten auch ohne Laufindex arbeiten. So ist z.B. die *Eulersche ϕ-Funktion* wie folgt definiert:

$$\phi(n) =_{\text{def}} n \prod_{p|n}(1 - \frac{1}{p})$$

In das Produkt gehen also alle Primteiler p von n mit dem Faktor $1 - 1/p$ ein.

Oft werden Summen- oder Produktsymbole verwendet bei denen der Startindex größer als der Stopindex ist. Solche Summen bzw. Produkte sind „leer“, d.h. es wird nichts summiert bzw. multipliziert. Sind dagegen Start- und Endindex gleich, so tritt nur genau ein Wert auf, d.h. das Summen- bzw. Produktsymbol hat in diesem Fall keine Auswirkung. Es ergeben sich also die folgenden Rechenregeln:

- Seien $n, m \in \mathbb{Z}$ und $n < m$, dann

$$\sum_{i=m}^{n} a_i = 0 \text{ und } \prod_{i=m}^{n} a_i = 1$$

- Sei $n \in \mathbb{Z}$, dann

$$\sum_{i=n}^{n} a_i = a_n = \prod_{i=n}^{n} a_i$$

Beispiel 26: *Die folgende Identität wird Euler zugeschrieben. Erstaunlicherweise kann man damit eine Summe von Kehrwerten von Potenzen mit einem Produkt von Primzahlen in Verbindung bringen.*

$$\sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p} \frac{1}{1 - \frac{1}{p^s}}$$

Diese Identiät kann man Die Summe auf der linken Seite ist auch als Riemannsche Zetafuntion $\zeta(s)$ *bekannt. Erstaunlicherweise gilt dann der folgende Zusammenhang*

$$\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \prod_{p} \frac{1}{1 - \frac{1}{p^2}} = \frac{\pi^2}{6},$$

denn durch diese Gleichung wird die Menge der Primzahlen in Beziehung zur Kreiszahl π gebracht.

2.4 Logarithmieren, Potenzieren und Radizieren

Die Schreibweise a^b ist eine Abkürzung für

$$a^b =_{\text{def}} \underbrace{a \cdot a \cdot \ldots \cdot a}_{b-\text{mal}}$$

und wird als *Potenzierung* bezeichnet. Dabei ist a die *Basis*, b der *Exponent* und a^b die b-te *Potenz* von a. Seien nun $r, s, t \in \mathbb{R}$ und $r, t \geq 0$ durch die folgende Gleichung verbunden:

$$r^s = t.$$

Dann lässt sich diese Gleichung wie folgt umstellen und es gelten die folgenden Rechenregeln:

Logarithmieren	Potenzieren	Radizieren
$\mathbf{s = \log_r t}$	$\mathbf{t = r^s}$	$\mathbf{r = \sqrt[s]{t}}$
i) $\log_r(\frac{u}{v}) = \log_r u - \log_r v$	i) $r^u \cdot r^v = r^{u+v}$	i) $\sqrt[s]{u} \cdot \sqrt[s]{v} = \sqrt[s]{u \cdot v}$
ii) $\log_r(u \cdot v) = \log_r u + \log_r v$	ii) $\frac{r^u}{r^v} = r^{u-v}$	ii) $\frac{\sqrt[s]{u}}{\sqrt[s]{v}} = \sqrt[s]{\left(\frac{u}{v}\right)}$
iii) $\log_r(t^u) = u \cdot \log_r t$	iii) $u^s \cdot v^s = (u \cdot v)^s$	iii) $\sqrt[u]{\sqrt[v]{t}} = \sqrt[u \cdot v]{t}$
iv) $\log_r(\sqrt[u]{t}) = \frac{1}{u} \cdot \log_r t$	iv) $\frac{u^s}{v^s} = \left(\frac{u}{v}\right)^s$	
v) $\frac{\log_r t}{\log_r u} = \log_u t$ (Basiswechsel)	v) $(r^u)^v = r^{u \cdot v}$	

Zusätzlich gilt: Wenn $r > 1$, dann ist $s_1 < s_2$ gdw. $r^{s_1} < r^{s_2}$ (Monotonie).

Da $\sqrt[s]{t} = t^{\left(\frac{1}{s}\right)}$ gilt, können die Gesetze für das Radizieren leicht aus den Potenzierungsgesetzen abgeleitet werden. Weiterhin legen wir spezielle Schreibweisen für die Logarithmen zur Basis 10, e (Eulersche Zahl) und 2 fest: $\lg t =_{\text{def}} \log_{10} t$, $\ln t =_{\text{def}} \log_e t$ und $\text{lb}\, t =_{\text{def}} \log_2 t$.

2.5 Gebräuchliche griechische Buchstaben

In der Informatik, Mathematik und Physik ist es üblich, griechische Buchstaben zu verwenden. Ein Grund hierfür ist, dass es so möglich wird mit einer größeren Anzahl von Unbekannten arbeiten zu können, ohne unübersichtliche und oft unhandliche Indizes benutzen zu müssen.

Kleinbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
α	Alpha	β	Beta	γ	Gamma
δ	Delta	ϕ	Phi	φ	Phi
ξ	Xi	ζ	Zeta	ϵ	Epsilon
θ	Theta	λ	Lambda	π	Pi
σ	Sigma	η	Eta	μ	Mu

Großbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
Γ	Gamma	Δ	Delta	Φ	Phi
Ξ	Xi	Θ	Theta	Λ	Lambda
Π	Pi	Σ	Sigma	Ψ	Psi
Ω	Omega				

3 Einige (wenige) Grundlagen der elementaren Logik

Aussagen sind entweder *wahr* ($\triangleq 1$) oder *falsch* ($\triangleq 0$). So sind die Aussagen

„Wiesbaden liegt am Mittelmeer“ und „$1 = 7$“

sicherlich falsch, wogegen die Aussagen

„Wiesbaden liegt in Hessen“ und „$11 = 11$“

sicherlich wahr sind. Aussagen werden meist durch *Aussagenvariablen* formalisiert, die nur die Werte 0 oder 1 annehmen können. Oft verwendet man auch eine oder mehrere Unbekannte, um eine Aussage zu parametrisieren. So könnte „$P(x)$“ etwa für „Wiesbaden liegt im Bundesland x“ stehen, d.h. „$P(\text{Hessen})$“ wäre wahr, wogegen „$P(\text{Bayern})$“ eine falsche Aussage ist. Solche Konstrukte mit Parameter nennt man auch *Prädikat* oder *Aussageformen*.

Um die Verknüpfung von Aussagen auch formal aufschreiben zu können, werden die folgenden logischen Operatoren verwendet

Symbol	umgangssprachlicher Name	Name in der Logik
$\wedge$	und	Konjunktion
$\vee$	oder	Disjunktion / Alternative
$\neg$	nicht	Negation
$\rightarrow$	folgt	Implikation
$\leftrightarrow$	genau dann wenn (*gdw.*)	Äquivalenz

Zusätzlich werden noch die Quantoren $\exists$ („es existiert“) und $\forall$ („für alle“) verwendet, die z.B. wie folgt gebraucht werden können

$\forall x\colon P(x)$ bedeutet „Für alle x gilt die Aussage $P(x)$“.

$\exists x\colon P(x)$ bedeutet „Es existiert ein x, für das die Aussage $P(x)$ gilt“.

Üblicherweise läßt man sogar den Doppelpunkt weg und schreibt statt $\forall x\colon P(x)$ vereinfachend $\forall x P(x)$. Die Aussageform $P(x)$ wird auch als Prädikat (siehe Definition 16 auf Seite 11) bezeichnet, weshalb man von *Prädikatenlogik* spricht.

Beispiel 27: *Die Aussage „Jede gerade natürliche Zahl kann als Produkt von 2 und einer anderen natürlichen Zahl geschrieben werden“ lässt sich dann wie folgt schreiben*

$$\forall n \in \mathbb{N}\colon ((n \text{ ist gerade}) \rightarrow (\exists m \in \mathbb{N}\colon n = 2 \cdot m))$$

Die folgende logische Formel wird wahr gdw. n eine ungerade natürliche Zahl ist.

$$\exists m \in \mathbb{N}\colon (n = 2 \cdot m + 1)$$

Für die logischen Konnektoren sind die folgenden Wahrheitswertetafeln festgelegt:

p	$\neg p$
0	1
1	0

und

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Jetzt kann man Aussagen auch etwas komplexer verknüpfen:

Beispiel 28: *Nun wird der $\wedge$-Operator verwendet werden. Dazu soll die Aussage „Für alle natürlichen Zahlen n und m gilt, wenn n kleiner gleich m und m kleiner gleich n gilt, dann ist m gleich n“*

$$\forall n, m \in \mathbb{N}\,(((n \leq m) \wedge (m \leq n)) \rightarrow (n = m))$$

Oft benutzt man noch den negierten Quantor $\nexists$ („es existiert kein“).

Cubum autem in duos cubos, aut quadrato-quadratum in duos quadrato-quadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos eiusdem nominis fas est dividere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Beispiel 29 („Großer Satz von Fermat“): *Die Richtigkeit dieser Aussage konnte erst 1994 nach mehr als 350 Jahren von Andrew Wiles und Richard Taylor gezeigt werden:*

$$\forall n \in \mathbb{N}\nexists a, b, c \in \mathbb{N}\,(((n > 2) \wedge (a \cdot b \cdot c \neq 0)) \rightarrow a^n + b^n = c^n)$$

Für den Fall $n = 2$ hat die Gleichung $a^n + b^n = c^n$ unendlich viele ganzzahlige Lösungen (die so genannten Pythagoräischen Zahlentripel) wie z.B. $3^2 + 4^2 = 5^2$. Diese sind seit mehr als 3500 Jahren bekannt und haben z.B. geholfen die Cheops-Pyramide zu bauen.

4 Einige formale Grundlagen von Beweistechniken

Praktisch arbeitende Informatiker glauben oft völlig ohne (formale) Beweistechniken auskommen zu können. Dabei meinen sie sogar, dass formale Beweise keinerlei Berechtigung in der Praxis der Informatik haben und bezeichnen solches Wissen als „in der Praxis irrelevantes Zeug, das nur von und für seltsame Wissenschaftler erfunden wurde“. Studenten in den ersten Semestern unterstellen sogar oft, dass mathematische Grundlagen und Beweistechniken nur als „Filter“ dienen, um die Anzahl der Studenten zu reduzieren. Oft stellen sich beide Gruppen auf den Standpunkt, dass die Korrektheit von Programmen und Algorithmen durch „Lassen wir es doch mal laufen und probieren es aus!“ ($\triangleq$ Testen) belegt werden könne[16]. Diese Einstellung zeigt sich oft auch darin, dass Programme mit Hilfe einer IDE schnell „testweise“ übersetzt werden, in der Hoffnung oder (wesentlich schlimmer) in der Überzeugung, dass jedes übersetzbare Programm ($\triangleq$ syntaktisch korrekt) automatisch auch semantisch korrekt ist. Das diese Einstellung völlig falsch ist zeigen Myriaden in der Praxis auftretende Softwarefehler eindrücklich.

Theoretiker, die sich mit den Grundlagen der Informatik beschäftigen, vertreten oft den Standpunkt, dass die Korrektheit *jedes* Programms rigoros *bewiesen* werden muss. Wahrscheinlich ist die Position zwischen diesen beiden Extremen richtig, denn zum einen ist der formale Beweis von (großen) Programmen oft nicht praktikabel oder aufgrund der enormen Komplexität nicht möglich und zum anderen kann das Testen mit einer (relativ kleinen) Menge von Eingaben sicherlich nicht belegen, dass ein Programm vollständig den Spezifikationen entspricht. Im praktischen Einsatz ist es dann oft mit Eingaben konfrontiert, die zu einer fehlerhaften Reaktion führen oder es sogar abstürzen[17] lassen. Bei einfacher Anwendersoftware sind solche Fehler ärgerlich, aber oft zu verschmerzen. Bei sicherheitskritischer Software (z.B. bei der Regelung von Atomkraftwerken, Airbags und Bremssystemen in Autos, in der Medizintechnik, bei Finanztransaktionssystemen oder bei der Steuerung von Raumsonden) gefährden solche Fehler menschliches Leben oder führen zu extrem hohen finanziellen Verlusten und müssen deswegen unbedingt vermieden werden.

Für den Praktiker bringen Kenntnisse über formale Beweise aber noch andere Vorteile. Viele Beweise beschreiben direkt den zur Lösung benötigten Algorithmus, d.h. eigentlich wird die Richtigkeit einer Aussage durch die (implizite) Angabe eines Algorithmus gezeigt. Aber es gibt noch einen anderen Vorteil. Ist der umzusetzende Algorithmus komplex (z.B. aufgrund einer komplizierten Schleifenstruktur oder einer verschachtelten Rekursion), so ist es unwahrscheinlich, eine korrekte Implementation an den Kunden liefern zu können, ohne die Hintergründe ($\triangleq$ Beweis) verstanden zu haben. All dies zeigt, dass auch ein praktischer Informatiker Einblicke in Beweistechniken haben sollte. Interessanterweise zeigt die persönliche Erfahrung im praktischen Umfeld auch, dass solches (theoretisches) Wissen über die Hintergründe oft zu klarer strukturierten und effizienteren Programmen führt.

Aus diesen Gründen sollen in den folgenden Abschnitten einige grundlegende Beweistechniken mit Hilfe von Beispielen (unvollständig) kurz vorgestellt werden.

4.1 Direkte Beweise

Um einen *direkten Beweis* zu führen, müssen wir, beginnend von einer initialen Aussage ($\triangleq$ Hypothese), durch Angabe einer Folge von (richtigen) Zwischenschritten zu der zu beweisenden Aussage ($\triangleq$ Folgerung) gelangen. Jeder Zwischenschritt ist dabei entweder unmittelbar klar oder muss wieder durch einen weiteren (kleinen) Beweis belegt werden. Dabei müssen nicht alle Schritte völlig formal beschrieben werden, sondern es kommt darauf an, dass sich dem Leser die eigentliche Strategie erschließt.

[16] Diese Bemerkung ist natürlich etwas polemisch, da *richtiges* Testen natürlich die Qualität einer Software verbessert. Weiterhin werden solche Tests systematisch durchgeführt und haben somit eine ganz andere Qualität als „herumprobieren“.

[17] Dies wird eindrucksvoll durch viele Softwarepakete und verbreitete Betriebssysteme im PC-Umfeld belegt.

Satz 30: *Sei $n \in \mathbb{N}$. Falls $n \geq 4$, dann ist $2^n \geq n^2$.*

Wir müssen also, in Abhängigkeit des Parameters n, die Richtigkeit dieser Aussage belegen. Einfaches Ausprobieren ergibt, dass $2^4 = 16 \geq 16 = 4^2$ und $2^5 = 32 \geq 25 = 5^2$, d.h. intuitiv scheint die Aussage richtig zu sein. Wir wollen die Richtigkeit der Aussage nun durch eine Reihe von (kleinen) Schritten belegen:
Beweis:

Wir haben schon gesehen, dass die Aussage für $n = 4$ und $n = 5$ richtig ist. Erhöhen wir n auf $n + 1$, so verdoppelt sich der Wert der linken Seite der Ungleichung von 2^n auf $2 \cdot 2^n = 2^{n+1}$. Für die rechte Seite ergibt sich ein Verhältnis von $(\frac{n+1}{n})^2$. Je größer n wird, desto kleiner wird der Wert $\frac{n+1}{n}$, d.h. der maximale Wert ist bei $n = 4$ mit 1.25 erreicht. Wir wissen $1.25^2 = 1.5625$, d.h. immer wenn wir n um eins erhöhen, verdoppelt sich der Wert der linken Seite, wogegen sich der Wert der rechten Seite um maximal das 1.5625-fache erhöht. Damit muss die linke Seite der Ungleichung immer größer als die rechte Seite sein. #

Dieser Beweis war nur wenig formal, aber sehr ausführlich und wurde am Ende durch das Symbol „#“ markiert. Im Laufe der Zeit hat es sich eingebürgert, das Ende eines Beweises mit einem besonderen Marker abzuschließen. Besonders bekannt ist hier „qed“, eine Abkürzung für die lateinische Floskel „quod erat demonstrandum“, die mit „was zu beweisen war“ übersetzt werden kann. In neuerer Zeit werden statt „qed“ mit der gleichen Bedeutung meist die Symbole „□“ oder „#“ verwendet.

Nun stellt sich die Frage: „Wie formal und ausführlich muss ein Beweis sein?“ Diese Frage kann so einfach nicht beantwortet werden, denn das hängt u.a. davon ab, welche Lesergruppe durch den Beweis von der Richtigkeit einer Aussage überzeugt werden soll und wer den Beweis schreibt. Ein Beweis für ein Übungsblatt sollte auch auf Kleinigkeiten Rücksicht nehmen, wogegen ein solcher Stil für eine wissenschaftliche Zeitschrift vielleicht nicht angebracht wäre, da die potentielle Leserschaft über ganz andere Erfahrungen und viel mehr Hintergrundwissen verfügt.

Nun noch eine Bemerkung zum Thema „Formalismus“. Die menschliche Sprache ist unpräzise, mehrdeutig und Aussagen können oft auf verschiedene Weise interpretiert werden, wie das tägliche Zusammenleben der Menschen und die „Juristerei“ eindrucksvoll demonstriert. Diese Defizite sollen Formalismen[18] ausgleichen, d.h. die Antwort muss lauten: „So viele Formalismen wie notwendig und so wenige wie möglich!“. Durch Übung und Praxis lernt man die Balance zwischen diesen Anforderungen zu halten und es zeigt sich bald, dass „Geübte“ die formale Beschreibung sogar wesentlich leichter verstehen.

