Zwei Kapitel in "Algorithms of Computer Science"[1]

JÖRG ROTHE[2]

13. Dezember 2004

[1]Erscheint in "Informatikai algoritmusok," Antal Iványi, Editor, ELTE Eötvös Kiadó, Budapest, 2004. Übersetzung ins Ungarische durch Zsuzsa Láng und Csaba Sidló.

[2]Email: rothe@cs.uni-duesseldorf.de. Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, 40225 Düsseldorf, Germany. Unterstützt durch die Deutsche Forschungsgemeinschaft unter Kennzeichen RO 1202/9-1.

Inhaltsverzeichnis

1 Algorithmen in der Kryptologie **5**
- 1.1 Einleitung . . . 5
- 1.2 Grundlagen . . . 6
  - 1.2.1 Kryptographie . . . 7
  - 1.2.2 Kryptoanalyse . . . 10
  - 1.2.3 Algorithmik . . . 12
  - 1.2.4 Algebra, Zahlentheorie und Graphentheorie . . . 17
- 1.3 Schlüsseltausch nach Diffie und Hellman . . . 23
- 1.4 RSA und Faktorisierung . . . 26
  - 1.4.1 RSA . . . 26
  - 1.4.2 Digitale Signaturen mit RSA . . . 29
  - 1.4.3 Sicherheit von RSA und mögliche Angriffe auf RSA . . . 29
- 1.5 Die Protokolle von Rivest, Rabi und Sherman . . . 31
- 1.6 Interaktive Beweissysteme und Zero-Knowledge . . . 32
  - 1.6.1 Interaktive Beweissysteme, Arthur-Merlin-Spiele und Zero-Knowledge-Protokolle . . . 32
  - 1.6.2 Zero-Knowledge-Protokoll für Graphisomorphie . . . 34

2 Algorithmen in der Komplexitätstheorie **39**
- 2.1 Einleitung . . . 39
- 2.2 Grundlagen . . . 40
- 2.3 NP-Vollständigkeit . . . 46
- 2.4 Das Erfüllbarkeitsproblem der Aussagenlogik . . . 52
  - 2.4.1 Deterministische Zeitkomplexität von 3-SAT . . . 52
  - 2.4.2 Probabilistische Zeitkomplexität von 3-SAT . . . 53
- 2.5 Graphisomorphie und Lowness . . . 56
  - 2.5.1 Reduzierbarkeiten und Komplexitätshierarchien . . . 57
  - 2.5.2 Graphisomorphie ist in der Low-Hierarchie . . . 62
  - 2.5.3 Graphisomorphie ist in SPP . . . 64

Kapitel 1

Algorithmen in der Kryptologie

1.1 Einleitung

In diesem Kapitel werden kryptographische Protokolle und die ihnen zugrunde liegenden Probleme und Algorithmen vorgestellt. Ein typisches Szenario in der Kryptographie ist in Abbildung 1.1 dargestellt. Alice und Bob wollen Nachrichten über einen unsicheren Kanal austauschen, z.B. über eine öffentliche Telefonleitung oder über elektronische Post zwischen zwei vernetzten Computern. Erich hat die Leitungen angezapft und belauscht den Datentransfer. Da Alice und Bob dies wissen, verschlüsseln sie ihre Botschaften.

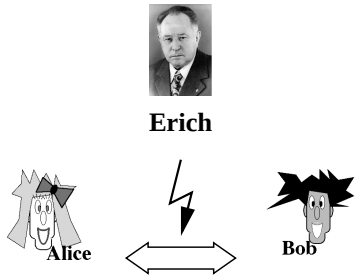


Abbildung 1.1: Typisches Szenario in der Kryptographie.

In Abschnitt 1.2 werden verschiedene symmetrische Kryptosysteme präsentiert, die ein unbefugtes Entschlüsseln verhindern sollen. Symmetrisch heißt ein Kryptosystem, falls derselbe Schlüssel zum Ver- und zum Entschlüsseln verwendet werden kann. Wie aber sollen sich Alice und Bob auf einen gemeinsamen Schlüssel einigen, wenn sie doch nur über einen unsicheren Kanal miteinander kommunizieren können? Würde Alice zum Beispiel einen Schlüssel wählen und verschlüsselt (etwa mit einem symmetrischen Kryptosystem) an Bob schicken, so ergäbe sich sofort die Frage, welchen Schlüssel sie denn zum Verschlüsseln dieses Schlüssels verwenden soll.

Diese paradox anmutende Frage, die ein wenig an die Frage erinnert, ob es das Ei vor der Henne oder die Henne vor dem Ei gab, ist als das *Problem des Schlüsseltauschs* bekannt. Seit den Anfängen der Kryptographie galt dieses Problem als unlösbar. Um so größer war das Erstaunen, als Whitfield Diffie und Martin Hellman es 1976 lösten. Sie schlugen ein Protokoll vor, bei dem Alice und Bob Informationen tauschen, mittels derer sie schließlich ihren gemeinsamen Schlüssel berechnen können. Der Lauscher Erich jedoch hat, auch wenn er jedes einzelne Bit ihrer Datenübertragung abfangen konnte, von ihrem Schlüssel keine Ahnung. Abschnitt 1.3 präsentiert das Diffie–Hellman-Protokoll.

Es ist eine kleine Ironie der Geschichte, dass gerade dieses Protokoll, das das für unlösbar gehaltene Problem des Schlüsseltauschs löste, welches in der symmetrischen Kryptographie so wichtig ist, der Erfindung eines anderen Protokolls den Weg ebnete, bei dem der geheime Schlüsseltausch über unsichere Kanäle gar keine Rolle mehr spielt. Diffie und Hellman hatten mit ihrer Arbeit [12] die Tür zur modernen *public-key*-Kryptographie aufgeschlossen, und schon zwei Jahre später, 1978, stießen Rivest, Shamir und Adleman diese Tür weit auf, als sie mit ihrem berühmten RSA-System das erste *public-key*-Kryptosystem schufen. Abschnitt 1.4 beschreibt das RSA-System sowie ein auf RSA beruhendes Protokoll für digitale Unterschriften. Mit einem solchen Protokoll kann Alice ihre Botschaften an Bob so signieren, dass dieser ihre Identität als Senderin der Botschaft überprüfen kann. Digitale Unterschriften sollen verhindern, dass Erich Alice' Botschaften fälscht und so tut, als hätte Alice sie gesendet.

Die Sicherheit des Diffie–Hellman-Protokolls beruht auf der Annahme, dass der diskrete Logarithmus nicht effizient berechnet werden kann. Daher gilt die modulare Exponentiation, deren Umkehrfunktion der diskrete Logarithmus ist, als ein Kandidat für eine Einwegfunktion. Auch die Sicherheit von RSA beruht auf der vermuteten Härte eines Problems, nämlich auf der Annahme, dass große Zahlen nicht effizient in ihre Primfaktoren zerlegt werden können. Der legale Empfänger Bob kann den Schlüsseltext jedoch effizient entschlüsseln, indem er die Faktorisierung einer von ihm gewählten Zahl als seine private "Falltür"-Information nutzt.

In Abschnitt 1.5 wird ein Protokoll von Rivest und Sherman vorgestellt, das auf so genannten stark nichtinvertierbaren assoziativen Einwegfunktionen beruht und ähnlich wie das Protokoll von Diffie und Hellman dem geheimen Schlüsseltausch dient. Auch dieses Protokoll kann in ein Protokoll für digitale Unterschriften umgeformt werden.

Abschnitt 1.6 schließlich führt in das interessante Gebiet der interaktiven Beweissysteme und Zero-Knowledge-Protokolle ein, das praktische Anwendungen in der Kryptographie hat, speziell beim Problem der Authentikation. Insbesondere wird ein Zero-Knowledge-Protokoll für das Graphisomorphieproblem vorgestellt. Andererseits gehört dieses Gebiet aber auch in die Komplexitätstheorie und wird daher in Kapitel 2 erneut aufgegriffen, wieder im Zusammenhang mit dem Graphisomorphieproblem.

1.2 Grundlagen

Kryptographie ist die Jahrtausende alte Kunst und Wissenschaft vom Verschlüsseln von Texten oder Botschaften in Geheimschrift, so dass das Entschlüsseln durch Unbefugte verhindert wird. In diesem Abschnitt werden zwei klassische symmetrische Kryptosysteme vorgestellt, während die folgenden Abschnitte einige der wichtigsten heute gebräuchlichen Protokolle und asymmetrischen Kryptosysteme präsentieren. Unter einem Protokoll versteht man dabei einen Dialog zwischen zwei oder mehreren Parteien, wobei eine "Partei" ein Mensch, aber auch ein Computer sein kann. Kryptographische Protokolle dienen der Verschlüsselung von Texten, so dass der legitime Empfänger den Schlüsseltext einfach und effizient entschlüsseln kann. Protokolle können auch als Algorithmen aufgefasst werden, an deren Ausführung mehrere Parteien beteiligt sind.

Kryptoanalyse ist die Jahrtausende alte Kunst und Wissenschaft vom (unbefugten) Entschlüsseln verschlüsselter Botschaften und vom Brechen bestehender Kryptosysteme. Die *Kryptologie* umfasst diese beiden Gebiete, die Kryptographie und die Kryptoanalyse. Insbesondere auf *kryptographische* Algorithmen, die eine sichere Verschlüsselung ermöglichen sollen, konzentrieren wir uns in diesem Kapitel. Algorithmen der Kryptoanalyse, mit denen man versucht, kryptographische Protokolle und Systeme zu brechen, werden hier zwar auch erwähnt, aber nicht so umfassend und genau untersucht.

1.2.1 Kryptographie

Ein typisches Szenario in der Kryptographie ist in Abbildung 1.1 in Abschnitt 1.1 zu sehen: Alice und Bob kommunizieren über einen unsicheren Kanal, der von Erich abgehört wird, und verschlüsseln daher ihre Botschaften mit einem Kryptosystem.

Definition 1.1 (Kryptosystem) *Ein* Kryptosystem *ist ein Quintuple* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ *mit den folgenden Eigenschaften:*

1. $\mathcal{P}$*,* $\mathcal{C}$ *und* $\mathcal{K}$ *sind endliche Mengen, wobei* $\mathcal{P}$ *der* Klartextraum, $\mathcal{C}$ *der* Schlüsseltextraum *und* $\mathcal{K}$ *der* Schlüsselraum *ist. Die Elemente von* $\mathcal{P}$ *heißen* Klartext *(*"plaintext"*) und die Elemente von* $\mathcal{C}$ *heißen* Schlüsseltext *(*"ciphertext"*). Eine* Botschaft *(*"message"*) ist ein Wort von Klartextsymbolen.*

2. $\mathcal{E} = \{E_k \mid k \in \mathcal{K}\}$ *ist eine Familie von Funktionen* $E_k : \mathcal{P} \to \mathcal{C}$*, die für die Verschlüsselung benutzt werden.* $\mathcal{D} = \{D_k \mid k \in \mathcal{K}\}$ *ist eine Familie von Funktionen* $D_k : \mathcal{C} \to \mathcal{P}$*, die für die Entschlüsselung benutzt werden.*

3. *Für jeden Schlüssel* $e \in \mathcal{K}$ *gibt es einen Schlüssel* $d \in \mathcal{K}$*, so dass für jeden Klartext* $p \in \mathcal{P}$ *gilt:*

$$D_d(E_e(p)) = p. \tag{1.1}$$

Ein Kryptosystem heißt symmetrisch *(oder* "private-key"*), falls entweder* $d = e$ *oder falls* d *zumindest "leicht" aus* e *berechnet werden kann. Ein* Kryptosystem *heißt* asymmetrisch *(oder* "public-key"*), falls* $d \neq e$ *und es "praktisch nicht machbar" ist, den Schlüssel* d *aus dem Schlüssel* e *zu berechnen. Hier heißt* e öffentlicher Schlüssel *und* d *der* zu e gehörige private Schlüssel.

Zuweilen benutzt man auch verschiedene Schlüsselräume für die Ver- und Entschlüsselung, was dann zu einer entsprechenden Modifizierung der obigen Definition führt.

Um nun einige Beispiele für klassische Kryptosysteme vorzustellen, betrachten wir das Alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ sowohl für den Klartextraum als auch für den Schlüsseltextraum. Damit wir mit Buchstaben rechnen können, als ob sie Zahlen wären, identifizieren wir Σ mit $\mathbf{Z}_{26} = \{0, 1, \ldots, 25\}$. Die Zahl 0 entspricht also dem Buchstaben A, die 1 entspricht B und so weiter. Diese Codierung der Klartextsymbole durch natürliche Zahlen gehört natürlich nicht zur eigentlichen Verschlüsselung bzw. Entschlüsselung.

Botschaften sind Elemente von Σ^*, wobei Σ^* die Menge der Wörter über Σ bezeichnet. Wird eine Botschaft $m \in \Sigma^*$ in Blöcke der Länge n aufgeteilt und blockweise verschlüsselt, wie es in vielen Kryptosystemen üblich ist, so können die einzelnen Blöcke von m als Elemente von $\mathbf{Z}_{26}^n$ aufgefasst werden.

Beispiel 1.2 (Verschiebungschiffre) *Das erste Beispiel ist ein monoalphabetisches symmetrisches Kryptosystem. Seien* $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbf{Z}_{26}$*. Die* Verschiebungschiffre *verschlüsselt Botschaften, indem jedes Zeichen des Klartextes um dieselbe Anzahl* k *von Buchstaben (modulo* 26*) im Alphabet verschoben wird. Dabei ist* $k \in \mathbf{Z}_{26}$ *der Schlüssel. Die Rückverschiebung eines jeden Zeichens im Schlüsseltext unter Benutzung desselben Schlüssels* k *enthüllt den Klartext. Die Verschlüsselungsfunktion* E_k *und die Entschlüsselungsfunktion* D_k *sind für jeden Schlüssel* $k \in \mathbf{Z}_{26}$ *definiert durch:*

$$\begin{aligned} E_k(m) &= (m + k) \bmod 26 \\ D_k(c) &= (c - k) \bmod 26, \end{aligned}$$

wobei die Addition und die Subtraktion mit k *modulo* 26 *zeichenweise ausgeführt werden. Tabelle 1.1*

m	B	U	D	A	P	E	S	T	I	S	T	D	A	S	P	A	R	I	S	D	E	S	O	S	T	E	N	S
c	E	X	G	D	S	H	V	W	L	V	W	G	D	V	S	D	U	L	V	G	H	V	R	V	W	H	Q	V

Tabelle 1.1: Beispiel einer Verschlüsselung mit der Caesar-Chiffre.

zeigt die Verschlüsselung c einer deutschen Botschaft m mit dem Schlüssel $k = 3$. Die Verschiebungschiffre mit diesem speziellen Schlüssel $k = 3$ ist auch als die Caesar-Chiffre *bekannt, weil sie der römische Imperator in seinen Kriegen zur Geheimhaltung militärischer Botschaften benutzt haben soll.*[1] *Sie ist eine sehr einfache Substitutionschiffre, in der jeder Buchstabe im Klartext durch einen bestimmten Buchstaben des Geheimalphabets ersetzt wird.*

Da der Schlüsselraum sehr klein ist, kann die Verschiebungschiffre sehr leicht gebrochen werden. Sie ist bereits anfällig für Angriffe, bei denen ein Angreifer lediglich den Schlüsseltext kennt ("*ciphertext-only attacks*"). Ein einfaches Durchprobieren der 26 möglichen Schlüssel offenbart, welcher Schlüssel einen sinnvollen Klartext ergibt, sofern der Schlüsseltext lang genug ist, um eine eindeutige Entschlüsselung zu erlauben.

Die Caesar-Chiffre ist ein monoalphabetisches Kryptosystem, weil jeder Buchstabe im Klartext stets durch denselben Buchstaben im Schlüsseltext ersetzt wird. Bei einem polyalphabetischen Kryptosystem ist es dagegen möglich, dass dieselben Klartextsymbole jeweils durch verschiedene Schlüsseltextsymbole verschlüsselt werden, abhängig davon, an welcher Stelle im Text sie stehen. Ein solches polyalphabetisches Kryptosystem, das auf der Verschiebungschiffre aufbaut, jedoch viel schwerer zu brechen ist, wurde von dem französischen Diplomaten Blaise de Vigenère (1523 bis 1596) vorgeschlagen. Sein System baut auf Vorarbeiten des italienischen Mathematikers Leon Battista Alberti (geb. 1404), des deutschen Abtes Johannes Trithemius (geb. 1492) und des italienischen Wissenschaftlers Giovanni Porta (geb. 1535) auf. Es funktioniert wie die Verschiebungschiffre, nur dass der Buchstabe, mit dem ein Symbol des Klartextes verschlüsselt wird, nun mit dessen Position im Text variiert.

Beispiel 1.3 (Vigenère-Chiffre) *Für dieses symmetrische polyalphabetische Kryptosystem verwendet man ein so genanntes* Vigenère-Quadrat*, siehe Tabelle 1.2. Das ist eine Matrix, die aus 26 Zeilen und 26 Spalten besteht. Jede Zeile enthält die 26 Buchstaben des Alphabets, von Zeile zu Zeile um jeweils eine Position nach links verschoben. Das heißt, die einzelnen Zeilen können als eine Verschiebungschiffre mit den Schlüsseln $0, 1, \ldots, 25$ aufgefasst werden. Welche Zeile des Vigenère-Quadrats man für die Verschlüsselung eines Klartextsymbols benutzt, hängt von dessen Position im Text ab.*

Botschaften werden in Blöcke einer festen Länge n aufgeteilt und blockweise verschlüsselt, d.h., $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbf{Z}_{26}^n$. Die Blocklänge n heißt auch die Periode *des Systems. Den i-ten Buchstaben in einem Wort w bezeichnen wir mit w_i.*

Die Verschlüsselungsfunktion E_k und die Entschlüsselungsfunktion D_k, die beide von $\mathbf{Z}_{26}^n$ in $\mathbf{Z}_{26}^n$ abbilden, sind für jeden Schlüssel $k \in \mathbf{Z}_{26}^n$ definiert durch:

$$\begin{aligned} E_k(b) &= (b + k) \bmod 26 \\ D_k(c) &= (c - k) \bmod 26, \end{aligned}$$

wobei die Addition und die Subtraktion mit k modulo 26 wieder zeichenweise ausgeführt werden. Das heißt, der vereinbarte Schlüssel $k \in \mathbf{Z}_{26}^n$ wird Zeichen für Zeichen über die Symbole des Blockes $b \in \mathbf{Z}_{26}^n$

[1] Historische Anmerkung: Gaius Julius Caesar berichtet in seinem Werk "De Bello Gallico", wie er während der Gallischen Kriege (58 bis 50 v. Chr.) eine verschlüsselte Botschaft an Q. Tullius Cicero (Bruder des berühmten Redners) schickte, der mit seiner Legion belagert wurde. Das verwendete System war monoalphabetisch und ersetzte lateinische Buchstaben durch griechische, jedoch geht aus Caesars Schriften nicht hervor, ob es sich wirklich um die Verschiebungschiffre mit Schlüssel $k = 3$ handelte. Diese Information wurde später von Suetonius gegeben.

0	A	B	C	D	E	F	G	**H**	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	**T**	U	V	W	X	Y	Z	**A**	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Tabelle 1.2: Vigenère-Quadrat: Klartext “H” wird mit Schlüssel “T” als “A” verschlüsselt.

geschrieben. Hat der letzte Block des Klartextes weniger als n Symbole, so werden auch entsprechend weniger Symbole des Schlüssels verwendet. Zur Verschlüsselung des i-ten Klartextsymbols b_i, über dem das Schlüsselsymbol k_i steht, verwendet man dann die i-te Zeile des Vigenère-Quadrats wie in der Verschiebungschiffre.

k	T	O	N	Y	T	O	N	Y	T		O	N		Y	T	O	N	Y	T	O	N		Y	T	O	N	Y
m	H	U	N	G	A	R	I	A	N		I	S		A	L	L		G	R	E	E	K		T	O	G	E	R	M	A	N	S
c	A	I	A	E	T	F	V	Y	G		W	F		Y	E	Z		T	P	X	S	X		R	H	U	R	P	F	O	A	Q

Tabelle 1.3: Beispiel einer Verschlüsselung mit der Vigenère-Chiffre.

Wähle zum Beispiel die Blocklänge $n = 4$ und den Schlüssel $k =$ TONY. *Tabelle 1.3 zeigt die Verschlüsselung eines englischen Klartextes m, der aus sieben Blöcken besteht, in einen Schlüsseltext c mit der Vigenère-Chiffre unter Verwendung des Schlüssels k. Dem ersten Buchstaben des Klartextes,* “H”*, ist das Schlüsselsymbol* “T” *zugeordnet. Der Schnittpunkt der* “H”*-Spalte mit der* “T”*-Zeile des Vigenère-Quadrats ergibt* “A” *als das erste Symbol des Schlüsseltextes, siehe Tabelle 1.2.*

Es gibt eine Vielzahl weiterer klassischer Kryptosysteme, die aber an dieser Stelle nicht näher betrachtet werden sollen. Auch gibt es mehrere Möglichkeiten, Kryptosysteme nach der Art ihres Entwurfs und nach ihren Eigenschaften zu klassifizieren. Definition 1.1 erklärt die Unterscheidung in *symmetrische* und *asymmetrische* Kryptosysteme. Die beiden vorgestellten symmetrischen Systeme (die Verschiebungs- und die Vigenère-Chiffre) verdeutlichen den Unterschied zwischen *monoalphabetischen*

und *polyalphabetischen* Systemen. Beide sind *Substitutionschiffren*. Diese kann man den *Permutationschiffren* (auch *Transpositionschiffren* genannt) gegenüberstellen, bei denen die Buchstaben im Klartext nicht durch bestimmte Buchstaben aus dem Geheimalphabet ersetzt werden, sondern lediglich ihre Position im Text ändern, sonst aber unverändert bleiben. Verschlüsselt man blockweise mit Periode n und verwendet die Menge aller Permutationen von Σ^n als Schlüsselraum, wobei Σ ein Alphabet mit m Buchstaben ist, so gibt es $(m^n)!$ Möglichkeiten, einen Schlüssel zu wählen.

Weiterhin kann man *Blockchiffren*, die wie das Vigenère-System den Klartext in Blöcke aufteilen und blockweise verschlüsseln, mit *Stromchiffren* vergleichen, die abhängig vom Kontext im Klartext einen kontinuierlichen Schlüsselstrom erzeugen. Auch können verschiedene Typen von Blockchiffren betrachtet werden. Ein wichtiger Typ sind die *affin linearen Blockchiffren*. Diese sind dadurch definiert, dass ihre Verschlüsselungsfunktionen $E_{(A,\vec{b})}$ und ihre Entschlüsselungsfunktionen $D_{(A^{-1},\vec{b})}$, die beide von $\mathbf{Z}_m^n$ in $\mathbf{Z}_m^n$ abbilden, affin linear sind, d.h., sie sind von der folgenden Form:

$$\begin{aligned} E_{(A,\vec{b})}(\vec{x}) &= A\vec{x} + \vec{b} \bmod m, \\ D_{(A^{-1},\vec{b})}(\vec{y}) &= A^{-1}(\vec{y} - \vec{b}) \bmod m. \end{aligned} \tag{1.2}$$

Dabei sind $(A, \vec{b})$ und $(A^{-1}, \vec{b})$ die Schlüssel zum Verschlüsseln bzw. zum Entschlüsseln; A ist eine $(n \times n)$-Matrix mit Einträgen aus $\mathbf{Z}_m$; A^{-1} ist die zu A inverse Matrix; $\vec{x}$, $\vec{y}$ und $\vec{b}$ sind Vektoren in $\mathbf{Z}_m^n$ und alle Arithmetik wird modulo m ausgeführt. Hierzu einige mathematische Erläuterungen (siehe auch Definition 1.7 in Abschnitt 1.2.4): Eine $(n \times n)$-Matrix A über dem Ring $\mathbf{Z}_m$ hat genau dann ein multiplikatives Inverses, wenn $\text{ggT}(\det A, m) = 1$. Die zu A *inverse Matrix* ist definiert als $A^{-1} = (\det A)^{-1} A_{\texttt{adj}}$, wobei $\det A$ die Determinante von A und $A_{\texttt{adj}} = ((-1)^{i+j} \det A_{j,i})$ die zu A *adjungierte Matrix* ist. Die *Determinante* $\det A$ von A ist rekursiv definiert: Für $n = 1$ und $A = (a)$ ist $\det A = a$; für $n > 1$ und jedes $i \in \{1, 2, \ldots, n\}$ ist $\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \det A_{i,j}$, wobei $a_{i,j}$ der (i,j)-Eintrag von A ist und die $(n-1) \times (n-1)$-Matrix $A_{i,j}$ aus A durch Streichen der i-ten Zeile und der j-ten Spalte entsteht. Die Determinante einer Matrix kann effizient berechnet werden, siehe Aufgabe 1.2.3.

Beispielsweise ist die Vigenère-Chiffre eine affin lineare Chiffre, deren Schlüsselraum $\mathcal{K} = \mathbf{Z}_m^n$ genau m^n Elemente hat, siehe Beispiel 1.3. Ist $\vec{b}$ in (1.2) der Nullvektor, so handelt es sich um eine *lineare Blockchiffre*. Ein klassisches Beispiel dafür ist die *Hill-Chiffre*, die 1929 von Lester Hill erfunden wurde. Der Schlüsselraum ist hier die Menge aller $(n \times n)$-Matrizen A mit Einträgen aus $\mathbf{Z}_m$, für die $\text{ggT}(\det A, m) = 1$ gilt. Damit wird die Invertierbarkeit der als Schlüssel erlaubten Matrizen garantiert, denn die inverse Matrix A^{-1} dient zum Entschlüsseln der mit A verschlüsselten Botschaft. Die Hill-Chiffre ist definiert durch die Verschlüsselungsfunktion $E_A(\vec{x}) = A\vec{x} \bmod m$ und die Entschlüsselungsfunktion $D_{A^{-1}}(\vec{y}) = A\vec{y} \bmod m$. Damit ist sie die allgemeinste lineare Chiffre. Permutationschiffren sind ebenfalls lineare Chiffren und somit ein Spezialfall der Hill-Chiffre.

1.2.2 Kryptoanalyse

Die Aufgabe der Kryptoanalyse besteht darin, vorhandene Kryptosysteme zu brechen und insbesondere die für die Entschlüsselung nötigen Schlüssel zu ermitteln. Gemäß der für die Kryptoanalyse vorliegenden Information kann man verschiedene Typen von Angriffen unterscheiden und damit die Sicherheit bzw. die Verletzbarkeit des betrachteten Kryptosystems charakterisieren. Im Zusammenhang mit der Verschiebungschiffre wurden bereits die "*ciphertext-only*"-Angriffe erwähnt. Das ist die schwächste Form eines Angriffs, und ein Kryptosystem, das diesem Angriff nicht widersteht, ist nicht viel wert.

Affin lineare Blockchiffren wie die Vigenère- und die Hill-Chiffre sind anfällig für Angriffe, bei denen der Angreifer den zu einem abgefangenen Schlüsseltext zugehörigen Klartext kennt ("*known-plaintext attacks*") und daraus Schlüsse auf die verwendeten Schlüssel ziehen kann. Noch anfälliger sind

sie für Angriffe, bei denen der Angreifer sogar selbst einen Klartext wählen kann ("*chosen-plaintext attacks*") und dann sieht, in welchen Schlüsseltext dieser verschlüsselt wird. Die vierte mögliche Art eines Angriffs ist vor allem für asymmetrische Kryptosysteme relevant. In einem solchen *encryption-key*-Angriff kennt der Angreifer lediglich den öffentlich bekannten Schlüssel, aber noch keinerlei verschlüsselte Botschaften, und versucht, allein aus dieser Information den privaten Schlüssel zu bestimmen. Der Unterschied besteht darin, dass der Angreifer nun jede Menge Zeit für seine Berechnungen hat, während er sich bei den anderen Angriffen beeilen muss, da die Botschaft bereits gesendet wurde. Deshalb müssen bei asymmetrischen Kryptosysteme Schlüssel sehr großer Länge gewählt werden, damit die Sicherheit des Systems gewährleistet bleibt. Daraus resultiert in vielen praktischen Fällen eine geringere Effizienz der asymmetrischen Systeme.

Oft ist bei solchen Angriffen die Häufigkeitsanalyse der in den verschlüsselten Texten vorkommenden Buchstaben nützlich. Dabei wird die Redundanz der für den Klartext verwendeten natürlichen Sprache ausgenutzt. Zum Beispiel kommt in vielen natürlichen Sprachen der Buchstabe "E" statistisch signifikant am häufigsten vor. Gemittelt über lange, "typische" Texte erscheint das "E" mit der Häufigkeit von 12.31% im Englischen, von 15.87% im Französischen und sogar von 18.46% im Deutschen, siehe [53]. In anderen Sprachen können andere Buchstaben mit der größten Häufigkeit auftreten. Zum Beispiel ist mit 12.06% das "A" der häufigste Buchstabe in "typischen" finnischen Texten [53].

Die Nützlichkeit der Häufigkeitsanalyse bei Angriffen auf monoalphabetische Kryptosysteme ist offensichtlich. Tritt zum Beispiel bei einem mit der Verschiebungschiffre verschlüsselten längeren deutschen Text der im Deutschen (wie auch in den meisten anderen Sprachen) seltene Buchstabe "Y" am häufigsten im Schlüsseltext auf, so kann man davon ausgehen, dass er das "E" verschlüsselt, der verwendete Schlüssel also das "U" ($k = 20$) ist, siehe Tabelle 1.2. Neben der Häufigkeit einzelner Buchstaben kann man auch die Häufigkeit des Auftretens von Buchstabenpaaren (Digrammen), von Buchstabentripeln (Trigrammen) und so weiter analysieren. Diese Art von Angriff funktioniert auch bei polyalphabetischen Kryptosystemen, sofern die Periode (also die Blocklänge) bekannt ist.

Polyalphabetische Kryptosysteme mit unbekannter Periode bieten dagegen mehr Sicherheit. Die Vigenère-Chiffre zum Beispiel widerstand lange Zeit jedem Versuch, sie zu brechen. Erst 1863, etwa 300 Jahre nach ihrer Erfindung, fand der deutsche Kryptoanalytiker Friedrich Wilhelm Kasiski eine Methode zum Brechen der Vigenère-Chiffre. Er zeigte, wie man die verwendete Periode, auch wenn sie anfangs unbekannt ist, aus Wiederholungen derselben Teilwörter im Schlüsseltext bestimmen kann. Anschließend kann man dann den Schlüsseltext mit der Häufigkeitsanalyse entschlüsseln. Singh [63] schreibt, dass der britische Exzentriker Charles Babbage, von vielen als ein Genie seiner Zeit betrachtet, Kasiskis Methode vermutlich schon früher entdeckte, nämlich um 1854, seine Arbeit allerdings nicht veröffentlichte.

Als ein Meilenstein in der Geschichte der Kryptographie sei noch die bahnbrechende Arbeit [62] von Claude Shannon (1916 bis 2001) erwähnt, dem Vater der modernen Codierungs- und Informationstheorie. Shannon bewies, dass es Kryptosysteme gibt, die in einem streng mathematischen Sinn *perfekte Geheimhaltung* ermöglichen. Genauer gesagt, leistet ein Kryptosystem genau dann perfekte Geheimhaltung, wenn $|\mathcal{C}| = |\mathcal{K}|$ gilt, die Schlüssel in $\mathcal{K}$ gleichverteilt sind und es für jedes $p \in \mathcal{P}$ und für jedes $c \in \mathcal{C}$ *genau einen* Schlüssel $k \in \mathcal{K}$ mit $E_k(p) = c$ gibt. Das bedeutet, dass ein solches Kryptosystem für die meisten praktischen Zwecke nicht brauchbar ist, denn um perfekte Geheimhaltung zu garantieren, müsste jeder Schlüssel erstens mindestens so lang wie die zu verschlüsselnde Botschaft sein und zweitens nach einmaligem Gebrauch weggeworfen werden. Für praktische Zwecke geeignetere Kryptosysteme werden später in diesem Kapitel vorgestellt.

1.2.3 Algorithmik

Was ist ein Algorithmus? Diese in gewissem Sinne philosophische Frage soll vorerst lediglich pragmatisch und informal betrachtet werden, denn jeder hat wohl eine intuitive Vorstellung vom Begriff des Algorithmus. In Abschnitt 2.2 wird die *Turingmaschine* vorgestellt, ein Modell, das den Algorithmenbegriff und den Begriff der Berechenbarkeit von Funktionen bzw. der Lösbarkeit von Problemen präzise formalisiert. Die Bezeichnung "Algorithmus" hat sich durch Sprachtransformation aus dem Namen des persisch-arabischen Wissenschaftlers[2] Muhammed Ibn Musa Abu Djáfar al Choresmi (773 bis 850) entwickelt, der als Hofmathematiker des Kalifenreiches in Bagdad im Jahre 820 das höchst einflussreiche Buch "Über die indischen Zahlen" verfasste, welches das Dezimalsystem (mit der Zahl 0) erklärt. Die deutsche Übersetzung "Rechnung auf der Linie" von Adam Ries (1492 bis 1559) erschien 1518.

Intuitiv versteht man unter einem Algorithmus ein Rechenverfahren, also eine endliche Folge von Anweisungen, deren Ausführung entweder eine "sinnvolle" Eingabe in endlich vielen Schritten in die gewünschte, einer Lösung des betrachteten Problems entsprechende Ausgabe transformiert oder aber eine "unsinnige" Eingabe verwirft. Es ist auch möglich, dass Algorithmen nie zu einem Ergebnis kommen und bei bestimmten Eingaben in eine Endlosschleife geraten. Interessant ist in diesem Zusammenhang das so genannte *Halteproblem*. Ein fundamentales Resultat der Berechenbarkeitstheorie sagt, dass das Halteproblem algorithmisch nicht entscheidbar ist, d.h., es gibt beweisbar keinen Algorithmus, der entscheiden könnte, ob ein gegebener Algorithmus bei einer gegebenen Eingabe jemals anhält. Dieses Resultat ist das erste in einer langen Liste von so genannten Unentscheidbarkeitsresultaten, die die Grenzen des algorithmisch Machbaren aufzeigen.

```
EUKLID(n, m) {
  if (m = 0) return n;
    else return EUKLID(m, n mod m);
}
```

Abbildung 1.2: Euklidischer Algorithmus.

Sehen wir uns ein Beispiel an. Einer der einfachsten und grundlegendsten Algorithmen ist seit der Antike bekannt und geht auf Euklid von Alexandria (ca. 325 bis 265 v. Chr.) zurück. Trotz seines Alters ist der Euklidische Algorithmus auch heute noch sehr nützlich und findet vielfach Verwendung, zum Beispiel im Abschnitt 1.4, in dem das populäre *public-key* Kryptosystem RSA vorgestellt wird.

Seien $\mathbf{N} = \{0, 1, 2, \ldots\}$ die Menge der natürlichen und $\mathbf{Z} = \{0, \pm 1, \pm 2, \ldots\}$ die Menge der ganzen Zahlen. Der Euklidische Algorithmus bestimmt den größten gemeinsamen Teiler zweier gegebener ganzer Zahlen m und n mit $m \leq n$, also die größte natürliche Zahl k, für die es Zahlen $a, b \in \mathbf{Z}$ mit $m = a \cdot k$ und $n = b \cdot k$ gibt. Dieses k wird als $\mathrm{ggT}(n, m)$ bezeichnet und ist die Ausgabe des Algorithmus, der in Abbildung 1.2 in Pseudocode dargestellt ist. Als Eingabe erhält er die Zahlen n und m, die er sukzessive verändert, indem er sich selbst mit den neuen Werten m (statt n) und $n \bmod m$ (statt m) rekursiv aufruft.[3] Dies wird so lange getan, bis die Abbruchbedingung ($m = 0$) erreicht ist. Zu diesem Zeitpunkt ist der aktuelle Wert von n gerade der größte gemeinsame Teiler $\mathrm{ggT}(n, m)$ der ursprünglich gegebenen Werte n und m, siehe Gleichung (1.3) unten. Diese rekursive Darstellung des Euklidischen Algorithmus ist zweifellos elegant; man kann ihn jedoch natürlich auch *iterativ* implementieren. Das

[2] Andere Schreibweisen seines Namens sind überliefert, z.B. Abu Ja'far Mohammed Ibn Musa Al-Khowarizmi [58]. Zum Wort "Algorithmus" hat sich dieser Name über die lateinische Redensart "*dixit algorizmi*" entwickelt, die man etwa mit "Also sprach al Choresmi" übersetzen kann und die eine Art Qualitätssiegel für die Korrektheit einer Rechnung bedeutete.

[3] Die Bezeichnung "$n \bmod m$" und die Arithmetik in Restklassenringen werden in Problem 1.1 am Ende des Kapitels erklärt.

bedeutet, es gibt keine rekursiven Aufrufe, sondern die berechneten Zwischenwerte werden explizit gespeichert, siehe auch Aufgabe 1.2.3. Hier ist ein Testlauf des Euklidischen Algorithmus für $n = 170$ und $m = 102$:

n	m	$n \bmod m$
170	102	68
102	68	34
68	34	0
34	0	

In diesem Fall liefert der Algorithmus die korrekte Lösung, denn $\text{ggT}(170, 102) = 34$. Doch schon Edsger Dijkstra (1930 bis 2002) wusste, dass solche Tests für einzelne Eingaben höchstens die Anwesenheit von Fehlern zeigen können, nicht aber ihre Abwesenheit. Man überzeuge sich in Aufgabe 1.2.1 davon, dass der Algorithmus tatsächlich *korrekt* ist, d.h., dass er den größten gemeinsamen Teiler von beliebigen Zahlen m und n berechnet. Dieser Korrektheitsbeweis beruht auf der folgenden Gleichung:

$$\text{ggT}(n, m) = \text{ggT}(m, n \bmod m). \tag{1.3}$$

Der Euklidische Algorithmus folgt, wie gesagt, einer rekursiven Strategie nach Art von Teile-und-Herrsche (*divide and conquer* bzw. *divide et impera*), da die zu behandelnden Zahlen mit jedem rekursiven Aufruf strikt kleiner werden. Das allgemeine Schema von Teile-und-Herrsche-Algorithmen ist:

1. **Teile** das Problem in paarweise disjunkte kleinere Teilprobleme derselben Art.

2. **Herrsche** durch rekursives Lösen dieser kleineren Teilprobleme.

3. **Verschmelze** die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

Im Gegensatz zu den meisten anderen Anwendungen von Teile-und-Herrsche entfällt beim Euklidischen Algorithmus allerdings der Verschmelzungsschritt. Ein Verschmelzen der rekursiv gefundenen Lösungen von kleineren Teilproblemen zur Lösung des Ausgangsproblems ist hier offenbar nicht nötig.

```
ERWEITERTER-EUKLID(n, m) {
  if (m = 0) return (n, 1, 0);
    else {
      (g, x', y') := ERWEITERTER-EUKLID(m, n mod m);
      x := y';
      y := x' − y' ∗ ⌊n/m⌋;
      return (g, x, y);
    }
}
```

Abbildung 1.3: Erweiterter Algorithmus von Euklid.

Bei der in Abbildung 1.3 dargestellten erweiterten Version dieses Algorithmus entfällt der Verschmelzungsschritt dagegen nicht. Dieser erweiterte Euklidische Algorithmus berechnet die Vielfachsummendarstellung zweier Zahlen m und n, d.h., er findet Zahlen x und y, so dass $\text{ggT}(n, m) = x \cdot n + y \cdot m$. Dies ist in vielen Anwendungen sehr nützlich, zum Beispiel beim RSA-Verschlüsselungsverfahren in Abschnitt 1.4. Der Verschmelzungsschritt besteht nun in der Berechnung der Werte

x und y aus den rekursiv berechneten Werten x' und y'. Die folgende Tabelle zeigt einen Testlauf des erweiterten Euklidischen Algorithmus für $n = 170$ und $m = 102$:

n	m	g	x	y
170	102	$\boxed{\mathbf{34}}$	$\boxed{\mathbf{-1}}$	$\boxed{\mathbf{2}}$
102	68	34	1	-1
68	34	34	0	1
34	0	34	1	0

Die beiden linken Spalten der Tabelle werden wie beim Euklidischen Algorithmus von oben nach unten gefüllt, wobei der Algorithmus ERWEITERTER-EUKLID jeweils mit neuen Werten für n und m rekursiv aufgerufen wird. In der letzten Zeile ist mit $n = 34$ und $m = 0$ die Abbruchbedingung erreicht. Es wird kein weiterer rekursiver Aufruf initiiert, sondern $(g, x, y) := (34, 1, 0)$ gesetzt, und während ERWEITERTER-EUKLID aus seinen rekursiven Aufrufen zurückkehrt, werden die rechten drei Spalten von unten nach oben gefüllt. Das Ergebnis von ERWEITERTER-EUKLID$(170, 102)$ ist also das Tripel $(34, -1, 2)$ in der ersten Zeile. Für dieses Testbeispiel ist dieses Ergebnis korrekt, denn es gilt:

$$(-1) \cdot 170 + 2 \cdot 102 = 34 = \text{ggT}(170, 102).$$

Die Notation "$\lfloor n/m \rfloor$" in Abbildung 1.3 bezeichnet die größte ganze Zahl $\leq n/m$. Der Nachweis der Korrektheit des erweiterten Algorithmus von Euklid wird dem Leser als Aufgabe 1.2.1 überlassen. Eine weiteres wichtiges Charakteristikum eines Algorithmus neben der Korrektheit ist seine Laufzeit. Ist er effizient? Oder braucht er für bestimmte "harte" Eingaben oder im Mittel sehr lange, um zum Ergebnis zu kommen? Die Laufzeiten des Euklidischen Algorithmus und seiner erweiterten Version sind im Wesentlichen gleich; daher untersuchen wir nur die des ersteren. Offenbar genügt es, die Anzahl der rekursiven Aufrufe des Euklidischen Algorithmus abzuschätzen, um seine Laufzeit zu analysieren. Dazu brauchen wir noch einige Vorbereitungen.

Definition 1.4 *Die Folge $\mathfrak{F} = \{F_n\}_{n \geq 0}$ der* Fibonacci-Zahlen *ist induktiv definiert durch:*

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \quad \textit{für } n \geq 2. \end{aligned}$$

Die Fibonacci-Zahlen sind, mathematisch gesprochen, durch eine homogene lineare Rekurrenzgleichung zweiter Ordnung definiert. Das heißt, die Glieder der Folge $\mathfrak{F}$ sind von der Form:

$$\begin{aligned} T(0) &= r, \\ T(1) &= s, \\ T(n) &= p \cdot T(n-1) + q \cdot T(n-2) \quad \text{für } n \geq 2, \end{aligned} \tag{1.4}$$

wobei p, q, r und s reelle Konstanten mit $p \neq 0$ und $q \neq 0$ sind.

Satz 1.5 *Die Lösung der Rekurrenzgleichung (1.4) hat die Form:*

$$T(n) = \begin{cases} A \cdot \alpha^n - B \cdot \beta^n & \textit{falls } \alpha \neq \beta \\ (A \cdot n + B)\alpha^n & \textit{falls } \alpha = \beta, \end{cases}$$

wobei α und β die zwei reellen Lösungen der quadratischen Gleichung $a^2 - p \cdot a - q = 0$ und die Zahlen A und B wie folgt definiert sind:

$$A = \begin{cases} \frac{s - r\cdot\beta}{\alpha - \beta} & \textit{falls } \alpha \neq \beta \\ \frac{s - r\cdot\alpha}{\alpha} & \textit{falls } \alpha = \beta \end{cases} \qquad \textit{und} \qquad B = \begin{cases} \frac{s - r\cdot\alpha}{\alpha - \beta} & \textit{falls } \alpha \neq \beta \\ r & \textit{falls } \alpha = \beta. \end{cases}$$

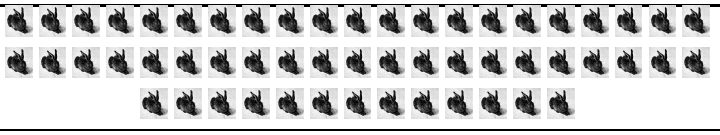
0. Monat		$F_0 = 0$
1. Monat		$F_1 = 1$
2. Monat		$F_2 = 1$
3. Monat		$F_3 = 2$
4. Monat		$F_4 = 3$
5. Monat		$F_5 = 5$
6. Monat		$F_6 = 8$
7. Monat		$F_7 = 13$
8. Monat		$F_8 = 21$
9. Monat		$F_9 = 34$
10. Monat		$F_{10} = 55$

Tabelle 1.4: Die Fibonacci-Zahlen vermehren sich wie die Karnickel und umgekehrt.

Der Beweis von Satz 1.5 wird dem Leser als Aufgabe 1.2.2 überlassen. Im Falle der Fibonacci-Zahlen gilt für die Konstanten $p = q = s = 1$ und $r = 0$. Die ersten 20 Werte dieser Folge sind:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181

Die Vermutung liegt nahe, dass die Folge $\mathfrak{F} = \{F_n\}_{n\geq 0}$ exponentiell schnell wächst. Bevor wir dies beweisen, sehen wir uns ein illustrierendes Beispiel an. Bei "exponentiellem Wachstum" könnte einem die Vermehrung von Kaninchen in den Sinn kommen. Und tatsächlich war dies die ursprüngliche Motivation für die Untersuchungen von Leonardo Pisano (1170 bis 1250), dessen Spitzname Fibonacci war. Die heute seinen Namen tragende Zahlenfolge liefert eine mathematische Beschreibung für die Vermehrung von Kaninchen unter bestimmten vereinfachenden Annahmen. In Fibonaccis Modell bringt ein jedes Kaninchen jeden Monat – mit Ausnahme der ersten beiden seiner Lebensmonate – ein weiteres Kaninchen zur Welt. Außerdem wird in Vereinfachung der Wirklichkeit angenommen, dass alle Kaninchen vom gleichen Geschlecht und unsterblich sind und keine natürlichen Feinde haben. Beginnt man die Zucht mit nur einem einzigen Kaninchen, so ist man nach n Monaten bereits im Besitz einer Population von genau F_n Kaninchen. Tabelle 1.4 veranschaulicht dies bildlich für die Anfangsglieder der Folge.

Nun wollen wir unsere Vermutung beweisen, dass die Folge $\mathfrak{F} = \{F_n\}_{n\geq 0}$ exponentiell in n wächst. Zu zeigen ist also, dass für geeignete Konstanten a und c und alle hinreichend großen n gilt:

$$F_n \geq c \cdot a^n. \tag{1.5}$$

Einsetzen der Induktionsvoraussetzung in die Rekurrenzgleichung für F_n ergibt:

$$F_n \quad = \quad F_{n-1} + F_{n-2} \geq c \cdot a^{n-1} + c \cdot a^{n-2} = c \cdot a^n \cdot \frac{a+1}{a^2} \geq c \cdot a^n.$$

Können wir die letzte Ungleichung zeigen, so ist der Induktionsschritt korrekt vollzogen. Diese letzte Ungleichung ist aber äquivalent dazu, dass $a^2 - a - 1 \leq 0$ gilt. Eine quadratische Gleichung der Form $a^2 + p \cdot a + q = 0$ hat die beiden reellwertigen Lösungen $-\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$. In unserem Fall ($a^2 - a - 1 = 0$)

ist $p = -1 = q$, und es ergeben sich die Lösungen:

$$\begin{aligned} \alpha &= \frac{1}{2} + \sqrt{\frac{1}{4} + 1} = \frac{1 + \sqrt{1+4}}{2} = \frac{1+\sqrt{5}}{2}, \\ \beta &= \frac{1}{2} - \sqrt{\frac{1}{4} + 1} = \frac{1 - \sqrt{1+4}}{2} = \frac{1-\sqrt{5}}{2}. \end{aligned}$$

Der Induktionsschritt ist somit für alle $a \leq \alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$ gezeigt. Die Zahl $\alpha = \frac{1+\sqrt{5}}{2}$ tritt in vielen Zusammenhängen in der Mathematik und Informatik auf. In der Geometrie zum Beispiel ergibt sie sich bei der Zerlegung eines Rechtecks in ein Quadrat und ein kleineres Rechteck, so dass die beiden Rechtecke dieselben Seitenverhältnisse haben. Diese Zerlegung, die man als den „goldenen Schnitt“ bezeichnet, wird in Aufgabe 1.2.3 beschrieben.

Nach Satz 1.5 können wir aus α und β die Zahlen $A = \frac{1}{\sqrt{5}}$ und $B = \frac{1}{\sqrt{5}}$ berechnen, und es folgt

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n \tag{1.6}$$

für die n-te Fibonacci-Zahl. Der zweite Term in (1.6) kann dabei vernachlässigt werden, weil $\beta = \frac{1-\sqrt{5}}{2}$ betragsmäßig kleiner als 1 ist und $\frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$ deshalb für wachsendes n gegen 0 strebt.

Nun zeigen wir, dass die Folge $\mathfrak{F} = \{F_n\}_{n \geq 0}$ der Fibonacci-Zahlen in einem engen Zusammenhang zur maximalen Anzahl der rekursiven Aufrufe des Euklidischen Algorithmus steht. Der folgende Satz bestimmt Zahlen, für die die Laufzeit des Euklidischen Algorithmus am schlechtesten ist. Der Beweis von Satz 1.6 wird dem Leser als Aufgabe 1.2.4 überlassen.

Satz 1.6 *Für alle $k \geq 1$ gilt:*

1. *Ein Aufruf von* EUKLID(F_{k+3}, F_{k+2}) *benötigt genau k rekursive Aufrufe.*
2. *Benötigt* EUKLID(n, m) *mindestens k rekursive Aufrufe, so ist $|n| \geq F_{k+3}$ und $|m| \geq F_{k+2}$.*

Nach Satz 1.6 ist die Anzahl der rekursiven Aufrufe von EUKLID(n, m) durch $\max\{k \mid F_{k+3} \leq n\}$ beschränkt. Wir wissen aus Ungleichung (1.5), dass $F_{k+3} \geq c \cdot \alpha^{k+3}$ für $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$ und eine Konstante c gilt. Logarithmieren wir die Ungleichung $c \cdot \alpha^{k+3} \leq F_{k+3} \leq n$ zur Basis α, so ergibt sich:

$$\log_\alpha c + \log_\alpha \alpha^{k+3} = \log_\alpha c + k + 3 \leq \log_\alpha n.$$

Gemäß Aufgabe 1.2.5 rechnet man $\log_\alpha n = (\log_\alpha 2)(\log_2 n)$ aus. Definiere für eine gegebene Funktion $g : \mathbf{N} \to \mathbf{N}$ die Funktionenklasse $O(g)$ durch

$$O(g) = \{f : \mathbf{N} \to \mathbf{N} \mid (\exists c > 0)\,(\exists n_0 \in \mathbf{N})\,(\forall n \geq n_0)\,[f(n) \leq c \cdot g(n)]\}.$$

Das heißt, eine Funktion $f \in O(g)$ wächst asymptotisch höchstens so schnell wie g. Die O-Notation abstrahiert somit von additiven Konstanten, von konstanten Faktoren und von endlich vielen Ausnahmen. Daher ist sie für die Laufzeitanalyse von Algorithmen besonders geeignet. Insbesondere ist in der O-Notation die Basis des Logarithmus irrelevant (siehe Aufgabe 1.2.5), weshalb wir vereinbaren, dass der Logarithmus $\log$ stets zur Basis 2 sei. Aus diesen Betrachtungen folgt, dass die Anzahl k der rekursiven Aufrufe von EUKLID(n, m) aus $O(\log n)$ und EUKLID somit ein effizienter Algorithmus ist.

1.2.4 Algebra, Zahlentheorie und Graphentheorie

Für das Verständnis einiger der Algorithmen und Probleme, die wir später behandeln, sind grundlegende Begriffe und Aussagen aus der Algebra und insbesondere aus der Gruppen- und der Zahlentheorie hilfreich. Das betrifft sowohl die Kryptosysteme und Zero-Knowledge-Protokolle in Kapitel 1 als auch einige der in Kapitel 2 betrachteten Probleme. Man kann den gegenwärtigen Abschnitt auch überspringen und die erforderlichen Begriffe und Resultate erst dann nachschlagen, wenn man später darauf stößt. Auf Beweise wird in diesem Abschnitt meist verzichtet.

Definition 1.7 (Gruppe, Ring und Körper)

- *Eine* Gruppe $\mathfrak{G} = (S, \circ)$ *ist definiert durch eine nichtleere Menge S und eine zweistellige Operation $\circ$ auf S, die die folgenden Axiome erfüllen:*
  - Abschluss: $(\forall x \in S)\,(\forall y \in S)\,[x \circ y \in S]$.
  - Assoziativität: $(\forall x \in S)\,(\forall y \in S)\,(\forall z \in S)\,[(x \circ y) \circ z = x \circ (y \circ z)]$.
  - Neutrales Element: $(\exists e \in S)\,(\forall x \in S)\,[e \circ x = x \circ e = x]$.
  - Inverses Element: $(\forall x \in S)\,(\exists x^{-1} \in S)\,[x \circ x^{-1} = x^{-1} \circ x = e]$.

  Das Element e heißt das neutrale Element der Gruppe $\mathfrak{G}$. *Das Element x^{-1} heißt das* zu x inverse Element. *$\mathfrak{G}$ ist eine* Halbgruppe, *falls $\mathfrak{G}$ die Assoziativität und die Abschlusseigenschaft unter $\circ$ erfüllt, auch wenn $\mathfrak{G}$ kein neutrales Element besitzt oder wenn nicht jedes Element in $\mathfrak{G}$ ein Inverses hat. Eine Gruppe bzw. eine Halbgruppe $\mathfrak{G} = (S, \circ)$ heißt* kommutativ *(oder* abelsch*), falls $x \circ y = y \circ x$ für alle $x, y \in S$ gilt. Die Anzahl der Elemente einer endlichen Gruppe $\mathfrak{G}$ heißt die* Ordnung von $\mathfrak{G}$ *und wird mit $|\mathfrak{G}|$ bezeichnet.*

- *$\mathfrak{H} = (T, \circ)$ heißt* Untergruppe einer Gruppe $\mathfrak{G} = (S, \circ)$ *(bezeichnet durch $\mathfrak{H} \leq \mathfrak{G}$), falls $T \subseteq S$ und $\mathfrak{H}$ die Gruppenaxiome erfüllt.*

- *Ein* Ring *ist ein Tripel $\mathfrak{R} = (S, +, \cdot)$, so dass $(S, +)$ eine abelsche Gruppe und $(S, \cdot)$ eine Halbgruppe ist und die Distributivgesetze gelten:*

  $$(\forall x \in S)\,(\forall y \in S)\,(\forall z \in S)\,[(x \cdot (y + z) = (x \cdot y) + (x \cdot z)) \wedge ((x + y) \cdot z = (x \cdot z) + (y \cdot z))].$$

  Ein Ring $\mathfrak{R} = (S, +, \cdot)$ heißt kommutativ, *falls die Halbgruppe $(S, \cdot)$ kommutativ ist. Das neutrale Element der Gruppe $(S, +)$ heißt* Nullelement *(kurz* Null*) des Ringes $\mathfrak{R}$. Ein neutrales Element der Halbgruppe $(S, \cdot)$ heißt* Einselement *(kurz* Eins*) des Ringes $\mathfrak{R}$.*

- *Sei $\mathfrak{R} = (S, +, \cdot)$ ein Ring mit Eins. Ein Element x von $\mathfrak{R}$ heißt genau dann* invertierbar *(oder* Einheit von $\mathfrak{R}$*), wenn es in der Halbgruppe $(S, \cdot)$ invertierbar ist. Ein Element x von $\mathfrak{R}$ heißt* Nullteiler, *falls es von Null verschieden ist und es ein von Null verschiedenes Element y von $\mathfrak{R}$ mit $x \cdot y = y \cdot x = 0$ gibt.*

- *Ein* Körper *ist ein kommutativer Ring mit Eins, in dem jedes von Null verschiedene Element invertierbar ist.*

Beispiel 1.8 (Gruppe, Ring und Körper)

- *Sei $k \in \mathbf{N}$. Die Menge $\mathbf{Z}_k = \{0, 1, \ldots, k-1\}$ ist bezüglich der Addition von Zahlen modulo k eine endliche Gruppe mit dem neutralen Element 0. Bezüglich der Addition und der Multiplikation von Zahlen modulo k ist $\mathbf{Z}_k$ ein kommutativer Ring mit Eins, siehe Problem 1.1 am Ende des Kapitels. Ist p eine* Primzahl *(d.h., $p \geq 2$ ist nur durch 1 und durch p teilbar), so ist $\mathbf{Z}_p$ bezüglich der Addition und der Multiplikation von Zahlen modulo k ein Körper.*

- *Der größte gemeinsame Teiler* $\mathrm{ggT}(n, m)$ *zweier Zahlen* m *und* n *wurde bereits in Abschnitt 1.2.3 betrachtet. Definiere für* $k \in \mathbf{N}$ *die Menge* $\mathbf{Z}_k^* = \{i \,|\, 1 \leq i \leq k-1$ *und* $\mathrm{ggT}(i,k) = 1\}$*. Bezüglich der Multiplikation von Zahlen modulo* k *ist* $\mathbf{Z}_k^*$ *eine endliche Gruppe mit dem neutralen Element* 1*.*

Ist die Operation $\circ$ einer Gruppe $\mathfrak{G} = (S, \circ)$ aus dem Kontext heraus klar, so wird sie nicht explizit angegeben. Die Gruppe $\mathbf{Z}_k^*$ aus Beispiel 1.8 spielt insbesondere in Abschnitt 1.4 eine Rolle, in dem das RSA-Kryptosystem vorgestellt wird. Die *Euler-Funktion* φ gibt die Ordnung dieser Gruppe an, d.h., $\varphi(k) = |\mathbf{Z}_k^*|$. Die folgenden Eigenschaften von φ folgen aus der Definition:

- $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ für alle $m, n \in \mathbf{N}$ mit $\mathrm{ggT}(m, n) = 1$ und
- $\varphi(p) = p - 1$ für alle Primzahlen p.

Der Beweis dieser Eigenschaften wird dem Leser als Aufgabe 1.2.8 überlassen. Insbesondere werden wir in Abschnitt 1.4.1 den folgenden Fakt benutzen, der unmittelbar aus diesen Eigenschaften folgt.

Fakt 1.9 *Ist* $n = p \cdot q$ *für Primzahlen* p *und* q*, so gilt* $\varphi(n) = (p-1)(q-1)$*.*

Eulers Satz unten ist ein Spezialfall (für die Gruppe $\mathbf{Z}_n^*$) des Satzes von Lagrange, welcher sagt, dass für ein jedes Gruppenelement a einer endlichen multiplikativen Gruppe $\mathfrak{G}$ der Ordnung $|\mathfrak{G}|$ und mit dem neutralen Element e gilt, dass $a^{|\mathfrak{G}|} = e$. Der Spezialfall von Eulers Satz mit einer Primzahl n, die a nicht teilt, ist als der Kleine Fermat bekannt.

Satz 1.10 (Euler) *Für jedes* $a \in \mathbf{Z}_n^*$ *gilt* $a^{\varphi(n)} \equiv 1 \bmod n$*.*

Korollar 1.11 (Kleiner Fermat) *Ist* p *eine Primzahl und ist* $a \in \mathbf{Z}_p^*$*, dann gilt* $a^{p-1} \equiv 1 \bmod p$*.*

In Abschnitt 2.5 werden Algorithmen für das Graphisomorphieproblem vorgestellt. Dieses Problem, welches auch bei den Zero-Knowledge-Protokollen aus Abschnitt 1.6.2 eine wichtige Rolle spielt, kann als ein Spezialfall bestimmter gruppentheoretischer Probleme aufgefasst werden. Dabei sind insbesondere *Permutationsgruppen* von Interesse. Erläuternde Beispiele dieser abstrakten Begriffe folgen später.

Definition 1.12 (Permutationsgruppe)

- *Eine* Permutation *ist eine bijektive Abbildung einer Menge in sich selbst. Für eine natürliche Zahl* $n \geq 1$ *sei* $[n] = \{1, 2, \ldots, n\}$*. Die Menge aller Permutationen von* $[n]$ *wird mit* $\mathfrak{S}_n$ *bezeichnet. Für algorithmische Zwecke werden Permutationen* $\pi \in \mathfrak{S}_n$ *als Listen von* n *geordneten Paaren* $(i, \pi(i))$ *aus* $[n] \times [n]$ *repräsentiert.*
- *Definiert man als Operation auf* $\mathfrak{S}_n$ *die Komposition von Permutationen, so wird* $\mathfrak{S}_n$ *eine Gruppe. Sind etwa* π *und* τ *zwei Permutationen aus* $\mathfrak{S}_n$*, so ist ihre* Komposition $\pi\tau$ *als diejenige Permutation in* $\mathfrak{S}_n$ *definiert, die sich ergibt, wenn zuerst* π *und dann* τ *auf die Elemente in* $[n]$ *angewandt wird, d.h.,* $(\pi\tau)(i) = \tau(\pi(i))$ *für alle* $i \in [n]$*. Das neutrale Element der Permutationsgruppe* $\mathfrak{S}_n$ *ist die* identische Permutation*, die definiert ist als* $\mathrm{id}(i) = i$ *für alle* $i \in [n]$*. Die Untergruppe von* $\mathfrak{S}_n$*, die* id *als das einzige Element enthält, wird durch* **id** *bezeichnet.*
- *Für eine Teilmenge* $\mathfrak{T}$ *von* $\mathfrak{S}_n$ *ist die* durch $\mathfrak{T}$ erzeugte Permutationsgruppe $\langle\mathfrak{T}\rangle$ *definiert als die kleinste Untergruppe von* $\mathfrak{S}_n$*, die* $\mathfrak{T}$ *enthält. Untergruppen* $\mathfrak{G}$ *von* $\mathfrak{S}_n$ *werden durch ihre erzeugenden Mengen repräsentiert, welche auch* Generatoren von $\mathfrak{G}$ *genannt werden. In* $\mathfrak{G}$ *ist der* Orbit eines Elements $i \in [n]$ *definiert als* $\mathfrak{G}(i) = \{\pi(i) \,|\, \pi \in \mathfrak{G}\}$.

- *Für eine Teilmenge T von $[n]$ bezeichne $\mathfrak{S}_n^T$ die Untergruppe von $\mathfrak{S}_n$, die jedes Element von T auf sich selbst abbildet. Insbesondere ist für $i \leq n$ und eine Untergruppe $\mathfrak{G}$ von $\mathfrak{S}_n$ der* (punktweise) Stabilisator von $[i]$ in $\mathfrak{G}$ *definiert durch:*

$$\mathfrak{G}^{(i)} = \{\pi \in \mathfrak{G} \mid \pi(j) = j \text{ für alle } j \in [i]\}.$$

  Es gilt $\mathfrak{G}^{(n)} = \mathbf{id}$ und $\mathfrak{G}^{(0)} = \mathfrak{G}$.

- *Seien $\mathfrak{G}$ und $\mathfrak{H}$ Permutationsgruppen mit $\mathfrak{H} \leq \mathfrak{G}$. Für $\tau \in \mathfrak{G}$ heißt $\mathfrak{H}\tau = \{\pi\tau \mid \pi \in \mathfrak{H}\}$ eine* rechte co-Menge von $\mathfrak{H}$ in $\mathfrak{G}$. *Je zwei rechte co-Mengen von $\mathfrak{H}$ in $\mathfrak{G}$ sind entweder identisch oder disjunkt. Daher wird die Permutationsgruppe $\mathfrak{G}$ durch die rechten co-Mengen von $\mathfrak{H}$ in $\mathfrak{G}$ zerlegt:*

$$\mathfrak{G} = \mathfrak{H}\tau_1 \cup \mathfrak{H}\tau_2 \cup \cdots \cup \mathfrak{H}\tau_k. \tag{1.7}$$

  Jede rechte co-Menge von $\mathfrak{H}$ in $\mathfrak{G}$ hat die Kardinalität $|\mathfrak{H}|$. Die Menge $\{\tau_1, \tau_2, \ldots, \tau_k\}$ in (1.7) heißt die vollständige rechte Transversale von $\mathfrak{H}$ in $\mathfrak{G}$.

Der Begriff der punktweisen Stabilisatoren ist besonders wichtig für den Entwurf von Algorithmen für Probleme auf Permutationsgruppen. Die wesentliche Struktur, die man dabei ausnutzt, ist der so genannte "*Turm von Stabilisatoren*" einer Permutationsgruppe $\mathfrak{G}$:

$$\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}.$$

Für jedes i mit $1 \leq i \leq n$ sei $\mathfrak{T}_i$ die vollständige rechte Transversale von $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$. Dann nennt man $\mathfrak{T} = \bigcup_{i=1}^{n-1} \mathfrak{T}_i$ einen *starken Generator von* $\mathfrak{G}$, und es gilt $\mathfrak{G} = \langle\mathfrak{T}\rangle$. Jedes $\pi \in \mathfrak{G}$ hat dann eine eindeutige Faktorisierung $\pi = \tau_1\tau_2\cdots\tau_n$ mit $\tau_i \in \mathfrak{T}_i$. Die folgenden grundlegenden algorithmischen Resultate über Permutationsgruppen werden später in Abschnitt 2.5 nützlich sein.