Oft kann man andere, schon bekannte, Aussagen dazu verwenden, die Richtigkeit einer neuen (evtl. kompliziert wirkenden) Aussage zu belegen.

Satz 31: *Sei $n \in \mathbb{N}$ die Summe von 4 Quadratzahlen, die größer als 0 sind, dann muss $2^n \geq n^2$ sein.*

Beweis: Die Menge der Quadratzahlen ist $\mathbb{S} = \{0, 1, 4, 9, 16, 25, 36, \dots\}$, d.h. 1 ist die kleinste Quadratzahl, die größer als 0 ist. Damit muss unsere Summe von 4 Quadratzahlen größer oder gleich als 4 sein. Die Aussage folgt direkt aus Satz 30. #

4.1.1 Die Kontraposition

Mit Hilfe von direkten Beweisen haben wir Zusammenhänge der Form „Wenn Aussage H richtig ist, dann folgt daraus die Aussage C“ untersucht. Manchmal ist es schwierig, einen Beweis für eine solchen Zusammenhang zu finden. Völlig gleichwertig ist die Behauptung „Wenn die Aussage C falsch ist, dann ist die Aussage H falsch“ und oft ist eine solche Aussage leichter zu zeigen.

[18] In diesem Zusammenhang sind Programmiersprachen auch Formalismen, die eine präzise Beschreibung von Algorithmen erzwingen und die durch einen Compiler verarbeitet werden können.

Die *Kontraposition* von Satz 30 ist also die folgende Aussage: „Wenn nicht $2^n \geq n^2$, dann gilt nicht $n \geq 4$.“. Das entspricht der Aussage: „Wenn $2^n < n^2$, dann gilt $n < 4$.“, was offensichtlich zu der ursprünglichen Aussage von Satz 30 gleichwertig ist.

Diese Technik ist oft besonders hilfreich, wenn man die Richtigkeit einer Aussage zeigen soll, die aus zwei Teilaussagen zusammengesetzt und die durch ein „genau dann wenn“[19] verknüpft sind. In diesem Fall sind zwei Teilbeweise zu führen, denn zum einen muss gezeigt werden, dass aus der ersten Aussage die zweite folgt und umgekehrt muss gezeigt werden, dass aus der zweiten Aussage die erste folgt.

Satz 32: *Eine natürliche Zahl n ist durch drei teilbar genau dann, wenn die Quersumme ihrer Dezimaldarstellung durch drei teilbar ist.*

Beweis: Für die Dezimaldarstellung von n gilt

$$n = \sum_{i=0}^{k} a_i \cdot 10^i, \text{ wobei } a_i \in \{0, 1, \ldots, 9\} \text{ („Ziffern“) und } 0 \leq i \leq k.$$

Mit $\mathrm{QS}(n)$ wird die Quersumme von n bezeichnet, d.h. $\mathrm{QS}(n) = \sum_{i=0}^{k} a_i$. Mit Hilfe einer einfachen vollständigen Induktion kann man zeigen, dass für jedes $i \geq 0$ ein $b \in \mathbb{N}$ existiert, sodass $10^i = 9b + 1$. Damit gilt $n = \sum_{i=0}^{k} a_i \cdot 10^i = \sum_{i=0}^{k} a_i(9b_i + 1) = \mathrm{QS}(n) + 9\sum_{i=0}^{k} a_i b_i$, d.h. es existiert ein $c \in \mathbb{N}$, so dass $n = \mathrm{QS}(n) + 9c$.

„$\Rightarrow$“: Wenn n durch 3 teilbar ist, dann muss auch $\mathrm{QS}(n) + 9c$ durch 3 teilbar sein. Da $9c$ sicherlich durch 3 teilbar ist, muss auch $\mathrm{QS}(n) = n - 9c$ durch 3 teilbar sein.

„$\Leftarrow$“: Dieser Fall soll durch Kontraposition gezeigt werden. Sei nun n nicht durch 3 teilbar, dann darf $\mathrm{QS}(n)$ nicht durch 3 teilbar sein, denn sonst wäre $n = 9c + \mathrm{QS}(n)$ durch 3 teilbar. #

4.2 Der Ringschluss

Oft findet man mehrere Aussagen, die zueinander äquivalent sind. Ein Beispiel dafür ist Satz 33. Um die Äquivalenz dieser Aussagen zu beweisen, müssten jeweils zwei „genau dann wenn“ Beziehungen untersucht werden, d.h. es werden vier Teilbeweise notwendig. Dies kann mit Hilfe eines so genannten *Ringschlusses* abgekürzt werden, denn es reicht zu zeigen, dass aus der ersten Aussage die zweite folgt, aus der zweiten Aussage die dritte und dass schließlich aus der dritten Aussage wieder die erste folgt. Im Beweis zu Satz 33 haben wir deshalb nur drei anstatt vier Teilbeweise zu führen, was zu einer Arbeitsersparnis führt. Diese Arbeitsersparnis wird um so größer, je mehr äquivalente Aussagen zu untersuchen sind. Dabei ist die Reihenfolge der Teilbeweise nicht wichtig, solange die einzelnen Teile zusammen einen Ring bilden.

Satz 33: *Seien A und B zwei beliebige Mengen, dann sind die folgenden drei Aussagen äquivalent:*

i) $A \subseteq B$

ii) $A \cup B = B$

iii) $A \cap B = A$

Beweis: Im folgenden soll ein Ringschluss verwendet werden, um die Äquivalenz der drei Aussagen zu zeigen:
„i) $\Rightarrow$ ii)“: Da nach Voraussetzung $A \subseteq B$ ist, gilt für jedes Element $a \in A$ auch $a \in B$, d.h. in der Vereinigung $A \cup B$ sind alle Elemente nur aus B, was $A \cup B = B$ zeigt.

[19] Oft wird „genau dann wenn“ durch *gdw.* abgekürzt.

„ii) $\Rightarrow$ iii)“: Wenn $A \cup B = B$ gilt, dann ergibt sich durch Einsetzen und mit den Regeln aus Abschnitt 2.1.4 (Absorptionsgesetz) direkt $A \cap B = A \cap (A \cup B) = A$.

„iii) $\Rightarrow$ i)“: Sei nun $A \cap B = A$, dann gibt es kein Element $a \in A$ für das $a \notin B$ gilt. Dies ist aber gleichwertig zu der Aussage $A \subseteq B$.

Damit hat sich ein Ring von Aussagen „i) $\Rightarrow$ ii)“, „ii) $\Rightarrow$ iii)“ und „iii) $\Rightarrow$ i)“ gebildet, was die Äquivalenz aller Aussagen zeigt. #

4.3 Widerspruchsbeweise

Obwohl die Technik der Widerspruchsbeweise auf den ersten Blick sehr kompliziert erscheint, ist sie meist einfach anzuwenden, extrem mächtig und liefert oft sehr kurze Beweise. Angenommen wir sollen die Richtigkeit einer Aussage „aus der Hypothese H folgt C“ zeigen. Dazu beweisen wir, dass sich ein Widerspruch ergibt, wenn wir, von H und der Annahme, dass C falsch ist, ausgehen. Also war die Annahme falsch, und die Aussage C muss richtig sein.

Anschaulicher wird diese Beweistechnik durch folgendes Beispiel: Nehmen wir einmal an, dass Alice eine bürgerliche Frau ist und deshalb auch keine Krone trägt. Es ist klar, dass jede Königin eine Krone trägt. Wir sollen nun beweisen, dass Alice keine Königin ist. Dazu nehmen wir an, dass Alice eine Königin ist, d.h. Alice trägt eine Krone. Dies ist ein Widerspruch! Also war unsere Annahme falsch, und wir haben gezeigt, dass Alice keine Königin sein kann.

Der Beweis zu folgendem Satz verwendet diese Technik:

Satz 34: *Sei S eine endliche Untermenge einer unendlichen Menge U. Sei T das Komplement von S bzgl. U, dann ist T eine unendliche Menge.*

Beweis: Hier ist unsere Hypothese „S endlich, U unendlich und T Komplement von S bzgl. U“ und unsere Folgerung ist „T ist unendlich“. Wir nehmen also an, dass T eine endliche Menge ist. Da T das Komplement von S ist, gilt $S \cap T = \emptyset$, also ist $\#(S) + \#(T) = \#(S \cap T) + \#(S \cup T) = \#(S \cup T) = n$, wobei n eine Zahl aus $\mathbb{N}$ ist (siehe Abschnitt 2.1.6). Damit ist $S \cup T = U$ eine endliche Menge. Dies ist ein Widerspruch zu unserer Hypothese! Also war die Annahme „T ist endlich“ falsch. #

4.4 Der Schubfachschluss

Der *Schubfachschluss* ist auch als *Dirichlets Taubenschlagprinzip* bekannt. Werden $n > k$ Tauben auf k Boxen verteilt, so gibt es mindestens eine Box in der sich wenigstens zwei Tauben aufhalten. Allgemeiner formuliert sagt das Taubenschlagprinzip, dass wenn n Objekte auf k Behälter aufgeteilt werden, dann gibt es mindestens eine Box die mindestens $\lceil \frac{n}{k} \rceil$ Objekte enthält.

Beispiel 35: *Auf einer Party unterhalten sich 8 Personen ($\triangleq$ Objekte), dann gibt es mindestens einen Wochentag ($\triangleq$ Box) an dem $\lceil \frac{8}{7} \rceil = 2$ Personen aus dieser Gruppe Geburtstag haben.*

4.5 Gegenbeispiele

Im wirklichen Leben wissen wir nicht, ob eine Aussage richtig oder falsch ist. Oft sind wir dann mit einer Aussage konfrontiert, die auf den ersten Blick richtig ist und sollen dazu ein Programm entwickeln. Wir müssen also entscheiden, ob diese Aussage wirklich richtig ist, denn sonst ist evtl. alle Arbeit umsonst und hat hohen Aufwand verursacht. In solchen Fällen kann man versuchen, ein einziges Beispiel dafür zu finden, dass die Aussage falsch ist, um so unnötige Arbeit zu sparen.

Wir zeigen, dass die folgenden Vermutungen falsch sind:

Vermutung 36: *Wenn $p \in \mathbb{N}$ eine Primzahl ist, dann ist p ungerade.*

Gegenbeispiel: Die natürliche Zahl 2 ist eine Primzahl und 2 ist gerade. #

Vermutung 37: *Es gibt keine Zahlen $a, b \in \mathbb{N}$, sodass a mod $b = b$ mod a.*

Gegenbeispiel: Für $a = b = 2$ gilt $a \bmod b = b \bmod a = 0$. #

4.6 Induktionsbeweise und das Induktionsprinzip

Sei nun eine Menge von natürlichen Zahlen X mit den folgenden zwei Eigenschaften gegeben:

(IA) $0 \in X$
(IS) Ist eine beliebige natürliche Zahl n ein Element von X, so ist auch die Zahl $n+1$ ein Element von X.

Man kann sich nun leicht überlegen, dass dann X alle natürlichen Zahlen enthält, d.h. es gilt $X = \mathbb{N}$. Mit Hilfe dieser Beobachtung konstruieren wir eine der nützlichsten Beweismethoden in der Informatik bzw. Mathematik: Das *Induktionsprinzip* bzw. die Methode des *Induktionsbeweises*. Die Idee funktioniert mit folgender Beobachtung:

> Angenommen man kann nachweisen, dass 0 die Eigenschaft E hat[20] (kurz: $E(0)$) und weiterhin, dass wenn n die Eigenschaft E hat, dann gilt auch $E(n+1)$. Ist dies der Fall, so muss jede natürliche Zahl die Eigenschaft E haben.

Im Folgenden wollen wir nachweisen, dass für jedes $n \in \mathbb{N}$ eine bestimmte Eigenschaft E gilt. Wir schreiben also abkürzend $E(n)$ für die Aussage „n besitzt die Eigenschaft E“, d.h. der Schreibweise $E(0)$ drücken wir also aus, dass die erste natürliche Zahl 0 die Eigenschaft E besitzt, dann erhalten wir die folgende Vorgehensweise:

Induktionsprinzip: Es gelten
(IA) $E(0)$
(IS) Für $n \geq 0$ gilt, wenn $E(n)$ korrekt ist, dann ist auch $E(n+1)$ richtig.

Sind diese beiden Aussagen erfüllt, so hat jede natürliche Zahl die Eigenschaft E. Dabei ist **IA** die Abkürzung für *Induktionsanfang* und **IS** ist die Kurzform von *Induktionsschritt*. Die Voraussetzung ($\triangleq$ Hypothese) $E(n)$ ist korrekt für n und wird im Induktionsschritt als *Induktionsvoraussetzung* benutzt (kurz: **IV**). Hat man also den Induktionsanfang und den Induktionsschritt gezeigt, dann ist es anschaulich, dass jede natürliche Zahl die Eigenschaft E haben muss.

Es gibt verschiedene Versionen von Induktionsbeweisen. Die bekannteste Version ist die vollständige Induktion, bei der Aussagen über natürliche Zahlen gezeigt werden.

4.6.1 Die vollständige Induktion

Wie in Piratenfilmen üblich, seien Kanonenkugeln in einer Pyramide mit quadratischer Grundfläche gestapelt. Wir stellen uns die Frage, wieviele Kugeln (in Abhängigkeit von der Höhe) in einer solchen Pyramide gestapelt sind.

Satz 38: *Mit einer quadratische Pyramide aus Kanonenkugeln der Höhe $n \geq 1$ als Munition, können wir $\frac{n(n+1)(2n+1)}{6}$ Schüsse abgeben.*

Beweis: Einfacher formuliert: wir sollen zeigen, dass $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
(IA) Eine Pyramide der Höhe $n = 1$ enthält $\frac{1 \cdot 2 \cdot 3}{6} = 1$ Kugel, d.h. wir haben die Eigenschaft für $n = 1$ verifiziert.

[20] Mit E wird also ein Prädikat oder Aussagenform bezeichnet (siehe Abschnitt 2.1.2)

(IV) Für $k \leq n$ gilt $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$.

(IS) Wir müssen nun zeigen, dass $\sum_{i=1}^{n+1} i^2 = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}$ gilt und dabei muss die Induktionsvoraussetzung $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ benutzt werden. Es ergeben sich die folgenden Schritte:

$$\begin{aligned}
\sum_{i=1}^{n+1} i^2 &= \sum_{i=1}^{n} i^2 + (n+1)^2 \\
&\overset{\text{(IV)}}{=} \frac{n(n+1)(2n+1)}{6} + (n^2+2n+1) \\
&= \frac{2n^3+3n^2+n}{6} + (n^2+2n+1) \\
&= \frac{2n^3+9n^2+13n+6}{6} \\
&= \frac{(n+1)(2n^2+7n+6)}{6} \quad (\star) \\
&= \frac{(n+1)(n+2)(2n+3)}{6} \quad (\star\star) \\
&= \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}
\end{aligned}$$

Die Zeile $\star$ (bzw. $\star\star$) ergibt sich, indem man $2n^3+9n^2+13n+6$ durch $n+1$ teilt (bzw. $2n^2+7n+6$ durch $n+2$). #

Das Induktionsprinzip kann man auch variieren. Dazu geht man davon aus, dass die Eigenschaft E für alle Zahlen $k \leq n$ erfüllt ist (Induktionsvoraussetzung). Obwohl dies auf den ersten Blick wesentlich komplizierter erscheint, vereinfacht dieser Ansatz oft den Beweis:

Verallgemeinertes Induktionsprinzip: Wenn die zwei Aussagen (also Induktionsanfang und Induktionsschritt)
(IA) $E(0)$
(IS) Wenn für alle $0 \leq k \leq n$ die Eigenschaft $E(k)$ gilt, dann ist auch $E(n+1)$ richtig,

gelten, dann haben alle natürliche Zahlen die Eigenschaft E. Damit ist das verallgemeinerte Induktionsprinzip eine Verallgemeinerung des weiter oben vorgestellten Induktionsprinzips, wie das folgende Beispiel veranschaulicht:

Satz 39: *Jede natürliche Zahl $n \geq 2$ läßt sich als Produkt von Primzahlen schreiben.*

Beweis: Das verallgemeinerte Induktionsprinzip wird wie folgt verwendet:
(IA) Offensichtlich ist 2 das Produkt von einer Primzahl.
(IV) Jede natürliche Zahl m mit $2 \leq m \leq n$ kann als Produkt von Primzahlen geschrieben werden.
(IS) Nun wird eine Fallunterscheidung durchgeführt:

i) Sei $n+1$ wieder eine Primzahl, dann ist nichts zu zeigen, da $n+1$ direkt ein Produkt von Primzahlen ist.

ii) Sei $n+1$ keine Primzahl, dann existieren mindestens zwei Zahlen p und q mit $2 \leq p, q < n+1$ und $p \cdot q = n+1$. Nach Induktionsvoraussetzung sind dann p und q wieder als Produkt von Primzahlen darstellbar. Etwa $p = p_1 \cdot p_2 \cdot \ldots \cdot p_s$ und $q = q_1 \cdot q_2 \cdot \ldots \cdot q_t$. Damit ist aber $n+1 = p \cdot q = p_1 \cdot p_2 \cdot \ldots \cdot p_s \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_t$ ein Produkt von Primzahlen. #

Induktionsbeweise treten z.B. bei der Analyse von Programmen immer wieder auf und spielen deshalb für die Informatik eine ganz besonders wichtige Rolle.

4.6.2 Induktive Definitionen

Das Induktionsprinzip kann man aber auch dazu verwenden, (Daten-)Strukturen formal zu spezifizieren. Dies macht diese Technik für Anwendungen in der Informatik besonders interessant. Dazu werden in einem ersten Schritt ($\triangleq$ Induktionsanfang) die „atomaren" Objekte definiert und dann in einem zweiten Schritt die zusammengesetzten Objekte ($\triangleq$ Induktionsschritt). Diese Technik ist als *induktive Definition* bekannt.

Beispiel 40: *Die Menge der* binären Bäume *ist wie folgt definiert:*

(IA) *Ein einzelner Knoten w ist ein* Baum *und w ist die* Wurzel *dieses Baums.*
(IS) *Seien $T_1, T_2, \ldots, T_n$ Bäume mit den Wurzeln $k_1, \ldots, k_n$ und w ein einzelner neuer Knoten. Verbinden wir den Knoten w mit allen Wurzeln $k_1, \ldots, k_n$, dann entsteht ein neuer Baum mit der Wurzel w. Nichts sonst ist ein Baum.*

Beispiel 41: *Die Menge der* arithmetischen Ausdrücke *ist wie folgt definiert:*

(IA) *Jeder Buchstabe und jede Zahl ist ein arithmetischer Ausdruck.*
(IS) *Seien E und F Ausdrücke, so sind auch $E + F$, $E * F$ und $[E]$ Ausdrücke. Nichts sonst ist ein Ausdruck.*

D.h. x, $x + y$, $[2 * x + z]$ sind arithmetische Ausdrücke, aber beispielsweise sind $x+$, yy, $][x + y$ sowie $x + *z$ keine arithmetischen Ausdrücke im Sinn dieser Definition.

Beispiel 42: *Die Menge der* aussagenlogischen Formeln *ist wie folgt definiert:*

(IA) *Jede aussagenlogische Variable $x_1, x_2, x_3, \ldots$ ist eine aussagenlogische Formel.*
(IS) *Seien H_1 und H_2 aussagenlogische Formeln, so sind auch $(H_1 \wedge H_2)$, $(H_1 \vee H_2)$, $\neg H_1$, $(H_1 \leftrightarrow H_2)$, $(H_1 \rightarrow H_2)$ und $(H_1 \oplus H_2)$ aussagenlogische Formeln. Nichts sonst ist eine aussagenlogische Formel.*

Bei diesen Beispielen ahnt man schon, dass solche Techniken zur präzisen und eleganten Definition von Programmiersprachen und Dateiformaten gute Dienste leisten. Es zeigt sich, dass die im Compilerbau verwendeten Chomsky-Grammatiken eine andere Art von induktiven Definitionen darstellen. Darüber hinaus bieten induktive Definitionen noch weitere Vorteile, denn man kann oft relativ leicht Induktionsbeweise konstruieren, die Aussagen über induktiv definierte Objekte belegen / beweisen.

4.6.3 Die strukturelle Induktion

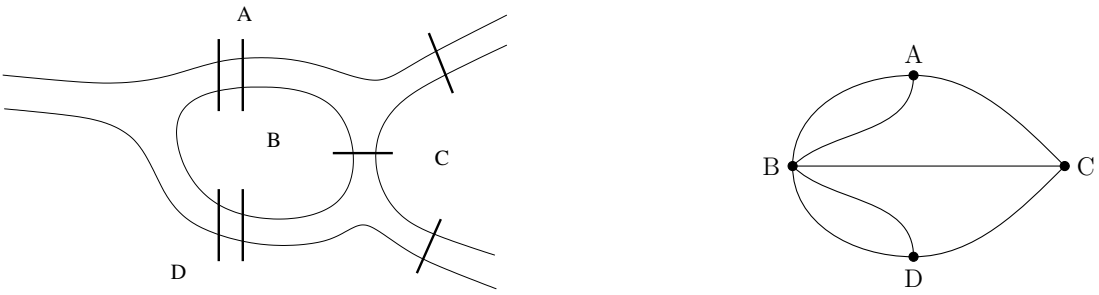
Satz 43: *Die Anzahl der öffnenden Klammern eines arithmetischen Ausdrucks stimmt mit der Anzahl der schließenden Klammern überein.*

Es ist offensichtlich, dass diese Aussage richtig ist, denn in Ausdrücken wie $(x + y)/2$ oder $x + ((y/2) * z)$ muss ja zu jeder öffnenden Klammer eine schließende Klammer existieren. Der nächste Beweis verwendet diese Idee um die Aussage von Satz 43 mit Hilfe einer *strukturellen Induktion* zu zeigen.
Beweis: Wir bezeichnen die Anzahl der öffnenden Klammern eines Ausdrucks E mit $\#_[(E)$ und verwenden die analoge Notation $\#_](E)$ für die Anzahl der schließenden Klammern.
(IA) Die einfachsten Ausdrücke sind Buchstaben und Zahlen. Die Anzahl der öffnenden und schließenden Klammern ist in beiden Fällen gleich 0.
(IV) Sei E ein Ausdruck, dann gilt $\#_[(E) = \#_](E)$.

(IS) Für einen Ausdruck $E+F$ gilt $\#_[(E+F) = \#_[(E)+\#_[(F) \overset{\text{IV}}{=} \#_](E)+\#_](F) = \#_](E+F)$. Völlig analog zeigt man dies für $E * F$. Für den Ausdruck $[E]$ ergibt sich $\#_[([E]) = \#_[(E) + 1 \overset{\text{IV}}{=} \#_](E) + 1 = \#_]([E])$. In jedem Fall ist die Anzahl der öffnenden Klammern gleich der Anzahl der schließenden Klammern. #

Mit Hilfe von Satz 43 können wir nun leicht ein Programm entwickeln, das einen Plausibilitätscheck (z.B. direkt in einem Editor) durchführt und die Klammern zählt, bevor die Syntax von arithmetischen Ausdrücken überprüft wird. Definiert man eine vollständige Programmiersprache induktiv, dann werden ganz ähnliche Induktionsbeweise möglich, d.h. man kann die Techniken aus diesem Beispiel relativ leicht auf die Praxis der Informatik übertragen.

Man überlegt sich leicht, dass die natürlichen Zahlen auch induktiv definiert werden können (vgl. Peano-Axiome). Damit zeigt sich, dass die vollständige Induktion eigentlich nur ein Spezialfall der strukturellen Induktion ist.



(a) grober Stadtplan (b) formalisierter Stadtplan

Abbildung 1: Das Königsberger-Brückenproblem

5 Graphen und Graphenalgorithmen

5.1 Einführung

Sehr viele Probleme lassen sich durch Objekte und Verbindungen oder Beziehungen zwischen diesen Objekten beschreiben. Ein schönes Beispiel hierfür ist das *Königsberger Brückenproblem*, das 1736 von Leonhard Euler[21] formuliert und gelöst wurde. Zu dieser Zeit hatte Königsberg[22] genau sieben Brücken, wie die sehr grobe Karte in Abbildung 1 zeigt.

Die verschiedenen Stadtteile sind dabei mit A-D bezeichnet. Euler stellte sich nun die Frage, ob es möglich ist, einen Spaziergang in einem beliebigen Stadtteil zu beginnen, jede Brücke *genau einmal* zu überqueren und den Spaziergang am Startpunkt zu beenden. Ein solcher Weg soll *Euler-Spaziergang* heißen. Die Frage lässt sich leicht beantworten, wenn der Stadtplan wie nebenstehend formalisiert wird.

Die Stadtteile sind bei der Formalisierung zu Knoten geworden und die Brücken werden durch Kanten zwischen den Knoten symbolisiert[23]. Angenommen es gäbe in Königsberg einen Euler-Spaziergang, dann müsste für jeden Knoten in Abbildung 1 die folgende Eigenschaft erfüllt sein: die Anzahl der Kanten die mit einem Knoten verbunden sind ist gerade, weil für jede Ankunft (über eine Brücke) in einem Stadtteil ein Verlassen eines Stadtteil (über eine Brücke) notwendig ist.

5.2 Grundlagen

Die Theorie der Graphen ist heute zu einem unverzichtbaren Bestandteil der Informatik geworden. Viele Probleme, wie z.B. das Verlegen von Leiterbahnen auf einer Platine, die Modellierung von Netzwerken oder die Lösung von Routingproblemen in Verkehrsnetzen benutzen Graphen oder Algorithmen, die Graphen als Datenstruktur verwenden. Auch schon bekannte Datenstrukturen wie Listen und Bäume können als Graphen aufgefasst werden. All dies gibt einen Anhaltspunkt, dass die Graphentheorie eine sehr zentrale Rolle für die Informatik spielt und vielfältige Anwendungen hat. In diesem Kontext ist es wichtig zu bemerken, dass der Begriff des Graphen in der Informatik *nicht* im Sinne von Graph einer Funktion gebraucht wird, sondern wie folgt definiert ist:

Definition 44: *Ein* gerichteter Graph $G = (V, E)$ *ist ein Paar, das aus einer Menge von* Knoten V *und einer Menge von* Kanten $E \subseteq V \times V$ *(*Kantenrelation*) besteht. Eine Kante $k = (u, v)$ aus E kann als Verbindung zwischen den Knoten $u, v \in V$ aufgefasst werden. Aus diesem Grund nennt man u auch* Startknoten *und v* Endknoten*. Zwei Knoten, die durch eine Kante verbunden sind, heißen auch* benachbart *oder* adjazent*.*

[21] Der Schweizer Mathematiker Leonhard Euler wurde 1707 in Basel geboren und starb 1783 in St. Petersburg.

[22] Königsberg heißt heute Kaliningrad.

[23] Abbildung 1 nennt man *Multigraph*, denn hier starten mehrere Kanten von *einem* Knoten und enden in *einem* anderen Knoten.

Ein Graph $H = (V', E')$ mit $V' \subseteq V$ und $E' \subset E$ heißt Untergraph *von G.*

Ein Graph (V, E) heißt *endlich* gdw. die Menge der Knoten V endlich ist. Obwohl man natürlich auch unendliche Graphen betrachten kann, werden wir uns in diesem Abschnitt nur mit endlichen Graphen beschäftigen, da diese für den Informatiker von großem Nutzen sind.

Da wir eine Kante (u, v) als Verbindung zwischen den Knoten u und v interpretieren können, bietet es sich an, Graphen durch Diagramme darzustellen. Dabei wird die Kante (u, v) durch einen Pfeil von u nach v dargestellt. Drei Beispiele für eine bildliche Darstellung von gerichteten Graphen finden sich in Abbildung 2.

5.3 Einige Eigenschaften von Graphen

Der Graph in Abbildung 2(c) hat eine besondere Eigenschaft, denn offensichtlich kann man die Knotenmenge $V_{1c} = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ in zwei disjunkte Teilmengen $V_{1c}^l = \{0, 1, 2, 3\}$ und $V_{1c}^r = \{4, 5, 6, 7, 8\}$ so aufteilen, dass keine Kante zwischen zwei Knoten aus V_{1c}^l oder V_{1c}^r verläuft.

Definition 45: *Ein Graph $G = (V, E)$ heißt* bipartit, *wenn gilt:*

1. *Es gibt zwei Teilmengen V^l und V^r von V mit $V = V^l \cup V^r$, $V^l \cap V^r = \emptyset$ und*
2. *für jede Kante $(u, v) \in E$ gilt $u \in V^l$ und $v \in V^r$.*

Bipartite Graphen haben viele Anwendungen, weil man jede binäre Relation $R \subseteq A \times B$ mit $A \cap B = \emptyset$ ganz natürlich als bipartiten Graph auffassen kann, dessen Kanten von Knoten aus A zu Knoten aus B laufen.

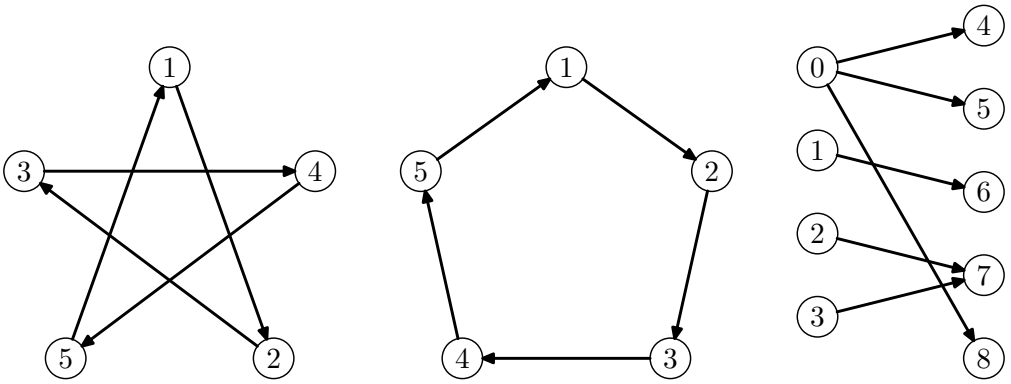
Beispiel 46: *Gegeben sei ein bipartiter Graph $G = (V, E)$ mit $V = V^F \cup V^M$ und $V^F \cap V^M = \emptyset$. Die Knoten aus V^F symbolisieren Frauen und V^M symbolisiert eine Menge von Männern. Kann sich eine Frau vorstellen einen Mann zu heiraten, so wird der entsprechende Knoten aus V^F mit dem passenden Knoten aus V^M durch eine Kante verbunden. Eine* Heirat *ist nun eine Kantenmenge $H \subseteq E$, so dass keine zwei Kanten aus H einen gemeinsamen Knoten besitzen. Das* Heiratsproblem *ist nun die Aufgabe für G eine Heirat H zu finden, so dass alle Frauen heiraten können, d.h. es ist das folgende Problem zu lösen:*

Problem:	*MARRIAGE*
Eingabe:	*Bipartiter Graph $G = (V, E)$ mit $V = V^F \cup V^M$ und $V^F \cap V^M = \emptyset$*
Ausgabe:	*Eine Heirat H mit $\#H = \#V^F$*

Im Beispielgraphen 2(c) gibt es keine Lösung für das Heiratsproblem, denn für die Knoten ($\triangleq$ Kandidatinnen) 2 und 3 existieren nicht ausreichend viele Partner, d.h. keine Heirat in diesem Graphen enthält zwei Kanten die sowohl 2 als auch 3 als Startknoten haben.

Obwohl dieses Beispiel auf den ersten Blick nur von untergeordneter Bedeutung erscheint, kann man es auf eine Vielfalt von Anwendungen übertragen. Immer wenn die Elemente zweier disjunkter Mengen durch eine Beziehung verbunden sind, kann man dies als bipartiten Graphen auffassen. Sollen nun die Bedürfnisse der einen Menge völlig befriedigt werden, so ist dies wieder ein Heiratsproblem. Beispiele mit mehr praktischem Bezug finden sich u.a. bei Beziehungen zwischen Käufern und Anbietern.

Oft beschränken wir uns auch auf eine Unterklasse von Graphen, bei denen die Kanten keine „Richtung“ haben (siehe Abbildung 3) und einfach durch eine Verbindungslinie symbolisiert werden können:



(a) Ein gerichteter Graph mit 5 Knoten

(b) Ein planarer gerichteter Graph mit 5 Knoten

(c) Ein gerichteter bipartiter Graph

Abbildung 2: Beispiele für gerichtete Graphen

Definition 47: *Sei $G = (V, E)$ ein Graph. Ist die Kantenrelation E* symmetrisch*, d.h. gibt es zu jeder Kante $(u, v) \in E$ auch eine Kante $(v, u) \in E$ (siehe auch Abschnitt 2.2.1), dann bezeichnen wir G als* ungerichteten Graphen *oder kurz als* Graph*.*

Es ist praktisch, die Kanten (u, v) und (v, u) eines ungerichteten Graphen als Menge $\{u, v\}$ mit zwei Elementen aufzufassen. Diese Vorgehensweise führt zu einem kleinen technischen Problem. Eine Kante (u, u) mit gleichem Start- und Endknoten nennen wir, entsprechend der intuitiven Darstellung eines Graphens als Diagramm, *Schleife*. Wandelt man nun solch eine Kante in eine Menge um, so würde nur eine einelementige Menge entstehen. Aus diesem Grund legen wir fest, dass ungerichtete Graphen *schleifenfrei* sind.

Definition 48: *Der (ungerichtete) Graph $K = (V, E)$ heißt* vollständig*, wenn für alle $u, v \in V$ mit $u \neq v$ auch $(u, v) \in E$ gilt, d.h. jeder Knoten des Graphen ist mit allen anderen Knoten verbunden. Ein Graph $O = (V, \emptyset)$ ohne Kanten wird als* Nullgraph *bezeichnet.*

Mit dieser Definition ergibt sich, dass die Graphen in Abbildung 3(a) und Abbildung 3(b) vollständig sind. Der Nullgraph $(V, \emptyset)$ ist Untergraph jedes beliebigen Graphen (V, E). Diese Definitionen lassen sich natürlich auch analog auf gerichtete Graphen übertragen.