Satz 1.13 *Gegeben sei eine Permutationsgruppe $\mathfrak{G} \leq \mathfrak{S}_n$ durch ihre erzeugende Menge. Dann gilt:*

1. *Für jedes $i \in [n]$ kann der Orbit $\mathfrak{G}(i)$ von i in $\mathfrak{G}$ in Polynomialzeit berechnet werden.*
2. *Der Turm von Stabilisatoren $\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}$ kann in einer Zeit polynomiell in n berechnet werden, d.h., für jedes i mit $1 \leq i \leq n$ werden die vollständigen rechten Transversalen $\mathfrak{T}_i$ von $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$ und somit ein starker Generator von $\mathfrak{G}$ bestimmt.*

Die in Definition 1.12 eingeführten Begriffe zu Permutationsgruppen werden nun anhand konkreter Beispiele aus der Graphentheorie erläutert. Insbesondere betrachten wir die Automorphismengruppe und die Menge der Isomorphismen von Graphen. Zunächst benötigen wir einige Begriffe aus der Graphentheorie.

Definition 1.14 (Graphisomorphie und Graphautomorphie) *Ein Graph G besteht aus einer endlichen Menge von Knoten, $V(G)$, und einer endlichen Menge von Kanten, $E(G)$, die bestimmte Knoten miteinander verbinden. Wir nehmen an, dass keine Mehrfachkanten und keine Schlaufen auftreten. In diesem Kapitel werden ausschließlich ungerichtete Graphen betrachtet, d.h., die Kanten haben keine Orientierung und können als ungeordnete Knotenpaare aufgefasst werden. Die* disjunkte Vereinigung $G \cup H$ zweier Graphen G und H, *deren Knotenmengen $V(G)$ und $V(H)$ durch Umbenennen disjunkt gemacht werden, ist definiert als der Graph mit Knotenmenge $V(G) \cup V(H)$ und Kantenmenge $E(G) \cup E(H)$.*

Seien G und H zwei Graphen mit der gleichen Anzahl von Knoten. Ein Isomorphismus zwischen G und H *ist eine kantenerhaltende Bijektion von der Knotenmenge von G auf die von H. Unter der*

Vereinbarung, dass $V(G) = \{1, 2, \ldots, n\} = V(H)$, sind G und H also genau dann isomorph (kurz $G \cong H$), wenn es eine Permutation $\pi \in \mathfrak{S}_n$ gibt, so dass für alle Knoten $i, j \in V(G)$ gilt:

$$\{i, j\} \in E(G) \iff \{\pi(i), \pi(j)\} \in E(H). \tag{1.8}$$

Ein Automorphismus von G *ist eine kantenerhaltende Bijektion von der Knotenmenge von G auf sich selbst. Jeder Graph enthält den trivialen Automorphismus* id. *Mit* $\mathrm{Iso}(G, H)$ *bezeichnen wir die Menge aller Isomorphismen zwischen G und H, und mit* $\mathrm{Aut}(G)$ *bezeichnen wir die Menge aller Automorphismen von G. Definiere die Probleme* Graphisomorphie *(kurz* `GI`*) und* Graphautomorphie *(kurz* `GA`*) durch:*

$$\begin{aligned} \texttt{GI} &= \{(G, H) \mid G \text{ und } H \text{ sind isomorphe Graphen}\}; \\ \texttt{GA} &= \{G \mid G \text{ enthält einen nichttrivialen Automorphismus}\}. \end{aligned}$$

Für algorithmische Zwecke werden Graphen entweder durch ihre Knoten- und Kantenlisten oder ihre Adjazenzmatrix repräsentiert, die an der Stelle (i, j) den Eintrag 1 hat, falls $\{i, j\}$ eine Kante ist, und den Eintrag 0 sonst. Diese Darstellung eines Graphen wird geeignet über dem Alphabet $\Sigma = \{0, 1\}$ codiert. Um Paare von Graphen darzustellen, verwenden wir eine bijektive Paarungsfunktion $(\cdot, \cdot)$ von $\Sigma^* \times \Sigma^*$ auf Σ^*, die in Polynomialzeit berechenbar ist und in Polynomialzeit berechenbare Inverse hat.

Beispiel 1.15 (Graphisomorphie und Graphautomorphie) *Die Graphen G und H in Abbildung 1.4 sind isomorph. Ein Isomorphismus $\pi : V(G) \to V(H)$, der die Adjazenz der Knoten gemäß (1.8) erhält,*

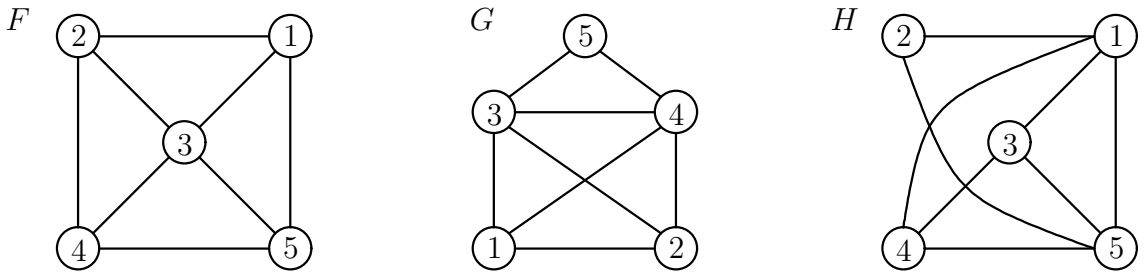


Abbildung 1.4: Drei Graphen: G ist isomorph zu H, aber nicht zu F.

ist zum Beispiel gegeben durch $\pi = \left(\begin{smallmatrix} 1&2&3&4&5 \\ 3&4&1&5&2 \end{smallmatrix}\right)$ bzw. in der Zyklenschreibweise $\pi = (1\,3)(2\,4\,5)$. Es gibt noch drei weitere Isomorphismen zwischen G und H, d.h., $|\mathrm{Iso}(G, H)| = 4$, siehe Aufgabe 1.2.9. Jedoch ist weder G noch H isomorph zu F. Das sieht man sofort daran, dass sich die Folge der Knotengrade *(also die Anzahl der auslaufenden Kanten eines jeden Knoten) von G bzw. H von der Folge der Knotengrade von F unterscheidet: Bei G und H ist diese Folge $(2, 3, 3, 4, 4)$, während sie $(3, 3, 3, 3, 4)$ bei F ist. Ein nichttrivialer Automorphismus $\varphi : V(G) \to V(G)$ von G ist z.B. gegeben durch $\varphi = \left(\begin{smallmatrix} 1&2&3&4&5 \\ 2&1&4&3&5 \end{smallmatrix}\right)$ bzw. $\varphi = (1\,2)(3\,4)(5)$, ein anderer durch $\tau = \left(\begin{smallmatrix} 1&2&3&4&5 \\ 1&2&4&3&5 \end{smallmatrix}\right)$ bzw. $\tau = (1)(2)(3\,4)(5)$. Es gibt noch zwei weitere Automorphismen von G, d.h., $|\mathrm{Aut}(G)| = 4$, siehe Aufgabe 1.2.9.*

Die Permutationsgruppen $\mathrm{Aut}(F)$, $\mathrm{Aut}(G)$ und $\mathrm{Aut}(H)$ sind Untergruppen von $\mathfrak{S}_5$. Der Turm $\mathrm{Aut}(G)^{(5)} \leq \mathrm{Aut}(G)^{(4)} \leq \cdots \leq \mathrm{Aut}(G)^{(1)} \leq \mathrm{Aut}(G)^{(0)}$ der Stabilisatoren von $\mathrm{Aut}(G)$ besteht aus den Untergruppen $\mathrm{Aut}(G)^{(5)} = \mathrm{Aut}(G)^{(4)} = \mathrm{Aut}(G)^{(3)} = \mathbf{id}$, $\mathrm{Aut}(G)^{(2)} = \mathrm{Aut}(G)^{(1)} = \langle\{\mathrm{id}, \tau\}\rangle$ und $\mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$. In der Automorphismengruppe $\mathrm{Aut}(G)$ von G haben die Knoten 1 und 2 den Orbit $\{1, 2\}$, die Knoten 3 und 4 den Orbit $\{3, 4\}$, und der Knoten 5 hat den Orbit $\{5\}$.

$\mathrm{Iso}(G, H)$ und $\mathrm{Aut}(G)$ haben genau dann dieselbe Zahl von Elementen, wenn G und H isomorph sind. Denn sind G und H isomorph, so folgt $|\mathrm{Iso}(G, H)| = |\mathrm{Aut}(G)|$ aus $\mathrm{Aut}(G) = \mathrm{Iso}(G, G)$. Ist andernfalls $G \ncong H$, so ist $\mathrm{Iso}(G, H)$ leer, während $\mathrm{Aut}(G)$ stets den trivialen Automorphismus id enthält.

Daraus folgt die Aussage (1.9) aus Lemma 1.16, das wir später in Abschnitt 2.5 benötigen. Für die Aussage (1.10) nehmen wir an, dass G und H zusammenhängend seien; andernfalls betrachten wir statt G bzw. H die co-Graphen $\overline{G}$ bzw. $\overline{H}$, siehe Aufgabe 1.2.10. Ein Automorphismus von $G \cup H$, der die Knoten von G und H vertauscht, setzt sich zusammen aus einem Isomorphismus in $\mathrm{Iso}(G, H)$ und einem Isomorphismus in $\mathrm{Iso}(H, G)$. Somit ist $|\mathrm{Aut}(G \cup H)| = |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| + |\mathrm{Iso}(G, H)|^2$, woraus die Aussage (1.10) mittels (1.9) folgt.

Lemma 1.16 *Für je zwei Graphen G und H gilt:*

$$|\mathrm{Iso}(G,H)| = \begin{cases} |\mathrm{Aut}(G)| = |\mathrm{Aut}(H)| & \textit{falls } G \cong H \\ 0 & \textit{falls } G \ncong H; \end{cases} \tag{1.9}$$

$$|\mathrm{Aut}(G \cup H)| = \begin{cases} 2 \cdot |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| & \textit{falls } G \cong H \\ |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| & \textit{falls } G \ncong H. \end{cases} \tag{1.10}$$

Sind G und H isomorphe Graphen und ist $\tau \in \mathrm{Iso}(G, H)$, so ist $\mathrm{Iso}(G, H) = \mathrm{Aut}(G)\tau$. Das heißt, $\mathrm{Iso}(G, H)$ ist eine rechte co-Menge von $\mathrm{Aut}(G)$ in $\mathfrak{S}_n$. Da zwei rechte co-Mengen entweder disjunkt oder gleich sind, kann $\mathfrak{S}_n$ gemäß (1.7) in rechte co-Mengen von $\mathrm{Aut}(G)$ zerlegt werden:

$$\mathfrak{S}_n = \mathrm{Aut}(G)\tau_1 \cup \mathrm{Aut}(G)\tau_2 \cup \cdots \cup \mathrm{Aut}(G)\tau_k, \tag{1.11}$$

wobei $|\mathrm{Aut}(G)\tau_i| = |\mathrm{Aut}(G)|$ für alle i, $1 \leq i \leq k$. Die Menge $\{\tau_1, \tau_2, \ldots, \tau_k\}$ von Permutationen in $\mathfrak{S}_n$ ist also eine vollständige rechte Transversale von $\mathrm{Aut}(G)$ in $\mathfrak{S}_n$. Bezeichnen wir mit $\pi(G)$ den Graphen $H \cong G$, der sich ergibt, wenn die Permutation $\pi \in \mathfrak{S}_n$ auf die Knoten von G angewandt wird, so folgt $\{\tau_i(G) \mid 1 \leq i \leq k\} = \{H \mid H \cong G\}$. Da es genau $n! = n(n-1)\cdots 2 \cdot 1$ viele Permutationen in $\mathfrak{S}_n$ gibt, folgt aus (1.11):

$$|\{H \mid H \cong G\}| = k = \frac{|\mathfrak{S}_n|}{|\mathrm{Aut}(G)|} = \frac{n!}{|\mathrm{Aut}(G)|}.$$

Damit haben wir das folgende Korollar gezeigt.

Korollar 1.17 *Zu jedem Graphen G mit n Knoten sind genau $n!/|\mathrm{Aut}(G)|$ Graphen isomorph.*

Zu dem Graphen G aus Abbildung 1.4 in Beispiel 1.15 sind also genau $5!/4 = 30$ Graphen isomorph. Das folgende Lemma wird später in Abschnitt 2.5 benötigt.

Lemma 1.18 *Seien G und H zwei Graphen mit n Knoten. Definiere die Menge*

$$A(G,H) = \{(F,\varphi) \mid F \cong G \textit{ und } \varphi \in \mathrm{Aut}(F)\} \cup \{(F,\varphi) \mid F \cong H \textit{ und } \varphi \in \mathrm{Aut}(F)\}.$$

Dann gilt:

$$|A(G,H)| = \begin{cases} n! & \textit{falls } G \cong H \\ 2n! & \textit{falls } G \ncong H. \end{cases}$$

Beweis. Sind F und G isomorph, so ist $|\mathrm{Aut}(F)| = |\mathrm{Aut}(G)|$. Folglich gilt nach Korollar 1.17:

$$|\{(F,\varphi) \mid F \cong G \text{ und } \varphi \in \mathrm{Aut}(F)\}| = \frac{n!}{|\mathrm{Aut}(F)|} \cdot |\mathrm{Aut}(F)| = n!,$$

und ganz analog zeigt man: $|\{(F,\varphi) \mid F \cong H \text{ und } \varphi \in \mathrm{Aut}(F)\}| = n!$. Sind G und H isomorph, so ist

$$\{(F,\varphi) \mid F \cong G \text{ und } \varphi \in \mathrm{Aut}(F)\} = \{(F,\varphi) \mid F \cong H \text{ und } \varphi \in \mathrm{Aut}(F)\}.$$

Daraus folgt $|A(G,H)| = n!$. Sind G und H nicht isomorph, so sind $\{(F,\varphi) \mid F \cong G \text{ und } \varphi \in \mathrm{Aut}(F)\}$ und $\{(F,\varphi) \mid F \cong H \text{ und } \varphi \in \mathrm{Aut}(F)\}$ disjunkte Mengen. Folglich ist $|A(G,H)| = 2n!$. ∎

Übungsaufgaben

Aufgabe 1.2.1 **(a)** Zeige die Korrektheit des Euklidischen Algorithmus aus Abbildung 1.2. Dazu genügt es, die Gleichung (1.3) zu zeigen.

(b) Zeige die Korrektheit des erweiterten Algorithmus von Euklid aus Abbildung 1.3.

Aufgabe 1.2.2 Beweise Satz 1.5 durch Induktion.

Aufgabe 1.2.3 **Goldener Schnitt:** Gegeben sei ein Rechteck mit den Seitenlängen a und b, wobei $a < b$. Schneidet man aus diesem Rechteck ein Quadrat mit Seitenlänge a heraus, so ergibt sich ein weiteres Rechteck mit den Seitenlängen $b-a$ und a. Die Frage ist, für welche Zahlen a und b sich in dieser Weise zwei Rechtecke ergeben, deren Seitenlängen genau dasselbe Verhältnis zueinander haben, d.h., so dass $a/b = (b-a)/a$ gilt.

(a) Zeige, dass die Zahl $(1+\sqrt{5})/2$ dieses Verhältnis erfüllt.

(b) Welche andere (negative) Zahl erfüllt dieses Verhältnis ebenfalls?

Aufgabe 1.2.4 Beweise Satz 1.6 durch Induktion.

Aufgabe 1.2.5 Zeige, dass in der O-Notation die Basis des Logarithmus irrelevant ist: Sind etwa $a, b > 1$ zwei Basen, dann gilt $\log_b n = (\log_b a)(\log_a n)$, woraus $\log_b n \in O(\log_a n)$ folgt.

Aufgabe 1.2.6 Tabelle 1.5 zeigt zwei abgefangene Schlüsseltexte, c_1 und c_2. Man weiß, dass beide denselben Klartext m verschlüsseln und dass der eine mit der Verschiebungschiffre, der andere mit der Vigenère-Chiffre verschlüsselt wurde. Entschlüssele die beiden Texte.

Hinweis: Nach der Entschlüsselung stellt man fest, dass sich bei der einen Verschlüsselungsmethode eine wahre, bei der anderen eine falsche Aussage ergibt. Ist vielleicht eine Häufigkeitsanalyse sinnvoll?

c_1	W K L V V H Q W H Q F H L V H Q F U B S W H G E B F D H V D U V N H B
c_2	N U O S J Y A Z E E W R O S V H P X Y G N R J B P W N K S R L F Q E P

Tabelle 1.5: Zwei Schlüsseltexte zum selben Klartext aus Aufgabe 1.2.6.

Aufgabe 1.2.7 Zeige, dass $\mathbf{Z}$ bezüglich der gewöhnlichen Addition und Multiplikation ein Ring ohne Nullteiler ist. Ist $\mathbf{Z}$ ein Körper? Was kann man über die Eigenschaften der algebraischen Strukturen $(\mathbf{N}, +)$, $(\mathbf{N}, \cdot)$ und $(\mathbf{N}, +, \cdot)$ sagen?

Aufgabe 1.2.8 Beweise die angegebenen Eigenschaften der Eulerschen φ-Funktion:

(a) $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ für alle $m, n \in \mathbf{N}$ mit $\text{ggT}(m, n) = 1$.

(b) $\varphi(p) = p - 1$ für alle Primzahlen p.

Beweise mittels dieser Eigenschaften Fakt 1.9.

Aufgabe 1.2.9 Gegeben seien die Graphen F, G und H aus Abbildung 1.4 in Beispiel 1.15.

(a) Bestimme alle Isomorphismen zwischen G und H.

(b) Bestimme alle Automorphismen von F, G und H.

(c) Für welchen Isomorphismus zwischen G und H ist $\text{Iso}(G, H)$ eine rechte co-Menge von $\text{Aut}(G)$ in $\mathfrak{S}_5$, d.h., für welches $\tau \in \text{Iso}(G, H)$ ist $\text{Iso}(G, H) = \text{Aut}(G)\tau$? Bestimme die vollständigen rechten Transversalen von $\text{Aut}(F)$, $\text{Aut}(G)$ und $\text{Aut}(H)$ in $\mathfrak{S}_5$.

(d) Bestimme den Orbit aller Knoten von F in $\text{Aut}(F)$ und den Orbit aller Knoten von H in $\text{Aut}(H)$.

(e) Bestimme den Turm von Stabilisatoren der Untergruppen $\mathrm{Aut}(F) \leq \mathfrak{S}_5$ und $\mathrm{Aut}(H) \leq \mathfrak{S}_5$.

(f) Wie viele Graphen mit 5 Knoten sind zu F isomorph?

Aufgabe 1.2.10 Der *co-Graph* $\overline{G}$ eines Graphen G ist definiert durch die Knotenmenge $V(\overline{G}) = V(G)$ und die Kantenmenge $E(\overline{G}) = \{\{i,j\} \mid i,j \in V(\overline{G}) \text{ und } \{i,j\} \notin E(G)\}$. Zeige:

(a) $\mathrm{Aut}(G) = \mathrm{Aut}(\overline{G})$.

(b) $\mathrm{Iso}(G,H) = \mathrm{Iso}(\overline{G},\overline{H})$.

(c) $\overline{G}$ ist zusammenhängend, falls G nicht zusammenhängend ist.

1.3 Schlüsseltausch nach Diffie und Hellman

In diesem und den folgenden Abschnitten benötigen wir die zahlentheoretischen Grundlagen aus Abschnitt 1.2.4. Insbesondere sei an die multiplikative Gruppe $\mathbf{Z}_k^*$ aus Beispiel 1.8 und an die Euler-Funktion φ erinnert; die Arithmetik in Restklassenringen wird am Ende des Kapitels in Problem 1.1 erklärt.

Abbildung 1.5 zeigt das Diffie–Hellman-Protokoll für den Schlüsseltausch. Es beruht auf der modularen Exponentialfunktion zur Basis r und mit dem Modul p, wobei p eine Primzahl und r eine primitive Wurzel von p ist. Eine *primitive Wurzel einer Zahl* n ist ein Element $r \in \mathbf{Z}_n^*$, für das $r^d \not\equiv 1 \bmod n$ für jedes d mit $1 \leq d < \varphi(n)$ gilt. Eine primitive Wurzel r von n erzeugt die ganze Gruppe $\mathbf{Z}_n^*$, d.h., $\mathbf{Z}_n^* = \{r^i \,|\, 0 \leq i < \varphi(n)\}$. Man erinnere sich daran, dass für eine Primzahl p die Gruppe $\mathbf{Z}_p^*$ die Ordnung $\varphi(p) = p-1$ hat. $\mathbf{Z}_p^*$ hat genau $\varphi(p-1)$ primitive Wurzeln, siehe auch Aufgabe 1.3.1.

Beispiel 1.19 *Betrachte* $\mathbf{Z}_5^* = \{1,2,3,4\}$. *Wegen* $\mathbf{Z}_4^* = \{1,3\}$ *ist* $\varphi(4) = 2$, *und die beiden primitiven Wurzeln von* 5 *sind* 2 *und* 3. *Sowohl* 2 *als auch* 3 *erzeugt ganz* $\mathbf{Z}_5^*$, *denn:*

$$\begin{array}{llll} 2^0 = 1; & 2^1 = 2; & 2^2 = 4; & 2^3 \equiv 3 \bmod 5; \\ 3^0 = 1; & 3^1 = 3; & 3^2 \equiv 4 \bmod 5; & 3^3 \equiv 2 \bmod 5. \end{array}$$

Nicht jede Zahl hat eine primitive Wurzel; die Zahl 8 ist das kleinste solche Beispiel. Man weiß aus der elementaren Zahlentheorie, dass eine Zahl n genau dann eine primitive Wurzel hat, wenn n entweder aus $\{1,2,4\}$ ist oder die Form $n = q^k$ oder $n = 2q^k$ für eine ungerade Primzahl q hat.

Definition 1.20 (Diskreter Logarithmus) *Seien* p *eine Primzahl und* r *eine primitive Wurzel von* p. *Die* modulare Exponentialfunktion zur Basis r und mit dem Modul p *ist die Funktion* $\mathtt{exp}_{r,p}$ *von* $\mathbf{Z}_{p-1}$ *in* $\mathbf{Z}_p^*$, *die durch* $\mathtt{exp}_{r,p}(a) = r^a \bmod p$ *definiert ist. Ihre Umkehrfunktion heißt der* diskrete Logarithmus *und bildet für festes* p *und* r *den Wert* $\mathtt{exp}_{r,p}(a)$ *auf* a *ab. Man schreibt* $a = \log_r \mathtt{exp}_{r,p}(a) \bmod p$.

Das Protokoll aus Abbildung 1.5 funktioniert, denn wegen $k_A = \beta^a = r^{ba} = r^{ab} = \alpha^b = k_B$ (in der Arithmetik modulo p) berechnet Alice tatsächlich denselben Schlüssel wie Bob. Es fällt ihnen auch nicht schwer, diesen Schlüssel zu berechnen, denn die modulare Exponentialfunktion $\mathtt{exp}_{r,p}$ kann mit dem "square-and-multiply"-Verfahren aus Abbildung 1.6 effizient berechnet werden. Erich dagegen stößt beim Versuch, ihren Schlüssel zu bestimmen, auf Schwierigkeiten, denn der diskrete Logarithmus gilt als ein sehr schweres Problem. Die modulare Exponentialfunktion $\mathtt{exp}_{r,p}$ ist daher ein Kandidat für eine *Einwegfunktion*, eine Funktion, die zwar leicht berechenbar, aber nur schwer invertierbar ist. Es ist schlimm: Man weiß bis heute nicht, ob es Einwegfunktionen überhaupt gibt. Es ist noch schlimmer: Obwohl man nicht weiß, ob es sie gibt, spielen Einwegfunktionen eine Schlüsselrolle in der Kryptographie, denn die Sicherheit vieler Kryptosysteme basiert auf der Annahme, dass es Einwegfunktionen wirklich gibt.

Schritt	Alice	Erich	Bob
1	Alice und Bob vereinbaren eine große Primzahl p und eine primitive Wurzel r von p; p und r sind öffentlich		
2	wählt zufällig eine große geheime Zahl a und berechnet $\alpha = r^a \bmod p$		wählt zufällig eine große geheime Zahl b und berechnet $\beta = r^b \bmod p$
3		$\alpha \Rightarrow$ $\Leftarrow \beta$	
4	berechnet $k_A = \beta^a \bmod p$		berechnet $k_B = \alpha^b \bmod p$

Abbildung 1.5: Das Protokoll von Diffie und Hellman für den Schlüsseltausch.

SQUARE_AND-MULTIPLY(a, b, m) {

// m ist der Modul, $b < m$ ist die Basis und a ist der Exponent

Bestimme die Binärentwicklung des Exponenten $a = \sum_{i=0}^{k} a_i 2^i$, wobei $a_i \in \{0, 1\}$;

Berechne sukzessive b^{2^i}, wobei $0 \leq i \leq k$, unter Benutzung von $b^{2^{i+1}} = \left(b^{2^i}\right)^2$;

Berechne $b^a = \prod_{\substack{i=0 \\ a_i=1}}^{k} b^{2^i}$ in der Arithmetik modulo m;

// sobald ein Faktor b^{2^i} im Produkt und $b^{2^{i+1}}$ berechnet sind, kann b^{2^i} gelöscht werden

`return` b^a;

}

Abbildung 1.6: Der "square-and-multiply"-Algorithmus.

Die Berechnung von $b^a = \prod_{\substack{i=0 \\ a_i=1}}^{k} b^{2^i}$ ist dabei korrekt, denn in der Arithmetik modulo m gilt:

$$b^a = b^{\sum_{i=0}^{k} a_i 2^i} = \prod_{i=0}^{k} \left(b^{2^i}\right)^{a_i} = \prod_{\substack{i=0 \\ a_i=1}}^{k} b^{2^i}.$$

Warum kann die modulare Exponentialfunktion $\exp_{r,p}(a) = r^a \bmod p$ effizient berechnet werden? Wenn man diese Berechnung naiv ausführt, sind möglicherweise viele Multiplikationen nötig, je nach Größe des Exponenten a. Mit dem "square-and-multiply"-Algorithmus aus Abbildung 1.6 jedoch muss Alice nicht $a - 1$ Multiplikationen wie beim naiven Verfahren ausführen, sondern lediglich höchstens $2 \log a$ Multiplikationen. Der "square-and-multiply"-Algorithmus beschleunigt die modulare Exponentiation also um einen exponentiellen Faktor.

Beispiel 1.21 (Square-and-Multiply im Diffie–Hellman-Protokoll) *Alice und Bob haben die Primzahl $p = 5$ und die primitive Wurzel $r = 3$ von 5 gewählt. Alice wählt die geheime Zahl $a = 17$. Um ihre Zahl α an Bob zu schicken, möchte Alice nun $\alpha = 3^{17} = 129140163 \equiv 3 \bmod 5$ berechnen. Die Binärentwicklung des Exponenten ist $17 = 1 + 16 = 2^0 + 2^4$. Alice berechnet sukzessive die Werte:*

$$3^{2^0} = 3;\ 3^{2^1} = 3^2 \equiv 4 \bmod 5;\ 3^{2^2} \equiv 4^2 \equiv 1 \bmod 5;\ 3^{2^3} \equiv 1^2 \equiv 1 \bmod 5;\ 3^{2^4} \equiv 1^2 \equiv 1 \bmod 5.$$

Daraus berechnet sie $3^{17} \equiv 3^{2^0} \cdot 3^{2^4} \equiv 3 \cdot 1 \equiv 3 \bmod 5$. Beachte, dass Alice nicht 16-mal multipliziert, sondern lediglich viermal quadriert und einmal multipliziert hat, um $\alpha = 3 \bmod 5$ zu bestimmen. Hat Bob seinerseits den geheimen Exponenten $b = 23$ gewählt, so kann er mit demselben Verfahren

seinen Teil des Schlüssels berechnen, also $\beta = 3^{23} = 94143178827 \equiv 2 \bmod 5$, *und Alice und Bob können anschließend ihren gemeinsamen geheimen Schlüssel gemäß des Diffie–Hellman-Protokolls aus Abbildung 1.5 ermitteln; siehe Aufgabe 1.3.2. Natürlich ist die Sicherheit des Protokolls in diesem Fall nicht gewährleistet, da mit* $p = 5$ *eine sehr kleine Primzahl gewählt wurde; dieses Spielzeugbeispiel dient nur dem leichteren Verständnis.*

Wenn Erich der Kommunikation zwischen Alice und Bob im Diffie–Hellman-Protokoll aus Abbildung 1.5 aufmerksam lauscht, so kennt er die Werte p, r, α und β, aus denen er gern ihren gemeinsamen geheimen Schlüssel $k_A = k_B$ berechnen möchte. Dieses Problem Erichs nennt man das *Diffie–Hellman-Problem*. Könnte Erich die privaten Zahlen $a = \log_r \alpha \bmod p$ und $b = \log_r \beta \bmod p$ bestimmen, so könnte er wie Alice und Bob den Schlüssel $k_A = \beta^a \bmod p = \alpha^b \bmod p = k_B$ berechnen und hätte das Diffie–Hellman-Problem gelöst. Dieses Problem ist also nicht schwerer als das Problem des diskreten Logarithmus. Die umgekehrte Frage, ob das Diffie–Hellman-Problem *mindestens* so schwer wie der diskrete Logarithmus ist, ob also beide Probleme gleich schwer sind, ist bisher lediglich eine unbewiesene Vermutung. Wie viele andere Protokolle ist das Diffie–Hellman-Protokoll bislang nicht beweisbar sicher.

Da zum Glück bisher weder der diskrete Logarithmus noch das Diffie–Hellman-Problem effizient gelöst werden können, stellt dieser direkte Angriff jedoch keine wirkliche Bedrohung dar. Andererseits gibt es auch andere, indirekte Angriffe, bei denen der Schlüssel nicht unmittelbar aus den im Diffie–Hellman-Protokoll übertragenen Werten α bzw. β berechnet wird. So ist Diffie–Hellman zum Beispiel unsicher gegen "*Man-in-the-middle*"-Angriffe. Anders als der oben beschriebene *passive* Angriff ist dieser *aktiv* in dem Sinn, dass der Angreifer Erich sich nicht mit passivem Lauschen begnügt, sondern aktiv versucht, das Protokoll zu seinen Gunsten zu verändern. Er stellt sich gewissermaßen "in die Mitte" zwischen Alice und Bob und fängt Alice' Botschaft $\alpha = r^a \bmod p$ an Bob sowie Bobs Botschaft $\beta = r^b \bmod p$ an Alice ab. Statt α und β leitet er dann seine eigenen Werte $\alpha_E = r^c \bmod p$ an Bob sowie $\beta_E = r^d \bmod p$ an Alice weiter, wobei Erich die privaten Zahlen c und d selbst gewählt hat. Wenn nun Alice den Schlüssel $k_A = (\beta_E)^a \bmod p$ berechnet, den sie vermeintlich mit Bob teilt, so ist k_A in Wirklichkeit ein Schlüssel zur künftigen Kommunikation mit Erich, denn dieser ermittelt denselben Schlüssel wie sie (in der Arithmetik modulo p) durch $k_E = \alpha^d = r^{ad} = r^{da} = (\beta_E)^a = k_A$. Ebenso kann Erich unbemerkt einen Schlüsseltausch mit Bob vollziehen und künftig mit diesem kommunizieren, ohne dass Bob das Geringste bemerken würde. Mit diesem Problem der *Authentikation* beschäftigen wir uns später im Abschnitt 1.6 über Zero-Knowledge-Protokolle genauer.

Übungsaufgaben

Aufgabe 1.3.1 **(a)** Wie viele primitive Wurzeln hat $\mathbf{Z}_{13}^*$ bzw. $\mathbf{Z}_{14}^*$?

(b) Bestimme alle primitiven Wurzeln von $\mathbf{Z}_{13}^*$ sowie von $\mathbf{Z}_{14}^*$ und beweise, dass es sich tatsächlich um primitive Wurzeln handelt.