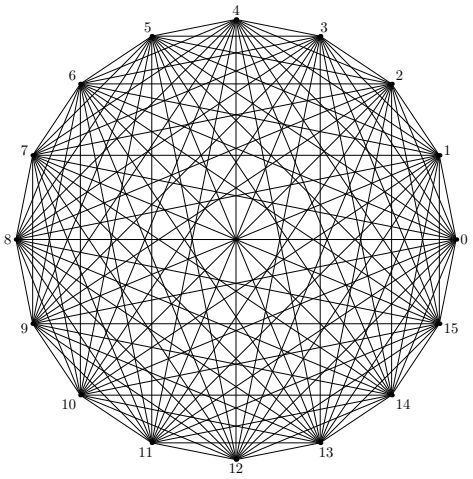
Definition 49: *Sei $G = (V, E)$ ein gerichteter Graph und $v \in V$ ein beliebiger Knoten. Der* Ausgrad *von v (kurz:* $\text{outdeg}(v)$*) ist dann die Anzahl der Kanten in G, die v als Startknoten haben. Analog ist der* Ingrad *von v (kurz:* $\text{indeg}(v)$*) die Anzahl der Kanten in G, die v als Endknoten haben.*

Bei ungerichteten Graphen gilt für jeden Knoten $\text{outdeg}(v) = \text{indeg}(v)$*. Aus diesem Grund schreiben wir kurz* $\deg(v)$ *und bezeichnen dies als* Grad von v*. Ein Graph G heißt* regulär *gdw. alle Knoten von G den gleichen Grad haben.*

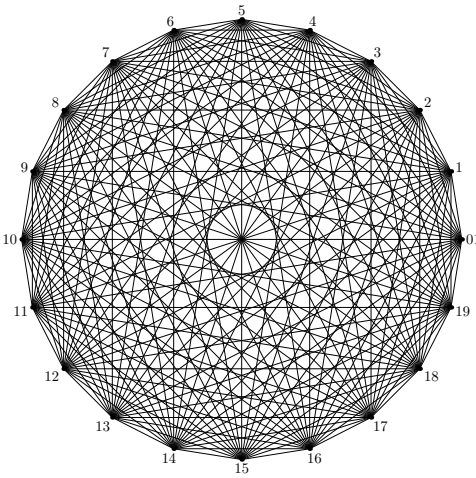
Die Diagramme der Graphen in den Abbildungen 2 und 3 haben die Eigenschaft, dass sich einige Kanten schneiden. Es stellt sich die Frage, ob man diese Diagramme auch so zeichnen kann, dass keine Überschneidungen auftreten. Diese Eigenschaft von Graphen wollen wir durch die folgende Definition festhalten:

Definition 50: *Ein Graph G heißt* planar*, wenn sich sein Diagramm ohne Überschneidungen zeichnen läßt.*

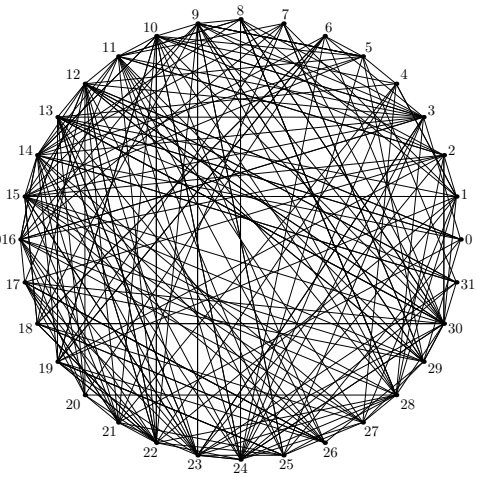
Beispiel 51: *Der Graph in Abbildung 2(a) ist, wie man leicht nachprüfen kann, planar, da die Diagramme aus Abbildung 2(a) und 2(b) den gleichen Graphen repräsentieren.*



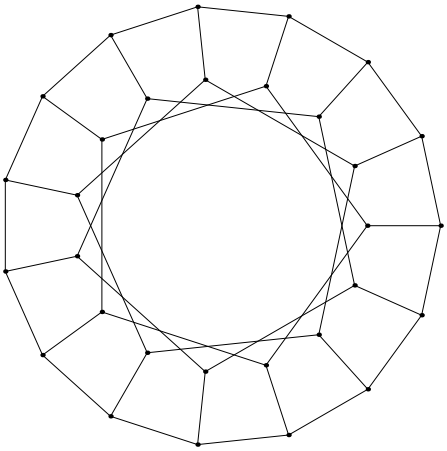
(a) Vollständiger ungerichteter Graph K_{16}



(b) Vollständiger ungerichteter Graph K_{20}



(c) Zufälliger Graph mit 32 Knoten



(d) Regulärer Graph mit Grad 3

Abbildung 3: Beispiele für ungerichtete Graphen

Auch planare Graphen haben eine anschauliche Bedeutung. Der Schaltplan einer elektronischen Schaltung kann als Graph aufgefasst werden. Die Knoten entsprechen den Stellen an denen die Bauteile aufgelötet werden müssen, und die Kanten entsprechen den Leiterbahnen auf der Platine. In diesem Zusammenhang bedeutet planar, ob man die Leiterbahnen kreuzungsfrei verlegen kann, d.h. ob es möglich ist, eine Platine zu fertigen, die mit einer Kupferschicht auskommt. In der Praxis kommen oft Platinen mit mehreren Schichten zum Einsatz („Multilayer-Platine“). Ein Grund dafür kann sein, dass der „Schaltungsgraph“ nicht planar war und deshalb mehrere Schichten benötigt werden. Da Platinen mit mehreren Schichten in der Fertigung deutlich teurer sind als solche mit einer Schicht, hat die Planaritätseigenschaft von Graphen somit auch unmittelbare finanzielle Auswirkungen.

5.4 Wege, Kreise, Wälder und Bäume

Definition 52: *Sei $G = (V, E)$ ein Graph und $u, v \in V$. Eine Folge von Knoten $u_0, \ldots, u_l \in V$ mit $u = u_0$, $v = u_l$ und $(u_i, u_{i+1}) \in E$ für $0 \leq i \leq l - 1$ heißt* Weg von u nach v der Länge l. *Der Knoten u wird* Startknoten *und v wird* Endknoten *des Wegs genannt.*

Ein Weg, bei dem Start- und Endknoten gleich sind, heißt geschlossener Weg. *Ein geschlossener Weg, bei dem kein Knoten außer dem Startknoten mehrfach enthalten ist, wird* Kreis *genannt.*

Mit Definition 52 wird klar, dass der Graph in Abbildung 2(a) den Kreis $1, 2, 3, \ldots, 5, 1$ mit Startknoten 1 hat.

Definition 53: *Sei $G = (V, E)$ ein Graph. Zwei Knoten $u, v \in V$ heißen* zusammenhängend, *wenn es einen Weg von u nach v gibt. Der Graph G heißt* zusammenhängend, *wenn jeder Knoten von G mit jedem anderen Knoten von G zusammenhängt.*

Sei G' ein zusammenhängender Untergraph von G mit einer besonderen Eigenschaft: Nimmt man einen weiteren Knoten von G zu G' hinzu, dann ist der neu entstandene Graph nicht mehr zusammenhängend, d.h. es gibt keinen Weg zu diesem neu hinzugekommenen Knoten. Solch einen Untergraph nennt man Zusammenhangskomponente.

Offensichtlich sind die Graphen in den Abbildungen 2(a), 3(a), 3(b) und 3(d) zusammenhängend und haben genau eine Zusammenhangskomponente. Man kann sich sogar leicht überlegen, dass die Eigenschaft „u hängt mit v“ zusammen eine *Äquivalenzrelation* (siehe Abschnitt 2.2.1) darstellt.

Mit Hilfe der Definition des geschlossenen Wegs lässt sich nun der Begriff der Bäume definieren, die eine sehr wichtige Unterklasse der Graphen darstellen.

Definition 54: *Ein Graph G heißt*

- Wald, *wenn es keinen geschlossenen Weg mit Länge ≥ 1 in G gibt und*
- Baum, *wenn G ein zusammenhängender Wald ist, d.h. wenn er nur genau eine Zusammenhangskomponente hat.*

5.5 Die Repräsentation von Graphen und einige Algorithmen

Nachdem Graphen eine große Bedeutung sowohl in der praktischen als auch in der theoretischen Informatik erlangt haben, stellt sich noch die Frage, wie man Graphen effizient als Datenstruktur in einem Computer ablegt. Dabei soll es möglich sein, Graphen effizient zu speichern und zu manipulieren.

Die erste Idee, Graphen als dynamische Datenstrukturen zu repräsentieren, scheitert an dem relativ ineffizienten Zugriff auf die Knoten und Kanten bei dieser Art der Darstellung. Sie ist nur von Vorteil, wenn ein Graph nur sehr wenige Kanten enthält. Die folgende Methode der Speicherung von Graphen hat sich als effizient erwiesen und ermöglicht auch die leichte Manipulation des Graphens:

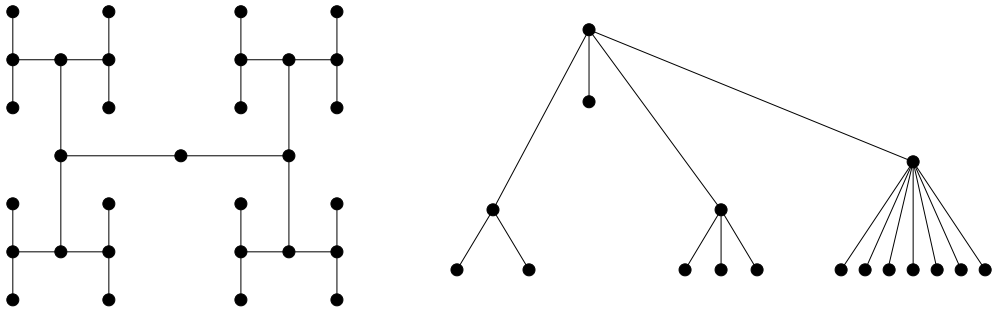


Abbildung 4: Ein Wald mit zwei Bäumen

Definition 55: *Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{v_1, \ldots, v_n\}$. Wir definieren eine $n \times n$ Matrix $A_G = (a_{i,j})_{1 \leq i,j, \leq n}$ durch*

$$a_{i,j} = \begin{cases} 1, & \text{falls } (v_i, v_j) \in E \\ 0, & \text{sonst} \end{cases}$$

Die so definierte Matrix A_G mit Einträgen aus der Menge $\{0, 1\}$ heißt Adjazenzmatrix *von G.*

Beispiel 56: *Für den gerichteten Graphen aus Abbildung 2(a) ergibt sich die folgende Adjazenzmatrix:*

$$A_{G_5} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Die Adjazenzmatrix eines ungerichteten Graphen erkennt man daran, dass sie spiegelsymmetrisch zur Diagonale von links oben nach rechts unten ist (die Kantenrelation ist symmetrisch) und dass die Diagonale aus 0-Einträgen besteht (der Graph hat keine Schleifen). Für den vollständigen Graphen K_{16} aus Abbildung 3(a) ergibt sich offensichtlich die folgende Adjazenzmatrix:

$$A_{K_{16}} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Mit Hilfe der Adjazenzmatrix und Algorithmus 1 kann man leicht berechnen, ob ein Weg von einem Knoten u zu einem Knoten v existiert. Mit einer ganz ähnlichen Idee kann man auch leicht

Algorithmus 1: Erreichbarkeit in Graphen

Eingabe: Ein Graph $G = (V, E)$ und zwei Knoten $u, v \in V$
Ergebnis: `true` wenn es einen Weg von u nach v gibt, `false` sonst

```
markiert = true;
markiere Startknoten u ∈ V;
while (markiert) do
    markiert = false;
    for (alle markierten Knoten w ∈ V) do
        if (w ∈ V ist adjazent zu einem unmarkierten Knoten w' ∈ V) then
            markiere Knoten w';
            markiert = true;
        end
    end
end
if (v ist markiert) then
    return true;
else
    return false;
end
```

Algorithmus 2: Zusammenhangskomponenten

Eingabe: Ein Graph $G = (V, E)$
Ergebnis: Anzahl der Zusammenhangskomponenten von G

```
kFarb = 0;
while (es gibt einen unmarkierten Knoten u ∈ V) do
    kFarb++;
    markiere u ∈ V mit kFarb;
    markiert=true;
    while (markiert) do
        markiert=false;
        for (alle mit kFarb markierten Knoten v ∈ V) do
            if (v ∈ V ist adjazent zu einem unmarkierten Knoten v' ∈ V) then
                markiere Knoten v' ∈ V mit kFarb;
                markiert=true;
            end
        end
    end
end
return kFarb;
```

die Anzahl der Zusammenhangskomponenten berechnen (siehe Algorithmus 2). Dieser Algorithmus markiert die Knoten der einzelnen Zusammenhangskomponenten auch mit unterschiedlichen „Farben“, die hier durch Zahlen repräsentiert werden.

Definition 57: *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Funktion der Form $f : V \to \{1, \ldots, k\}$ heißt* k-Färbung *des Graphen G. Anschaulich ordnet die Funktion f jedem Knoten eine von k verschiedenen Farben zu, die hier durch die Zahlen $1, \ldots, k$ symbolisiert werden. Eine Färbung heißt* verträglich, *wenn für alle Kanten $(u, v) \in E$ gilt, dass $f(u) \neq f(v)$, d.h. zwei adjazente Knoten werden nie mit der gleichen Farbe markiert.*

Für viele Probleme der Graphentheorie gibt es mit hoher Wahrscheinlichkeit keinen effizienten Algorithmus. Mehr Informationen zu diesem Thema finden sich in Abschnitt 6.

6 Komplexität

Für viele ständig auftretende Berechnungsprobleme, wie Sortieren, die arithmetischen Operationen (Addition, Multiplikation, Division), Fourier-Transformation etc., sind sehr effiziente Algorithmen konstruiert worden. Für wenige andere praktische Probleme weiß man, dass sie nicht oder nicht effizient algorithmisch lösbar sind. Im Gegensatz dazu ist für einen sehr großen Teil von Fragestellungen aus den verschiedensten Anwendungsbereichen wie Operations Research, Netzwerkdesign, Programmoptimierung, Datenbanken, Betriebssystem-Entwicklung und vielen mehr jedoch nicht bekannt, ob sie effiziente Algorithmen besitzen (vgl. die Abbildungen 9 und 10). Diese Problematik hängt mit der seit über vierzig Jahren[24] offenen $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage zusammen, wahrscheinlich gegenwärtig das wichtigste ungelöste Problem der theoretischen Informatik. Es wurde sogar vor einiger Zeit auf Platz 1 der Liste der so genannten *Millennium Prize Problems* des Clay Mathematics Institute gesetzt[25]. Diese Liste umfasst sieben offene Probleme aus der gesamten Mathematik. Das Clay Institute zahlt jedem, der eines dieser Probleme löst, eine Million US-Dollar.

In diesem Abschnitt werden die wesentlichen Begriffe aus dem Kontext des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems und des Begriffes der **NP**-Vollständigkeit erläutert, um so die Grundlagen für das Verständnis derartiger Probleme zu schaffen und deren Beurteilung zu ermöglichen.

6.1 Effizient lösbare Probleme: die Klasse P

Jeder, der schon einmal mit der Aufgabe konfrontiert wurde, einen Algorithmus für ein gegebenes Problem zu entwickeln, kennt die Hauptschwierigkeit dabei: Wie kann ein effizienter Algorithmus gefunden werden, der das Problem mit möglichst wenigen Rechenschritten löst? Um diese Frage beantworten zu können, muss man sich zunächst einige Gedanken über die verwendeten Begriffe, nämlich „Problem“, „Algorithmus“, „Zeitbedarf“ und „effizient“, machen.

Was ist ein „Problem“? Jedem Programmierer ist diese Frage intuitiv klar: Man bekommt geeignete Eingaben, und das Programm soll die gewünschten Ausgaben ermitteln. Ein einfaches Beispiel ist das Problem MULT. (Jedes Problem soll mit einem eindeutigen Namen versehen und dieser in Großbuchstaben geschrieben werden.) Hier bekommt man zwei ganze Zahlen als Eingabe und soll das Produkt beider Zahlen berechnen, d.h. das Programm berechnet einfach eine zweistellige Funktion. Es hat sich gezeigt, dass man sich bei der Untersuchung von Effizienzfragen auf eine abgeschwächte Form von Problemen beschränken kann, nämlich sogenannte *Entscheidungsprobleme*. Hier ist die Aufgabe, eine gewünschte Eigenschaft der Eingaben zu testen. Hat die aktuelle Eingabe die gewünschte Eigenschaft, dann gibt man den Wert 1 ($\triangleq$ `true`) zurück (man spricht dann auch von einer *positiven Instanz* des Problems), hat die Eingabe die Eigenschaft nicht, dann gibt man den Wert 0 ($\triangleq$ `false`) zurück. Oder anders formuliert: Das Programm berechnet eine Funktion, die den Wert 0 oder 1 zurück gibt und partitioniert damit die Menge der möglichen Eingaben in zwei Teile: die Menge der Eingaben mit der gewünschten Eigenschaft und die Menge der Eingaben, die die gewünschte Eigenschaft nicht besitzen. Folgendes Beispiel soll das Konzept verdeutlichen:

Problem:	PARITY
Eingabe:	Positive Integerzahl x
Ausgabe:	Ist die Anzahl der Ziffern 1 in der Binärdarstellung von x ungerade?

[24] Interessanterweise reicht die Geschichte des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems viel weiter in die Geschichte zurück. So fragte 1956 Kurt Gödel in einem Brief an John von Neumann schon „Given a first-order formula F and a natural number n, determine if there is a proof of F of length n“. Gödel war an der Anzahl von Schritten interessiert, die eine Turing-Maschine benötigt, um dieses Problem zu lösen. Dazu fragte er, ob dies in linearer oder quadratischer Zeit möglich ist, was man als eine Frühform der $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage auffassen könnte.

[25] siehe http://www.claymath.org/millennium-problems

Es soll also ein Programm entwickelt werden, das die Parität einer Integerzahl x berechnet. Eine mögliche Entscheidungsproblem-Variante des Problems MULT ist die folgende:

Problem:	MULT_D
Eingabe:	Integerzahlen x, y, positive Integerzahl i
Ausgabe:	Ist das i-te Bit in $x \cdot y$ gleich 1?

Offensichtlich sind die Probleme MULT und MULT_D gleich schwierig (oder leicht) zu lösen.

Im Weiteren wollen wir uns hauptsächlich mit Problemen beschäftigen, die aus dem Gebiet der Graphentheorie stammen. Das hat zwei Gründe. Zum einen können, wie sich noch zeigen wird, viele praktisch relevante Probleme mit Hilfe von Graphen modelliert werden, und zum anderen sind sie anschaulich und oft relativ leicht zu verstehen. Ein (ungerichteter) Graph G besteht aus einer Menge von Knoten V und einer Menge von Kanten E, die diese Knoten verbinden. Man schreibt: $G = (V, E)$. Ein wohlbekanntes Beispiel ist der Nikolausgraph: $G_N = \big(\{1,2,3,4,5\}, \{(1,2),(1,3),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)\}\big)$. Es gibt also fünf Knoten $V = \{1,2,3,4,5\}$, die durch die Kanten in $E = \{(1,2),(1,3),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)\}$ verbunden werden (siehe Abbildung 5).

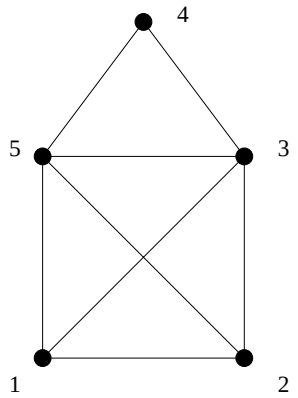


Abbildung 5: Der Graph G_N

Ein prominentes Problem in der Graphentheorie ist es, eine sogenannte Knotenfärbung zu finden. Dabei wird jedem Knoten eine Farbe zugeordnet, und man verbietet, dass zwei Knoten, die durch eine Kante verbunden sind, die gleiche Farbe zugeordnet wird. Natürlich ist die Anzahl der Farben, die verwendet werden dürfen, durch eine feste natürliche Zahl k beschränkt. Genau wird das Problem, ob ein Graph k-färbbar ist, wie folgt beschrieben:

Problem:	kCOL
Eingabe:	Ein Graph $G = (V, E)$
Ausgabe:	Hat G eine Knotenfärbung mit höchstens k Farben?

Offensichtlich ist der Beispielgraph G_N nicht mit drei Farben färbbar (aber mit 4 Farben, wie man leicht ausprobieren kann), und jedes Programm für das Problem 3COL müsste ermitteln, dass G_N die gewünschte Eigenschaft (3-Färbbarkeit) nicht hat.

Man kann sich natürlich fragen, was das künstlich erscheinende Problem 3COL mit der Praxis zu tun hat. Das folgende einfache Beispiel soll das verdeutlichen. Man nehme das Szenario an, dass ein großer Telefonprovider in einer Ausschreibung drei Funkfrequenzen für einen neuen Mobilfunkstandard erworben hat. Da er schon über ein Mobilfunknetz verfügt, sind die Sendemasten schon gebaut. Aus technischen Gründen dürfen Sendemasten, die zu eng stehen, nicht mit der gleichen Frequenz funken, da sie sich sonst stören würden. In der graphentheoretischen Welt modelliert man die Sendestationen mit Knoten eines Graphen, und „nahe“ zusammenstehende Sendestationen symbolisiert man mit einer Kante zwischen den Knoten, für die sie stehen. Die Aufgabe des Mobilfunkplaners ist es nun, eine 3-Färbung für den entstehenden Graphen zu finden. Offensichtlich kann das Problem verallgemeinert werden, wenn man sich nicht auf drei Farben/Frequenzen festlegt; dann aber ergeben sich genau die oben definierten Probleme kCOL für beliebige Zahlen k.

Als nächstes ist zu klären, was unter einem „Algorithmus“ zu verstehen ist. Ein Algorithmus ist eine endliche, formale Beschreibung einer Methode, die ein Problem löst (z.B. ein Programm in einer beliebigen Programmiersprache). Diese Methode muss also alle Eingaben mit der gesuchten Eigenschaft von den Eingaben, die diese Eigenschaft nicht haben, unterscheiden können. Man legt

fest, dass der Algorithmus für erstere den Wert 1 und für letztere den Wert 0 ausgeben soll. Wie soll die „Laufzeit“ eines Algorithmus gemessen werden? Um dies festlegen zu können, muss man sich zunächst auf ein sogenanntes *Berechnungsmodell* festlegen. Das kann man damit vergleichen, welche Hardware für die Implementation des Algorithmus verwendet werden soll. Für die weiteren Analysen soll das folgende einfache C-artige Modell verwendet werden: Es wird (grob!) die Syntax von C verwendet und festgelegt, dass jede Anweisung in einem Schritt abgearbeitet werden kann. Gleichzeitig beschränkt man sich auf zwei Datentypen: einen Integer-Typ und zusätzlich Arrays dieses Integer-Typs (wobei Array-Grenzen nicht deklariert werden müssen, sondern sich aus dem Gebrauch ergeben). Dieses primitive Maschinenmodell ist deshalb geeignet, weil man zeigen kann, dass jeder so formulierte Algorithmus auf realen Computern implementiert werden kann, ohne eine substantielle Verlangsamung zu erfahren. (Dies gilt zumindest, wenn die verwendeten Zahlen nicht übermäßig wachsen, d.h., wenn alle verwendeten Variablen nicht zu viel Speicher belegen. Genaueres zu dieser Problematik – man spricht von der Unterscheidung zwischen *uniformem Komplexitätsmaß* und *Bitkomplexität* – findet sich in [Scho1, S. 62f].)

Umgekehrt kann man ebenfalls sagen, dass dieses einfache Modell die Realität genau genug widerspiegelt, da auch reale Programme ohne allzu großen Zeitverlust auf diesem Berechnungsmodell simuliert werden können. Offensichtlich ist die *Eingabe* der Parameter, von dem die Rechenzeit für einen festen Algorithmus abhängt. In den vergangenen Jahrzehnten, in denen das Gebiet der Analyse von Algorithmen entstand, hat die Erfahrung gezeigt, dass die Länge der Eingabe, also die Anzahl der Bits, die benötigt werden um die Eingabe zu speichern, ein geeignetes und robustes Maß ist, in der die Rechenzeit gemessen werden kann. Auch der Aufwand, die Eingabe selbst festzulegen (zu konstruieren), hängt schließlich von ihrer Länge ab, nicht davon, ob sich irgendwo in der Eingabe eine 0 oder 1 befindet.

6.1.1 Das Problem der 2-Färbbarkeit

Das Problem der 2-Färbbarkeit ist wie folgt definiert:

Problem:	2COL
Eingabe:	Ein Graph $G = (V, E)$
Ausgabe:	Hat G eine Knotenfärbung mit höchstens 2 Farben?

Es ist bekannt, dass dieses Problem mit einem sogenannten *Greedy-Algorithmus* gelöst werden kann: Beginne mit einem beliebigen Knoten in G (z.B. v_1) und färbe ihn mit Farbe 1. Färbe dann die Nachbarn dieses Knoten mit 2, die Nachbarn dieser Nachbarn wieder mit 1, usw. Falls G aus mehreren Komponenten (d.h. zusammenhängenden Teilgraphen) besteht, muss dieses Verfahren für jede Komponente wiederholt werden. G ist schließlich 2-färbbar, wenn bei obiger Prozedur keine inkorrekte Färbung entsteht. Diese Idee führt zu Algorithmus 3.

Die Laufzeit von Algorithmus 3 kann wie folgt abgeschätzt werden: Die erste **for**-Schleife benötigt n Schritte. In der **while**-Schleife wird entweder mindestens ein Knoten gefärbt und die Schleife dann erneut ausgeführt, oder es wird kein Knoten gefärbt und die Schleife dann verlassen; also wird diese Schleife höchstens n-mal ausgeführt. Innerhalb der **while**-Schleife finden sich drei ineinander verschachtelte **for**-Schleifen, die alle jeweils n-mal durchlaufen werden, und eine **while**-Schleife, die maximal n-mal durchlaufen wird.

Damit ergibt sich also eine Gesamtlaufzeit der Größenordnung n^4, wobei n die Anzahl der Knoten des Eingabe-Graphen G ist. Wie groß ist nun die Eingabelänge, also die Anzahl der benötigten Bits zur Speicherung von G? Sicherlich muss jeder Knoten in dieser Speicherung vertreten sein, d.h. also, dass mindestens n Bits zur Speicherung von G benötigt werden. Die Eingabelänge ist also mindestens n. Daraus folgt, dass die Laufzeit des Algorithmus also höchstens von der Größenordnung N^4 ist, wenn N die Eingabelänge bezeichnet.

Algorithmus 3: Algorithmus zur Berechnung einer 2-Färbung eines Graphen

Eingabe: Graph $G = (\{v_1, \ldots v_n\}, E)$;
Ergebnis: 1 wenn es eine 2-Färbung für G gibt, 0 sonst

```
begin
    for (i = 1 to n) do
        Farbe[i] = 0;
    end
    Farbe[1] = 1;
    repeat
        aktKompoBearbeiten = false;
        for (i = 1 to n) do
            for (j = 1 to n) do
                                    /* Kante mit noch ungefärbten Knoten? */
                if (((v_i, v_j) ∈ E) und (Farbe[i] ≠ 0) und (Farbe[j] = 0)) then
                                    /* v_j bekommt eine andere Farbe als v_i */
                    Farbe[j] = 3 − Farbe[i];
                    aktKompoBearbeiten = true;
                                  /* Alle direkten Nachbarn von v_j prüfen */
                    for (k = 1 to n) do
                                    /* Kollision beim Färben aufgetreten? */
                        if (((v_j, v_k) ∈ E) und (Farbe[i] = Farbe[k])) then
                                      /* Kollision! Graph nicht 2-färbbar */
                            return 0;
                        end
                    end
                end
            end
        end
          /* Ist die aktuelle Zusammenhangskomponente völlig gefärbt?
        */
        if (not(aktKompoBearbeiten)) then
            i = 1;
              /* Suche nach einer weiteren Zusammenhangskomponente von
            G */
            repeat
                /* Liegt v_i in einer neuen Zusammenhangskomponente von
                G? */
                if (Farbe[i] = 0) then
                    Farbe[i] = 1;
                            /* Neue Zusammenhangskomponente bearbeiten */
                    aktKompoBearbeiten = true;
                              /* Suche nach neuer Zusammenhangskomponente
                    abbrechen */
                    weiterSuchen = false;
                end
                i = i + 1;
            until (not(weiterSuchen) und (i ≤ n));
        end
    until (aktKompoBearbeiten);
                                                    /* 2-Färbung gefunden */
    return 1.
end
```

Anzahl der Takte	Eingabelänge n					
	10	20	30	40	50	60
n	0,00001 Sekunden	0,00002 Sekunden	0,00003 Sekunden	0,00004 Sekunden	0,00005 Sekunden	0,00006 Sekunden
n^2	0,0001 Sekunden	0,0004 Sekunden	0,0009 Sekunden	0,0016 Sekunden	0,0025 Sekunden	0,0036 Sekunden
n^3	0,001 Sekunden	0,008 Sekunden	0,027 Sekunden	0,064 Sekunden	0,125 Sekunden	0,216 Sekunden
n^5	0,1 Sekunden	3,2 Sekunden	24,3 Sekunden	1,7 Minuten	5,2 Minuten	13,0 Minuten
2^n	0,001 Sekunden	1 Sekunde	17,9 Minuten	12,7 Tage	35,7 Jahre	366 Jahrhunderte
3^n	0,059 Sekunden	58 Minuten	6,5 Jahre	3855 Jahrhunderte	$2 \cdot 10^8$ Jahrhunderte	$1,3 \cdot 10^{13}$ Jahrhunderte

Abbildung 6: Rechenzeitbedarf von Algorithmen auf einem „1-MIPS“-Rechner

Tatsächlich sind (bei Verwendung geeigneter Datenstrukturen wie Listen oder Queues) wesentlich effizientere Verfahren für 2COL möglich. Aber auch schon das obige einfache Verfahren zeigt: 2COL hat einen Polynomialzeitalgorithmus, also 2COL $\in$ **P**.

Alle Probleme für die Algorithmen existieren, die eine Anzahl von Rechenschritten benötigen, die durch ein beliebiges Polynom beschränkt ist, bezeichnet man mit **P** („**P**“ steht dabei für „**P**olynomialzeit“). Auch dabei wird die Rechenzeit in der Länge der Eingabe gemessen, d.h. in der Anzahl der Bits, die benötigt werden, um die Eingabe zu speichern (zu kodieren). Die Klasse **P** wird auch als Klasse der *effizient lösbaren Probleme* bezeichnet. Dies ist natürlich wieder eine idealisierte Auffassung: Einen Algorithmus mit einer Laufzeit n^{57}, wobei n die Länge der Eingabe bezeichnet, kann man schwer als effizient bezeichnen. Allerdings hat es sich in der Praxis gezeigt, dass für fast alle bekannten Probleme in **P** auch Algorithmen existieren, deren Laufzeit durch ein Polynom kleinen[26] Grades beschränkt ist.

In diesem Licht ist die Definition der Klasse **P** auch für praktische Belange von Relevanz. Dass eine polynomielle Laufzeit etwas substanziell Besseres darstellt als exponentielle Laufzeit (hier beträgt die benötigte Rechenzeit $2^{c \cdot n}$ für eine Konstante c, wobei n wieder die Länge der Eingabe bezeichnet), zeigt die Tabelle „Rechenzeitbedarf von Algorithmen“. Zu beachten ist, dass bei einem Exponentialzeit-Algorithmus mit $c = 1$ eine Verdoppelung der „Geschwindigkeit“ der verwendeten Maschine (also Taktzahl pro Sekunde) es nur erlaubt, eine um höchstens 1 Bit längere Eingabe in einer bestimmten Zeit zu bearbeiten. Bei einem Linearzeit-Algorithmus hingegen verdoppelt sich auch die mögliche Eingabelänge; bei einer Laufzeit von n^k vergrößert sich die mögliche Eingabelänge immerhin noch um den Faktor $\sqrt[k]{2}$. Deswegen sind Probleme, für die nur Exponentialzeit-Algorithmen existieren, praktisch nicht lösbar; daran ändert sich auch nichts Wesentliches durch die Einführung von immer schnelleren Rechnern.

Nun stellt sich natürlich sofort die Frage: Gibt es für jedes Problem einen effizienten Algorithmus? Man kann relativ leicht zeigen, dass die Antwort auf diese Frage „Nein“ ist. Die Schwierigkeit bei dieser Fragestellung liegt aber darin, dass man von vielen Problemen *nicht weiß*, ob sie effizient lösbar sind. Ganz konkret: Ein effizienter Algorithmus für das Problem 2COL ist Algorithmus 3. Ist es möglich, ebenfalls einen Polynomialzeitalgorithmus für 3COL zu finden? Viele Informatiker beschäftigen sich seit den 60er Jahren des letzten Jahrhunderts intensiv mit dieser Frage. Dabei kristallisierte sich heraus, dass viele praktisch relevante Probleme, für die kein effizienter Algorithmus

[26] Aktuell *scheint* es kein praktisch relevantes Problem aus **P** zu geben, für das es keinen Algorithmus mit einer Laufzeit von weniger als n^{12} gibt.

bekannt ist, eine gemeinsame Eigenschaft besitzen, nämlich die der *effizienten Überprüfbarkeit* von geeigneten Lösungen. Auch 3COL gehört zu dieser Klasse von Problemen, wie sich in Kürze zeigen wird. Aber wie soll man zeigen, dass für ein Problem kein effizienter Algorithmus existiert? Nur weil kein Algorithmus bekannt ist, bedeutet das noch nicht, dass keiner existiert.

Es ist bekannt, dass die oberen Schranken (also die Laufzeit von bekannten Algorithmen) und die unteren Schranken (mindestens benötigte Laufzeit) für das Problem PARITY (und einige wenige weitere, ebenfalls sehr einfach-geartete Probleme) sehr nahe zusammen liegen. Das bedeutet also, dass nur noch unwesentliche Verbesserungen der Algorithmen für des PARITY-Problem erwartet werden können. Beim Problem 3COL ist das ganz anders: Die bekannten unteren und oberen Schranken liegen extrem weit auseinander. Deshalb ist nicht klar, ob nicht doch (extrem) bessere Algorithmen als heute bekannt im Bereich des Möglichen liegen. Aber wie untersucht man solch eine Problematik? Man müsste ja über unendlich viele Algorithmen für das Problem 3COL Untersuchungen anstellen. Dies ist äußerst schwer zu handhaben und deshalb ist der einzige bekannte Ausweg, das Problem mit einer Reihe von weiteren (aus bestimmten Gründen) interessierenden Problemen zu vergleichen und zu zeigen, dass unser zu untersuchendes Problem nicht leichter zu lösen ist als diese anderen. Hat man das geschafft, ist eine untere Schranke einer speziellen Art gefunden: Unser Problem ist nicht leichter lösbar, als alle Probleme der Klasse von Problemen, die für den Vergleich herangezogen wurden. Nun ist aus der Beschreibung der Aufgabe aber schon klar, dass auch diese Aufgabe schwierig zu lösen ist, weil ein Problem nun mit unendlich vielen anderen Problemen zu vergleichen ist. Es zeigt sich aber, dass diese Aufgabe nicht aussichtslos ist. Bevor diese Idee weiter ausgeführt wird, soll zunächst die Klasse von Problemen untersucht werden, die für diesen Vergleich herangezogen werden sollen, nämlich die Klasse **NP**.