(c) Zeige für jede primitive Wurzel von 13 bzw. von 14, dass sie ganz $\mathbf{Z}_{13}^*$ bzw. ganz $\mathbf{Z}_{14}^*$ erzeugt.

Aufgabe 1.3.2 **(a)** Bestimme Bobs Zahl $\beta = 3^{23} = 94143178827 \equiv 2 \bmod 5$ aus Beispiel 1.21 mit dem "square-and-multiply"-Algorithmus aus Abbildung 1.6.

(b) Bestimme für die Zahlen α und β aus Beispiel 1.21 den gemeinsamen geheimen Schlüssel von Alice und Bob gemäß dem Diffie–Hellman-Protokoll aus Abbildung 1.5.

1.4 RSA und Faktorisierung

1.4.1 RSA

Das RSA-Kryptosystem, benannt nach seinen Erfindern Ron Rivest, Adi Shamir und Leonard Adleman [49], ist das erste bekannte *public-key*-Kryptosystem. Auch heute noch ist es sehr populär und wird in vielen kryptographischen Anwendungen eingesetzt. Abbildung 1.7 fasst die einzelnen Schritte des RSA-Protokolls zusammen, die anschließend im Detail beschrieben werden, siehe auch Beispiel 1.25.

Schritt	Alice	Erich	Bob
1			wählt zufällig große Primzahlen p und q und berechnet $n = pq$ und $\varphi(n) = (p-1)(q-1)$, seinen öffentlichen Schlüssel (n, e) und seinen privaten Schlüssel d, die (1.12) und (1.13) erfüllen
2		$\Leftarrow (n, e)$	
3	verschlüsselt m als $c = m^e \bmod n$		
4		$c \Rightarrow$	
5			entschlüsselt c als $m = c^d \bmod n$

Abbildung 1.7: Das RSA-Protokoll.

1. Schlüsselerzeugung: Bob wählt zufällig zwei große Primzahlen, p und q mit $p \neq q$, und berechnet ihr Produkt $n = pq$. Dann wählt Bob einen Exponenten $e \in \mathbf{N}$ mit

$$1 < e < \varphi(n) = (p-1)(q-1) \quad \text{und} \quad \text{ggT}(e, \varphi(n)) = 1 \tag{1.12}$$

und bestimmt die zu $e \bmod \varphi(n)$ inverse Zahl, d.h. die eindeutige Zahl d, für die gilt:

$$1 < d < \varphi(n) \quad \text{und} \quad e \cdot d \equiv 1 \bmod \varphi(n). \tag{1.13}$$

Das Paar (n, e) ist Bobs *öffentlicher Schlüssel*, und d ist Bobs *privater Schlüssel*.

2. Verschlüsselung: Wie schon in Abschnitt 1.2 sind Botschaften Wörter über einem Alphabet Σ und werden blockweise mit fester Blocklänge als natürliche Zahlen in $|\Sigma|$-adischer Darstellung codiert. Diese Zahlen werden dann verschlüsselt. Sei $m < n$ die Zahl, die einen Block der Botschaft codiert, welche Alice an Bob schicken möchte. Alice kennt Bobs öffentlichen Schlüssel (n, e) und verschlüsselt m als die Zahl $c = E_{(n,e)}(m)$, wobei die Verschlüsselungsfunktion definiert ist durch:

$$E_{(n,e)}(m) = m^e \bmod n.$$

3. Entschlüsselung: Sei c mit $0 \leq c < n$ die Zahl, die einen Block des Schlüsseltextes codiert, den Bob empfängt und der von Erich erlauscht wird. Bob entschlüsselt c mit Hilfe seines privaten Schlüssels d und der folgenden Entschlüsselungsfunktion:

$$D_d(c) = c^d \bmod n.$$

Dass das oben beschriebene RSA-Verfahren tatsächlich ein Kryptosystem im Sinne der Definition 1.1 ist, wird in Satz 1.22 festgestellt. Der Beweis dieses Satzes wird dem Leser als Aufgabe 1.4.1 überlassen.

Satz 1.22 *Seien (n, e) der öffentliche und d der private Schlüssel im RSA-Protokoll. Dann gilt für jede Botschaft m mit $0 \leq m < n$, dass $m = (m^e)^d \bmod n$. Somit ist RSA ein (public-key) Kryptosystem.*

Für die Effizienz des RSA-Verfahrens verwendet man wieder den "square-and-multiply"-Algorithmus aus Abbildung 1.6, um schnell zu potenzieren. Wie sind die Primzahlen p und q im RSA-Protokoll aus Abbildung 1.7 zu wählen? Zunächst müssen sie groß genug sein, denn könnte Erich die Zahl n in Bobs öffentlichem Schlüssel (n, e) faktorisieren und die Primfaktoren p und q von n bestimmen, so könnte er mit dem erweiterten Euklidischen Algorithmus aus Abbildung 1.3 leicht Bobs privaten Schlüssel d ermitteln, der ja das eindeutige Inverse von $e \bmod \varphi(n)$ ist, wobei $\varphi(n) = (p-1)(q-1)$. Die Primzahlen p und q müssen also geheim bleiben und deshalb hinreichend groß sein. In der Praxis sollten p und q jeweils wenigstens 80 Dezimalstellen haben. Man erzeugt zufällig Zahlen dieser Größe und testet mit einem der bekannten randomisierten Primzahltests, ob die gewählten Zahlen wirklich prim sind. Da es nach dem Primzahltheorem ungefähr $N/\ln N$ Primzahlen gibt, die kleiner als N sind, stehen die Chancen recht gut, dass man nach nicht allzu vielen Versuchen tatsächlich auf eine Primzahl stößt.

Theoretisch kann man die Primalität von p und q auch *deterministisch* in Polynomialzeit entscheiden. Agrawal et al. [1] zeigten kürzlich das überraschende Resultat, dass das Primzahlproblem, definiert durch $\texttt{PRIMES} = \{\text{bin}(n) \mid n \text{ ist prim}\}$, in der Klasse P liegt. Ihr Durchbruch löste ein lange offenes Problem der Komplexitätstheorie, denn neben dem Graphisomorphieproblem galt das Primzahlproblem lange als ein Kandidat für ein Problem, das weder in P liegt noch NP-vollständig ist.[4] Für praktische Zwecke jedoch ist dieser Algorithmus immer noch nicht effizient genug. Die Laufzeit $O(n^{12})$ des Algorithmus in der ursprünglichen Arbeit [1] konnte inzwischen zu $O(n^6)$ verbessert werden, doch auch das ist für praktische Anwendungen noch viel zu ineffizient.

```
MILLER_RABIN(n) {
  Ermittle die Darstellung n − 1 = 2^k m, wobei m und n ungerade sind;
  Wähle eine zufällige Zahl z ∈ {1, 2, ..., n − 1} unter Gleichverteilung;
  Berechne x = z^m mod n;
  if (x ≡ 1 mod n) return "n ist eine Primzahl" und halte;
    else {
      for (j = 0, 1, ..., k − 1) {
        if (x ≡ −1 mod n) return "n ist eine Primzahl" und halte;
          else x := x^2 mod n;
      }
      return "n ist keine Primzahl" und halte;
    }
}
```

Abbildung 1.8: Der Primzahltest von Miller und Rabin.

Einer der beliebtesten randomisierten Primzahltests ist der Algorithmus von Rabin [47], der in Abbildung 1.8 dargestellt ist und auf den Ideen des deterministischen Algorithmus von Miller [41] aufbaut. Der Miller–Rabin-Test ist ein so genannter Monte-Carlo-Algorithmus, denn "nein"-Antworten des Algorithmus sind stets verlässlich, aber "ja"-Antworten haben eine gewisse Fehlerwahrscheinlichkeit. Eine Alternative zum Miller–Rabin-Test ist der Primzahltest von Solovay und Strassen [64]. Beide Primzahltests arbeiten in der Zeit $O(n^3)$. Der Solovay–Strassen-Test ist jedoch weniger populär, da er in der Praxis nicht ganz so effizient ist wie der Miller–Rabin-Test und auch weniger akkurat.

[4] Die Komplexitätsklassen P und NP werden in Abschnitt 2.2 und der Begriff der NP-Vollständigkeit in Abschnitt 2.3 definiert.

Die Klasse der Probleme, die sich mittels Monte-Carlo-Algorithmen mit stets verlässlichen "ja"-Antworten lösen lassen, hat einen Namen: RP, ein Akronym für *R*andomisierte *P*olynomialzeit. Die komplementäre Klasse, coRP $= \{L \mid \overline{L} \in \mathrm{RP}\}$, enthält alle die Probleme, die sich durch Monte-Carlo-Algorithmen mit stets verlässlichen "nein"-Antworten lösen lassen. Formal definiert man RP durch nichtdeterministische polynomialzeitbeschränkte Turingmaschinen (kurz NPTMs; siehe Abschnitt 2.2 und insbesondere die Definitionen 2.1, 2.2 und 2.4), deren Berechnung als ein Zufallsprozess aufgefasst wird: Bei jeder nichtdeterministischen Verzweigung wirft die Maschine sozusagen eine faire Münze und folgt jeder der beiden Folgekonfigurationen mit der Wahrscheinlichkeit $1/2$. Je nach Anzahl der akzeptierenden Pfade der NPTM ergibt sich so eine bestimmte Akzeptierungswahrscheinlichkeit für jede Eingabe, wobei auch Fehler auftreten können. Die Definition von RP verlangt, dass die Fehlerwahrscheinlichkeit für zu akzeptierende Eingaben den Wert $1/2$ nie überschreiten darf, während bei abzulehnenden Eingaben überhaupt kein Fehler auftreten darf.

Definition 1.23 (Randomisierte Polynomialzeit) *Die Klasse* RP *enthält genau die Probleme A, für die es eine NPTM M gibt, so dass für jede Eingabe x gilt: Ist $x \in A$, so ist akzeptiert $M(x)$ mit einer Wahrscheinlichkeit $\geq 1/2$, und ist $x \notin A$, so akzeptiert $M(x)$ mit der Wahrscheinlichkeit 0.*

Satz 1.24 `PRIMES` *ist in* coRP.

Der obige Satz sagt, dass der Miller–Rabin-Test ein Monte-Carlo-Algorithmus für das Primzahlproblem ist. Der Beweis wird hier nur skizziert. Wir zeigen, dass der Miller–Rabin-Test `PRIMES` mit einseitiger Fehlerwahrscheinlichkeit akzeptiert: Ist die (binär dargestellte) Eingabe n eine Primzahl, so kann der Algorithmus nicht irrtümlich antworten, dass n keine Primzahl sei. Um einen Widerspruch zu erhalten, nehmen wir also an, dass n prim ist, aber der Miller–Rabin-Test hält mit der Ausgabe: "n ist keine Primzahl". Folglich muss $z^m \not\equiv 1 \bmod n$ gelten. Da x in jedem Durchlauf der `for`-Schleife quadriert wird, werden modulo n nacheinander die Werte z^m, z^{2m}, …, $z^{2^{k-1}m}$ getestet. Für keinen dieser Werte antwortet der Algorithmus, dass n prim wäre. Daraus folgt, dass $z^{2^j m} \not\equiv -1 \bmod n$ für jedes j mit $0 \leq j \leq k-1$ gilt. Da $n - 1 = 2^k m$ gilt, folgt $z^{2^k m} \equiv 1 \bmod n$ aus dem Kleinen Fermat, siehe Korollar 1.11. Somit ist $z^{2^{k-1}m}$ eine Quadratwurzel von 1 modulo n. Da n prim ist, gibt es nur zwei Quadratwurzeln von 1 modulo n, nämlich $\pm 1 \bmod n$, siehe Aufgabe 1.4.2. Wegen $z^{2^{k-1}m} \not\equiv -1 \bmod n$ muss also $z^{2^{k-1}m} \equiv 1 \bmod n$ gelten. Aber dann ist wiederum $z^{2^{k-2}m}$ eine Quadratwurzel von 1 modulo n. Wie oben folgt daraus $z^{2^{k-2}m} \equiv 1 \bmod n$. Indem wir dieses Argument wiederholt anwenden, erhalten wir schließlich $z^m \equiv 1 \bmod n$, ein Widerspruch. Folglich antwortet der Miller–Rabin-Test für jede Primzahl korrekt. Ist n keine Primzahl, so kann man zeigen, dass die Fehlerwahrscheinlichkeit des Miller–Rabin-Tests die Schwelle $1/4$ nicht überschreitet. Durch wiederholte unabhängige Testläufe kann man – natürlich auf Kosten der Laufzeit, die gleichwohl polynomiell in $\log n$ bleibt – die Fehlerwahrscheinlichkeit auf einen Wert beliebig nahe bei Null drücken.

Beispiel 1.25 (RSA) *Bob wählt die Primzahlen $p = 67$ und $q = 11$. Somit ist $n = 67 \cdot 11 = 737$ und $\varphi(n) = (p-1)(q-1) = 66 \cdot 10 = 660$. Wählt Bob nun den für $\varphi(n) = 660$ kleinstmöglichen Exponenten, $e = 7$, so ist sein öffentlicher Schlüssel das Paar $(n, e) = (737, 7)$. Der erweiterte Euklidische Algorithmus aus Abbildung 1.3 liefert Bobs privaten Schlüssel $d = 283$, und es gilt $e \cdot d = 7 \cdot 283 = 1981 \equiv 1 \bmod 660$; siehe Aufgabe 1.4.3. Wie in Abschnitt 1.2 identifizieren wir das Alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ mit der Menge $\mathbf{Z}_{26} = \{0, 1, \ldots, 25\}$. Botschaften sind Wörter über Σ und werden blockweise mit fester Blocklänge als natürliche Zahlen in 26-adischer Darstellung codiert. In unserem Beispiel ist die Blocklänge $\ell = \lfloor \log_{26} n \rfloor = \lfloor \log_{26} 737 \rfloor = 2$.*

Ein Block $b = b_1 b_2 \cdots b_\ell$ der Länge ℓ mit $b_i \in \mathbf{Z}_{26}$ wird durch die Zahl $m_b = \sum_{i=1}^{\ell} b_i \cdot 26^{\ell - i}$

dargestellt. Wegen der Definition der Blocklänge $\ell = \lfloor \log_{26} n \rfloor$ *gilt dabei:*

$$0 \leq m_b \leq 25 \cdot \sum_{i=1}^{\ell} 26^{\ell-i} = 26^{\ell} - 1 < n.$$

Mit der RSA-Verschlüsselungsfunktion wird der Block b *bzw. die entsprechende Zahl* m_b *verschlüsselt durch* $c_b = (m_b)^e \bmod n$*. Der Schlüsseltext für den Block* b *ist dann* $c_b = c_0 c_1 \cdots c_\ell$ *mit* $c_i \in \mathbf{Z}_{26}$*. RSA bildet also Blöcke der Länge* ℓ *injektiv auf Blöcke der Länge* $\ell+1$ *ab. Tabelle 1.6 zeigt, wie eine Botschaft der Länge* 34 *in* 17 *Blöcke der Länge* 2 *zerlegt wird und wie die einzelnen Blöcke als Zahlen dargestellt und verschlüsselt werden. Beispielsweise wird der erste Block, "*RS*", so in eine Zahl verwandelt: dem "*R*" entspricht die* 17 *und dem "*S*" die* 18*, und es gilt* $17 \cdot 26^1 + 18 \cdot 26^0 = 442 + 18 = 460$*.*

Botschaft	R S	A I	S T	H E	K E	Y T	O P	U B	L I	C K	E Y	C R	Y P	T O	G R	A P	H Y
m_b	460	8	487	186	264	643	379	521	294	62	128	69	639	508	173	15	206
c_b	697	387	229	340	165	223	586	5	189	600	325	262	100	689	354	665	673

Tabelle 1.6: Beispiel einer Verschlüsselung mit dem RSA-System.

Die resultierende Zahl c_b *wird wieder in* 26*-adischer Darstellung geschrieben und kann die Länge* $\ell + 1$ *haben:* $c_b = \sum_{i=0}^{\ell} c_i \cdot 26^{\ell-i}$*, wobei* $c_i \in \mathbf{Z}_{26}$*, siehe auch Aufgabe 1.4.3. So wird der erste Block,* $697 = 676 + 21 = 1 \cdot 26^2 + 0 \cdot 26^1 + 21 \cdot 26^0$*, in den Schlüsseltext "*BAV*" verwandelt.*

Entschlüsselt wird ebenfalls blockweise. Um etwa den ersten Block mit dem privaten Schlüssel $d = 283$ *zu entschlüsseln, wird* $697^{283} \bmod 737$ *berechnet, wieder mit der schnellen Potenzierung aus Abbildung 1.6. Damit die Zahlen nicht zu groß werden, empfiehlt es sich dabei, nach jeder Multiplikation modulo* $n = 737$ *zu reduzieren. Die Binärentwicklung des Exponenten ist* $283 = 2^0 + 2^1 + 2^3 + 2^4 + 2^8$*, und es ergibt sich wie gewünscht:*

$$697^{283} \equiv 697^{2^0} \cdot 697^{2^1} \cdot 697^{2^3} \cdot 697^{2^4} \cdot 697^{2^8} \equiv 697 \cdot 126 \cdot 9 \cdot 81 \cdot 15 \equiv 460 \bmod 737.$$

1.4.2 Digitale Signaturen mit RSA

Das *public-key*-Kryptosystem RSA aus Abbildung 1.7 kann so modifiziert werden, dass es als ein Protokoll für digitale Unterschriften verwendet werden kann. Dieses ist in Abbildung 1.9 dargestellt. Man überzeuge sich davon, dass das Protokoll funktioniert; siehe Aufgabe 1.4.4. Dieses Protokoll ist anfällig für Angriffe, bei denen der Angreifer selbst einen zu verschlüsselnden Klartext wählen kann ("*chosen-plaintext attacks*"). Dieser Angriff wird zum Beispiel in [51] beschrieben.

1.4.3 Sicherheit von RSA und mögliche Angriffe auf RSA

Wie bereits erwähnt, hängt die Sicherheit des RSA-Kryptosystems entscheidend davon ab, dass große Zahlen nicht effizient faktorisiert werden können. Da trotz hartnäckiger Versuche bisher kein effizienter Faktorisierungsalgorithmus gefunden werden konnte, vermutet man, dass es keinen solchen Algorithmus gibt, das Faktorisierungsproblem also hart ist. Ein Beweis dieser Vermutung steht indes noch aus. Selbst wenn diese Vermutung bewiesen wäre, würde daraus nicht folgen, dass das RSA-System sicher ist. Man weiß, dass das Brechen von RSA höchstens so schwer wie das Faktorisierungsproblem ist, jedoch nicht, ob es genauso schwer ist. Es wäre ja denkbar, dass man RSA auch brechen kann, ohne n zu faktorisieren.

Statt einer Liste von möglichen Angriffen auf das RSA-System, die ohnehin unvollständig bleiben müsste, verweisen wir auf die weiterführende einschlägige Literatur, siehe z.B. [6, 51], und auf Problem 1.4 am Ende des Kapitels. Für jeden der bisher bekannten Angriffe auf RSA gibt es geeignete Gegenmaßnahmen, um ihn zu vereiteln oder praktisch wirkungslos, also die Erfolgswahrscheinlichkeit

Schritt	Alice	Erich	Bob
1	wählt $n = pq$, ihren öffentlichen Schlüssel (n, e) und ihren privaten Schlüssel d wie Bob im RSA-Protokoll aus Abbildung 1.7		
2		$(n, e) \Rightarrow$	
3	signiert die Botschaft m mit $\text{sig}_A(m) = m^d \bmod n$		
4		$(m, \text{sig}_A(m)) \Rightarrow$	
5			verifiziert Alice' Signatur durch $m \equiv (\text{sig}_A(m))^e \bmod n$

Abbildung 1.9: Digitale Unterschriften mit RSA.

des Angriffs vernachlässigbar klein zu machen. Insbesondere müssen dafür die Primzahlen p und q, der Modul n, der Exponent e und der private Schlüssel d mit einer gewissen Sorgfalt gewählt werden.

Da die *Faktorisierungsangriffe* auf RSA eine besonders zentrale Rolle spielen, stellen wir einen einfachen solchen Angriff vor, der auf der $(p-1)$-Methode von John Pollard [45] beruht. Die $(p-1)$-Methode funktioniert für zusammengesetzte Zahlen n mit einem Primfaktor p, so dass die Primfaktoren von $p-1$ klein sind. Dann kann man nämlich ein Vielfaches ν von $p-1$ bestimmen, ohne p zu kennen, und nach dem Kleinen Fermat (siehe Korollar 1.11) gilt $a^\nu \equiv 1 \bmod p$ für alle ganzen Zahlen a, die zu p teilerfremd sind. Folglich ist p ein Teiler von $a^\nu - 1$. Ist n kein Teiler von $a^\nu - 1$, so ist $\text{ggT}(a^\nu - 1, n)$ ein echter Teiler von n und die Zahl n somit faktorisiert.

Wie kann das Vielfache ν von $p-1$ bestimmt werden? Das $(p-1)$-Verfahren von Pollard verwendet als Kandidaten für ν die Produkte aller Primzahlpotenzen unterhalb einer gewählten Schranke S:

$$\nu = \prod_{q \text{ ist prim, } q^k \le S} q^k.$$

Sind alle Primzahlpotenzen, die $p-1$ teilen, kleiner als S, so ist ν ein Vielfaches von $p-1$. Der Algorithmus berechnet $\text{ggT}(a^\nu - 1, n)$ für eine geeignete Basis a. Wird dabei kein echter Teiler von n gefunden, so wird der Algorithmus mit einer neuen Schranke $S' > S$ neu gestartet.

Andere Faktorisierungsmethoden wie etwa das *quadratische Sieb* werden z.B. in [65] beschrieben. Sie beruhen auf der folgenden einfachen Idee. Mit einem bestimmten Sieb werden für die zu faktorisierende Zahl n Zahlen a und b ermittelt, für die gilt:

$$a^2 \equiv b^2 \bmod n \quad \text{und} \quad a \not\equiv \pm b \bmod n. \tag{1.14}$$

Folglich teilt n die Zahl $a^2 - b^2 = (a-b)(a+b)$, aber weder $a-b$ noch $a+b$. Somit ist $\text{ggT}(a-b, n)$ ein nichttrivialer Faktor von n. Neben dem quadratischen Sieb gibt es auch andere Siebmethoden, die sich von diesem in der spezifischen Art unterscheiden, wie die Zahlen a und b ermittelt werden, die (1.14) erfüllen. Ein Beispiel ist das "allgemeine Zahlkörpersieb", siehe [37].

Übungsaufgaben

Aufgabe 1.4.1 Beweise Satz 1.22.

Hinweis: Zeige $(m^e)^d \equiv m \bmod p$ und $(m^e)^d \equiv m \bmod q$ mit Korollar 1.11, dem Kleinen Fermat. Da p und q Primzahlen mit $p \neq q$ sind und $n = pq$ gilt, folgt die Behauptung $(m^e)^d \equiv m \bmod n$ nun aus dem Chinesischen Restsatz, siehe zum Beispiel Stinson [65].

Aufgabe 1.4.2 In der Beweisskizze von Satz 1.24 wurde benutzt, dass die Primzahl n nur zwei Quadratwurzeln von 1 modulo n hat, nämlich $\pm 1 \bmod n$. Zeige dies.

Hinweis: Benutze dabei, dass r genau dann eine Quadratwurzel von 1 modulo n ist, wenn n die Zahl $(r-1)(r+1)$ teilt.

Aufgabe 1.4.3 **(a)** Zeige, dass sich für die Werte $\varphi(n) = 660$ und $e = 7$ aus Beispiel 1.25 mit dem erweiterten Euklidischen Algorithmus aus Abbildung 1.3 tatsächlich der private Schlüssel $d = 283$ ergibt, der das Inverse zu $7 \bmod 660$ ist.

(b) Bestimme für den Klartext aus Tabelle 1.6 in Beispiel 1.25 die Codierung des Schlüsseltextes durch Buchstaben aus $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ für sämtliche 17 Blöcke.

(c) Entschlüssele sämtliche 17 Blöcke des Schlüsseltextes aus Tabelle 1.6 und zeige, dass sich wieder der ursprüngliche Klartext ergibt.

Aufgabe 1.4.4 Zeige, dass das RSA-Protokoll für digitale Unterschriften aus Abbildung 1.9 funktioniert.

1.5 Die Protokolle von Rivest, Rabi und Sherman

Rivest, Rabi und Sherman schlugen Protokolle für den Schlüsseltausch und digitale Unterschriften vor. Das Schlüsseltauschprotokoll aus Abbildung 1.10 geht auf Rivest und Sherman zurück. Es kann leicht zu einem Protokoll für digitale Unterschriften modifiziert werden, siehe Aufgabe 1.5.1.

Schritt	Alice	Erich	Bob
1	wählt zufällig zwei große Zahlen x und y, hält x geheim und berechnet $x\sigma y$		
2		$(y, x\sigma y) \Rightarrow$	
3			wählt zufällig eine große Zahl z, hält z geheim und berechnet $y\sigma z$
4		$\Leftarrow y\sigma z$	
5	berechnet $k_A = x\sigma(y\sigma z)$		berechnet $k_B = (x\sigma y)\sigma z$

Abbildung 1.10: Das Rivest–Sherman-Protokoll für den Schlüsseltausch, basierend auf σ.

Das Protokoll von Rivest und Sherman beruht auf einer *totalen, stark nichtinvertierbaren, assoziativen Einwegfunktion*. Darunter versteht man das Folgende. Eine totale (also überall definierte) Funktion σ, die von $\mathbf{N}\times\mathbf{N}$ in $\mathbf{N}$ abbildet, heißt genau dann *assoziativ*, wenn $(x\sigma y)\sigma z = x\sigma(y\sigma z)$ für alle $x, y, z \in \mathbf{N}$ gilt, wobei wir statt der Präfixnotation $\sigma(x, y)$ die Infixnotation $x\sigma y$ verwenden. Aus dieser Eigenschaft folgt, dass das Protokoll funktioniert, denn wegen $k_A = x\sigma(y\sigma z) = (x\sigma y)\sigma z = k_B$ berechnen Alice und Bob tatsächlich denselben Schlüssel.

Den Begriff der starken Nichtinvertierbarkeit wollen wir hier nicht formal definieren. Informal gesprochen, heißt σ *stark nichtinvertierbar*, wenn σ nicht nur eine Einwegfunktion ist, sondern selbst dann nicht effizient invertiert werden kann, wenn zusätzlich zum Funktionswert eines der beiden zu diesem Funktionswert gehörigen Argumente bekannt ist. Diese Eigenschaft soll verhindern, dass Erich aus der Kenntnis von y und $x\sigma y$ bzw. $y\sigma z$ die geheimen Zahlen x bzw. z berechnen kann, mit deren Hilfe er dann leicht den Schlüssel $k_A = k_B$ bestimmen könnte.

Übungsaufgaben

Aufgabe 1.5.1 Modifiziere Rivest und Shermans Protokoll für den Schlüsseltausch aus Abbildung 1.10 zu einem Protokoll für digitale Unterschriften.

Aufgabe 1.5.2 Welcher direkte Angriff wäre gegen das Rivest–Sherman-Protokoll aus Abbildung 1.10 denkbar, und wie kann er verhindert werden?

Hinweis: Argumentiere über den Begriff der "starken Nichtinvertierbarkeit" der assoziativen Einwegfunktion σ, auf der das Protokoll beruht.

Aufgabe 1.5.3 **(a)** Gib eine formale Definition des Begriffs der "starken Nichtinvertierbarkeit" an.

(b) Gib eine formale Definition des Begriffs der "Assoziativität" für *partielle* Funktionen von $\mathbf{N} \times \mathbf{N}$ in $\mathbf{N}$ an. Eine partielle Definition muss nicht notwendig überall definiert sein. Was ist am folgenden Versuch einer Definition problematisch: "Eine (möglicherweise partielle) Funktion $\sigma : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ heißt *assoziativ*, falls $x\sigma(y\sigma z) = (x\sigma y)\sigma z$ für alle $x, y, z \in \mathbf{N}$ gilt, für die jedes der vier Paare (x, y), (y, z), $(x, y\sigma z)$ und $(x\sigma y, z)$ im Definitionsbereich von σ liegt.

Hinweis: Eine ausführliche Diskussion dieser Begriffe findet man in den Arbeiten [28, 26].

1.6 Interaktive Beweissysteme und Zero-Knowledge

1.6.1 Interaktive Beweissysteme, Arthur-Merlin-Spiele und Zero-Knowledge-Protokolle

Das Problem beim "*Man-in-the-middle*"-Angriff auf das Diffie–Hellman-Protokoll, das in Abschnitt 1.3 erwähnt wurde, liegt offenbar darin, dass Bob sich vor der Durchführung des Protokolls nicht von der wahren Identität seines Gesprächspartners überzeugt hat. Vermeintlich führt er das Protokoll mit Alice, in Wirklichkeit jedoch mit Erich durch. Anders formuliert, besteht die Aufgabe für Alice darin, Bob zweifelsfrei von der Echtheit ihrer Identität zu überzeugen. Diese Aufgabe der Kryptographie nennt man *Authentikation*. Im Gegensatz zur digitalen Signatur, die die Echtheit von elektronisch übermittelten *Dokumenten* wie etwa emails authentifiziert, geht es nun um die Authentikation von *Individuen*, die als Parteien an einem Protokoll teilnehmen. Diese Begriffe sind in einem weiteren Sinne zu verstehen: Als "Individuum" oder "Partei" fasst man nicht nur eine lebende Person auf, sondern zum Beispiel auch einen Computer, der mit einem anderen Computer ein Protokoll automatisch durchführt.

Um sich zu authentifzieren, könnte Alice ihre Identität durch eine nur ihr bekannte geheime Information beweisen, etwa durch ihre PIN ("*P*ersonal *I*dentifaction *N*umber") oder eine andere private Information, die niemand außer ihr kennt. Doch es gibt da einen Haken. Zum Beweis der Echtheit ihrer Identität müsste Alice Bob ihr Geheimnis verraten. Aber dann wäre es kein Geheimnis mehr! Bob könnte in einem Protokoll mit einer dritten Partei, etwa Chris, vorgeben, er selbst sei Alice, denn er kennt ja nun ihr Geheimnis. Die Frage ist also, wie man die Kenntnis eines Geheimnisses beweisen kann, ohne dieses zu verraten. Genau darum geht es bei den Zero-Knowledge-Protokollen. Diese sind spezielle interaktive Beweissysteme, die von Goldwasser, Micali und Rackoff [20] eingeführt wurden. Unabhängig entwickelten Babai und Moran [4, 3] das im Wesentlichen äquivalente Konzept der Arthur-Merlin-Spiele, die zunächst informal beschrieben werden.

Der mächtige Zauberer Merlin, repräsentiert durch eine NP-Maschine M, und der misstrauische König Arthur, repräsentiert durch eine randomisierte Polynomialzeit-Maschine A, wollen gemeinsam ein Problem L lösen, also entscheiden, ob eine Eingabe x zu L gehört oder nicht. Sie spielen um jede Eingabe, wobei sie abwechselnd ziehen. Merlins Absicht ist es dabei stets, Arthur davon zu überzeugen, dass ihr gemeinsames Eingabewort x zu L gehört, egal, ob dies nun so ist oder nicht. Ein Zug von Merlin ist die Angabe eines (behaupteten) Beweises für "$x \in L$". Diesen erhält er durch Simulation von

$M(x, y)$, wobei x die Eingabe und y die bisherigen Spielzüge codiert. Das Wort y beschreibt also die bisherigen nichtdeterministischen Wahlen von M bzw. die bisherigen Zufallswahlen von A.

König Arthur jedoch ist misstrauisch. Natürlich kann er die behaupteten Beweise des mächtigen Zauberers nicht unmittelbar selbst überprüfen; dazu fehlt ihm die Berechnungskraft. Aber er kann Merlins Beweise anzweifeln und ihm mit einer geschickten Herausforderung antworten, indem er zu *zufällig ausgewählten* Details der von Merlin gelieferten Beweise ein von ihm überprüfbares Zertifikat verlangt. Um Arthur zufriedenzustellen, muss Merlin ihn mit überwältigender Wahrscheinlichkeit von der Korrektheit seiner Beweise überzeugen. Ein Zug von Arthur besteht also darin, die Berechnung von $A(x, y)$ zu simulieren, wobei wieder x die Eingabe ist und y den bisherigen Spielverlauf beschreibt.