6.2 Effizient überprüfbare Probleme: die Klasse NP

Wie schon erwähnt, gibt es eine große Anzahl verschiedener Probleme, für die kein effizienter Algorithmus bekannt ist, die aber eine gemeinsame Eigenschaft haben: die *effiziente Überprüfbarkeit von Lösungen* für dieses Problem. Diese Eigenschaft soll an dem schon bekannten Problem 3COL veranschaulicht werden: Angenommen, man hat einen beliebigen Graphen G gegeben; wie bereits erwähnt ist kein effizienter Algorithmus bekannt, der entscheiden kann, ob der Graph G eine 3-Färbung hat (d.h., ob der fiktive Mobilfunkprovider mit 3 Funkfrequenzen auskommt). Hat man aber aus irgendwelchen Gründen eine *potenzielle* Knotenfärbung vorliegen, dann ist es leicht, diese potenzielle Knotenfärbung zu überprüfen und festzustellen, ob sie eine *korrekte* Färbung des Graphen ist, wie Algorithmus 4 zeigt.

Das Problem 3COL hat also die Eigenschaft, dass eine potenzielle Lösung leicht daraufhin überprüft werden kann, ob sie eine tatsächliche, d.h. korrekte, Lösung ist. Viele andere praktisch relevante Probleme, für die kein effizienter Algorithmus bekannt ist, besitzen ebenfalls diese Eigenschaft. Dies soll noch an einem weiteren Beispiel verdeutlicht werden, dem so genannten *Hamiltonkreis-Problem*.

Sei wieder ein Graph $G = \big(\{v_1, \ldots v_n\}, E\big)$ gegeben. Diesmal ist eine Rundreise entlang der Kanten von G gesucht, die bei einem Knoten v_{i_1} aus G startet, wieder bei v_{i_1} endet und jeden Knoten genau einmal besucht. Genauer wird diese Rundreise im Graphen G durch eine Folge von n Knoten $(v_{i_1}, v_{i_2}, v_{i_3}, \ldots, v_{i_{n-1}}, v_{i_n},)$ beschrieben, wobei gelten soll, dass alle Knoten $v_{i_1}, \ldots, v_{i_n}$ verschieden sind und die Kanten $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \ldots, (v_{i_{n-1}}, v_{i_n})$ und (v_{i_n}, v_{i_1}) in G vorkommen. Eine solche Folge von Kanten wird als *Hamiltonscher Kreis* bezeichnet. Ein Hamiltonscher Kreis in einem Graphen G ist also ein Kreis, der jeden Knoten des Graphen genau einmal besucht. Das Problem, einen Hamiltonschen Kreis in einem Graphen zu finden, bezeichnet man mit HAMILTON:

Problem:	HAMILTON
Eingabe:	Ein Graph G

Algorithmus 4: Ein Algorithmus zur Überprüfung einer potentiellen Färbung

Eingabe: Graph $G = (\{v_1, \ldots v_n\}, E)$ und eine potenzielle Knotenfärbung
Ergebnis: 1 wenn die Färbung korrekt, 0 sonst

```
begin
    /* Teste systematisch alle Kanten */
    for (i = 1 to n) do
        for (j = 1 to n) do
            /* Test auf Verletzung der Knotenfärbung */
            if (((v_i, v_j) ∈ E) und (v_i und v_j sind gleich gefärbt)) then
                return 0;
            end
        end
    end
    return 1;
end
```

Frage: Hat G einen Hamiltonschen Kreis?

Auch für dieses Problem ist kein effizienter Algorithmus bekannt. Aber auch hier ist offensichtlich: Bekommt man einen Graphen gegeben und eine Folge von Knoten, dann kann man sehr leicht überprüfen, ob sie ein Hamiltonscher Kreis ist – dazu ist lediglich zu testen, ob alle Knoten genau einmal besucht werden und auch alle Kanten im gegebenen Graphen vorhanden sind.

Hat man erst einmal die Beobachtung gemacht, dass viele Probleme die Eigenschaft der effizienten Überprüfbarkeit haben, ist es naheliegend, sie in einer Klasse zusammenzufassen und gemeinsam zu untersuchen. Die Hoffnung dabei ist, dass sich alle Aussagen, die man über diese Klasse herausfindet, sofort auf alle Probleme anwenden lassen. Solche Überlegungen führten zur Geburt der Klasse **NP**, in der man alle effizient überprüfbaren Probleme zusammenfasst. Aber wie kann man solch eine Klasse untersuchen? Man hat ja noch nicht einmal ein Maschinenmodell (oder eine Programmiersprache) zur Verfügung, um solch eine Eigenschaft zu modellieren. Um ein Programm für effizient überprüfbare Probleme zu schreiben, braucht man erst eine Möglichkeit, die zu überprüfenden möglichen Lösungen zu ermitteln und sie dann zu testen, d.h. man muss die Programmiersprache für **NP** in einer geeigneten Weise mit mehr „Berechnungskraft“ ausstatten.

Die erste Lösungsidee für **NP**-Probleme, nämlich alle in Frage kommenden Lösungen in einer **for**-Schleife aufzuzählen, führt zu Exponentialzeit-Lösungsalgorithmen, denn es gibt im Allgemeinen einfach so viele potenzielle Lösungen. Um erneut auf das Problem 3COL zurückzukommen: Angenommen, G ist ein Graph mit n Knoten. Dann gibt es 3^n potenzielle Färbungen, die überprüft werden müssen, denn es gibt 3 Möglichkeiten den ersten Knoten zu färben, 3 Möglichkeiten den zweiten Knoten zu färben, usw., und damit 3^n viele zu überprüfende potenzielle Färbungen. Würde man diese in einer **for**-Schleife aufzählen und auf Korrektheit testen, so führte das also zu einem Exponentialzeit-Algorithmus. Auf der anderen Seite gibt es aber Probleme, die in Exponentialzeit gelöst werden können, aber nicht zu der Intuition der effizienten Überprüfbarkeit der Klasse **NP** passen. Das Berechnungsmodell für **NP** kann also nicht einfach so gewonnen werden, dass exponentielle Laufzeit zugelassen wird, denn damit wäre man über das Ziel hinausgeschossen.

Hätte man einen Parallelrechner zur Verfügung mit so vielen Prozessoren wie es potenzielle Lösungen gibt, dann könnte man das Problem schnell lösen, denn jeder Prozessor kann unabhängig von allen anderen Prozessoren eine potenzielle Färbung überprüfen. Es zeigt sich aber, dass auch dieses Berechnungsmodell zu mächtig wäre. Es gibt Probleme, die wahrscheinlich nicht im obigen Sinne effizient überprüfbar sind, aber mit solch einem Parallelrechner trotzdem (effizient) gelöst

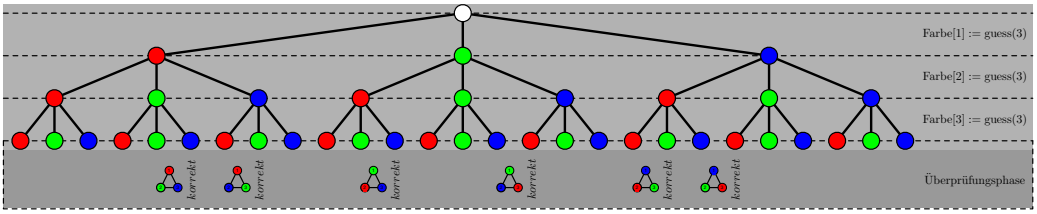


Abbildung 7: Ein Berechnungsbaum für das 3COL-Problem

werden könnten. In der Praxis würde uns ein derartiger paralleler Algorithmus auch nichts nützen, da man einen Rechner mit exponentiell vielen Prozessoren, also enormem Hardwareaufwand, zu konstruieren hätte. Also muss auch dieses Berechnungsmodell wieder etwas schwächer gemacht werden.

Eine Abschwächung der gerade untersuchten Idee des Parallelrechners führt zu folgendem Vorgehen: Man „rät“ für den ersten Knoten eine beliebige Farbe, dann für den zweiten Knoten auch wieder eine beliebige Farbe, solange bis für den letzten Knoten eine Farbe gewählt wurde. Danach überprüft man die geratene Färbung und akzeptiert die Eingabe, wenn die geratene Färbung eine korrekte Knotenfärbung ist. Die Eingabe ist eine positive Eingabeinstanz des **NP**-Problems 3COL, falls es eine potenzielle Lösung (Färbung) gibt, die sich bei der Überprüfung als korrekt herausstellt, d.h. im beschriebenen Rechnermodell: falls es eine Möglichkeit zu raten gibt, sodass am Ende akzeptiert (Wert 1 ausgegeben) wird. Man kann also die Berechnung durch einen Baum mit 3-fachen Verzweigungen darstellen (vgl. Abbildung 7). An den Kanten des Baumes findet sich das Resultat der Rateanweisung der darüberliegenden Verzweigung. Jeder Pfad in diesem sogenannten *Berechnungsbaum* entspricht daher einer Folge von Farbzuordnungen an die Knoten, d.h. einer potenziellen Färbung. Der Graph ist 3-färbbar, falls sich auf mindestens einem Pfad eine korrekte Färbung ergibt, falls also auf mindestens einem Pfad die Überprüfungsphase erfolgreich ist; der Beispielgraph besitzt sechs korrekte 3-Färbungen, ist also eine positive Instanz des 3COL-Problems.

Eine weitere, vielleicht intuitivere Vorstellung für die Arbeitsweise dieser **NP**-Maschine ist die, dass bei jedem Ratevorgang 3 verschiedene unabhängige Prozesse gestartet werden, die aber nicht miteinander kommunizieren dürfen. In diesem Sinne hat man es hier mit einem eingeschränkten Parallelrechner zu tun: Beliebige Aufspaltung (fork) ist erlaubt, aber keine Kommunikation zwischen den Prozessen ist möglich. Würde man Kommunikation zulassen, hätte man erneut den allgemeinen Parallelrechner mit exponentiell vielen Prozessoren von oben, der sich ja als zu mächtig für **NP** herausgestellt hat.

Es hat sich also gezeigt, dass eine Art „Rateanweisung“ benötigt wird. In der Programmiersprache für **NP** verwendet man dazu das neue Schlüsselwort **guess**(m), wobei m die Anzahl von Möglichkeiten ist, aus denen eine geraten wird, und legt fest, dass auch die Anweisung **guess**(m) nur einen Takt Zeit für ihre Abarbeitung benötigt. Berechnungen, die, wie soeben beschrieben, verschiedene Möglichkeiten raten können, heißen *nichtdeterministisch*. Es sei wiederholt, dass *festgelegt* (definiert) wird, dass ein nichtdeterministischer Algorithmus bei einer Eingabe den Wert 1 berechnet, falls *eine Möglichkeit* geraten werden kann, sodass der Algorithmus auf die Anweisung „**return** 1“ stößt. Die Klasse **NP** umfasst nun genau die Probleme, die von nichtdeterministischen Algorithmen mit polynomieller Laufzeit gelöst werden können. „**NP**“ steht dabei für „**n**ichtdeterministische **P**olynomialzeit“, nicht etwa, wie mitunter zu lesen, für „Nicht-Polynomialzeit“. (Eine formale Präsentation der Äquivalenz zwischen effizienter Überprüfbarkeit und Polynomialzeit in der **NP**-Programmiersprache findet sich z.B. in [GJ79, Kapitel 2.3].)

Mit Hilfe eines nichtdeterministischen Algorithmus kann das 3COL-Problem in Polynomialzeit gelöst werden (siehe Algorithmus 5). Die zweite Phase von Algorithmus 5, die *Überprüfungsphase*, entspricht dabei genau dem oben angegebenen Algorithmus zum effizienten Überprüfen von möglichen Lösungen des 3COL-Problems (vgl. Algorithmus 4).

Algorithmus 5: Ein nichtdeterministischer Algorithmus für 3COL

Eingabe: Graph $G = (\{v_1, \ldots v_n\}, E)$
Ergebnis: 1 wenn eine Färbung existiert, 0 sonst

```
begin
    /* Ratephase */
    for (i = 1 to n) do
        Farbe[i] = guess(3);
    end
    /* Überprüfungsphase */
    for (i = 1 to n) do
        for (j = 1 to n) do
            if (((v_i, v_j) ∈ E) und (v_i und v_j sind gleich gefärbt)) then
                return 0;
            end
        end
    end
    return 1;
end
```

Dieser nichtdeterministische Algorithmus läuft in Polynomialzeit, denn man benötigt für einen Graphen mit n Knoten mindestens n Bits, um ihn zu speichern (kodieren), und der Algorithmus braucht im schlechtesten Fall $O(n)$ (Ratephase) und $O(n^2)$ (Überprüfungsphase), also insgesamt $O(n^2)$ Takte Zeit. Damit ist gezeigt, dass 3COL in der Klasse **NP** enthalten ist, denn es wurde ein nichtdeterministischer Polynomialzeitalgorithmus gefunden, der 3COL löst. Ebenso einfach könnte man nun einen nichtdeterministischen Polynomialzeitalgorithmus entwickeln, der das Problem HAMILTON löst: Der Algorithmus wird in einer ersten Phase eine Knotenfolge raten und dann in einer zweiten Phase überprüfen, dass die Bedingungen, die an einen Hamiltonschen Kreis gestellt werden, bei der geratenen Folge erfüllt sind. Dies zeigt, dass auch HAMILTON in der Klasse **NP** liegt.

Dass eine nichtdeterministische Maschine nicht gebaut werden kann, spielt hier keine Rolle. Nichtdeterministische Berechnungen sollen hier lediglich als Gedankenmodell für unsere Untersuchungen herangezogen werden, um Aussagen über die (Nicht-) Existenz von effizienten Algorithmen machen zu können.

6.3 Schwierigste Probleme in NP: der Begriff der NP-Vollständigkeit

Es ist nun klar, was es bedeutet, dass ein Problem in **NP** liegt. Es liegt aber auch auf der Hand, dass alle Probleme aus **P** auch in **NP** liegen, da bei der Einführung von **NP** ja nicht verlangt wurde, dass die **guess**-Anweisung verwendet werden muss. Damit ist jeder deterministische Algorithmus automatisch auch ein (eingeschränkter) nichtdeterministischer Algorithmus. Nun ist aber auch schon bekannt, dass es Probleme in **NP** gibt, z.B. 3COL und weitere Probleme, von denen nicht bekannt ist, ob sie in **P** liegen. Das führt zu der Vermutung, dass $\mathbf{P} \neq \mathbf{NP}$.

Es gibt also in **NP** anscheinend unterschiedlich schwierige Probleme: einerseits die **P**-Probleme (also die leichten Probleme), und andererseits die Probleme, von denen man nicht weiß, ob sie in **P** liegen (die schweren Probleme). Es liegt also nahe, eine allgemeine Möglichkeit zu suchen, Probleme in **NP** bezüglich ihrer Schwierigkeit zu vergleichen. Ziel ist, wie oben erläutert, eine Art von unterer Schranke für Probleme wie 3COL: Es soll gezeigt werden, dass 3COL mindestens so schwierig ist, wie jedes andere Problem in **NP**, also in gewissem Sinne ein *schwierigstes Problem in* **NP** ist.

Für diesen Vergleich der Schwierigkeit ist die erste Idee natürlich, einfach die Laufzeit von (bekannten) Algorithmen für das Problem heranzuziehen. Dies ist jedoch nicht erfolgversprechend, denn was soll eine „größte“ Laufzeit sein, die Programme für „schwierigste“ Probleme in **NP** ja haben müssten? Außerdem hängt die Laufzeit eines Algorithmus vom verwendeten Berechnungsmodell ab. So kennen Turingmaschinen keine Arrays im Gegensatz zu der hier verwendeten C-Variante. Also würde jeder Algorithmus, der Arrays verwendet, auf einer Turingmaschine mühsam simuliert werden müssen und damit langsamer abgearbeitet werden, als bei einer Hochsprache, die Arrays enthält. Obwohl sich die Komplexität eines Problems nicht ändert, würde man sie verschieden messen, je nachdem welches Berechnungsmodell verwendet würde. Ein weiterer Nachteil dieses Definitionsversuchs wäre es, dass die Komplexität (Schwierigkeit) eines Problems mit bekannten Algorithmen gemessen würde. Das würde aber bedeuten, dass jeder neue und schnellere Algorithmus Einfluss auf die Komplexität hätte, was offensichtlich so keinen Sinn macht. Aus diesen und anderen Gründen führt die erste Idee nicht zum Ziel.

Eine zweite, erfolgversprechendere Idee ist die folgende: Ein Problem A ist nicht (wesentlich) schwieriger als ein Problem B, wenn man A mit der Hilfe von B (als Unterprogramm) effizient lösen kann. Ein einfaches Beispiel ist die Multiplikation von n Zahlen. Angenommen, man hat schon ein Programm, dass zwei Zahlen multiplizieren kann; dann ist es nicht wesentlich schwieriger, auch n Zahlen zu multiplizieren, wenn die Routine für die Multiplikation von zwei Zahlen verwendet wird. Dieser Ansatz ist unter dem Namen *relative Berechenbarkeit* bekannt, der genau den oben beschriebenen Sachverhalt widerspiegelt: Multiplikation von n Zahlen (so genannte *iterierte Multiplikation*) ist relativ zur Multiplikation zweier Zahlen (leicht) berechenbar.