Die Idee der Arthur-Merlin-Spiele lässt sich durch alternierende existenzielle und probabilistische Quantoren ausdrücken, wobei erstere die NP-Berechnung Merlins und letztere die randomisierte Polynomialzeitberechnung Arthurs formalisieren.[5] In dieser Weise kann eine Hierarchie von Komplexitätsklassen definiert werden, die so genannte Arthur-Merlin-Hierarchie. Wir beschränken uns hier auf die Definition der Klasse MA, die einem Arthur-Merlin-Spiel aus zwei Zügen entspricht, bei dem Merlin zuerst zieht.

Definition 1.26 (MA in der Arthur-Merlin-Hierarchie) *Die Klasse* MA *enthält genau die Probleme L, für die es eine NPTM M und eine randomisierte Polynomialzeit-Turingmaschine A gibt, so dass für jede Eingabe x gilt:*

- *Ist $x \in L$, so existiert ein Pfad y von $M(x)$, so dass $A(x, y)$ mit Wahrscheinlichkeit $\geq 3/4$ akzeptiert (d.h., Arthur kann Merlins Beweis y für "$x \in L$" nicht widerlegen, und Merlin gewinnt).*
- *Ist $x \notin L$, so gilt für alle Pfade y von $M(x)$, dass $A(x, y)$ mit Wahrscheinlichkeit $\geq 3/4$ ablehnt (d.h., Arthur lässt sich von Merlins falschen Beweisen für "$x \in L$" nicht täuschen und gewinnt).*

Entsprechend kann man die Klassen AM, MAM, AMA, ... *definieren, siehe Aufgabe 1.6.1.*

Die Wahrscheinlichkeitsschwelle $3/4$, mit der Arthur akzeptiert bzw. ablehnt, ist in Definition 1.26 willkürlich gewählt und erscheint zunächst nicht groß genug. Tatsächlich kann man die Erfolgswahrscheinlichkeit jedoch verstärken und beliebig nahe an 1 bringen. Anders gesagt, könnte man in der Definition auch die Wahrscheinlichkeit $1/2 + \varepsilon$ verwenden, für eine beliebige, feste Konstante $\varepsilon > 0$, und man erhielte immer noch genau dieselbe Klasse. Weiter ist bekannt, dass für eine konstante Zahl von Zügen diese Hierarchie auf die Klasse AM kollabiert: NP $\subseteq$ MA $\subseteq$ AM $=$ AMA $=$ MAM $= \cdots$. Ob die Inklusionen NP $\subseteq$ MA $\subseteq$ AM echt sind oder nicht, ist jedoch offen.

Die oben erwähnten *interaktiven Beweissysteme* sind ein alternatives Modell zu den Arthur-Merlin-Spielen. Ein (unerheblicher) Unterschied besteht in der Terminologie: Merlin heißt hier "Prover" und Arthur "Verifier", und die Kommunikation läuft nicht als Spiel ab, sondern in Form eines Protokolls. Ein auf den ersten Blick erheblicher Unterschied zwischen beiden Modellen besteht darin, dass Arthurs Zufallsbits öffentlich – und insbesondere Merlin – bekannt sind, wohingegen die Zufallsbits des Verifiers bei den interaktiven Beweissystemen privat sind. Jedoch haben Goldwasser und Sipser [21] gezeigt, dass es in Wirklichkeit doch unerheblich ist, ob die Zufallsbits privat oder öffentlich sind. Arthur-Merlin-Spiele sind somit äquivalent zu den interaktiven Beweissystemen.

Lässt man statt einer konstanten Anzahl von Spielzügen polynomiell viele zu, und mehr sind wegen der polynomiellen Zeitbeschränkung nicht möglich, so erhält man die Klasse IP. Nach Definition enthält IP ganz NP und insbesondere das Graphisomorphieproblem. Wir werden später sehen, dass IP auch Probleme aus coNP $= \{\overline{L} \mid L \in \text{NP}\}$ enthält, von denen man annimmt, dass sie nicht in NP liegen.

[5] Dies ähnelt der Charakterisierung der Stufen der Polynomialzeit-Hierarchie durch alternierende $\exists$ und $\forall$ Quantoren, siehe Abschnitt 2.5 und insbesondere Teil 3 von Satz 2.22.

Insbesondere zeigt der Beweis von Satz 2.27, dass das Komplement des Graphisomorphieproblems in AM und somit in IP liegt. Ein berühmtes Resultat von Shamir [61] sagt, dass IP sogar mit PSPACE übereinstimmt, der Klasse der Probleme, die sich in polynomialem Raum entscheiden lassen.

Aber kehren wir nun zum oben geschilderten Problem der Authentikation und zum Begriff der Zero-Knowledge-Protokolle zurück. Hier ist die Idee. Angenommen, Arthur und Merlin spielen eines ihrer Spiele. In der Terminologie der interaktiven Beweissysteme schickt Merlin in diesem IP-Protokoll schwierige Beweise an Arthur. Woher er diese Beweise zaubert, ist Merlins Geheimnis, und da nur er selbst es kennt, kann er sich so Arthur gegenüber authentifizieren.

Was beide nicht wissen: Der böse Zauberer Marvin hat sich mittels eines Zaubertranks in das genaue Ebenbild Merlins verwandelt und will sich Arthur gegenüber als Merlin ausgeben. Er kennt jedoch Merlins Geheimnis nicht. Auch verfügt Marvin nicht über Merlins gewaltige Zauberkräfte, seine Magie ist nicht mächtiger als die Berechnungskraft einer gewöhnlichen randomisierten Polynomialzeit-Turingmaschine. Ebenso wenig wie Arthur kann Marvin Merlins Beweise selbst finden. Dennoch versucht er, die Kommunikation zwischen Merlin und Arthur zu simulieren. Ein IP-Protokoll hat genau dann die *Zero-Knowledge-Eigenschaft,* wenn die Information, die zwischen Marvin und Arthur ausgetauscht wird, nicht von der Kommunikation zwischen Merlin und Arthur zu unterscheiden ist. Denn Marvin, der Merlins geheime Beweise ja nicht kennt, kann natürlich keinerlei Information über sie in sein simuliertes Protokoll einfließen lassen. Da er dennoch in der Lage ist, das Originalprotokoll perfekt zu kopieren, ohne dass ein unabhängiger Beobachter irgendeinen Unterschied feststellen könnte, kann dem Protokoll auch keinerlei Information entzogen werden: Wo nichts drin ist, kann man nichts herausholen!

Definition 1.27 (Zero-Knowledge-Protokoll) *Für $L \in$* IP *seien M eine NPTM und A eine randomisierte Polynomialzeit-Turingmaschine, so dass (M, A) ein interaktives Beweissystem für L ist. Das* IP*-Protokoll (M, A) ist genau dann ein* Zero-Knowledge-Protokoll für L*, wenn es eine randomisierte Polynomialzeit-Turingmaschine $\widehat{M}$ gibt, so dass $(\widehat{M}, A)$ das Originalprotokoll (M, A) simuliert, und für jedes $x \in L$ sind die Tupel $(m_1, m_2, \ldots, m_k)$ und $(\widehat{m}_1, \widehat{m}_2, \ldots, \widehat{m}_k)$, die die Kommunikation in (M, A) bzw. in $(\widehat{M}, A)$ repräsentieren, identisch über die Münzwürfe in (M, A) bzw. in $(\widehat{M}, A)$ verteilt.*

Der oben definierte Begriff heißt in der Literatur "*honest-verifier perfect zero-knowledge*". Das heißt: (a) man nimmt an, dass der Verifier Arthur *ehrlich* ist (was in kryptographischen Anwendungen nicht unbedingt der Fall sein muss), und (b) man verlangt, dass die im simulierten Protokoll kommunizierte Information *perfekt* mit der im Originalprotokoll kommunizierten Information übereinstimmt. Die erste Annahme ist etwas idealistisch, die zweite möglicherweise etwas zu streng. Deshalb werden auch andere Varianten von Zero-Knowledge-Protokollen betrachtet, siehe die Notizen am Ende dieses Kapitels.

1.6.2 Zero-Knowledge-Protokoll für Graphisomorphie

Nun betrachten wir ein konkretes Beispiel. Wie bereits erwähnt, sind GI in NP und das komplementäre Problem, $\overline{\texttt{GI}}$, in AM, siehe den Beweis von Satz 2.27. Somit sind beide Probleme in IP. Wir geben nun ein Zero-Knowledge-Protokoll für GI an, das auf Goldreich, Micali und Wigderson [18] zurückgeht.

Auch wenn derzeit kein effizienter Algorithmus für GI bekannt ist, kann Merlin dieses Problem lösen, da GI in NP ist. Doch das muss er gar nicht. Er kann einfach einen großen Graphen G_0 mit n Knoten sowie eine Permutation $\pi \in \mathfrak{S}_n$ zufällig wählen und den Graphen $G_1 = \pi(G_0)$ berechnen. Das Paar (G_0, G_1) macht er öffentlich bekannt, den Isomorphismus π zwischen G_0 und G_1 hält er als seine private Information geheim. Abbildung 1.11 zeigt das IP-Protokoll zwischen Merlin und Arthur.

Natürlich kann Merlin nicht einfach π an Arthur schicken, denn dann wäre sein Geheimnis verraten. Um zu beweisen, dass die gegebenen Graphen, G_0 und G_1, tatsächlich isomorph sind, wählt Merlin zufällig unter Gleichverteilung einen Isomorphismus ρ und ein Bit a und berechnet den Graphen $H = \rho(G_a)$. Dann schickt er H an Arthur. Dieser antwortet mit einer Herausforderung: Er schickt ein

zufällig unter Gleichverteilung gewähltes Bit b an Merlin und verlangt von diesem einen Isomorphismus σ zwischen G_b und H. Genau dann, wenn Merlins σ tatsächlich $\sigma(G_b) = H$ erfüllt, akzeptiert Arthur.

Schritt	Merlin		Arthur
1	wählt zufällige Permutation ρ auf $V(G_0)$ und ein Bit $a \in \{0,1\}$, berechnet $H = \rho(G_a)$		
2		$H \Rightarrow$	
3			wählt Bit $b \in \{0,1\}$ zufällig und verlangt einen Isomorphismus zwischen G_b und H
4		$\Leftarrow b$	
5	berechnet Isomorphismus σ mit $\sigma(G_b) = H$: falls $b = a$, so ist $\sigma = \rho$; falls $0 = b \neq a = 1$, so ist $\sigma = \pi\rho$; falls $1 = b \neq a = 0$, so ist $\sigma = \pi^{-1}\rho$.		
6		$\sigma \Rightarrow$	
7			verifiziert, dass $\sigma(G_b) = H$, und akzeptiert entsprechend

Abbildung 1.11: Das Zero-Knowledge-Protokoll für GI von Goldreich, Micali und Wigderson.

Das Protokoll funktioniert, da Merlin seinen geheimen Isomorphismus π und seine zufällig gewählte Permutation ρ kennt. Es ist also kein Problem für Merlin, den Isomorphismus σ zwischen G_b und H zu berechnen und sich so Arthur gegenüber zu authentifizieren. Das Geheimnis π wird dabei nicht verraten. Da G_0 und G_1 isomorph sind, akzeptiert Arthur mit Wahrscheinlichkeit 1. Der Fall zweier nicht isomorpher Graphen muss hier gar nicht betrachtet werden, da Merlin ja laut Protokoll isomorphe Graphen G_0 und G_1 wählt; siehe auch den Beweis von Satz 2.27.

Angenommen, Marvin möchte sich Arthur gegenüber als Merlin ausgeben. Er kennt die Graphen G_0 und G_1, aber nicht den geheimen Isomorphismus π. Dennoch möchte er die Kenntnis von π vortäuschen. Stimmt das von Arthur gewählte Bit b zufällig mit dem Bit a überein, auf das sich Marvin *zuvor* festgelegt hat, so gewinnt er. Gilt jedoch $b \neq a$, so erfordert die Berechnung von $\sigma = \pi\rho$ oder $\sigma = \pi^{-1}\rho$ die Kenntnis von π. Da GI selbst für eine randomisierte Polynomialzeit-Turingmaschine zu hart und nicht effizient lösbar ist, kann Marvin den Isomorphismus π für hinreichend große Graphen G_0 und G_1 nicht ermitteln. Ohne π zu kennen, kann er jedoch nur raten. Seine Chancen, zufällig ein Bit b mit $b = a$ zu erwischen, sind höchstens $1/2$. Natürlich kann Marvin immer raten, und daher ist seine Erfolgswahrscheinlichkeit genau $1/2$. Wenn Arthur verlangt, dass r unabhängige Runden des Protokolls absolviert werden müssen, kann die Betrugswahrscheinlichkeit auf den Wert 2^{-r} gedrückt werden. Schon für $r = 20$ ist dies verschwindend gering: Marvins Erfolgswahrscheinlichkeit ist dann kleiner als eins zu einer Million.

Es bleibt zu zeigen, dass das Protokoll aus Abbildung 1.11 ein Zero-Knowledge-Protokoll ist. Abbildung 1.12 zeigt ein simuliertes Protokoll mit Marvin, der Merlins Geheimnis π nicht kennt, es zu kennen aber vortäuscht. Die Information, die in einer Runde des Protokolls kommuniziert wird, hat die Form eines Tripels: (H, b, σ). Wählt Marvin zufällig ein Bit a mit $a = b$, so schickt er einfach $\sigma = \rho$ und gewinnt: Arthur oder irgendein anderer, unabhängiger Beobachter wird keinerlei Unregelmäßigkeiten entdecken. Ist andererseits $a \neq b$, fliegt Marvins Schwindel auf. Doch das ist kein Problem für den tückischen Zauberer: Er löscht einfach diese Runde aus dem simulierten Protokoll und wiederholt den Versuch. So kann er eine Folge von Tripeln der Form (H, b, σ) erzeugen, die von der entsprechenden Folge von Tripeln im Originalprotokoll zwischen Merlin und Arthur ununterscheidbar sind. Folglich ist das Protokoll für GI von Goldreich, Micali und Wigderson ein Zero-Knowledge-Protokoll.

Schritt	Marvin		Arthur
1	wählt zufällige Permutation ρ auf $V(G_0)$ und ein Bit $a \in \{0,1\}$, berechnet $H = \rho(G_a)$		
2		$H \Rightarrow$	
3			wählt Bit $b \in \{0,1\}$ zufällig und verlangt einen Isomorphismus zwischen G_b und H
4		$\Leftarrow b$	
5	ist $b \neq a$, so löscht $\widehat{M}$ alle zuvor in dieser Runde übermittelten Informationen und wiederholt; ist $b = a$, so schickt $\widehat{M}$ $\sigma = \rho$		
6		$\sigma \Rightarrow$	
7			$b = a$ impliziert $\sigma(G_b) = H$, daher akzeptiert Arthur die falsche Identität Marvins

Abbildung 1.12: Simulation des Zero-Knowledge-Protokolls für `GI` ohne Kenntnis von π.

Übungsaufgaben

Aufgabe 1.6.1 Arthur-Merlin-Hierarchie:

(a) Definiere die Klassen AM, MAM, AMA, ... der Arthur-Merlin-Hierarchie analog zur Klasse MA aus Definition 1.26.

(b) Welche Klassen erhält man bei einem Arthur-Merlin-Spiel, das aus nur einem Zug besteht, den Merlin bzw. den Arthur macht?

Aufgabe 1.6.2 Zero-Knowledge-Protokoll für Graphisomorphie:

(a) Führe das Zero-Knowledge-Protokoll aus Abbildung 1.11 mit den Graphen $G_0 = G$ und $G_1 = H$ sowie dem Isomorphismus $\pi = \left(\begin{smallmatrix}1&2&3&4&5\\3&4&1&5&2\end{smallmatrix}\right)$ zwischen G und H aus, wobei G und H die Graphen aus Beispiel 1.15 in Abschnitt 1.2.4 sind. Spiele dieses Arthur-Merlin-Spiel mit einem selbst gewählten Isomorphismus ρ für alle Kombinationen von $a \in \{0,1\}$ und $b \in \{0,1\}$ durch. Wiederhole dieses Spiel mit einem dir unbekannten Isomorphismus ρ, den jemand anderes gewählt hat.

(b) Modifiziere das Protokoll aus Abbildung 1.11 so, dass die Betrugswahrscheinlichkeit Marvins auf einen Wert unter 2^{-10} gedrückt wird. Gib dabei eine formal korrekte Analyse dieser Wahrscheinlichkeit an.

Probleme

Problem 1.1 Arithmetik im Restklassenring $\mathbf{Z}_k$: Seien $k \in \mathbf{N}$ und $x, y, z \in \mathbf{Z}$. Man sagt, x *ist kongruent zu* y *modulo* k (kurz: $x \equiv y \bmod k$), falls k die Differenz $y - x$ teilt.

Zum Beispiel ist $-3 \equiv 16 \bmod 19$ und $8 \equiv 0 \bmod 2$. Die Kongruenz $\equiv$ modulo k definiert eine *Äquivalenzrelation* auf $\mathbf{Z}$, d.h., sie ist *reflexiv* ($x \equiv x \bmod k$), *symmetrisch* (aus $x \equiv y \bmod k$ folgt $y \equiv x \bmod k$) und *transitiv* (aus $x \equiv y \bmod k$ und $y \equiv z \bmod k$ folgt $x \equiv z \bmod k$).

Die Menge $x + k\mathbf{Z} = \{y \in \mathbf{Z} \mid y \equiv x \bmod k\}$ heißt *Restklasse von* $x \bmod k$. Zum Beispiel ist die Restklasse von $3 \bmod 7$ die Menge $3 + 7\mathbf{Z} = \{3, 3 \pm 7, 3 \pm 2 \cdot 7, \ldots\} = \{3, 10, -4, 17, -11, \ldots\}$. *Repräsentant einer Restklasse von* $x \bmod k$ sei stets die kleinste natürliche Zahl in $x + k\mathbf{Z}$; z.B. repräsentiert 3 die Restklasse von $3 \bmod 7$. Die Menge aller Restklassen $\bmod k$ ist $\mathbf{Z}_k = \{0, 1, \ldots, k-1\}$.

Auf $\mathbf{Z}_k$ definieren wir die *Addition* durch $(x + k\mathbf{Z}) + (y + k\mathbf{Z}) = (x + y) + k\mathbf{Z}$ und die *Multiplikation* durch $(x + k\mathbf{Z}) \cdot (y + k\mathbf{Z}) = (x \cdot y) + k\mathbf{Z}$. In der Arithmetik modulo 7 ist zum Beispiel $(3 + 7\mathbf{Z}) + (6 + 7\mathbf{Z}) = (3 + 6) + 7\mathbf{Z} = 2 + 7\mathbf{Z}$ und $(3 + 7\mathbf{Z}) \cdot (4 + 7\mathbf{Z}) = (3 \cdot 4) + 7\mathbf{Z} = 5 + 7\mathbf{Z}$.
Beweise, dass in der Arithmetik modulo k:

(a) $(\mathbf{Z}_k, +, \cdot)$ ein kommutativer Ring mit Eins ist;

(b) die in Beispiel 1.8 definierte Menge $\mathbf{Z}_k^*$ eine multiplikative Gruppe ist;

(c) $(\mathbf{Z}_p, +, \cdot)$ für eine jede Primzahl p sogar ein Körper ist. Was ist mit $(\mathbf{Z}_p^* \cup \{0\}, +, \cdot)$?

(d) Beweise, dass das neutrale Element einer Gruppe sowie das Inverse eines jeden Gruppenelements eindeutig bestimmt sind.

(e) Zeige, dass die invertierbaren Elemente eines kommutativen Rings $\mathfrak{R}$ mit Eins eine Gruppe bilden, die so genannte *Einheitengruppe von* $\mathfrak{R}$. Was ist die Einheitengruppe des Rings $\mathbf{Z}_k$?

(f) Bestimme die Nullteiler im Restklassenring $\mathbf{Z}_k$. Zeige, dass $\mathbf{Z}_k$ keine Nullteiler hat, falls k eine Primzahl ist.

Problem 1.2 Baumisomorphie: Auf speziellen Graphklassen, z.B. auf der Klasse der Bäume, ist das Problem `GI` effizient lösbar. Ein (ungerichteter) *Baum* ist ein zusammenhängender, zyklenfreier Graph, wobei ein *Zyklus* aus aufeinanderfolgenden Kanten besteht, so dass man zum Ausgangsknoten zurückkehrt. Die *Blätter* eines Baumes sind die Knoten mit Knotengrad 1. Entwirf einen effizienten Algorithmus für das Problem *Baumisomorphie*, das in P liegt. Dieses Problem ist definiert durch:

$$\texttt{TI} \quad = \quad \{(G, H) \mid G \text{ und } H \text{ sind isomorphe Bäume}\}.$$

Hinweis: Markiere sukzessive die Knoten der gegebenen Bäume mit geeigneten Zahlenfolgen und vergleiche auf jeder Stufe die resultierenden Markierungsfolgen. Beginne dabei mit den Blättern und arbeite dich Stufe um Stufe zum Inneren der Bäume vor, bis alle Knoten markiert sind; siehe auch [34].

Problem 1.3 Berechnung der Determinante: Entwirf einen Algorithmus in Pseudocode, der die Determinante einer Matrix effizient berechnet. Implementiere den Algorithmus in einer Programmiersprache deiner Wahl. Kann die Inverse einer Matrix effizient berechnet werden?

Problem 1.4 Low-Exponent-Angriff:

(a) Aus Effizienzgründen erfreut sich der Exponent $e = 3$ beim RSA-System aus Abbildung 1.7 einer gewissen Beliebtheit. Das kann jedoch gefährlich sein. Angenommen, Alice, Bob und Chris verschlüsseln dieselbe Botschaft m mit demselben öffentlichen Exponenten $e = 3$, aber womöglich mit verschiedenen Moduln, n_A, n_B und n_C. Erich fängt die drei entstehenden Schlüsseltexte ab: $c_i = m^3 \bmod n_i$ für $i \in \{A, B, C\}$. Dann kann Erich die Botschaft m leicht entschlüsseln. Wie?
Hinweis: Erich kennt den Chinesischen Restsatz [65], der schon in Aufgabe 1.6.1 nützlich war. Ein empfohlener Wert für den Exponenten ist $e = 2^{16} + 1$, dessen Binärentwicklung nur zwei Einsen hat, wodurch der "square-and-multiply"-Algorithmus besonders schnell arbeitet.

(b) Der oben beschriebene Angriff kann auf k Schlüsseltexte erweitert werden, die miteinander in Beziehung stehen. Seien etwa a_i und b_i für $1 \leq i \leq k$ bekannt, und es werden k Botschaften $c_i = (a_i m + b_i)^e \bmod n_i$ übermittelt und abgefangen, wobei $k > e(e + 1)/2$ und $\min(n_i) > 2^{e^2}$ gilt. Wie kann ein Angreifer dann die ursprüngliche Botschaft m ermitteln?
Hinweis: Mit so genannten Gitterreduktionstechniken, siehe z.B. [40]. Der hier erwähnte Angriff geht auf Johan Håstad [24] zurück und wurde von Don Coppersmith [10] verschärft.

(c) Wie kann man diese Angriffe verhindern?

Notizen zum Kapitel

Das Buch von Singh [63] gibt einen schönen Einblick in die geschichtliche Entwicklung der Kryptologie, von ihren antiken Wurzeln bis zu modernen Verschlüsselungsverfahren. Beispielsweise kann man dort nachlesen, dass die Communications Electronics Security Group (CESG) des British Government Communications Head Quarters (GCHQ) behauptet, dass ihre Mitarbeiter Ellis, Cocks und Williamson sowohl das RSA-System aus Abbildung 1.7 als auch das Diffie–Hellman-Protokoll aus Abbildung 1.5 eher als Rivest, Shamir und Adleman bzw. eher als Diffie und Hellman erfunden haben, interessanterweise in umgekehrter Reihenfolge. Das RSA-System wird in wohl jedem Buch über Kryptographie beschrieben. Eine umfassendere Liste von Angriffen gegen RSA als die in Abschnitt 1.4 angegebene findet man z.B. in den Übersichtsartikeln [6, 51].

Primalitätstests wie der von Miller und Rabin aus Abbildung 1.8 und Faktorisierungsalgorithmen werden ebenfalls in vielen Büchern beschrieben, z.B. in [65, 53, 17].

Der Begriff der stark nichtinvertierbaren assoziativen Einwegfunktionen, auf denen das Protokoll für den geheimen Schlüsseltausch aus Abbildung 1.10 beruht, geht auf Rivest und Sherman zurück. Die Modifikation dieses Protokolls zu einem solchen für digitale Unterschriften ist Rabi und Sherman zu verdanken, die in ihrer Arbeit [46] auch bewiesen, dass kommutative, assoziative Einwegfunktionen genau dann existieren, wenn $P \neq NP$ gilt. Allerdings sind die von ihnen konstruierten Einwegfunktionen weder total noch stark nichtinvertierbar, selbst unter der Bedingung $P \neq NP$ nicht. Hemaspaandra und Rothe [28] zeigten, dass es totale, stark nichtinvertierbare, kommutative, assoziative Einwegfunktionen genau dann gibt, wenn $P \neq NP$ gilt; siehe auch [5, 26].

Die beste und umfassendste Quelle für das Gebiet der interaktiven Beweissysteme und Zero-Knowledge-Protokolle, die Goldwasser, Micali und Rackoff in ihrer Arbeit [20] einführten, ist Kapitel 4 in Goldreichs Buch [17]. Ebenfalls schöne Darstellungen findet man z.B. in den Büchern von Köbler et al. [34] und Papadimitriou [43] sowie in den Übersichtsartikeln [16, 19, 51]. Arthur-Merlin-Spiele wurden insbesondere von Babai und Moran [3, 4] sowie von Zachos und Heller [71] untersucht.

Varianten von Zero-Knowledge-Protokollen, die sich in den technischen Details von Definition 1.27 unterscheiden, werden ausführlich in [17] und etwas knapper in z.B. [16, 19, 51] diskutiert.

Kapitel 2

Algorithmen in der Komplexitätstheorie

2.1 Einleitung

Wir haben in Kapitel 1 effiziente Algorithmen kennen gelernt, die für kryptographische Verfahren und Protokolle wichtig sind, zum Beispiel den Euklidischen Algorithmus und seine erweiterte Version, den "square-and-multiply"-Algorithmus und andere. Wenn der Entwurf eines effizienten Algorithmus gelingt, freut sich der Algorithmiker. Leider sträuben sich viele wichtige Probleme hartnäckig gegen die effiziente Lösbarkeit und trotzen störrisch allen Versuchen, effiziente Algorithmen für sie zu entwerfen. Beispiele solcher Probleme, auf die wir in diesem Kapitel näher eingehen, sind das Erfüllbarkeitsproblem für boolesche Ausdrücke, das Matching- und das Graphisomorphieproblem.

Solche Probleme gemäß ihrer *Berechnungskomplexität* zu klassifizieren, ist eine der wichtigsten Aufgaben der Komplexitätstheorie. Während der Algorithmiker zufrieden ist, wenn er durch den Entwurf eines konkreten Algorithmus mit einer spezifischen Laufzeit eine möglichst gute *obere Komplexitätsschranke* für sein Problem erhalten kann, versucht der Komplexitätstheoretiker, bestmögliche *untere Schranken* für dasselbe Problem zu finden. In diesem Sinn ergänzen sich Algorithmik und Komplexitätstheorie. Stimmen die obere und die untere Schranke überein, so ist das Problem klassifiziert.

Der Nachweis, dass ein Problem nicht effizient lösbar ist, wirkt oft "negativ" und gar nicht wünschenswert. Doch es gibt auch einen positiven Aspekt: Gerade in der Kryptographie (siehe Kapitel 1) ist man an den Anwendungen der Ineffizienz interessiert. Ein Beweis der Ineffizienz gewisser Probleme, wie etwa des Faktorisierungsproblems oder des diskreten Logarithmus, bedeutet hier einen Zuwachs an Sicherheit in der Übertragung verschlüsselter Nachrichten.

In Abschnitt 2.2 werden die Grundlagen der Komplexitätstheorie gelegt. Insbesondere werden dort die Komplexitätsklassen P und NP definiert. Die höchst wichtige Frage, ob diese beiden Klassen verschieden sind oder nicht, steht seit Jahrzehnten im Zentrum der Komplexitätstheorie und der gesamten Theoretischen Informatik. Bis heute ist weder ein Beweis der Vermutung $\mathrm{P} \neq \mathrm{NP}$ gelungen, noch konnte die Gleichheit von P und NP gezeigt werden. Abschnitt 2.3 gibt eine kurze Einführung in die Theorie der NP-Vollständigkeit, die diese Frage besonders intensiv untersucht.

Eines der berühmtesten NP-vollständigen Probleme ist `SAT`, das Erfüllbarkeitsproblem der Aussagenlogik: Kann eine gegebene boolesche Formel durch eine Belegung ihrer Variablen mit Wahrheitswerten *erfüllt* werden, d.h., macht die Belegung sie wahr? Wegen der NP-Vollständigkeit von `SAT` gilt es als sehr unwahrscheinlich, dass `SAT` effiziente (deterministische) Algorithmen hat. In Abschnitt 2.4 werden ein deterministischer und ein probabilistischer Algorithmus für `SAT` vorgestellt, die beide in Exponentialzeit arbeiten. Auch wenn diese Algorithmen *asymptotisch ineffizient* sind, also für sehr große Eingaben einen astronomisch großen Aufwand erfordern, kann man ihre Laufzeit für *praktisch relevante* Eingabegrößen erträglich halten.

In Abschnitt 2.5 greifen wir das Graphisomorphieproblem `GI` wieder auf, das in Definition 1.14 in

Abschnitt 1.2.4 definiert wurde und im Abschnitt 1.6.2 im Zusammenhang mit den Zero-Knowledge-Protokollen eine Rolle spielte. Dieses Problem ist eines der wenigen natürlichen Probleme in NP, die vermutlich (unter der plausiblen Annahme P $\neq$ NP) weder effizient lösbar noch NP-vollständig sind. In diesem Sinne genießt GI eine Sonderstellung unter den Problemen in NP. Die Indizien dafür entstammen der so genannten *Lowness*-Theorie, in die Abschnitt 2.5 einführt. Insbesondere wird gezeigt, dass GI in der Low-Hierarchie in NP liegt, was ein starker Hinweis darauf ist, dass GI nicht NP-vollständig sein kann. Außerdem wird gezeigt, dass GI in der Komplexitätsklasse SPP liegt und somit "*low*" für gewisse probabilistische Komplexitätsklassen ist. Informal gesagt, heißt eine Menge *low* für eine Komplexitätsklasse $\mathcal{C}$, wenn sie als "Orakel" keinerlei nützliche Information für die $\mathcal{C}$-Berechnungen liefert. Beim Beweis der genannten Resultate, dass GI low für bestimmte Komplexitätsklassen ist, erweisen sich gruppentheoretische Algorithmen als sehr nützlich.

2.2 Grundlagen

In der Einleitung wurde erwähnt, dass sich die Komplexitätstheorie unter anderem mit dem Nachweis von unteren Schranken beschäftigt. Schwierig daran ist, dass es nun nicht mehr genügt, die Laufzeit *eines* konkreten Algorithmus, der das betrachtete Problem löst, zu analysieren. Stattdessen muss man zeigen, dass *sämtliche* denkbaren Algorithmen für das betrachtete Problem eine *schlechtere* Laufzeit als die zu zeigende untere Schranke haben müssen. Dazu gehören auch solche Algorithmen, die womöglich noch gar nicht erfunden wurden. Folglich muss man zunächst den Algorithmenbegriff formal und mathematisch präzise fassen, denn sonst könnte man nicht über die Gesamtheit der denkbaren Algorithmen reden.

Es sind seit den 1930ern viele verschiedene formale Algorithmenmodelle vorgeschlagen worden. Alle diese Modelle sind in dem Sinne äquivalent, dass sich jedes solche Modell in ein beliebiges anderes dieser Modelle transformieren lässt. Etwas lax gesagt, könnte man diese Transformation als eine Art Übersetzung (Compilierung) zwischen verschiedenen Programmiersprachen auffassen. Wegen der Äquivalenz aller bisher bekannten Modelle postuliert die so genannte *These von Church*, dass ein jedes solches Algorithmenmodell den naturgemäß etwas vagen Begriff des "intuitiv Berechenbaren" präzise erfasst. Das in der Komplexitätstheorie übliche Algorithmenmodell ist die Turingmaschine, die 1936 von Alan Turing (1912 bis 1954) in seiner bahnbrechenden Arbeit [67] eingeführt wurde. Die Turingmaschine ist ein sehr einfaches, abstraktes Modell eines Computers. Im folgenden definieren wir dieses Modell durch Angabe seiner Syntax und Semantik, wobei wir zugleich zwei verschiedene Berechnungsparadigma einführen: Determinismus und Nichtdeterminismus. Es ist zweckmäßig, zuerst das allgemeinere Modell der nichtdeterministischen Turingmaschine zu beschreiben. Deterministische Turingmaschinen ergeben sich dann sofort als ein Spezialfall.