Da das Prinzip der relativen Berechenbarkeit so allgemein gehalten ist, gibt es innerhalb der theoretischen Informatik sehr viele verschiedene Ausprägungen dieses Konzepts. Für die **P**-**NP**-Problematik ist folgende Version der relativen Berechenbarkeit, d.h. die folgende Art von erlaubten „Unterprogrammaufrufen“, geeignet:
Seien zwei Probleme A und B gegeben. Das Problem A ist nicht schwerer als B, falls es eine effizient zu berechnende Transformation T gibt, die Folgendes leistet: Wenn x eine Eingabeinstanz von Problem A ist, dann ist $T(x)$ eine Eingabeinstanz für B. Weiterhin gilt: x ist *genau dann* eine positive Instanz von A (d.h. ein Entscheidungsalgorithmus für A muss den Wert 1 für Eingabe x liefern), wenn $T(x)$ eine positive Instanz von Problem B ist. Erneut soll „effizient berechenbar“ hier bedeuten: in Polynomialzeit berechenbar. Es muss also einen Polynomialzeitalgorithmus geben, der die Transformation T ausführt. Das Entscheidungsproblem A ist damit effizient transformierbar in das Problem B. Man sagt auch: A ist reduzierbar auf B; oder intuitiver: A ist nicht schwieriger als B, oder B ist mindestens so schwierig wie A. Formal schreibt man dann $A \leq B$.

Um für dieses Konzept ein wenig mehr Intuition zu gewinnen, sei erwähnt, dass man sich eine solche Transformation auch wie folgt vorstellen kann: A lässt sich auf B reduzieren, wenn ein Algorithmus für A angegeben werden kann, der ein Unterprogramm U_B für B genau so verwendet wie in Algorithmus 6 gezeigt.

Dabei ist zu beachten, dass das Unterprogramm für B nur genau einmal und zwar am Ende aufgerufen werden darf. Das Ergebnis des Algorithmus für A ist genau das Ergebnis, das dieser Unterprogrammaufruf liefert. Es gibt zwar, wie oben erwähnt, auch allgemeinere Ausprägungen der relativen Berechenbarkeit, die diese Einschränkung nicht haben, diese sind aber für die folgenden Untersuchungen nicht relevant.

Nachdem nun ein Vergleichsbegriff für die Schwierigkeit von Problemen aus **NP** gefunden wurde, kann auch definiert werden, was unter einem „schwierigsten“ Problem in **NP** zu verstehen ist. Ein Problem C ist ein schwierigstes Problem in **NP**, wenn alle anderen Probleme in **NP** höchstens so schwer wie C sind. Formaler ausgedrückt sind dazu zwei Eigenschaften von C nachzuweisen:

(1) C ist ein Problem aus **NP**.

(2) C ist mindestens so schwierig wie jedes andere **NP**-Problem A; d.h.: für alle Probleme A aus

NP gilt: $A \leq C$.

Algorithmus 6: Algorithmische Darstellung der Benutzung einer Reduktionsfunktion

Eingabe: Instanz x für das Problem A
Ergebnis: 1 wenn $x \in A$ und 0 sonst

```
begin
    /* T ist die Reduktionsfunktion (polynomialzeitberechenbar) */
    berechne y = T(x);
                                        /* y ist Instanz des Problems B */
    z = U_B(y);
                                    /* z ist 1 genau dann, wenn x ∈ A gilt */
    return z;
end
```

Solche schwierigsten Probleme in **NP** sind unter der Bezeichnung **NP**-*vollständige Probleme* bekannt. Nun sieht die Aufgabe, von einem Problem zu zeigen, dass es **NP**-vollständig ist, ziemlich hoffnungslos aus. Immerhin ist zu zeigen, dass für alle Probleme aus **NP** – und damit unendlich viele – gilt, dass sie höchstens so schwer sind wie das zu untersuchende Problem, und damit scheint man der Schwierigkeit beim Nachweis unterer Schranken nicht entgangen zu sein. Dennoch konnten der russische Mathematiker Leonid Levin und der amerikanische Mathematiker Stephen Cook Anfang der siebziger Jahre des letzten Jahrhunderts unabhängig voneinander die Existenz von solchen **NP**-vollständigen Problemen zeigen. Hat man nun erst einmal *ein* solches Problem identifiziert, ist die Aufgabe, *weitere* **NP**-vollständige Probleme zu finden, wesentlich leichter. Dies ist sehr leicht einzusehen: Ein **NP**-Problem C ist ein schwierigstes Problem in **NP**, wenn es ein anderes schwierigstes Problem B gibt, sodass C nicht leichter als B ist. Das führt zu folgendem „Kochrezept“:

Nachweis der NP-Vollständigkeit eines Problems C:

i) Zeige, dass C in **NP** enthalten ist, indem dafür ein geeigneter nichtdeterministischer Polynomialzeitalgorithmus konstruiert wird.

ii) Suche ein geeignetes „ähnliches“ schwierigstes Problem B in **NP** und zeige, dass C nicht leichter als B ist. Formal: Finde ein **NP**-vollständiges Problem B und zeige $B \leq C$ mit Hilfe einer geeigneten Transformation T.

Den zweiten Schritt kann man oft relativ leicht mit Hilfe von bekannten Sammlungen **NP**-vollständiger Problemen erledigen. Das Buch von Garey und Johnson [GJ79] ist eine solche Sammlung (siehe auch die Abbildungen 9 und 10), die mehr als 300 **NP**-vollständige Probleme enthält. Dazu wählt man ein möglichst ähnliches Problem aus und versucht dann eine geeignete Reduktionsfunktion für das zu untersuchende Problem zu finden.

6.3.1 Traveling Salesperson ist NP-vollständig

Wie kann man zeigen, dass Traveling Salesperson **NP**-vollständig ist? Dazu wird zuerst die genaue Definition dieses Problems benötigt:

Problem:	TRAVELING SALESPERSON (TSP)
Eingabe:	Eine Menge von Städten $C = \{c_1, \ldots, c_n\}$ und eine $n \times n$ Entfernungsmatrix D, wobei das Element $D[i, j]$ der Matrix D die Entfernung zwischen Stadt c_i und c_j angibt. Weiterhin eine Obergrenze $k \geq 0$ für die maximal erlaubte Länge der Tour

Aus dem Graphen G links berechnet die Transformation die rechte Eingabe für das TSP. Die dick gezeichneten Verbindungen deuten eine Entfernung von 1 an, wogegen dünne Linien eine Entfernung von 6 symbolisieren. Weil G den Hamiltonkreis $1, 2, 3, 4, 5, 1$ hat, gibt es rechts eine Rundreise $1, 2, 3, 4, 5, 1$ mit Gesamtlänge 5.

Im Gegensatz dazu berechnet die Transformation hier aus dem Graphen G' auf der linken eine Eingabe für das TSP auf der rechten Seite, die, wie man sich leicht überzeugt, keine Rundreise mit einer maximalen Gesamtlänge von 5 hat. Dies liegt daran, dass der ursprüngliche Graph G' keinen Hamiltonschen Kreis hatte.

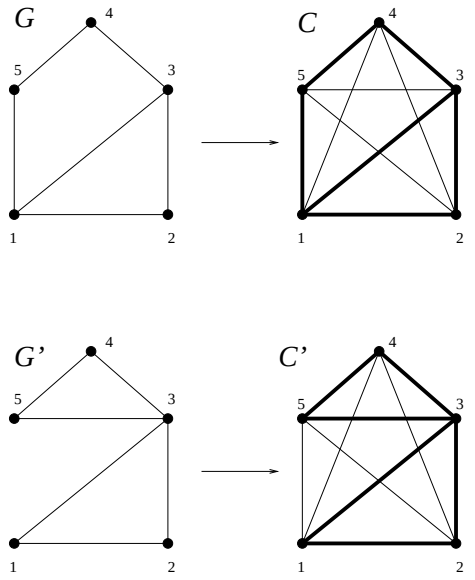


Abbildung 8: Beispiele für die Wirkungsweise von Algorithmus 7

Frage: Gibt es eine Rundreise, die einerseits alle Städte besucht, aber andererseits eine Gesamtlänge von höchstens k hat?

Nun zum ersten Schritt des Nachweises der **NP**-Vollständigkeit von TSP: Offensichtlich gehört auch das Traveling Salesperson Problem zur Klasse **NP**, denn man kann nichtdeterministisch eine Folge von n Städten raten (eine potenzielle Rundreise) und dann leicht überprüfen, ob diese potenzielle Tour durch alle Städte verläuft und ob die zurückzulegende Entfernung maximal k beträgt. Ein entsprechender nichtdeterministischer Polynomialzeitalgorithmus ist leicht zu erstellen. Damit ist der erste Schritt zum Nachweis der **NP**-Vollständigkeit von TSP getan und Punkt (1) des „Kochrezepts“ abgehandelt.

Als nächstes (Punkt (2)) soll von einem anderen **NP**-vollständigen Problem gezeigt werden, dass es effizient in TSP transformiert werden kann. Geeignet dazu ist das im Text betrachtete Hamitonkreis-Problem, das bekanntermaßen **NP**-vollständig ist. Es ist also zu zeigen: HAMILTON $\leq$ TSP.

Folgende Idee führt zum Ziel: Gegeben ist eine Instanz $G = (V, E)$ von HAMILTON. Transformiere G in folgende Instanz von TSP: Als Städtemenge C wählen wir die Knoten V des Graphen G. Die Entfernungen zwischen den Städten sind definiert wie folgt: $D[i, j] = 1$, falls es in E eine Kante von Knoten i zu Knoten j gibt, ansonsten setzt man $D[i, j]$ auf einen sehr großen Wert, also z.B. $n + 1$, wenn n die Anzahl der Knoten von G ist. Dann gilt klarerweise: Wenn G einen Hamiltonschen Kreis besitzt, dann ist der gleiche Kreis eine Rundreise in C mit Gesamtlänge n. Wenn G keinen Hamiltonschen Kreis besitzt, dann kann es keine Rundreise durch die Städte C mit Länge höchstens n geben, denn jede Rundreise muss mindestens eine Strecke von einer Stadt i nach einer Stadt j zurücklegen, die keiner Kante in G entspricht (denn ansonsten hätte G ja einen Hamiltonschen Kreis). Diese einzelne Strecke von i nach j hat dann aber schon Länge $n + 1$ und damit ist eine Gesamtlänge von n oder weniger nicht mehr erreichbar. Die Abbildung 8 zeigt zwei Beispiele für die Wirkungsweise der Transformation, die durch Algorithmus 7 in Polynomialzeit berechnet wird.

Algorithmus 7: Ein Algorithmus für die Reduktion von HAMILTON auf TSP

Eingabe: Graph $G = (V, E)$, wobei $V = \{1, \ldots, n\}$
Ergebnis: Eine Instanz (C, D, k) für TSP

```
begin
    /* Die Knoten entsprechen den Städten */
    C = V;
    /* Überprüfe alle potentiell existierenden Kanten */
    for (i = 1 to n) do
        for (j = 1 to n) do
            if ((v_i, v_j) ∈ E) then
                /* Kanten entsprechen kleinen Entfernungen */
                D[i][j] = 1;
            else
                /* nicht existierende Kante, dann sehr große Entfernung
                */
                D[i][j] = n + 1;
            end
        end
    end
    /* Gesamtlänge k der Rundreise ist Anzahl der Städte n */
    k = n;
    /* Gebe die berechnete TSP-Instanz zurück */
    return (C, D, k);
end
```

6.4 Die Auswirkungen der NP-Vollständigkeit

Welche Bedeutung haben nun die **NP**-vollständigen Probleme für die Klasse **NP**? Könnte jemand einen deterministischen Polynomialzeitalgorithmus $\mathcal{A}_C$ für ein **NP**-vollständiges Problem C angeben, dann hätte man für jedes **NP**-Problem einen Polynomialzeitalgorithmus gefunden (d.h. **P** $=$ **NP**). Diese überraschende Tatsache lässt sich leicht einsehen, denn für jedes Problem A aus **NP** gibt es eine Transformation T mit der Eigenschaft, dass x genau dann eine positive Eingabeinstanz von A ist, wenn $T(x)$ eine positive Instanz von C ist. Damit löst Algorithmus 8 das Problem A in Polynomialzeit. Es gilt also: Ist irgendein **NP**-vollständiges Problem effizient lösbar, dann ist **P** $=$ **NP**.

Algorithmus 8: Ein fiktiver Algorithmus für Problem A

Eingabe: Instanz x für das Problem A
Ergebnis: `true`, wenn $x \in A$, `false` sonst

begin
  `/* T ist die postulierte Reduktionsfunktion */`
  $y = T(x)$;
  $z = \mathcal{A}_C(y)$;
  return z;
end

Sei nun angenommen, dass jemand **P** $\neq$ **NP** gezeigt hat. In diesem Fall ist aber auch klar, dass dann für kein **NP**-vollständiges Problem ein Polynomialzeitalgorithmus existieren kann, denn sonst würde sich ja der Widerspruch **P** $=$ **NP** ergeben. Ist das Problem C also **NP**-vollständig, so gilt: C hat genau dann einen effizienten Algorithmus, wenn **P** $=$ **NP**, also wenn jedes Problem in **NP** einen effizienten Algorithmus besitzt. Diese Eigenschaft macht die **NP**-vollständigen Probleme für die Theoretiker so interessant, denn eine Klasse von unendlich vielen Problemen kann untersucht werden, indem man nur ein einziges Problem betrachtet. Man kann sich das auch wie folgt vorstellen: Alle relevanten Eigenschaften aller Probleme aus **NP** wurden in ein einziges Problem „destilliert". Die **NP**-vollständigen Probleme sind also in diesem Sinn *prototypische* **NP**-Probleme.

Trotz intensiver Bemühungen in den letzten 30 Jahren konnte bisher niemand einen Polynomialzeitalgorithmus für ein **NP**-vollständiges Problem finden. Dies ist ein Grund dafür, dass man heute **P** $\neq$ **NP** annimmt. Leider konnte auch dies bisher nicht gezeigt werden, aber in der theoretischen Informatik gibt es starke Indizien für die Richtigkeit dieser Annahme, sodass heute die große Mehrheit der Forscher von **P** $\neq$ **NP** ausgeht.

Für die Praxis bedeutet dies Folgendes: Hat man von einem in der Realität auftretenden Problem gezeigt, dass es **NP**-vollständig ist, dann kann man getrost aufhören, einen effizienten Algorithmus zu suchen. Wie wir ja gesehen haben, kann ein solcher nämlich (zumindest unter der gut begründbaren Annahme **P** $\neq$ **NP**) nicht existieren.

Nun ist auch eine Antwort für das 3COL-Problem gefunden. Es wurde gezeigt [GJ79], dass kCOL für $k \geq 3$ **NP**-vollständig ist. Der fiktive Mobilfunkplaner hat also Pech gehabt: Es ist unwahrscheinlich, dass er jemals ein korrektes effizientes Planungsverfahren finden wird.

Ein **NP**-Vollständigkeitsnachweis eines Problems ist also ein starkes Indiz für seine praktische Nicht-Handhabbarkeit. Auch die **NP**-Vollständigkeit eines Problems, das mit dem Spiel *Minesweeper* zu tun hat, bedeutet demnach lediglich, dass dieses Problem höchstwahrscheinlich nicht effizient lösbar sein wird. Ein solcher Vollständigkeitsbeweis hat nichts mit einem Schritt in Richtung auf eine Lösung des **P** $\stackrel{?}{=}$ **NP**-Problems zu tun, wie irreführenderweise gelegentlich zu lesen ist. Übrigens ist auch für eine Reihe weiterer Spiele ihre **NP**-Vollständigkeit bekannt. Dazu gehören u.a. bestimmte Puzzle- und Kreuzwortspiele. Typische Brettspiele, wie Dame, Schach oder GO, sind hingegen (verallgemeinert auf Spielbretter der Größe $n \times n$) **PSPACE**-vollständig. Die Klasse

PSPACE ist eine noch deutlich mächtigere Klasse als **NP**. Damit sind also diese Spiele noch viel komplexer als Minesweeper und andere **NP**-vollständige Probleme.

6.5 Der Umgang mit NP-vollständigen Problemen in der Praxis

Viele in der Praxis bedeutsame Probleme sind **NP**-vollständig (vgl. die Abbildungen 9 und 10). Ein Anwendungsentwickler wird es aber sicher schwer haben, seinem Management mitteilen zu müssen, dass ein aktuelles Projekt nicht durchgeführt werden kann, weil keine geeigneten Algorithmen zur Verfügung stehen (Wahrscheinlich würden in diesem Fall einfach „geeignetere“ Entwickler eingestellt werden!). Es stellt sich daher also die Frage, wie man mit solchen **NP**-vollständigen Problemen in der Praxis umgeht. Zu dieser Fragestellung hat die theoretische Informatik ein ausgefeiltes Instrumentarium entwickelt.