Zunächst werden einige technische Details und die Arbeitsweise von Turingmaschinen beschrieben. Eine Turingmaschine ist mit k beidseitig unendlichen Arbeitsbändern ausgestattet, die in Felder unterteilt sind, in denen Buchstaben stehen können. Enthält ein Feld keinen Buchstaben, so wird dies durch ein spezielles Leerzeichen, das $\Box$-Symbol, signalisiert. Auf den Arbeitsbändern findet die eigentliche Rechnung statt. Zu Beginn einer Rechnung steht das Eingabewort auf einem bestimmten Band, dem Eingabeband, und alle anderen Felder enthalten das $\Box$-Zeichen. Am Ende der Rechnung erscheint das Ergebnis der Rechnung auf einem bestimmten Band, dem Ausgabeband.[1] Auf jedes Band kann je ein Schreib-Lese-Kopf zugreifen. Dieser kann in einem Takt der Maschine den aktuell gelesenen Buchstaben überschreiben und anschließend eine Bewegung um ein Feld nach rechts oder links ausführen oder aber auf dem aktuellen Feld stehenbleiben. Gleichzeitig kann sich der aktuelle Zustand der Maschine ändern,

[1] Man kann z.B. festlegen, dass auf dem Eingabeband nur gelesen und auf dem Ausgabeband nur geschrieben werden darf. Ebenso kann man eine Vielzahl weiterer Variationen der technischen Details festlegen. Zum Beispiel könnte man verlangen, dass bestimmte Köpfe nur in einer Richtung wandern dürfen oder dass die Bänder halbseitig unendlich sind und so weiter.

den sie sich in ihrem inneren Gedächtnis ("*finite control*") merkt. Abbildung 2.1 zeigt eine Turingmaschine mit zwei Bändern.

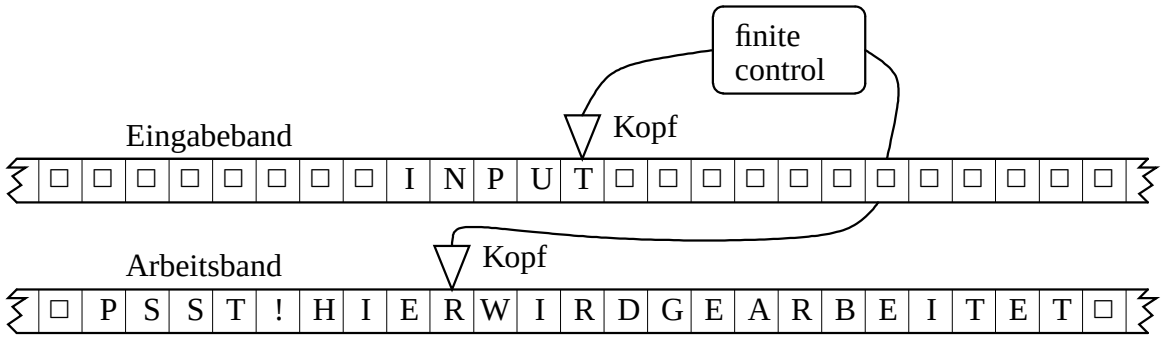


Abbildung 2.1: Eine Turingmaschine.

Definition 2.1 (Syntax von Turingmaschinen) *Eine* nichtdeterministische Turingmaschine mit k Bändern *(kurz k-Band-NTM) ist ein* 7*-Tupel* $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$*, wobei* Σ *das* Eingabealphabet, Γ *das Arbeitsalphabet mit* $\Sigma \subseteq \Gamma$*,* Z *eine endliche* Menge von Zuständen *mit* $Z \cap \Gamma = \emptyset$*,* $\delta : Z \times \Gamma^k \to \mathfrak{P}(Z \times \Gamma^k \times \{L, R, N\}^k)$ *die* Überführungsfunktion*,* $z_0 \in Z$ *der* Startzustand*,* $\square \in \Gamma - \Sigma$ *das* Leerzeichen *und* $F \subseteq Z$ *die* Menge der Endzustände *ist. Hier bezeichnet* $\mathfrak{P}(S)$ *die* Potenzmenge *einer Menge* S*, also die Menge aller Teilmengen von* S*.*

Statt $(z', b, x) \in \delta(z, a)$ *mit* $z, z' \in Z$*,* $x \in \{L, R, N\}$ *und* $a, b \in \Gamma$ *schreiben wir auch kurz* $(z, a) \mapsto (z', b, x)$*. Dieser Turingbefehl bedeutet das Folgende. Ist im Zustand* z *der Kopf auf einem Feld mit aktueller Inschrift* a*, so wird:*

- a *durch* b *überschrieben,*
- *der neue Zustand* z' *angenommen und*
- *eine Kopfbewegung gemäß* $x \in \{L, R, N\}$ *ausgeführt, d.h., der Kopf wandert entweder ein Feld nach* l*inks (*L*) oder ein Feld nach* r*echts (*R*) oder er bleibt auf dem aktuellen Feld stehen (*N *wie* n*eutral).*

Der Spezialfall der deterministischen Turingmaschine mit k Bändern *(kurz k-Band-DTM) ergibt sich, wenn die Überführungsfunktion* δ *von* $Z \times \Gamma^k$ *nach* $Z \times \Gamma^k \times \{L, R, N\}^k$ *abbildet.*

Für $k = 1$ ergibt sich die 1-Band-Turingmaschine, die wir einfach mit NTM bzw. DTM abkürzen. Jede k-Band-NTM bzw. k-Band-DTM kann durch eine entsprechende Maschine mit nur einem Band simuliert werden, wobei sich die Rechenzeit höchstens verdoppelt. Spielt die Effizienz eine Rolle, kann es dennoch sinnvoll sein, mehrere Bänder zu haben.

Turingmaschinen kann man sowohl als Akzeptoren auffassen, die Sprachen (also Wortmengen) akzeptieren, als auch zur Berechnung von Funktionen benutzen.

Definition 2.2 (Semantik von Turingmaschinen) *Sei* $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$ *eine NTM. Eine* Konfiguration von M *ist ein Wort* $k \in \Gamma^* Z \Gamma^*$*. Dabei bedeutet* $k = \alpha z \beta$*, dass* $\alpha\beta$ *die aktuelle Bandinschrift ist (also das Wort auf dem bereits vom Kopf besuchten Teil des Bandes), dass der Kopf auf dem ersten Symbol von* β *steht und dass* z *der aktuelle Zustand von* M *ist.*

Auf der Menge $\mathfrak{K}_M = \Gamma^* Z \Gamma^*$ *aller Konfigurationen von* M *definieren wir eine binäre Relation* $\vdash_M$*, die den Übergang von einer Konfiguration* $k \in \mathfrak{K}_M$ *in eine Konfiguration* $k' \in \mathfrak{K}_M$ *durch eine Anwendung der Überführungsfunktion* δ *beschreibt. Für alle Wörter* $\alpha = a_1 a_2 \cdots a_m$ *und* $\beta = b_1 b_2 \cdots b_n$

in Γ^**, wobei* $m \geq 0$ *und* $n \geq 1$*, und für alle* $z \in Z$ *sei*

$$\alpha z \beta \vdash_M \begin{cases} a_1 a_2 \cdots a_m z' c b_2 \cdots b_n & \text{falls } (z, b_1) \mapsto (z', c, N) \text{ und } m \geq 0 \text{ und } n \geq 1 \\ a_1 a_2 \cdots a_m c z' b_2 \cdots b_n & \text{falls } (z, b_1) \mapsto (z', c, R) \text{ und } m \geq 0 \text{ und } n \geq 2 \\ a_1 a_2 \cdots a_{m-1} z' a_m c b_2 \cdots b_n & \text{falls } (z, b_1) \mapsto (z', c, L) \text{ und } m \geq 1 \text{ und } n \geq 1. \end{cases}$$

Es sind noch zwei Sonderfälle zu betrachten:

1. *Ist* $n = 1$ *und* $(z, b_1) \mapsto (z', c, R)$ *(d.h.,* M *läuft nach rechts und trifft auf ein* $\Box$*-Symbol), so sei* $a_1 a_2 \cdots a_m z b_1 \vdash_M a_1 a_2 \cdots a_m c z' \Box$.
2. *Ist* $m = 0$ *und* $(z, b_1) \mapsto (z', c, L)$ *(d.h.,* M *läuft nach links und trifft auf ein* $\Box$*-Symbol), so sei* $z b_1 b_2 \cdots b_n \vdash_M z' \Box c b_2 \cdots b_n$.

Die Startkonfiguration von M bei Eingabe x *ist stets* $z_0 x$*. Die* Endkonfigurationen von M bei Eingabe x *haben die Form* $\alpha z \beta$ *mit* $z \in F$ *und* $\alpha, \beta \in \Gamma^*$.

Sei $\vdash_M^*$ *die reflexive, transitive Hülle von* $\vdash_M$*. Das heißt: Für* $k, k' \in \mathfrak{K}_M$ *gilt* $k \vdash_M^* k'$ *genau dann, wenn es eine endliche Folge* $k_0, k_1, \ldots, k_t$ *von Konfigurationen in* $\mathfrak{K}_M$ *gibt, so dass gilt:*

$$k = k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t = k',$$

wobei $k = k_0 = k_t = k'$ *möglich ist. Ist dabei* $k_0 = z_0 x$ *die Startkonfiguration von* M *bei Eingabe* x*, so heißt diese Folge von Konfigurationen* endliche Rechnung von $M(x)$*, und man sagt,* M hält bei Eingabe x an. *Die* von M akzeptierte Sprache *ist definiert als:*

$$L(M) = \{x \in \Sigma^* \mid z_0 x \vdash_M^* \alpha z \beta \text{ mit } z \in F \text{ und } \alpha, \beta \in \Gamma^*\}.$$

Man kann die Menge F *der Endzustände von* M *auch in die Menge* F_a *der* akzeptierenden Endzustände *und die Menge* F_r *der* ablehnenden Endzustände *unterteilen, wobei* $F = F_a \cup F_r$ *und* $F_a \cap F_r = \emptyset$ *gilt. Dann ist* $L(M) = \{x \in \Sigma^* \mid z_0 x \vdash_M^* \alpha z \beta \text{ mit } z \in F_a \text{ und } \alpha, \beta \in \Gamma^*\}$ *die von* M *akzeptierte Sprache.*

M berechnet eine Wortfunktion $f : \Sigma^* \to \Delta^*$*, falls für alle* $x \in \Sigma^*$ *und für alle* $y \in \Delta^*$ *gilt:*

1. $x \in D_f \iff M$ *hält bei Eingabe von* x *nach endlich vielen Schritten an;*
2. *für alle* $x \in D_f$ *gilt:* $f(x) = y \iff z_0 x \vdash_M^* z y$ *für ein* $z \in F$*,*

wobei D_f *den* Definitionsbereich von f *bezeichnet. Eine Wortfunktion, die von einer Turingmaschine berechnet wird, heißt* berechenbar*. Eine Funktion* $f : \mathbf{N}^k \to \mathbf{N}$ *heißt* berechenbar*, falls die durch*

$$g(bin(x_1)\#bin(x_2)\#\cdots\#bin(x_k)) = bin(f(x_1, x_2, \ldots, x_k))$$

definierte Wortfunktion $g : \{0, 1, \#\}^* \to \{0, 1\}^*$ *berechenbar ist. Dabei bezeichnet* $bin(n)$ *die Binärdarstellung (ohne führende Nullen) von* $n \in \mathbf{N}$*; z.B. ist* $bin(17) = 10001$.

Da im Falle einer NTM jede Konfiguration mehrere Folgekonfigurationen haben kann, ergibt sich ein *Berechnungsbaum*, dessen Wurzel die Startkonfiguration und dessen Blätter die Endkonfigurationen sind. Bäume sind spezielle Graphen (siehe Definition 1.14 in Abschnitt 1.2.4 und Aufgabe 2.2.2), bestehen also aus Knoten und Kanten. Die Knoten des Berechnungsbaums von $M(x)$ sind die Konfigurationen von M bei Eingabe x. Für zwei Konfigurationen k und k' aus $\mathfrak{K}_M$ gibt es genau dann eine gerichtete Kante von k nach k', wenn $k \vdash_M k'$ gilt. Ein Pfad im Berechnungsbaum von $M(x)$ ist eine Folge von Konfigurationen $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t \vdash_M \cdots$, also eine Rechnung von $M(x)$. Der Berechnungsbaum einer NTM kann unendliche Pfade haben. Im Falle einer DTM wird jede Konfiguration außer der Startkonfiguration eindeutig (*deterministisch*) durch ihre Vorgängerkonfiguration bestimmt. Deshalb entartet der Berechnungsbaum einer DTM zu einer linearen Kette, die mit der Startkonfiguration beginnt und mit einer Endkonfiguration endet, falls die Maschine bei dieser Eingabe hält; andernfalls geht die Kette ins Unendliche.

Beispiel 2.3 *Betrachte die Sprache $L = \{a^n b^n c^n \mid n \geq 1\}$. Eine Turingmaschine, die L akzeptiert, ist definiert durch*

$$M = (\{a, b, c\}, \{a, b, c, \$, \square\}, \{z_0, z_1, \ldots, z_6\}, \delta, z_0, \square, \{z_6\}),$$

wobei die Liste der Turingbefehle gemäß der Überführungsfunktion δ in Tabelle 2.1 angegeben ist. Tabelle 2.2 gibt die Bedeutung der einzelnen Zustände von M sowie die mit den einzelnen Zuständen verbundene Absicht an. Siehe auch Aufgabe 2.2.2.

$(z_0, a) \mapsto (z_1, \$, R)$	$(z_2, \$) \mapsto (z_2, \$, R)$	$(z_5, c) \mapsto (z_5, c, L)$
$(z_1, a) \mapsto (z_1, a, R)$	$(z_3, c) \mapsto (z_3, c, R)$	$(z_5, \$) \mapsto (z_5, \$, L)$
$(z_1, b) \mapsto (z_2, \$, R)$	$(z_3, \square) \mapsto (z_4, \square, L)$	$(z_5, b) \mapsto (z_5, b, L)$
$(z_1\$) \mapsto (z_1, \$, R)$	$(z_4, \$) \mapsto (z_4, \$, L)$	$(z_5, a) \mapsto (z_5, a, L)$
$(z_2, b) \mapsto (z_2, b, R)$	$(z_4, \square) \mapsto (z_6, \square, R)$	$(z_5, \square) \mapsto (z_0, \square, R)$
$(z_2, c) \mapsto (z_3, \$, R)$	$(z_4, c) \mapsto (z_5, c, L)$	$(z_0, \$) \mapsto (z_0, \$, R)$

Tabelle 2.1: Liste δ der Turingbefehle von M für die Sprache $L = \{a^n b^n c^n \mid n \geq 1\}$.

Z	Bedeutung	Absicht
z_0	Anfangszustand	neuer Zyklus
z_1	ein a gemerkt	nächstes b suchen
z_2	je ein a, b gemerkt	nächstes c suchen
z_3	je ein a, b, c getilgt	rechten Rand suchen
z_4	rechter Rand erreicht	Zurücklaufen und Test, ob alle a, b, c getilgt
z_5	Test nicht erfolgreich	Zurücklaufen zum linken Rand und neuer Zyklus
z_6	Test erfolgreich	Akzeptieren

Tabelle 2.2: Interpretation der Zustände von M.

Komplexitätstheoretiker sind ordentliche Menschen. Sie bringen gern Ordnung und Systematik in die ungeheure Vielfalt von wichtigen Problemen. Zu diesem Zweck klassifizieren und katalogisieren sie diese und ordnen sie in Komplexitätsklassen ein. Jede solche Klasse enthält alle die Probleme, die bezüglich eines bestimmten Komplexitätsmaßes etwa denselben Aufwand zur Lösung oder Berechnung erfordern. Die gängigsten Komplexitätsmaße sind das *Zeitmaß* (die nötige Anzahl von Schritten, die ein Algorithmus zur Lösung braucht) und das *Raummaß* (der dabei erforderliche Speicherplatz im Computer). Wir beschränken uns hier auf das Zeitmaß.

Unter der "Zeit", die ein Algorithmus zur Lösung eines Problems braucht, verstehen wir die Anzahl seiner Schritte als Funktion der Eingabegröße. Unser formales Algorithmenmodell ist die Turingmaschine, und ein Schritt oder Takt einer Turingmaschine ist eine Anwendung ihrer Überführungsfunktion δ, also ein Übergang von einer Konfiguration der Berechnung zur nächsten. Wir beschränken uns hier auf das traditionelle *worst-case*-Modell der Komplexität. Das heißt, dass man für die Zeitfunktion einer Turingmaschine unter allen Eingaben einer jeden Größe n gerade diejenigen Eingaben als entscheidend betrachtet, für die die Maschine am längsten braucht. Man nimmt also den schlimmsten Fall an. Im Gegensatz dazu untersucht man bei der *average-case*-Komplexität die erwartete Laufzeit eines Algorithmus im Mittel gemäß einer gegebenen Wahrscheinlichkeitsverteilung der Eingaben einer jeden Länge.

Nun werden deterministische und nichtdeterministische Zeitkomplexitätsklassen definiert.

Definition 2.4 (Deterministische und Nichtdeterministische Zeitkomplexität)

- *Sei M eine DTM mit $L(M) \subseteq \Sigma^*$ und sei $x \in \Sigma^*$ eine Eingabe. Definiere die* Zeitfunktion

von $M(x)$, *die von* Σ^* *in* $\mathbf{N}$ *abbildet, wie folgt:*

$$\text{Time}_M(x) \quad = \quad \begin{cases} m & \textit{falls } M(x) \textit{ genau } m+1 \textit{ Konfigurationen hat} \\ \textit{undefiniert} & \textit{sonst.} \end{cases}$$

Definiere die Funktion $\text{time}_M : \mathbf{N} \to \mathbf{N}$ *durch:*

$$\text{time}_M(n) \quad = \quad \begin{cases} \max_{x:|x|=n} \text{Time}_M(x) & \textit{falls } \text{Time}_M(x) \textit{ für jedes } x \\ & \qquad \textit{mit } |x| = n \textit{ definiert ist} \\ \textit{undefiniert} & \textit{sonst.} \end{cases}$$

- *Sei* M *eine NTM mit* $L(M) \subseteq \Sigma^*$ *und sei* $x \in \Sigma^*$ *eine Eingabe. Definiere die* Zeitfunktion von $M(x)$, *die von* Σ^* *in* $\mathbf{N}$ *abbildet, wie folgt:*

$$\text{NTime}_M(x) \quad = \quad \begin{cases} \min\{\text{Time}_M(x, \alpha) \mid M(x) \textit{ akzeptiert auf Pfad } \alpha\} & \textit{falls } x \in L(M) \\ \textit{undefiniert} & \textit{sonst.} \end{cases}$$

Definiere die Funktion $\text{ntime}_M : \mathbf{N} \to \mathbf{N}$ *durch:*

$$\text{ntime}_M(n) \quad = \quad \begin{cases} \max_{x:|x|=n} \text{NTime}_M(x) & \textit{falls } \text{NTime}_M(x) \textit{ für jedes } x \\ & \qquad \textit{mit } |x| = n \textit{ definiert ist} \\ \textit{undefiniert} & \textit{sonst.} \end{cases}$$

- *Sei* t *eine berechenbare Funktion, die von* $\mathbf{N}$ *in* $\mathbf{N}$ *abbildet. Definiere die* deterministischen und nichtdeterministischen Komplexitätsklassen mit Zeitfunktion t *durch:*

$$\text{DTIME}(t) \quad = \quad \left\{ A \;\middle|\; \begin{array}{l} A = L(M) \textit{ für eine DTM } M \textit{ und} \\ \textit{für alle } n \in \mathbf{N} \textit{ ist } \text{time}_M(n) \leq t(n) \end{array} \right\};$$

$$\text{NTIME}(t) \quad = \quad \left\{ A \;\middle|\; \begin{array}{l} A = L(M) \textit{ für eine NTM } M \textit{ und} \\ \textit{für alle } n \in \mathbf{N} \textit{ ist } \text{ntime}_M(n) \leq t(n) \end{array} \right\}.$$

- *Sei* $\mathbb{P}\text{ol}$ *die Menge aller Polynome. Definiere die Komplexitätsklassen* P *und* NP *wie folgt:*

$$\text{P} = \bigcup_{t \in \mathbb{P}\text{ol}} \text{DTIME}(t) \qquad \textit{und} \qquad \text{NP} = \bigcup_{t \in \mathbb{P}\text{ol}} \text{NTIME}(t).$$

DPTM *bzw.* NPTM *steht für polynomialzeitbeschränkte DTM bzw. NTM.*

Weshalb sind die Polynomialzeitklassen P und NP so wichtig? Erinnert man sich an Tabelle 1.4, die die Anfangsglieder der exponentiell wachsenden Fibonacci-Folge zeigt, so ahnt man, dass Algorithmen mit exponentieller Laufzeit nicht als effizient betrachtet werden können. Garey und Johnson [15] vergleichen für einige praxisrelevante Eingabegrößen die Wachstumsraten ausgewählter polynomieller und exponentieller Zeitfunktionen $t(n)$, siehe Tabelle 2.3. Dabei gehen sie von einem Computer aus, der pro Sekunde eine Million Operationen ausführen kann. Man sieht, dass alle durch Polynomialzeitfunktionen beschränkten Algorithmen bis zur Eingabegröße $n = 60$ das Ergebnis in vernünftiger Zeit liefern, wohingegen z.B. ein in der Zeit $t(n) = 3^n$ laufender Algorithmus bereits für die relativ bescheidene Problemgröße von $n = 30$ über 6 Jahre braucht. Bei der Problemgröße $n = 40$ benötigt er schon fast 400 Jahrtausende und ab etwa $n = 50$ eine wahrhaft astronomische Zeitspanne.

In den letzten Jahrzehnten konnte man eine eindrucksvolle Entwicklung der Computertechnik und der Hardwaretechnologie beobachten. Tabelle 2.4 aus [15] zeigt, dass dies nicht hilft, um die absolute Ausführungszeit exponentiell zeitbeschränkter Algorithmen wesentlich zu reduzieren, selbst wenn man

$t(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
n	.00001 sec	.00002 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0004 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.008 sec	.027 sec	.064 sec	.125 sec	.256 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
2^n	.001 sec	1.0 sec	17.9 min	12.7 Tage	35.7 Jahre	366 Jhdte.
3^n	.059 sec	58 min	6.5 Jahre	3855 Jhdte.	$2 \cdot 10^8$ Jhdte.	$1.3 \cdot 10^{13}$ Jhdte.

Tabelle 2.3: Vergleich einiger polynomieller und exponentieller Zeitfunktionen.

davon ausgeht, dass die bisherige Entwicklung von immer schnelleren Chips weiter anhält. Was würde geschehen, wenn man einen Computer benutzte, der 100-mal oder sogar 1000-mal schneller wäre als die schnellsten Computer von heute? Für die Funktionen $t_i(n)$, $1 \leq i \leq 6$, bezeichne N_i die maximale Größe der Probleme, die mit einem $t_i(n)$-zeitbeschränkten Algorithmus innerhalb einer Stunde gelöst werden können. Man sieht in Tabelle 2.4, dass selbst ein tausendfacher Geschwindigkeitszuwachs der Computer den Wert N_5 für $t_5(n) = 2^n$ um lediglich knapp 10 erhöht. Im Gegensatz dazu könnte ein n^5-zeitbeschränkter Algorithmus bei demselben Geschwindigkeitszuwachs in einer Stunde Probleme behandeln, die etwa viermal größer sind.

$t_i(n)$	Computer heute	100-mal schneller	1000-mal schneller
$t_1(n) = n$	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
$t_2(n) = n^2$	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
$t_3(n) = n^3$	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
$t_4(n) = n^5$	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
$t_5(n) = 2^n$	N_5	$N_5 + 6.64$	$N_5 + 9.97$
$t_6(n) = 3^n$	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Tabelle 2.4: Was, wenn die Computer schneller werden?

Das folgende Dogma drückt die weit verbreitete Überzeugung aus, dass Polynomialzeit-Algorithmen als effizient betrachtet werden, während Algorithmen, die nur exponentielle untere Schranken haben, ausgesprochen schlecht und ineffizient sind.

Dogma 2.5 *Polynomialzeit erfasst den intuitiven Begriff der Effizienz. Exponentialzeit erfasst den intuitiven Begriff der Ineffizienz.*

Natürlich ist ein Dogma nur ein Dogma, eine Sache des Glaubens, und daher sollte Dogma 2.5 kritisch diskutiert werden. Ein Algorithmus, der in $n^{10^{77}}$ Schritten arbeitet, ist zwar formal gesehen ein Polynom in n mit konstantem Grad. Jedoch ist der Grad dieses Polynoms zufällig so groß wie die derzeit geschätzte Anzahl der im gesamten sichtbaren Universum vorhandenen Atome. Deshalb ist ein solcher Algorithmus höchst ineffizient und praktisch nicht sinnvoll, selbst für kleinste Problemgrößen nicht. Andererseits ist eine exponentielle Zeitschranke wie $2^{0.00001 \cdot n}$ für in der Praxis wichtige Problemgrößen durchaus vernünftig. Irgendwann schlägt natürlich das exponentielle Wachstum zu, doch bei dem Exponenten $0.00001 \cdot n$ wird das erst für sehr große n der Fall sein. Diese beiden Extremfälle treten allerdings so gut wie nie in der Realität auf. Die überwältigende Mehrheit der natürlichen Probleme in P lässt sich durch Algorithmen lösen, deren Laufzeit ein Polynom geringen Grades ist, wie $O(n^2)$ oder $O(n^3)$. Polynome vierten, fünften oder noch höheren Grades treten sehr selten auf.

Die Klasse P umfasst nach Dogma 2.5 genau die effizient lösbaren Probleme. Die Klasse NP enthält viele in der Praxis wichtige Probleme, für die bisher keine effizienten Algorithmen gefunden werden konnten, so etwa das Erfüllbarkeits- und das Graphisomorphieproblem. Diese werden in Kapitel 2 genauer untersucht. Die Frage, ob die Klassen P und NP gleich sind oder nicht, ist bis heute ungelöst. Dies

ist die berühmte P-versus-NP-Frage, die als die wichtigste offene Frage der Theoretischen Informatik angesehen werden kann. Insbesondere spielt sie auch in der Kryptographie eine Rolle, denn die Sicherheit der meisten heute benutzten Kryptosysteme beruht auf der Annahme, dass bestimmte Probleme schwer lösbar sind. Dazu gehören das Faktorisierungsproblem sowie das Problem des diskreten Logarithmus, die in Kapitel 1 näher untersucht werden. Könnte man P $=$ NP beweisen, so wären all diese Kryptosysteme unsicher und damit nutzlos.

Die P-versus-NP-Frage hat insbesondere die Theorie der NP-Vollständigkeit ins Leben gerufen. Diese liefert Methoden zum Beweis unterer Schranken für die härtesten Probleme in NP. Dabei muss man nur von einem einzigen harten Problem in NP ausgehen. Die NP-Härte vieler anderer NP-Probleme folgt dann mittels einer Reduktion, die das eine Problem in das andere transformiert. Kurioserweise sind es effiziente Algorithmen – denn nichts anderes sind Reduktionen –, die den Nachweis der NP-Härte von schweren Problemen erlauben. Probleme, die in NP liegen und NP-hart sind, heißen NP-vollständig. Sie können nicht zu P gehören, also nicht effizient lösbar sein, außer wenn P $=$ NP gelten würde. Die Theorie der NP-Vollständigkeit wird in Abschnitt 2.3 vorgestellt.

Übungsaufgaben

Aufgabe 2.2.1 Kann man die These von Church je beweisen? Begründe deine Antwort.

Aufgabe 2.2.2 Betrachte die Turingmaschine M in Beispiel 2.3.

(a) Gib die Folge der Konfigurationen von M bei Eingabe $x = a^3b^3c^2$ bzw. $y = a^3b^3c^3$ an.

(b) Beweise die Korrektheit von M, d.h., zeige die Gleichheit $L(M) = \{a^n b^n c^n \mid n \geq 1\}$.

(c) Gib eine Abschätzung für die Laufzeit von M an.

Aufgabe 2.2.3 Gib eine Turingmaschine für den Euklidischen Algorithmus aus Abbildung 1.2 an.

Hinweis: Implementiere den Algorithmus *iterativ*, nicht *rekursiv*. Das heißt, es gibt keine rekursiven Aufrufe, sondern die berechneten Zwischenwerte werden explizit gespeichert.

Aufgabe 2.2.4 Zeige, dass die in Definition 1.14 definierten Probleme `GI` und `GA` in NP liegen.

2.3 NP-Vollständigkeit

Die Theorie der NP-Vollständigkeit liefert Methoden zum Nachweis unterer Schranken für Probleme in NP. Ein NP-Problem heißt *vollständig in* NP, falls es zu den härtesten Problemen dieser Klasse gehört. Um die NP-Härte eines Problems X zu beweisen, muss man also sämtliche Probleme aus NP mit X vergleichen und zeigen, dass X mindestens so schwer wie das jeweils betrachtete Problem ist. Die Komplexität zweier Probleme kann man mit Hilfe von polynomialzeitbeschränkten Reduktionen miteinander vergleichen. Unter den vielen verschiedenen Typen von Reduzierbarkeiten, die man definieren kann, ist hier die so genannte "*many-one*-Reduzierbarkeit" relevant, die mit $\leq^{\mathrm{p}}_{\mathrm{m}}$ bezeichnet wird. Da wir in diesem Abschnitt keinen anderen Reduzierbarkeitstyp als diesen betrachten, sprechen wir einfach von "Reduzierbarkeit". In Abschnitt 2.5 lernen wir allgemeinere Reduzierbarkeiten kennen, die so genannte *Turing-Reduzierbarkeit* und die (*starke*) *nichtdeterministische Turing-Reduzierbarkeit*.

Definition 2.6 (Reduzierbarkeit, NP-Vollständigkeit) *Eine Menge A ist genau dann* reduzierbar *auf eine Menge B (symbolisch $A \leq^{\mathrm{p}}_{\mathrm{m}} B$), wenn es eine in Polynomialzeit berechenbare Funktion r gibt, so dass für alle $x \in \Sigma^*$ gilt: $x \in A \iff r(x) \in B$. Eine Menge B heißt genau dann* $\leq^{\mathrm{p}}_{\mathrm{m}}$-hart für NP, *wenn $A \leq^{\mathrm{p}}_{\mathrm{m}} B$ für jede Menge $A \in$ NP gilt. Eine Menge B heißt genau dann* $\leq^{\mathrm{p}}_{\mathrm{m}}$-vollständig in NP *(oder kurz* NP-vollständig*), wenn $B \leq^{\mathrm{p}}_{\mathrm{m}}$-hart für* NP *ist und $B \in$* NP.

Anscheinend muss man zum Nachweis der NP-Härte von X unendlich viele effiziente Algorithmen finden, um ein jedes der unendlich vielen Probleme aus NP effizient auf X zu reduzieren. Ein grundlegendes Resultat sagt jedoch, dass es nicht nötig ist, *unendlich viele* solche Reduktionen auf X anzugeben. Es genügt, ein einziges NP-vollständiges Problem V auf X zu reduzieren. Da die $\leq_{\mathrm{m}}^{\mathrm{p}}$-Reduzierbarkeit transitiv ist (siehe Aufgabe 2.3.2) und da V NP-hart ist, folgt die NP-Härte von X mit der Reduktion $A \leq_{\mathrm{m}}^{\mathrm{p}} V \leq_{\mathrm{m}}^{\mathrm{p}} X$ für jedes NP-Problem A.

Stephen Cook fand 1971 ein erstes solches NP-vollständiges Problem: das Erfüllbarkeitsproblem für aussagenlogische Ausdrücke ("*satisfiability problem*"), kurz mit SAT bezeichnet. Für viele NP-Vollständigkeitsresultate ist es zweckmäßig, wenn man von 3-SAT ausgeht, der Einschränkung des Erfüllbarkeitsproblems, bei der die gegebene boolesche Formel in konjunktiver Normalform vorliegt und jede Klausel genau drei Literale enthält. Auch 3-SAT ist NP-vollständig. Ob eine boolesche Formel in disjunktiver Normalform erfüllbar ist, lässt sich effizient entscheiden.

Definition 2.7 (Erfüllbarkeitsproblem) *Die booleschen Konstanten* falsch *und* wahr *werden durch* 0 *und* 1 *repräsentiert. Seien* $x_1, x_2, \ldots, x_m$ *boolesche Variablen, d.h.,* $x_i \in \{0,1\}$ *für jedes* i*. Variablen und ihre Negationen heißen* Literale*. Eine boolesche Formel* φ *ist genau dann* erfüllbar*, wenn es eine Belegung der Variablen in* φ *gibt, die die Formel wahr macht. Eine boolesche Formel* φ *ist genau dann in* konjunktiver Normalform *(kurz* KNF*), wenn* φ *die Form* $\varphi(x_1, x_2, \ldots, x_m) = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{k_i} \ell_{i,j} \right)$ *hat, wobei die* $\ell_{i,j}$ *Literale über* $\{x_1, x_2, \ldots, x_m\}$ *sind. Die Disjunktionen* $\bigvee_{j=1}^{k_i} \ell_{i,j}$ *von Literalen heißen die* Klauseln *von* φ*. Eine boolesche Formel* φ *ist genau dann in* k-KNF*, wenn* φ *in KNF ist und jede Klausel von* φ *genau* k *Literale hat. Definiere die folgenden beiden Probleme:*

$$\begin{aligned} \texttt{SAT} &= \{\varphi \mid \varphi \textit{ ist eine erfüllbare boolesche Formel in KNF}\}; \\ \texttt{3-SAT} &= \{\varphi \mid \varphi \textit{ ist eine erfüllbare boolesche Formel in 3-KNF}\}. \end{aligned}$$

Beispiel 2.8 (Boolesche Ausdrücke) *Die folgenden beiden Formeln sind erfüllbare boolesche Ausdrücke (siehe auch Aufgabe 2.3.1):*

$$\begin{aligned} \varphi(w,x,y,z) &= (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (w \vee \neg y \vee z) \wedge (\neg w \vee \neg x \vee z); \\ \psi(w,x,y,z) &= (\neg w \vee x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z) \wedge (\neg w \vee y \vee z) \wedge (w \vee \neg x \vee \neg z). \end{aligned}$$

Dabei ist φ *eine Formel in 3-KNF, und somit ist* φ *in* 3-SAT*. Dagegen ist* ψ *nicht in 3-KNF, weil die erste Klausel vier Literale enthält. Daher ist* ψ *zwar in* SAT*, aber nicht in* 3-SAT*.*

Satz 2.9 ist das oben erwähnte Resultat von Cook, das mit SAT ein erstes NP-vollständiges Problem lieferte. Die Beweisidee besteht darin, die Berechnung einer beliebigen NPTM M bei Eingabe x in eine boolesche Formel $\varphi_{M,x}$ so zu codieren, dass $\varphi_{M,x}$ genau dann erfüllbar ist, wenn M die Eingabe x akzeptiert. Für viele Reduktionen, die vom Erfüllbarkeitsproblem ausgehen, ist es zweckmäßig, wenn die gegebene Formel in der strikten 3-KNF vorliegt. Dies ist möglich, weil SAT auf 3-SAT reduziert werden kann und 3-SAT somit ebenfalls NP-vollständig ist, siehe Aufgabe 2.3.3.

Satz 2.9 (Cook) *Die Probleme* SAT *und* 3-SAT *sind* NP*-vollständig.*

Es sind bisher mehrere tausend Probleme gefunden worden, die NP-vollständig sind. Eine Sammlung von Hunderten solcher Probleme findet sich im Buch von Garey und Johnson [15]. Zur Illustration wählen wir aus diesen vielen Problemen das *dreidimensionale Matching-Problem* aus und zeigen seine NP-Vollständigkeit durch eine Reduktion vom Problem 3-SAT. Beim Matching-Problem will man zueinander passende Paare oder Tripel bilden. Ein *zweidimensionales* (oder *bipartites*) *Matching* ist eine Menge zueinander passender Paare, ein *dreidimensionales* (oder *tripartites*) *Matching* ist eine Menge zueinander passender Tripel. Bipartite Matchings lassen sich gut anhand von (ungerichteten) Graphen veranschaulichen, siehe Definition 1.14 in Abschnitt 1.2.4.

Definition 2.10 (Zweidimensionales Matching-Problem) *Ein Graph G mit $2n$ Knoten heißt* bipartit, *falls seine Knotenmenge in zwei disjunkte Teilmengen V_1 und V_2 der Größe n zerlegt werden kann, die beide* unabhängige Mengen *sind, d.h., weder die Knoten in V_1 noch die Knoten in V_2 sind miteinander durch Kanten verbunden; nur zwischen den Knoten von V_1 und V_2 dürfen Kanten auftreten. Ein* (perfektes) bipartites Matching von G *ist eine Teilmenge $M \subseteq E(G)$ von n Kanten, so dass für je zwei verschiedene Kanten $\{v, w\}$ und $\{x, y\}$ in M gilt, dass $v \neq x$ und $w \neq y$. Das* bipartite Matching-Problem *fragt, ob in einem gegebenen bipartiten Graphen ein bipartites Matching existiert.*

Beispiel 2.11 (Zweidimensionales Matching-Problem) *Stellen wir uns n heiratswillige Damen und n heiratswillige Herren vor, die die Knoten eines bipartiten Graphen G bilden. Die Knotenmenge $V(G)$ wird also zerlegt in $V_{\text{Bräutigam}} = \{f_1, f_2, \ldots, f_n\}$ und $V_{\text{Braut}} = \{m_1, m_2, \ldots, m_n\}$, so dass $V(G) = V_{\text{Braut}} \cup V_{\text{Bräutigam}}$ und $V_{\text{Braut}} \cap V_{\text{Bräutigam}} = \emptyset$. Knoten in $V_{\text{Bräutigam}}$ können mit Knoten in V_{Braut} durch Kanten verbunden sein, aber es gibt keine Kanten zwischen Knoten in $V_{\text{Bräutigam}}$ oder zwischen Knoten in V_{Braut}. Ein bipartites Matching liegt vor, wenn es gelingt, n Hochzeiten zwischen den n Bräuten und den n Bräutigamen so zu arrangieren, dass (in den Worten von Garey und Johnson [15]) "Polygamie vermieden wird und alle eine akzeptable Gattin bzw. einen akzeptablen Gatten erhalten". Wegen dieser Interpretation wird das bipartite Matching-Problem auch das* Heiratsproblem *genannt. Abbildung 2.2 (links) zeigt eine Lösung des Heiratsproblems, wobei die fett gedruckten Kanten die vier frisch getrauten Ehepaare darstellen. Es ist bekannt, dass das Heiratsproblem effizient gelöst werden kann. Im wahren Leben findet man dieses Resultat oft bestätigt:* Heiraten ist leicht!

Nun verallgemeinern wir bipartite Graphen und Matchings auf drei Dimensionen.

Definition 2.12 (Dreidimensionales Matching-Problem) *Seien U, V und W drei paarweise disjunkte Mengen der Größe n. Sei $R \subseteq U \times V \times W$ eine ternäre Relation, d.h., R ist eine Menge von Tripeln (u, v, w) mit $u \in U$, $v \in V$ und $w \in W$. Ein* tripartites Matching von R *ist eine Teilmenge $M \subseteq R$ der Größe n, so dass für je zwei verschiedene Tripel (u, v, w) und $(\hat{u}, \hat{v}, \hat{w})$ in M gilt, dass $u \neq \hat{u}$, $v \neq \hat{v}$ und $w \neq \hat{w}$. Das heißt, keine zwei Elemente eines tripartiten Matchings stimmen in irgendeiner Koordinate überein. Definiere das* dreidimensionale Matching-Problem *wie folgt:*

$$\texttt{3-DM} \;=\; \left\{ (R, U, V, W) \;\middle|\; \begin{array}{l} U, V \text{ und } W \text{ sind paarweise disjunkte, nichtleere Mengen} \\ \text{gleicher Größe und } R \subseteq U \times V \times W \text{ ist eine ternäre} \\ \text{Relation, die ein tripartites Matching der Größe } |U| \text{ enthält} \end{array} \right\}.$$

Beispiel 2.13 (Dreidimensionales Matching-Problem) *Neun Monate sind vergangen. Eines Morgens sind unsere n glücklich verheirateten Paare auf dem Weg ins Stadtkrankenhaus. Einige Stunden später werden n Babies geboren, die sofort mit Schreien anfangen und die Komplexität im Leben ihrer Eltern beträchtlich erhöhen. Zum Beispiel dadurch, dass sie ihre Namensschilder vertauschen, auf denen steht, zu welchem Elternpaar sie gehören. Das verursacht ein großes Durcheinander im Kreißsaal. Schlimmer noch ist, dass jeder der frischen Väter – vielleicht von diesem aufregenden Moment verwirrt und von der Schönheit der anderen Frauen verführt – behauptet, er habe nie zuvor diese junge Dame gesehen, die starrsinnig darauf beharrt, gerade sein Kind zur Welt gebracht zu haben. Stattdessen behauptet er treulos, mit der* anderen *jungen Dame verheiratet zu sein, die gerade links neben der ersteren liegt. Das Chaos ist perfekt! Die Oberschwester im Kreißsaal steht einem schwierigen Problem gegenüber: Welches Baby gehört zu welchem Elternpaar? Anders gesagt: Um die n glücklichen, harmonischen und paarweise disjunkten Familien wieder herzustellen, muss sie ein dreidimensionales Matching zwischen den n Vätern, n Müttern und n Babies finden. Kein Wunder, dass das Problem* 3-DM NP*-vollständig ist, im Gegensatz zur effizienten Lösbarkeit des bipartiten Matching-Problems. Schließlich muss die Oberschwester, will sie das dreidimensionale Matching-Problem lösen, $3n$ Blutproben nehmen und raffinierte*

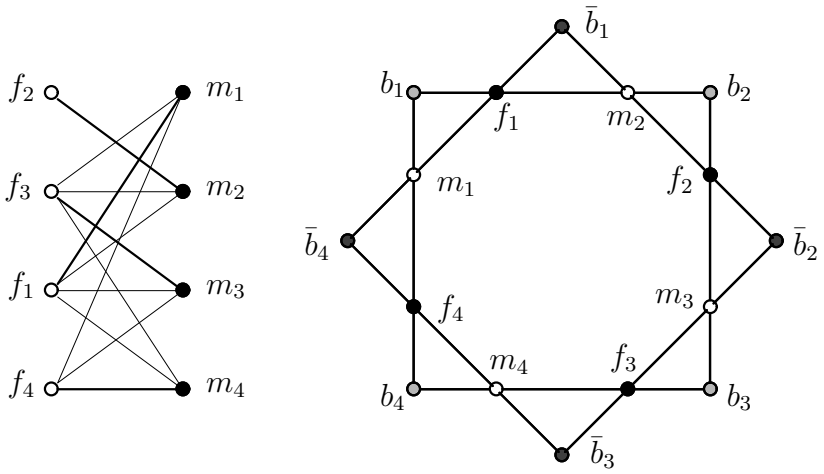


Abbildung 2.2: *Links*: Lösung des Heiratsproblems. *Rechts*: Wahrheitswertkomponente der Relation R.

DNA-Tests durchführen, deren Beschreibung den Rahmen dieses Buches sprengen würde. Und wieder entspricht die NP-*Vollständigkeit von* 3-DM *der Erfahrung im wirklichen Leben:* Wenn Kinder kommen, kann es eine sehr schwere Aufgabe sein, eine glückliche und harmonische Familie zu bleiben, disjunkt zu jeder anderen Familie!

Satz 2.14 3-DM *ist* NP-*vollständig.*

Beweis. Man kann sich leicht überlegen, dass 3-DM in NP ist, siehe Aufgabe 2.3.4. Die Intuition hinter dem Beweis der NP-Härte von 3-DM versteht man am besten, indem man sich zunächst ansieht, wie die Oberschwester im Kreißsaal vorgeht, um das tripartite Matching-Problem zu lösen. Zuerst versieht sie alle im Saal mit einem Namensschild, wobei sie sicherstellt, dass dieses nicht wieder entfernt werden kann. Angenommen, die Mütter erhalten die Namen $m_1, m_2, \ldots, m_n$, die Väter $f_1, f_2, \ldots, f_n$ und die Babies $b_1, b_2, \ldots, b_n$. Dann erzeugt die Oberschwester einen zweiten Satz von n Babies, $\{\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_n\}$, wobei jedes $\bar{b}_i$ ein identischer Klon[2] von b_i ist, d.h., b_i und $\bar{b}_i$ sehen identisch aus und ihre DNA trägt dieselbe Erbinformation. Anschließend stellt sie alle $4n$ Personen in zwei Kreisen auf. Die $2n$ Eltern formen einen inneren Kreis, in welchem sich Väter und Mütter abwechseln. Im äußeren Kreis stellen sich die n Babies und ihre n Klone auf, ebenfalls alternierend. Benachbarte Personen in diesen beiden Kreisen sind miteinander so verbunden, wie Abbildung 2.2 (rechts) dies für $n = 4$ zeigt: Jeder Vater ist mit zwei Müttern und mit zwei Babies verbunden.

Für jedes i modulo $n = 4$ gilt:[3] Vater f_i behauptet, mit Mutter m_{i-1} verheiratet zu sein und gemeinsam mit dieser das $(i-1)$-te Kind zu haben, während Mutter m_i darauf besteht, dass *sie* die Frau von f_i ist und ihr gemeinsames Kind das i-te Baby ist. Diese beiden widersprüchlichen Aussagen sind in Abbildung 2.2 (rechts) durch zwei Dreiecke dargestellt, deren Ecken f_i, m_{i-1} und $\bar{b}_{i-1}$ bzw. m_i, f_i und b_i sind. Jedes der $2n$ Dreiecke stellt eine potenzielle Familie dar. Die Oberschwester muss nun feststellen, welche Dreiecke die *ursprünglichen* n Familien repräsentieren und welche nicht. Die einzige Möglichkeit, n disjunkte Familien zu erhalten, ist, entweder jedes Dreieck mit einem Baby b_i oder aber jedes Dreieck mit einem Klonbaby $\bar{b}_i$ zu wählen. Indem sie $3n$ Blutproben nimmt und ihre oben erwähnten DNA-Tests auswertet, kann die Oberschwester die richtige Wahl treffen und jeden Vater seiner richtigen Frau und seinem richtigen Kind zuweisen. So stellt sie die n ursprünglichen Familien wieder her. Die

[2] Die technischen Details des Klonens von Babies sowie die Diskussion von damit verbundenen ethischen Fragen würden ebenfalls den Rahmen dieses Buches sprengen und werden daher schweigend übergangen.

[3] Die Arithmetik modulo n ist in Problem 1.1 am Ende von Kapitel 1 erklärt.

übrigen n Babies (und das ist die traurige Seite der Methode der Oberschwester – und des Babyklonens im Allgemeinen) werden in Waisenhäuser geschickt oder adoptiert.

Komplexitätstheoretiker wissen nicht viel über DNA-Tests oder Klonen. Glücklicherweise jedoch sind sie gut mit dem Erfüllbarkeitsproblem vertraut. Um die NP-Härte des Problems 3-DM zu zeigen, definieren wir nun eine Reduktion von 3-SAT auf 3-DM. Gegeben sei eine boolesche Formel φ in 3-KNF, d.h., $\varphi(x_1, x_2, \ldots, x_\ell) = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, wobei die Klauseln C_j von φ genau drei Literale haben. Zu konstruieren ist eine Instanz (R, U, V, W) von 3-DM, wobei $R \subseteq U \times V \times W$ eine ternäre Relation über den paarweise disjunkten, nichtleeren Mengen U, V und W gleicher Größe ist, so dass gilt:

$$\varphi \text{ ist erfüllbar} \iff R \text{ enthält ein tripartites Matching } M \text{ der Größe } |U|. \tag{2.1}$$

R besteht aus verschiedenen Arten von Tripeln, hinter denen sich jeweils eine andere Absicht verbirgt. Alle Tripel derselben Art werden zu einer Komponente zusammengefasst. Die erste Komponente besteht aus solchen Tripeln in R, deren Form eine bestimmte Belegung der Variablen der Formel φ erzwingt, so dass diese Belegung konsistent für sämtliche Klauseln von φ ist. Das heißt, wenn dieselbe Variable in verschiedenen Klauseln vorkommt, so sollen alle Vorkommen mit demselben Wahrheitswert belegt werden. Deshalb nennen wir diese Komponente die "*Wahrheitswertkomponente*" von R.

Erzeuge für jede Variable x_i in φ genau $2n$ Elemente $b_1^i, b_2^i, \ldots, b_n^i$ und $\bar{b}_1^i, \bar{b}_2^i, \ldots, \bar{b}_n^i$ in U, wobei n die Anzahl der Klauseln von φ ist. Dabei repräsentiert b_j^i das Vorkommen von x_i und $\bar{b}_j^i$ das Vorkommen von $\neg x_i$ in der j-ten Klausel C_j von φ. Da nicht jedes Literal in jeder Klausel vorkommt, entsprechen manche b_j^i oder $\bar{b}_j^i$ keinem Vorkommen eines Literals in φ. Außerdem werden für jede Variable x_i in φ weitere n Elemente $m_1^i, m_2^i, \ldots, m_n^i$ in V und n Elemente $f_1^i, f_2^i, \ldots, f_n^i$ in W erzeugt, welche den inneren Kreis in Abbildung 2.2 (rechts) bilden, wobei $n = 4$ und die oberen Indizes im Bild weggelassen sind. Verbinde nun die Elemente m_j^i, f_j^i und b_j^i miteinander sowie die Elemente f_j^i, m_{j-1}^i und $\bar{b}_{j-1}^i$, wie in Abbildung 2.2 (rechts) dargestellt. Die Dreiecke in der so konstruierten Komponente entsprechen den Tripeln in R. Die m_j^i und f_j^i aus dem inneren Kreis kommen nur in der Komponente vor, die der Variablen x_i entspricht, während die b_j^i und $\bar{b}_j^i$ aus dem äußeren Kreis auch in anderen Komponenten vorkommen können. Formal hat die Wahrheitswertkomponente X die Gestalt $X = \bigcup_{i=1}^{\ell} X_i$, wobei $X_i = F_i \cup T_i$ für jede Variable x_i in φ durch die folgenden zwei Mengen von Tripeln definiert ist:

$$\begin{aligned} F_i &= \{(b_j^i, m_j^i, f_j^i) \mid 1 \leq j \leq n\}; \\ T_i &= \{(\bar{b}_j^i, m_j^i, f_{j+1}^i) \mid 1 \leq j < n\} \cup \{(\bar{b}_n^i, m_n^i, f_1^i)\}. \end{aligned}$$

Da keines der Elemente m_j^i und f_j^i aus dem inneren Kreis in irgendeiner anderen Komponente als in X_i vorkommt, muss jedes Matching M von R genau n Tripel aus X_i enthalten, entweder alle Tripel aus F_i oder alle Tripel aus T_i. Diese Wahl eines Matchings zwischen F_i und T_i erzwingt eine Belegung der Variablen x_i mit dem Wahrheitswert entweder *falsch* oder *wahr*. Da alle Vorkommen von x_i in φ in X_i enthalten sind, ist diese Wahl der Wahrheitswerte für die ganze Formel konsistent. Folglich spezifiziert ein jedes Matching M von R eine Belegung der Formel φ, so dass jede Variable x_i unter der Belegung genau dann wahr gesetzt wird, wenn $M \cap X_i = T_i$.

Nun fügen wir zu R eine Menge $Y = \bigcup_{j=1}^{n} Y_j$ von Tripeln hinzu, so dass jedes Y_j die Erfüllbarkeit der Klausel C_j in φ überprüft. Deshalb heißt die Komponente Y die "*Erfüllbarkeitskomponente*" von R. Für jede Klausel C_j erzeugen wir dazu zwei Elemente, $v_j \in V$ und $w_j \in W$, die nur in Y_j vorkommen. Außerdem enthält Y_j drei weitere Elemente aus der Menge $\bigcup_{i=1}^{\ell} \Big(\{b_j^i\} \cup \{\bar{b}_j^i\}\Big)$, die den drei Literalen in C_j entsprechen und die auch in anderen Komponenten von R vorkommen dürfen. Formal ist Y_j für jede Klausel C_j von φ durch die folgende Menge von Tripeln definiert:

$$Y_j = \{(b_j^i, v_j, w_j) \mid x_i \text{ tritt in } C_j \text{ auf}\} \cup \{(\bar{b}_j^i, v_j, w_j) \mid \neg x_i \text{ tritt in } C_j \text{ auf}\}.$$

Da keines der Elemente v_j und w_j, $1 \leq j \leq n$, in irgendeinem anderen Tripel von R als in Y_j vorkommt, muss jedes Matching M von R genau ein Tripel aus Y_j enthalten, entweder (b_j^i, v_j, w_j) oder $(\bar{b}_j^i, v_j, w_j)$. Jedoch enthält M ein Tripel aus Y_j mit entweder b_j^i (falls x_i in C_j vorkommt) oder $\bar{b}_j^i$ (falls $\neg x_i$ in C_j vorkommt) genau dann, wenn dieses Element nicht in den Tripeln aus $M \cap X_i$ vorkommt. Dies ist aber genau dann der Fall, wenn die Belegung, die durch M mit der Wahrheitswertkomponente spezifiziert wird, die Klausel C_j erfüllt.

Bisher enthält U genau $2n\ell$ Elemente, aber sowohl V als auch W haben lediglich $n\ell + n$ Elemente. Fügen wir $n(\ell - 1)$ weitere Elemente sowohl zu V als auch zu W hinzu, so haben diese drei Mengen dieselbe Größe. Insbesondere fügen wir die Elemente $v_{n+1}, v_{n+2}, \ldots, v_{n\ell}$ zu V und die Elemente $w_{n+1}, w_{n+2}, \ldots, w_{n\ell}$ zu W hinzu. Außerdem wird R um die folgende Menge von Tripeln erweitert:

$$\begin{aligned} Z &= \{(b_j^i, v_k, w_k) \mid 1 \leq i \leq \ell \text{ und } 1 \leq j \leq n \text{ und } n+1 \leq k \leq n\ell\} \cup \\ &\quad \{(\bar{b}_j^i, v_k, w_k) \mid 1 \leq i \leq \ell \text{ und } 1 \leq j \leq n \text{ und } n+1 \leq k \leq n\ell\}. \end{aligned}$$

Der Witz ist, dass, wann immer ein Matching von $R - Z$ existiert, das sämtliche durch die Wahrheitswert- und die Erfüllbarkeitskomponente von R erzwungenen Bedingungen erfüllt, dieses Matching genau $n(\ell - 1)$ Elemente aus U frei lässt, die nun mit einem eindeutig bestimmten Paar (v_k, w_k) aus Z "gematcht" werden können. Diese Erweiterung des Matchings von $R - Z$ ergibt ein Matching von R. Formal sind die Mengen U, V und W folgendermaßen definiert:

$$\begin{aligned} U &= \{b_j^i \mid 1 \leq i \leq \ell \text{ und } 1 \leq j \leq n\} \cup \{\bar{b}_j^i \mid 1 \leq i \leq \ell \text{ und } 1 \leq j \leq n\}; \\ V &= \{m_j^i \mid 1 \leq i \leq \ell \text{ und } 1 \leq j \leq n\} \cup \{v_k \mid 1 \leq k \leq n\ell\}; \\ W &= \{f_j^i \mid 1 \leq i \leq \ell \text{ und } 1 \leq j \leq n\} \cup \{w_k \mid 1 \leq k \leq n\ell\}. \end{aligned}$$

Die Relation $R \subseteq U \times V \times W$ ist definiert durch $R = X \cup Y \cup Z$. Da R genau $2n\ell + 3n + 2n^2\ell(\ell - 1)$ Tripel enthält, also polynomiell viele in der Größe der gegebenen Formel φ, und da die Struktur von R leicht aus der Struktur von φ bestimmt werden kann, ist die Reduktion in Polynomialzeit berechenbar. Die Äquivalenz (2.1) folgt aus den Bemerkungen, die während der Konstruktion von R gemacht wurden. Ein formaler Beweis von (2.1) wird dem Leser als Aufgabe 2.3.5 überlassen. ∎

Übungsaufgaben

Aufgabe 2.3.1 Gib je eine erfüllende Belegung für die booleschen Formeln φ und ψ aus Beispiel 2.8 an.

Aufgabe 2.3.2 Zeige die Transitivität der $\leq_{\mathrm{m}}^{\mathrm{p}}$-Reduzierbarkeit: $(A \leq_{\mathrm{m}}^{\mathrm{p}} B \wedge B \leq_{\mathrm{m}}^{\mathrm{p}} C) \Longrightarrow A \leq_{\mathrm{m}}^{\mathrm{p}} C$.

Aufgabe 2.3.3 Gib eine Reduktion $\texttt{SAT} \leq_{\mathrm{m}}^{\mathrm{p}} \texttt{3-SAT}$ an. Forme dazu alle Klauseln einer gegebenen booleschen Formel in KNF, die nur ein oder zwei oder aber mehr als drei Literale enthalten, so in Klauseln mit genau drei Literalen um, dass sich dabei an der Erfüllbarkeit der Formel nichts ändert.

Aufgabe 2.3.4 Zeige, dass die Probleme SAT und 3-DM in NP liegen.

Aufgabe 2.3.5 Beweise die Äquivalenz (2.1) im Beweis von Satz 2.14.

2.4 Das Erfüllbarkeitsproblem der Aussagenlogik

2.4.1 Deterministische Zeitkomplexität von 3-SAT

Das Erfüllbarkeitsproblem SAT sowie seine Restriktion 3-SAT sind nach Satz 2.9 NP-vollständig. Wäre SAT in P, so würde also entgegen der allgemeinen Vermutung sofort P $=$ NP folgen. Daher gilt es als sehr unwahrscheinlich, dass es effiziente deterministische Algorithmen für SAT oder 3-SAT gibt. Aber welche Laufzeit haben denn die besten deterministischen Algorithmen für 3-SAT? Da offenbar die genaue Struktur der Formel einen Einfluss auf die Laufzeit haben kann, konzentrieren wir uns in diesem Abschnitt auf das Problem 3-SAT, bei dem jede Klausel aus genau drei Literalen besteht. Die hier vorgestellten Resultate lassen sich unmittelbar auf k-SAT übertragen, die Einschränkung von SAT mit genau k Literalen pro Klausel. Der "naive" deterministische Algorithmus für 3-SAT arbeitet so: Für eine gegebene boolesche Formel φ mit n Variablen werden nacheinander sämtliche möglichen Belegungen durchprobiert, wobei die Formel mit der jeweiligen Belegung ausgewertet wird. Macht eine dieser Belegungen φ wahr, so akzeptiert der Algorithmus. Sind andernfalls alle 2^n Belegungen erfolglos getestet worden, so lehnt der Algorithmus ab. Offensichtlich arbeitet dieser Algorithmus in der Zeit $O(2^n)$. Geht es besser?

Ja. Es geht besser. Doch bevor gezeigt wird *wie*, wollen wir zunächst die Frage stellen: *Warum?* Was hat man davon, die obere Zeitschranke für 3-SAT unter $O(2^n)$ zu drücken, etwa auf $O(c^n)$ für eine Konstante c mit $1 < c < 2$, was immer noch eine Exponentialzeitschranke ist? Man erreicht dadurch, dass sich der Schwellwert n_0 nach hinten verschiebt, bei dem die Exponentialzeit "zuschlägt" und die absolute Laufzeit des Algorithmus für Eingaben der Größe $n \geq n_0$ unerträglich groß wird. Kann man etwa die $O(2^n)$-Schranke des "naiven" deterministischen Algorithmus für 3-SAT so weit unterbieten, dass man mit einem $O(c^n)$-Algorithmus Eingaben doppelter Größe in derselben Zeit bearbeiten kann, so hat man in der Praxis viel gewonnen. Dies ist gerade für $c = \sqrt{2} \approx 1.4142$ der Fall, denn dann arbeitet der Algorithmus in der Zeit $O(\sqrt{2}^{2n}) = O(2^n)$, siehe auch Tabelle 2.5 auf Seite 69.

Nun wird ein deterministischer Algorithmus für 3-SAT vorgestellt, der auf dem algorithmischen Prinzip "*Backtracking*" beruht. Diese Algorithmenentwurfstechnik ist für Probleme geeignet, deren Lösungen sich aus n Komponenten zusammensetzen, für die es mehrere Wahlmöglichkeiten gibt. Beispielsweise besteht eine Lösung von 3-SAT aus den n Wahrheitswerten einer erfüllenden Belegung, und für jeden solchen Wahrheitswert gibt es zwei Wahlmöglichkeiten: *wahr* oder *falsch* bzw. 1 oder 0. Die Idee besteht nun darin, ausgehend von der leeren Lösung (der partiellen Belegung, die keine Variablen belegt) Schritt für Schritt durch rekursive Aufrufe der *Backtracking*-Prozedur eine immer größere partielle Lösung des Problems zu konstruieren, bis schließlich eine Gesamtlösung gefunden ist, sofern eine solche existiert. Im entstehenden Rekursionsbaum[4] ist die Wurzel mit der leeren Lösung markiert, während die vollständigen Lösungen des Problems auf der Blattebene vorliegen. Stellt man während der Ausführung des Algorithmus fest, dass der aktuelle Zweig des Rekursionsbaums "tot" ist, also dass sich die bisher konstruierte Teillösung auf keinen Fall zu einer Gesamtlösung des Problems fortsetzen lässt, so kann man den Teilbaum unter dem aktuell erreichten Knoten getrost abschneiden und in die aufrufende Prozedur zurücksetzen, um eine andere Fortsetzung der bisher konstruierten Teillösung zu versuchen. Diesem Zurücksetzen verdankt dieses algorithmische Prinzip den Namen "*Backtracking*", und durch das Abschneiden von "toten" Teilen des Rekursionsbaumes kann womöglich Zeit gespart werden.

Abbildung 2.3 zeigt den Algorithmus BACKTRACKING-SAT, der bei Eingabe einer booleschen Formel φ und einer partiellen Belegung β einiger Variablen von φ einen booleschen Wert liefert: 1, falls sich die partielle Belegung β zu einer erfüllenden Belegung aller Variablen von φ erweitern lässt, und 0 sonst.

[4] Um Verwechslungen auszuschließen, sei hier betont, dass ein Rekursionsbaum etwas anderes als der Berechnungsbaum einer NTM ist, d.h., der Algorithmus BACKTRACKING-SAT geht ganz deterministisch gemäß einer Tiefensuche im Rekursionsbaum vor. Die inneren Knoten eines solchen Baums repräsentieren die rekursiven Aufrufe des Algorithmus, seine Wurzel den ersten Aufruf, und an den Blättern terminiert der Algorithmus ohne weiteren Aufruf. Ein Knoten $\hat{k}$ im Rekursionsbaum ist genau dann Sohn eines Knoten k, wenn der Aufruf $\hat{k}$ innerhalb der durch k ausgelösten Berechnung des Algorithmus erfolgt.

```
BACKTRACKING-SAT(φ, β) {
  if (β belegt alle Variablen von φ) return φ(β);
    else if (β macht eine der Klauseln von φ falsch) return 0;     // "toter Zweig"
          else if (BACKTRACKING-SAT(φ, β0)) return 1;
                else return BACKTRACKING-SAT(φ, β1));
}
```

Abbildung 2.3: Backtracking-Algorithmus für 3-SAT.

Partielle Belegungen werden hierbei als Wörter der Länge $\leq n$ über dem Alphabet $\{0, 1\}$ aufgefasst. Der erste Aufruf des Algorithmus erfolgt durch BACKTRACKING-SAT(φ, λ), wobei λ die leere Belegung ist. Stellt sich heraus, dass die bisher konstruierte partielle Belegung β eine der Klauseln von φ falsch macht, so kann sie nicht mehr zu einer erfüllenden Belegung von φ erweitert werden, und der Teilbaum unter dem entsprechenden Knoten im Rekursionsbaum wird abgeschnitten, siehe auch Aufgabe 2.4.1.

Um die Laufzeit von BACKTRACKING-SAT nach oben abzuschätzen, betrachten wir eine beliebige feste Klausel C_j der gegebenen Formel φ. Jede erfüllende Belegung β von φ muss insbesondere die drei in C_j vorkommenden Variablen mit Wahrheitswerten belegen. Von den $2^3 = 8$ vielen Möglichkeiten, diese mit 0 oder 1 zu belegen, scheidet jedoch mit Sicherheit eine aus, nämlich die Belegung, die C_j falsch macht. Der entsprechende Knoten im Rekursionsbaum von BACKTRACKING-SAT(φ, β) führt also zu einem "toten" Teilbaum, der getrost abgeschnitten werden kann. Es kann je nach Struktur von φ noch weitere "tote" Teilbäume geben, die nicht mehr berücksichtigt werden müssen. Daraus ergibt sich für BACKTRACKING-SAT eine obere Schranke von $O\left(\left(2^3-1\right)^{n/3}\right) = O(\sqrt[3]{7}^n) \approx O(1.9129^n)$ im schlechtesten Fall, was die $O(2^n)$-Schranke des "naiven" Algorithmus für 3-SAT immerhin leicht verbessert.