Eine erste Idee wäre es, sich mit Algorithmen zufrieden zu geben, die mit Zufallszahlen arbeiten und die nur mit sehr großer Wahrscheinlichkeit die richtige Lösung berechnen, aber sich auch mit kleiner (vernachlässigbarer) Wahrscheinlichkeit irren dürfen. Solche Algorithmen sind als *probabilistische* oder *randomisierte Algorithmen* bekannt [MR95] und werden beispielsweise in der Kryptographie mit sehr großem Erfolg angewendet. Das prominenteste Beispiel hierfür sind Algorithmen, die testen, ob eine gegebene Zahl eine Primzahl ist und sich dabei fast nie irren. Primzahlen spielen bekanntermaßen im RSA-Verfahren und damit bei PGP und ähnlichen Verschlüsselungen eine zentrale Rolle. Es konnte aber gezeigt werden, dass probabilistische Algorithmen uns bei den **NP**-vollständigen Problemen wohl nicht weiterhelfen. So weiß man heute, dass die Klasse der Probleme, die sich mit probabilistischen Algorithmen effizient lösen lässt, höchstwahrscheinlich nicht die Klasse **NP** umfasst. Deshalb liegen (höchstwahrscheinlich) insbesondere alle **NP**-vollständigen Probleme außerhalb der Möglichkeiten von effizienten probabilistischen Algorithmen.

Nun könnte man auch versuchen, „exotischere“ Computer zu bauen. In der letzten Zeit sind zwei potenzielle Auswege bekannt geworden: DNA-Computer und Quantencomputer.

Es konnte gezeigt werden, dass DNA-Computer (siehe [Pău98]) jedes **NP**-vollständige Problem in Polynomialzeit lösen können. Für diese Berechnungsstärke hat man aber einen Preis zu zahlen: Die Anzahl und damit die Masse der DNA-Moleküle, die für die Berechnung benötigt werden, wächst exponentiell in der Eingabelänge. Das bedeutet, dass schon bei recht kleinen Eingaben mehr Masse für eine Berechnung gebraucht würde, als im ganzen Universum vorhanden ist. Bisher ist kein Verfahren bekannt, wie dieses Masseproblem gelöst werden kann, und es sieht auch nicht so aus, als ob es gelöst werden kann, wenn $\mathbf{P} \neq \mathbf{NP}$ gilt. Dieses Problem erinnert an das oben im Kontext von Parallelrechnern schon erwähnte Phänomen: Mit exponentiell vielen Prozessoren lassen sich **NP**-vollständige Probleme lösen, aber solche Parallelrechner haben natürlich explodierende Hardware-Kosten.

Der anderer Ausweg könnten Quantencomputer sein (siehe [Hom08, Gru99]). Hier scheint die Situation zunächst günstiger zu sein: Die Fortschritte bei der Quantencomputer-Forschung verlaufen immens schnell, und es besteht die berechtigte Hoffnung, dass Quantencomputer mittelfristig verfügbar sein werden. Aber auch hier sagen theoretische Ergebnisse voraus, dass Quantencomputer (höchstwahrscheinlich) keine **NP**-vollständigen Probleme lösen können. Trotzdem sind Quantencomputer interessant, denn es ist bekannt, dass wichtige Probleme existieren, für die kein Polynomialzeitalgorithmus bekannt ist und die wahrscheinlich nicht **NP**-vollständig sind, die aber auf Quantencomputern effizient gelöst werden können. Das prominenteste Beispiel hierfür ist die Aufgabe, eine ganze Zahl in ihre Primfaktoren zu zerlegen.

Die bisher angesprochenen Ideen lassen also die Frage, wie man mit **NP**-vollständigen Problemen umgeht, unbeantwortet. In der Praxis gibt es im Moment zwei Hauptansatzpunkte: Die erste Möglichkeit ist die, die Allgemeinheit des untersuchten Problems zu beschränken und eine spezielle Version zu betrachten, die immer noch für die geplante Anwendung ausreicht. Zum Beispiel sind Graphenprobleme oft einfacher, wenn man zusätzlich fordert, dass die Knoten des Graphen

Problemnummern in „[…]“ beziehen sich auf die Sammlung von Garey und Johnson [GJ79].

Problem: CLUSTER [GT19]
Eingabe: Netzwerk $G = (V, E)$, positive Integerzahl K
Frage: Gibt es eine Menge von mindestens K Knoten, die paarweise miteinander verbunden sind?

Problem: NETZ-AUFTEILUNG [ND16]
Eingabe: Netzwerk $G = (V, E)$, Kapazität für jede Kante in E, positive Integerzahl K
Frage: Kann man das Netzwerk so in zwei Teile zerlegen, dass die Gesamtkapazität aller Verbindungen zwischen den beiden Teilen mindestens K beträgt?

Problem: NETZ-REDUNDANZ [ND18]
Eingabe: Netzwerk $G = (V, E)$, Kosten für Verbindungen zwischen je zwei Knoten aus V, Budget B
Frage: Kann G so um Verbindungen erweitert werden, dass zwischen je zwei Knoten mindestens zwei Pfade existieren und die Gesamtkosten für die Erweiterung höchstens B betragen?

Problem: OBJEKTE SPEICHERN [SR1]
Eingabe: Eine Menge U von Objekten mit Speicherbedarf $s(u)$ für jedes $u \in U$; Kachelgröße S, positive Integerzahl K
Frage: Können die Objekte in U auf K Kacheln verteilt werden?

Problem: DATENKOMPRESSION [SR8]
Eingabe: Endliche Menge R von Strings über festgelegtem Alphabet, positive Integerzahl K
Frage: Gibt es einen String S der Länge höchstens K, sodass jeder String aus R als Teilfolge von S vorkommt?

Problem: K-SCHLÜSSEL [SR26]
Eingabe: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, positive Integerzahl K
Frage: Gibt es einen Schlüssel mit höchstens K Attributen?

Problem: BCNF [SR29]
Eingabe: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, Teilmenge $A' \subseteq A$
Frage: Verletzt die Menge A' die Boyce-Codd-Normalform?

Problem: MP-SCHEDULE [SS8]
Eingabe: Menge T von Tasks, Länge für jede Task, Anzahl m von Prozessoren, positive Integerzahl D („Deadline“)
Frage: Gibt es ein m-Prozessor-Schedule für T mit Ausführungszeit höchstens D?

Problem: PREEMPT-SCHEDULE [SS12]
Eingabe: Menge T von Tasks, Länge für jede Task, Präzedenzrelation auf den Tasks, Anzahl m von Prozessoren, positive Integerzahl D („Deadline“)
Frage: Gibt es ein m-Prozessor-Schedule für T, das die Präzedenzrelationen berücksichtigt und Ausführungszeit höchstens D hat?

Problem: DEADLOCK [SS22]
Eingabe: Menge von Prozessen, Menge von Ressourcen, aktuelle Zustände der Prozesse und aktuell allokierte Ressourcen
Frage: Gibt es einen Kontrollfluss, der zum Deadlock führt?

Problem: K-REGISTER [PO3]
Eingabe: Menge V von Variablen, die in einer Schleife benutzt werden, für jede Variable einen Gültigkeitsbereich, positive Integerzahl K
Frage: Können die Schleifenvariablen mit höchstens K Registern gespeichert werden?

Problem: REKURSION [PO20]
Eingabe: Menge A von Prozedur-Identifiern, Pascal-Programmfragment mit Deklarationen und Aufrufen der Prozeduren aus A
Frage: Ist eine der Prozeduren aus A formal rekursiv?

Abbildung 9: Eine kleine Sammlung **NP**-vollständiger Probleme (Teil 1)

Problemnummern in „[…]“ beziehen sich auf die Sammlung von Garey und Johnson [GJ79].

Problem:	LR(K)-GRAMMATIK [AL15]
Eingabe:	Kontextfreie Grammatik G, positive Integerzahl K (unär)
Frage:	Ist die Grammatik G nicht LR(K)?

Problem:	ZWANGSBEDINGUNG [LO5]
Eingabe:	Menge von Booleschen Constraints, positive Integerzahl K
Frage:	Können mindestens K der Constraints gleichzeitig erfüllt werden?

Problem:	INTEGER PROGRAM [MP1]
Eingabe:	Lineares Programm
Frage:	Hat das Programm eine Lösung, die nur ganzzahlige Werte enthält?

Problem:	KREUZWORTRÄTSEL [GP15]
Eingabe:	Menge W von Wörtern, Gitter mit schwarzen und weißen Feldern
Frage:	Können die weißen Felder des Gitters mit Wörtern aus W gefüllt werden?

Abbildung 10: Eine kleine Sammlung **NP**-vollständiger Probleme (Teil 2)

in der (Euklidischen) Ebene lokalisiert sind. Deshalb sollte die erste Idee bei der Behandlung von **NP**-vollständigen Problemen immer sein, zu untersuchen, welche Einschränkungen man an das Problem machen kann, ohne die praktische Aufgabenstellung zu verfälschen. Gerade diese Einschränkungen können dann effiziente Algorithmen ermöglichen.

Die zweite Möglichkeit sind sogenannte *Approximationsalgorithmen* (vgl. [ACG+99]). Die Idee hier ist es, nicht die optimalen Lösungen zu suchen, sondern sich mit einem kleinen garantierten Fehler zufrieden zu geben. Dazu folgendes Beispiel. Es ist bekannt, dass das TSP auch dann noch **NP**-vollständig ist, wenn man annimmt, dass die Städte in der Euklidischen Ebene lokalisiert sind, d.h. man kann die Städte in einer fiktiven Landkarte einzeichnen, sodass die Entfernungen zwischen den Städten proportional zu den Abständen auf der Landkarte sind. Das ist sicherlich in der Praxis keine einschränkende Abschwächung des Problems und zeigt, dass die oben erwähnte Methode nicht immer zum Erfolg führen muss: Hier bleibt auch das eingeschränkte Problem **NP**-vollständig. Aber für diese eingeschränkte TSP-Variante ist ein Polynomialzeitalgorithmus bekannt, der immer eine Rundreise berechnet, die höchstens um einen beliebig wählbaren Faktor schlechter ist, als die optimale Lösung. Ein Chip-Hersteller, der bei der Bestückung seiner Platinen die Wege der Roboterköpfe minimieren möchte, kann also beschließen, sich mit einer Tour zufrieden zu geben, die um 5 % schlechter ist als die optimale. Für dieses Problem existiert ein effizienter Algorithmus! Dieser ist für die Praxis völlig ausreichend.

⋆ ⋆ ⋆ Ende ⋆ ⋆ ⋆

Stichwortverzeichnis

Symbole

$2\mathbb{Z}$ . . . 8
A^n . . . 9
. . . 20
□ . . . 20
$\mathbb{N}$ . . . 8
$\mathbb{P}$ . . . 8
$\mathcal{P}(A)$ . . . 8
$\mathbb{Z}$ . . . 8
∩ . . . 8
. . . 10
∪ . . . 8
∅ . . . 8
¬ . . . 17
↔ . . . 17
→ . . . 17
∨ . . . 17
∧ . . . 17
≡ . . . 11
∃ . . . 17
∀ . . . 17
∈ . . . 7
$\lceil \cdot \rceil$ . . . 11
$\lfloor \cdot \rfloor$ . . . 11
∄ . . . 18
∋ . . . 7
∉ . . . 7
⊈ . . . 7
$\overline{A}$ . . . 8
π . . . 12
$\prod$ . . . 14
\ . . . 8
$\{\emptyset\}$ . . . 8
⊂ . . . 7
⊆ . . . 7
$\sum$ . . . 13
× . . . 9
$f(\cdot)$ . . . 11
k-Färbung . . . 34

A

adjazent . . . 27
Adjazenzmatrix . . . 32
Algorithmus . . .
    probabilistisch . . . 49
    randomisiert . . . 49
antisymmetrisch . . . 10
Approximationsalgorithmen . . . 51
Äquivalenzrelation . . . 10
Ausgrad . . . 29
Aussageformen . . . 17
Aussagenvariablen . . . 17

B

Basis . . . 15
Baum . . . 31
benachbart . . . 27
Berechnungsbaum . . . 42
Berechnungsmodell . . . 37
Beweis . . .
    direkt . . . 19
    Gegenbeispiel . . . 22
    Induktion . . .
        strukturell . . . 25
        vollständig . . . 23
    Kontraposition . . . 21
    Ringschluss . . . 21
    Schubfach . . . 22
    Widerspruch . . . 22
bijektiv . . . 12
Bing . . . 3
binäre Relation . . . 10
bipartit . . . 28
Bitkomplexität . . . 37
Brückenproblem . . . 27
Buchstabe . . .
    griechisch . . . 16
Buchstaben . . .
    griechische . . . 16

D

Deduktion . . . 4
deduktive Methode . . . 4
Definitionsbereich . . . 11
Differenz . . . 8
direkten Beweis . . . 19
Dirichlets Taubenschlagprinzip . . . 22
disjunkt . . . 8

E

Elemente . . . 7
Endknoten . . . 27, 31
endlich . . . 28
endliche Mengen . . . 10
Entscheidungsprobleme . . . 35
Erkenntnistheorie . . . 5
Eulersche ϕ-Funktion . . . 14
Exponent . . . 15

F

falsch . . . 17
Funktion . . . 11
Färbung . . . 34
    verträglich . . . 34

G

ganzen Zahlen . . . 8
gdw. . . . 7, 17, 21
Gegenbeispiel . . . 22
genau dann wenn . . . 7
gerichteter Graph . . . 27
geschlossener Weg . . . 31
Google . . . 3
Grad . . . 29
Graph . . . 29
    gerichtet . . . 27
    Null . . . 29
    ungerichtet . . . 29
Greedy-Algorithmus . . . 37
griechische Buchstaben . . . 16

H

Halbordnung . . . 10
Hamiltonkreis-Problem . . . 40
Hamiltonscher Kreis . . . 40
Heirat . . . 28
Heiratsproblem . . . 28
Hypothese . . . 4
Hüllenoperator . . . 12

I

indeg(v) . . . 29
Induktion . . .
    Prinzip . . . 23
        verallgemeinert . . . 24
    strukturelle . . . 25
    vollständige . . . 23
Induktionsanfang . . . 23
Induktionsbeweises . . . 23
Induktionsprinzip . . . 23
Induktionsschritt . . . 23
Induktionsvoraussetzung . . . 23
induktive Definition . . . 25
Ingrad . . . 29
injektiv . . . 12
Instanz . . .
    positiv . . . 35

K

Kanten . . . 27
Kantenrelation . . . 27, 32
Knoten . . . 27
    End . . . 27, 31
    Start . . . 27, 31
Komplement . . . 8
Konklusion . . . 4
Kontraposition . . . 21
Kreis . . . 31
Kreuzprodukt . . . 9
Königsberger Brückenproblem . . . 27

L

linear . . . 10
Logarithmus . . . 15
Logik . . .
    Prädikat . . . 17
logischer Operator . . . 17

M

Menge . . . 7
    Differenz- . . . 8
    endliche . . . 10
    Komplement- . . . 8
    Potenz- . . . 8
    Schnitt- . . . 8
    Teil- . . . 7
    Vereinigung- . . . 8
Methode . . .
    deduktive . . . 4
Mitschrift . . . 3
Multigraph . . . 27

N

natürlichen Zahlen . . . 8
nichtdeterministisch . . . 42
NP . . . 40
Nullgraph . . . 29

O

Operator . . . 12
    Hüllen . . . 12
    logisch . . . 17
Ordnung . . . 10
outdeg(v) . . . 29

P

P . . . 39
Paar . . . 10
Permutation . . . 12
planar . . . 29
positiven Instanz . . . 35
Potenz . . . 15
Potenzierung . . . 15
Potenzmenge . . . 8
Primzahlen . . . 8

Problem . . . . . .
Hamiltonkreis 40
Problemlösungskompetenz 2
Prädikat 7, 11, 17
Prädikatenlogik 17

Q

qed 20
Quadrupel 10
Quintupel 10

R

Radizieren 15
reflexiv 10
regulär 29
Relation 10
binär 10
Kante 27
Riemannsche Zetafuntion $\zeta(s)$ 15
Ringschluss 21

S

Schleife 29
schleifenfrei 29
Schnitt 8
Schubfachschluss 22
Startknoten 27, 31
Strenge 4
strukturellen Induktion 25
surjektiv 11
symmetrisch 10, 29

T

Taubenschlagprinzip 22
Teilmenge 7
total 11
transitiv 10
transitive Hülle 12
Tripel 10
Tupel 9
n-Tupel 9

U

Überprüfungsphase 42
ungerichteten Graphen 29
uniformem Komplexitätsmaß 37
Untergraph 28

V

Vereinigung 8
vollständig 29, 45
vollständige Induktion 23

W

wahr 17
Wald 31
Weg 31
geschlossen 31
Wertebereich 11
Widerspruchsbeweis 22
Wikipedia 3
Wurzel 15

Z

Zahlen . . . . . .
ganz 8
natürlich 8
Zusammenhangskomponente 31
zusammenhängend 31

Literatur

[ACG+99] G. Ausiello, P. Crescenzi, V. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability*. Springer Verlag, 1999.

[Can95] G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[Gru99] J. Gruska. *Quantum Computing*. McGraw-Hill, 1999.

[Hom08] M. Homeister. *Quantum Computing verstehen*. Vieweg, 2008.

[MB14] P. A. Mueller and D. M. Bernstein. The pen is mightier than the keyboard: Advantages of longhand over laptop note taking. *Psychological Science*, 2014.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Pău98] G. Păun. *Computing with Bio-Molecules*. Springer Series in Discrete Mathematics and Theoretical Computer Science. Springer Verlag, Singapore, 1998.

[Sch01] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001.