Die deterministische Zeitkomplexität für 3-SAT kann noch weiter nach unten gedrückt werden. Beispielsweise hat der Teile-und-Herrsche-Algorithmus von Monien und Speckenmeyer [42] eine obere Schranke von $O(1.618^n)$. Basierend auf einer lokalen Suche erzielten Dantsin et al. [11] mit $O(1.481^n)$ die bisher beste obere Schranke für einen deterministischen 3-SAT-Algorithmus und halten derzeit den Weltrekord. Es gibt auch andere, nicht deterministische Ansätze. Einer davon wird nun vorgestellt, ein "Random-Walk"-Algorithmus, der auf Schöning [56, 59] zurückgeht.

2.4.2 Probabilistische Zeitkomplexität von 3-SAT

Ein *random walk* ist eine (zufällige) Irrfahrt auf einer gegebenen Struktur, z.B. im euklidischen Raum, auf einem unendlichen Gitter oder auf einem Graphen. Hier sind wir an Irrfahrten auf Graphen interessiert, nämlich auf dem Graphen, der einen bestimmten stochastischen Automaten repräsentiert. Ein stochastischer Automat ist ein besonderer *endlicher Automat*.

Ein endlicher Automat kann durch seinen Zustandsgraphen veranschaulicht werden. Die Zustände des endlichen Automaten werden dabei durch Knoten und die Übergänge zwischen den Zuständen durch gerichtete, mit Symbolen aus einem Alphabet Σ beschriftete Kanten dargestellt. Ein Knoten ist als *Startzustand* ausgezeichnet. Bei diesem beginnt die Berechnung des Automaten, und sie endet, sobald die gesamte Eingabe verarbeitet ist, wobei in jedem Rechentakt genau ein Eingabesymbol gelesen wird. Manche Knoten des Zustandsgraphen sind als *Endzustände* gekennzeichnet. Wird ein solcher Endzustand am Ende der Berechnung erreicht, so hält der Automat akzeptierend an.

Man kann mit endlichen Automaten Wörter erkennen. Ein Wort $w = w_1 w_2 \cdots w_n$ aus Σ^* wird genau dann akzeptiert, wenn man ausgehend vom Startzustand die einzelnen Symbole w_i von w der Reihe nach

liest, wobei man jeweils den Zustandsübergang entlang der mit w_i beschrifteten Kante ausführt, und schließlich einen Endzustand erreicht. Die Sprache eines endlichen Automaten besteht aus genau den Wörtern, die in dieser Weise akzeptiert werden.

Die Kanten eines *stochastischen Automaten* $\mathcal{S}$ werden zusätzlich noch mit Zahlen beschriftet. Die Zahl $p_{u,v}$ mit $0 \leq p_{u,v} \leq 1$ neben einer Kante von u nach v im Zustandsgraphen von $\mathcal{S}$ gibt die Wahrscheinlichkeit an, mit der $\mathcal{S}$ vom Zustand u in den Zustand v übergeht. Den Prozess der (zufälligen) Zustandsübergänge eines stochastischen Automaten nennt man in der Stochastik auch eine *Markow-Kette*, und Endzustände heißen dort *absorbierende* Zustände. Im Falle eines stochastischen Automaten $\mathcal{S}$ erfolgt die Akzeptierung eines Wortes w (und somit die Definition der von $\mathcal{S}$ akzeptierten Sprache) natürlich nur mit einer gewissen Wahrscheinlichkeit gemäß der Beschriftung der beim Abarbeiten von w durchlaufenen Kanten.

```
RANDOM-SAT(φ) {
  for (i = 1, 2, ..., ⌈(4/3)^n⌉) {          // n ist die Anzahl der Variablen in φ
    Wähle zufällig eine Belegung β ∈ {0,1}^n unter Gleichverteilung;
    for (j = 1, 2, ..., n) {
      if (φ(β) = 1) return die erfüllende Belegung β von φ und halte;
        else {
          Wähle eine Klausel C = (x ∨ y ∨ z) mit C(β) = 0;
          Wähle zufällig ein Literal ℓ ∈ {x, y, z} unter Gleichverteilung;
          Bestimme das Bit β_ℓ ∈ {0,1} in β, das ℓ belegt;
          Ändere β_ℓ zu 1 − β_ℓ in β;
        }
    }
  }
  return "φ ist nicht erfüllbar";
}
```

Abbildung 2.4: Der Algorithmus RANDOM-SAT.

Hier sind wir jedoch nicht an der Spracherkennung durch einen stochastischen Automaten interessiert, sondern wir wollen ihn für eine Irrfahrt verwenden, die der probabilistische Algorithmus RANDOM-SAT ausführt, der in Abbildung 2.4 dargestellt ist. RANDOM-SAT versucht, für eine gegebene boolesche Formel φ mit n Variablen eine erfüllende Belegung zu finden, sofern eine solche existiert.

Bei Eingabe von φ rät RANDOM-SAT zunächst eine zufällige Anfangsbelegung β, wobei jedes Bit unabhängig und unter Gleichverteilung gewählt wird, d.h., jedes Bit von β nimmt den Wert 0 bzw. 1 mit Wahrscheinlichkeit $1/2$ an. Wieder werden Belegungen als Wörter der Länge n über $\{0,1\}$ aufgefasst. Angenommen, φ ist erfüllbar. Sei $\tilde{\beta}$ eine beliebige fest gewählte erfüllende Belegung von φ. Sei X diejenige Zufallsvariable, die den *Hammingabstand von β und $\tilde{\beta}$* ausdrückt, also die Anzahl der Bits, die in β und in $\tilde{\beta}$ nicht übereinstimmen. Offenbar kann X die Werte $j \in \{0,1,\ldots,n\}$ annehmen und ist binomialverteilt mit den Parametern n und $1/2$. Das heißt, die Wahrscheinlichkeit für $X = j$ ist gerade $\binom{n}{j}2^{-n}$.

Der Algorithmus RANDOM-SAT testet nun, ob die anfangs gewählte Belegung β die Formel φ bereits erfüllt, und akzeptiert, falls dies der Fall ist. Andernfalls, wenn also φ nicht durch β erfüllt wird, muss es eine Klausel in φ geben, die β nicht erfüllt. RANDOM-SAT wählt nun eine beliebige solche Klausel aus, wählt unter Gleichverteilung ein Literal in der gewählten Klausel und "flippt" dasjenige Bit in der aktuellen Belegung β, das dieses Literal mit einem Wahrheitswert belegt. Dies wird n-

mal wiederholt. Erfüllt die dann vorliegende aktuelle Belegung die Formel φ noch immer nicht, startet RANDOM-SAT mit einer neuen Anfangsbelegung und wiederholt den gesamten oben beschriebenen Versuch insgesamt t-mal, wobei $t = \lceil (4/3)^n \rceil$.

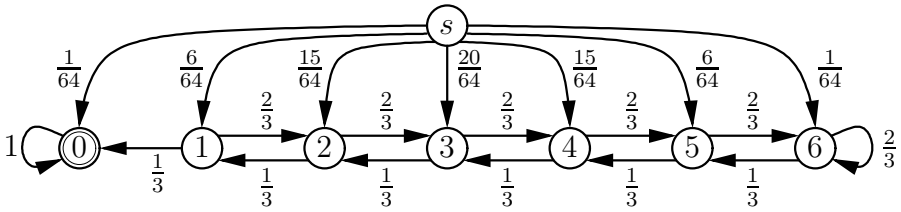


Abbildung 2.5: Zustandsgraph eines stochastischen Automaten für die Irrfahrt von RANDOM-SAT.

Abildung 2.5 zeigt einen stochastischen Automaten $\mathcal{S}$, dessen Kanten nicht mit Symbolen beschriftet sind, sondern nur mit Übergangswahrscheinlichkeiten. Die Berechnung von RANDOM-SAT bei Eingabe φ kann man sich folgendermaßen als eine Irrfahrt auf $\mathcal{S}$ vorstellen. Ausgehend vom Startzustand s, der später nie wieder erreicht wird, geht RANDOM-SAT(φ) zunächst gemäß der Binomialverteilung mit den Parametern n und $1/2$ in einen der Zustände $j \in \{0, 1, \ldots, n\}$ über; dies ist im oberen Teil der Abbildung für eine boolesche Formel φ mit $n = 6$ Variablen dargestellt. Ein solcher Zustand j bedeutet, dass die zufällig gewählte Anfangsbelegung β und die feste erfüllende Belegung $\tilde{\beta}$ den Hammingabstand j haben. Solange $j \neq 0$ ist, ändert RANDOM-SAT(φ) auf der Suche nach einer erfüllenden Belegung in jedem Durchlauf der inneren `for`-Schleife ein Bit β_ℓ zu $1 - \beta_\ell$ in der aktuellen Belegung β. Dem entspricht in der Irrfahrt auf $\mathcal{S}$ ein Schritt nach links in den Zustand $j - 1$ oder aber ein Schritt nach rechts in den Zustand $j + 1$, wobei nur Zustände kleiner oder gleich n erreicht werden können.

Die fest gewählte Belegung $\tilde{\beta}$ erfüllt φ, also macht sie in jeder Klausel von φ wenigstens ein Literal wahr. Fixieren wir in jeder Klausel *genau* eines dieser durch $\tilde{\beta}$ erfüllten Literale, so wird ein Schritt nach links genau dann gemacht, wenn dieses Literal ℓ durch RANDOM-SAT(φ) ausgewählt wurde. Offenbar ist die Wahrscheinlichkeit für einen Schritt nach links (von $j > 0$ nach $j - 1$) gleich $1/3$ und die Wahrscheinlichkeit für einen Schritt nach rechts (von j nach $j + 1$) gleich $2/3$.

Ist irgendwann der Zustand $j = 0$ erreicht, so haben β und $\tilde{\beta}$ den Hammingabstand 0. Somit erfüllt β die Formel φ, und RANDOM-SAT(φ) gibt β aus und hält akzeptierend. Man kann natürlich auch in einem Zustand $j \neq 0$ auf eine (von $\tilde{\beta}$ verschiedene) erfüllende Belegung treffen. Da diese Möglichkeit die Akzeptierungswahrscheinlichkeit nur erhöhen würde, lassen wir sie bei der folgenden Abschätzung der Akzeptierungswahrscheinlichkeit jedoch außer Acht. Wird der Zustand $j = 0$ nicht nach höchstens n Bitänderungen in β erreicht, so war die Anfangsbelegung so schlecht gewählt worden, dass RANDOM-SAT(φ) sie nun wegwirft und mit einer anderen Anfangsbelegung sein Glück neu versucht.

Da die Wahrscheinlichkeit dafür, sich weg vom (glücklichen) Endzustand 0 nach rechts zu bewegen, größer ist als die Wahrscheinlichkeit dafür, nach links Richtung 0 zu laufen, könnte man meinen, dass die Erfolgswahrscheinlichkeit von RANDOM-SAT nicht sehr groß ist. Jedoch darf man die Chance nicht unterschätzen, dass man bereits nach dem nullten Schritt von s aus in der Nähe von 0 landet! Je näher bei 0 man startet, um so größer ist die Wahrscheinlichkeit dafür, dass man im Verlauf der darauf folgenden zufälligen Bewegungen nach rechts oder links auf den Zustand 0 trifft.

Die Analyse der Erfolgswahrscheinlichkeit und der Laufzeit von RANDOM-SAT wird hier nicht in allen Details vorgeführt, sondern nur skizziert. Zur Vereinfachung nehmen wir an, dass n ein ganzzahliges Vielfaches von 3 ist. Sei p_i die Wahrscheinlichkeit dafür, dass RANDOM-SAT in n Schritten den Zustand 0 erreicht, unter der Bedingung, dass RANDOM-SAT im nullten Schritt der Irrfahrt (bei der Wahl der zufälligen Anfangsbelegung β) im Zustand $i \leq n/3$ landet. Landet man beispielsweise anfangs im Zustand $n/3$, so dürfen höchstens $n/3$ Schritte in die "falsche" Richtung nach rechts gemacht werden,

die man mit Schritten in die "richtige" Richtung nach links wieder ausgleichen kann. Andernfalls ließe sich der Zustand 0 nicht in n Schritten erreichen. Allgemein dürfen von Zustand i ausgehend höchstens $(n-i)/2$ Schritte nach rechts gemacht werden, und es ergibt sich für p_i:

$$p_i = \binom{n}{\frac{n-i}{2}} \left(\frac{2}{3}\right)^{\frac{n-i}{2}} \left(\frac{1}{3}\right)^{n-\frac{n-i}{2}}. \tag{2.2}$$

Sei ferner q_i die Wahrscheinlichkeit dafür, dass RANDOM-SAT im nullten Schritt der Irrfahrt im Zustand $i \leq n/3$ landet. Natürlich gilt:

$$q_i = \binom{n}{i} \cdot 2^{-n}. \tag{2.3}$$

Sei schließlich p die Erfolgswahrscheinlichkeit dafür, dass RANDOM-SAT in einem Durchlauf der äußeren `for`-Schleife den Zustand 0 erreicht. Dies ist auch von Zuständen $j > n/3$ aus möglich. Daher gilt:

$$p \geq \sum_{i=0}^{n/3} p_i \cdot q_i.$$

Approximiert man diese Summe mittels der Entropiefunktion sowie die Binomialkoeffizienten aus (2.2) und (2.3) in den einzelnen Summanden mit der Stirling-Formel, so erhält man schließlich $\Omega((3/4)^n)$ als untere Schranke für p.

Zur Fehlerreduktion führt RANDOM-SAT insgesamt t unabhängige Versuche aus, die jeweils mit einer neuen Anfangsbelegung starten und mindestens die oben angegebene Erfolgswahrscheinlichkeit von etwa $(3/4)^n$ haben. Da sich diese Wahrscheinlichkeiten wegen der Unabhängigkeit der Versuche multiplizieren, ist insgesamt die Erfolgswahrscheinlichkeit von RANDOM-SAT – also die Wahrscheinlichkeit dafür, eine erfüllende Belegung von φ auszugeben, falls eine solche existiert – sehr nahe bei 1. Ist übrigens φ nicht erfüllbar, so macht RANDOM-SAT nie einen Fehler, d.h., in diesem Fall ist die Ausgabe stets: "φ ist nicht erfüllbar".

Die Laufzeit des Algorithmus entspricht dem Kehrwert der Erfolgswahrscheinlichkeit $p \approx (3/4)^n$ in einem Durchlauf. Denn die Wahrscheinlichkeit für einen Fehler (dass also bei keinem der t Versuche eine erfüllende Belegung von φ gefunden wird, obwohl φ erfüllbar ist) lässt sich durch $(1-p)^t \leq e^{-t \cdot p}$ abschätzen. Will man eine fest vorgebene Fehlerwahrscheinlichkeit ε nicht überschreiten, genügt es also, t so zu wählen, dass $e^{-t \cdot p} \leq \varepsilon$ bzw. $t \geq \ln(1/\varepsilon)/p$ gilt. Abgesehen von konstanten Faktoren wird dies durch die Wahl von $t = \lceil (4/3)^n \rceil$ erreicht. Die Laufzeit des Algorithmus liegt also in $O\left((4/3)^n\right)$.

Übungsaufgaben

Aufgabe 2.4.1 Starte den Algorithmus BACKTRACKING-SAT aus Abbildung 2.3 für die boolesche Formel $\varphi = (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg u \vee y \vee z) \wedge (u \vee \neg y \vee z)$ und konstruiere Schritt für Schritt eine erfüllende Belegung von φ. Zeichne den entstehenden Rekursionsbaum und bestimme die Teile dieses Baumes, die abgeschnitten werden, weil sie nicht zu einer Lösung führen können.

2.5 Graphisomorphie und Lowness

In diesem Abschnitt benötigen wir die gruppen- und die graphentheoretischen Grundlagen aus Abschnitt 1.2.4. Insbesondere sei an den Begriff der Permutationsgruppe aus Definition 1.12 und an das Graphisomorphieproblem `GI` sowie das Graphautomorphieproblem `GA` aus Definition 1.14 erinnert; siehe auch Beispiel 1.15 in Kapitel 1.

2.5.1 Reduzierbarkeiten und Komplexitätshierarchien

In Abschnitt 2.3 haben wir das effizient lösbare Heiratsproblem sowie die NP-vollständigen Probleme SAT, 3-SAT und 3-DM kennen gelernt. Man kann sich leicht überlegen, dass P $=$ NP genau dann gilt, wenn *jedes* NP-Problem, einschließlich der NP-vollständigen Probleme, in P ist. Insbesondere kann kein NP-vollständiges Problem in P liegen, falls P $\neq$ NP. Ist unter der plausiblen Annahme P $\neq$ NP jedes NP-Problem entweder effizient lösbar, also in P, oder NP-vollständig? Oder aber kann es unter der Annahme P $\neq$ NP Probleme in NP geben, die weder effizient lösbar noch NP-vollständig sind? Ein Resultat von Ladner [35] gibt die Antwort.

Satz 2.15 (Ladner) *Ist* P $\neq$ NP*, so gibt es Probleme in* NP*, die weder in* P *noch* NP*-vollständig sind.*

Die von Ladner angegebenen Probleme sind etwas "künstlich" in dem Sinn, dass sie gerade zum Zweck des Beweises von Satz 2.15 konstruiert worden sind. Aber es gibt auch natürliche Probleme in NP, die gute Kandidaten dafür sind, weder in P zu liegen noch NP-vollständig zu sein. Eines davon ist GI, das Graphisomorphieproblem, und das wollen wir nun beweisen. Dazu definieren wir zwei Hierarchien von Komplexitätsklassen innerhalb von NP, die so genannte *Low-Hierarchie* und die *High-Hierarchie*, die von Schöning [54] eingeführt wurden. Damit diese beiden Hierarchien definiert werden können, müssen wir zunächst die *Polynomialzeit-Hierarchie* einführen, die auf NP aufbaut. Um diese wiederum definieren zu können, benötigen wir eine allgemeinere Reduzierbarkeit als die in Definition 2.6 eingeführte many-one-Reduzierbarkeit $\leq^{\mathrm{p}}_{\mathrm{m}}$, nämlich die *Turing-Reduzierbarkeit* $\leq^{\mathrm{p}}_{\mathrm{T}}$. Auch definieren wir die *nichtdeterministische* und die *starke nichtdeterministische Turing-Reduzierbarkeit*, $\leq^{\mathrm{NP}}_{\mathrm{T}}$ und $\leq^{\mathrm{NP}}_{\mathrm{sT}}$, die für die Polynomialzeit-Hierarchie und die High-Hierarchie von Bedeutung sind. Diese Reduzierbarkeiten beruhen auf dem Begriff der *Orakel-Turingmaschine*. Die genannten Begriffe werden nun definiert.

Definition 2.16 (Orakel-Turingmaschine) *Eine* Orakelmenge *(oder kurz ein* Orakel*) ist eine Menge von Wörtern. Eine* Orakel-Turingmaschine M*, etwa mit Orakel* B*, ist eine Turingmaschine, die über ein spezielles Arbeitsband verfügt, das so genannte* Frageband*, und deren Zustandsmenge einen speziellen* Fragezustand*,* $z_?$*, sowie die* Antwortzustände $z_{\mathtt{yes}}$ *und* $z_{\mathtt{no}}$ *enthält. Solange* M *nicht im Zustand* $z_?$ *ist, arbeitet sie genau wie eine gewöhnliche Turingmaschine. Erreicht sie im Laufe ihrer Berechnung jedoch den Fragezustand* $z_?$*, so unterbricht sie ihre Berechnung und fragt ihr Orakel nach dem Wort* q*, das zu diesem Zeitpunkt auf dem Frageband steht. Das Orakel* B *kann man sich als eine Art "Black Box" vorstellen:* B *gibt in einem Takt die Antwort, ob* q *in* B *ist oder nicht, unabhängig davon, wie schwer die Menge* B *zu entscheiden ist. Ist* $q \in B$*, so geht* M *im nächsten Takt in den Antwortzustand* $z_{\mathtt{yes}}$ *über und setzt ihre Berechnung fort. Andernfalls (wenn* $q \notin B$*) setzt* M *ihre Berechnung im Zustand* $z_{\mathtt{no}}$ *fort. Man sagt, die Berechnung von* M *bei Eingabe* x *erfolgt* relativ zum Orakel B*, und schreibt* $M^B(x)$*. Sei* $L(M^B)$ *die von* M^B *akzeptierte Sprache. Eine Komplexitätsklasse* $\mathcal{C}$ *heißt* relativierbar*, wenn sie in dieser Weise durch Orakel-Turingmaschinen (mit der leeren Orakelmenge) repräsentiert werden kann. Definiere für eine relativierbare Komplexitätsklasse* $\mathcal{C}$ *und ein Orakel* B *die Klasse* $\mathcal{C}$ relativ zu B *durch:*

$$\mathcal{C}^B = \{L(M^B) \mid M \text{ ist eine Orakel-Turingmaschine, die } \mathcal{C} \text{ repräsentiert}\}.$$

Ist $\mathcal{B}$ *eine Klasse von Mengen, so sei* $\mathcal{C}^{\mathcal{B}} = \bigcup_{B\in\mathcal{B}} \mathcal{C}^B$*.*

NPOTM (bzw. DPOTM) steht für *nichtdeterministische* (bzw. *deterministische*) *polynomialzeitbeschränkte Orakel-Turingmaschine*. Man kann beispielsweise die folgenden Klassen definieren:

$$\mathrm{NP}^{\mathrm{NP}} = \bigcup_{B\in\mathrm{NP}} \mathrm{NP}^B = \{L(M^B) \mid M \text{ ist eine NPOTM und } B \text{ ist in NP}\};$$

$$\mathrm{P}^{\mathrm{NP}} = \bigcup_{B\in\mathrm{NP}} \mathrm{P}^B = \{L(M^B) \mid M \text{ ist eine DPOTM und } B \text{ ist in NP}\}.$$

Im Falle der leeren Menge $\emptyset$ als Orakel erhalten wir die unrelativierten Klassen NP $=$ NP$^{\emptyset}$ bzw. P $=$ P$^{\emptyset}$ und sagen NPTM statt NPOTM bzw. DPTM statt DPOTM. Orakel-Turingmachinen können insbesondere für Suchtechniken eingesetzt werden, etwa bei einer Präfixsuche, wie das folgende Beispiel zeigt. Das verwendete NP-Orakel Pre-Iso liefert dabei die Information, wie man ausgehend vom leeren Wort Bit für Bit die kleinste Lösung des NP-Problems GI konstruiert, sofern überhaupt eine Lösung existiert.

Beispiel 2.17 (Präfixsuche nach dem kleinsten Isomorphismus mit einer Orakel-Turingmachine) *Das Graphisomorphieproblem* GI *wurde in Definition 1.14 in Abschnitt 1.2.4 definiert. Seien G und H zwei gegebene Graphen mit jeweils $n \geq 1$ Knoten. Ein Isomorphismus zwischen G und H heißt* Lösung von "$(G,H) \in$ GI"*. Die Menge der Isomorphismen* Iso(G,H) *enthält alle Lösungen von "$(G,H) \in$ GI", und es gilt:* Iso$(G,H) \neq \emptyset \iff (G,H) \in$ GI*. Unser Ziel ist es, die lexikographisch kleinste Lösung zu finden, falls $(G,H) \in$ GI; andernfalls soll "$(G,H) \notin$ GI" durch Ausgabe des leeren Wortes λ angezeigt werden. Das heißt, wir wollen die Funktion f berechnen, die folgendermaßen definiert ist:*

$$f(G,H) \;=\; \begin{cases} \min\{\pi \mid \pi \in \text{Iso}(G,H)\} & \textit{falls } (G,H) \in \texttt{GI} \\ \lambda & \textit{falls } (G,H) \notin \texttt{GI}, \end{cases}$$

wobei das Minimum bezüglich der lexikographischen Ordnung auf $\mathfrak{S}_n$ gebildet wird, die so definiert ist: Wir fassen eine Permutation $\pi \in \mathfrak{S}_n$ als das Wort $\pi(1)\pi(2)\cdots\pi(n)$ der Länge n über dem Alphabet $[n] = \{1,2,\ldots,n\}$ auf und schreiben $\pi < \sigma$ für $\pi, \sigma \in \mathfrak{S}_n$ genau dann, wenn es ein $j \in [n]$ gibt, so dass $\pi(i) = \sigma(i)$ für alle $i < j$ und $\pi(j) < \sigma(j)$ gilt. Streicht man aus einer Permutation $\sigma \in \mathfrak{S}_n$ einige der Paare $(i,\sigma(i))$ heraus, so entsteht eine partielle Permutation*, welche auch als Wort über $[n]$ aufgefasst wird. Ein* Präfix der Länge $k \leq n$ von $\sigma \in \mathfrak{S}_n$ *ist eine partielle Permutation von σ, die alle Paare $(i,\sigma(i))$ mit $i \leq k$ enthält, aber keines der Paare $(i,\sigma(i))$ mit $i > k$. Insbesondere sind im Fall $k = 0$ das leere Wort λ und im Fall $k = n$ die totale Permutation σ aus $\mathfrak{S}_n$ auch Präfixe von σ. Ist π ein Präfix der Länge $k < n$ von $\sigma \in \mathfrak{S}_n$ und ist $w = i_1 i_2 \cdots i_{|w|}$ ein Wort über $[n]$ der Länge $|w| \leq n-k$, so bezeichne πw die partielle Permutation, die π um die Paare $(k+1,i_1),(k+2,i_2),\ldots,(k+|w|,i_{|w|})$ erweitert. Gilt dabei $\sigma(k+j) = i_j$ für $1 \leq j \leq |w|$, so ist auch πw ein Präfix von σ. Definiere für die Graphen G und H die Menge der Präfixe von Isomorphismen in* Iso(G,H) *durch:*

$$\text{Pre-Iso} \;=\; \{(G,H,\pi) \mid (\exists w \in \{1,2,\ldots,n\}^*)\,[w = i_1 i_2 \cdots i_{n-|\pi|} \textit{ und } \pi w \in \text{Iso}(G,H)]\}.$$

```
N^Pre-Iso(G, H) {
  if ((G, H, λ) ∉ Pre-Iso) return λ;
    else {
      π := λ;   j := 0;
      while (j < n) {              // G und H haben jeweils n Knoten
        i := 1;
        while ((G, H, πi) ∉ Pre-Iso) {i := i + 1;}
        π := πi;   j := j + 1;
      }
      return π
    }
}
```

Abbildung 2.6: Präfixsuche nach dem kleinsten Isomorphismus in Iso(G,H).

Beachte, dass für $n \geq 1$ *das leere Wort* λ *keine Permutation in* $\mathfrak{S}_n$ *codiert und dass* $\mathrm{Iso}(G,H) = \emptyset$ *genau dann gilt, wenn* $(G,H,\lambda) \notin$ Pre-Iso*, was genau dann der Fall ist, wenn* $(G,H) \notin$ GI.

Mit dem Orakel Pre-Iso *berechnet die DPOTM* N *in Abbildung 2.6 die Funktion* f *durch Präfixsuche, siehe auch Aufgabe 2.5.2. Bezeichnen wir mit* FP *die Klasse aller in Polynomialzeit berechenbaren Funktionen, so folgt* $f \in \mathrm{FP}^{\text{Pre-Iso}}$*. Da* Pre-Iso *eine Menge in* NP *ist (siehe Aufgabe 2.5.2), folgt* $f \in \mathrm{FP}^{\mathrm{NP}}$.

Beispiel 2.17 zeigt, dass auch Turingmaschinen, die Funktionen berechnen, mit einem Orakel ausgestattet sein können und dass auch Funktionenklassen wie z.B. FP relativierbar sind. Andererseits können ebenso Funktionen statt Mengen als Orakel verwendet werden. Im Unterschied zu Definition 2.16 wird dann bei einer Frage q nicht die Antwort "ja" oder "nein" in einem Takt gegeben, sondern das Funktionenorakel $f : \Sigma^* \to \Sigma^*$ liefert als Antwort den Funktionswert $f(q)$ in $|f(q)|$ Takten. Der folgende Satz sagt, dass man mit einem Funktionenorakel f aus einem partiellen Isomorphismus zwischen zwei isomorphen Graphen einen totalen Isomorphismus konstruieren kann.

Satz 2.18 *Seien* G *und* H *zwei isomorphe Graphen. Sei* f *ein Funktionenorakel mit* $f(G,H) = (x,y)$*, wobei* $x \in V(G)$ *und* $y \in V(H)$ *mit* $\sigma(x) = y$ *für einen Isomorphismus* $\sigma \in \mathrm{Iso}(G,H)$ *gilt. Dann gibt es eine DPOTM* M*, die mit dem Orakel* f *einen Isomorphismus* $\varphi \in \mathrm{Iso}(G,H)$ *berechnet.*

Satz 2.18 sagt also, dass sich die Konstruktion einer vollständigen Lösung des NP-Problems GI auf die Konstruktion einer partiellen Lösung von GI reduzieren lässt; vgl. auch den Algorithmus für 3-SAT in Abbildung 2.3, der Bit für Bit partielle Lösungen von 3-SAT-Formeln erweitert, bis sie total sind.

Man stelle sich etwa vor, dass Merlin im Zero-Knowledge-Protokoll für GI aus Abbildung 1.11 in Abschnitt 1.6 einen Isomorphismus $\sigma \in \mathrm{Iso}(G,H)$ an Arthur schickt, wie von diesem verlangt. Leider gehen bei der Übertragung einige Bits verloren oder werden "verrauscht". Arthur empfängt also nur einen partiellen Isomorphismus π von σ. Dank Satz 2.18 kann er jedoch mit Merlins Hilfe aus π einen vollständigen Isomorphismus $\varphi \in \mathrm{Iso}(G,H)$ rekonstruieren, auch wenn π nur aus einem einzigen Knotenpaar besteht. Beachte, dass φ nicht der ursprünglich von Merlin geschickte Isomorphismus σ sein muss.

Beispiel 2.19 zeigt die Beweisidee anhand konkreter Graphen. Auf den formalen Beweis verzichten wir. Die wesentliche Eigenschaft, die man dabei ausnutzt, ist die so genannte *Selbstreduzierbarkeit* von GI. Ohne in technische Details zu gehen, kann man diesen wichtigen Begriff so erklären: Eine Menge A heißt genau dann *selbstreduzierbar*, wenn es eine DPOTM M gibt, die mit dem Orakel A die Menge A selbst akzeptiert. Könnte M das Orakel A einfach nach dem Eingabewort x fragen, so wäre die Entscheidung, ob x in A liegt oder nicht, natürlich trivial. Deshalb verbietet man in einer Selbstreduktion die Frage nach der Eingabe selbst. Stattdessen darf M das Orakel A nur nach solchen Wörtern fragen, die *kleiner* als die Eingabe sind, wobei "kleiner" in einem allgemeineren Sinn als nur bezüglich der gewöhnlichen lexikographischen Ordnung zu verstehen ist, siehe [52]. Diese Definition kann gemäß Satz 2.18 auch auf Funktionen statt Mengen übertragen werden.

Beispiel 2.19 (Konstruktion eines totalen Isomorphismus aus einem partiellen Isomorphismus) *Abbildung 2.7 gibt zwei isomorphe Graphen an,* G *und* H*, wobei* $\mathrm{Iso}(G,H) = \{\sigma, \varphi\}$*, mit* $\sigma = \left(\begin{smallmatrix}1&2&3&4&5\\1&5&4&3&2\end{smallmatrix}\right)$ *und* $\varphi = \left(\begin{smallmatrix}1&2&3&4&5\\5&1&4&3&2\end{smallmatrix}\right)$*. Zunächst wird ein naiver Ansatz beschrieben und erklärt, weshalb dieser fehlschlägt. Angenommen, der Algorithmus aus Satz 2.18 ermittelt durch Befragen des Orakels* f *ein Knotenpaar* (x,y) *mit* $\sigma(x) = y$ *oder* $\varphi(x) = y$*, merkt sich* (x,y)*, löscht* x *in* G *bzw.* y *in* H *und fährt in dieser Weise sukzessive fort, bis die Graphen leer sind. Damit wäre sichergestellt, dass er nach höchstens* $n = 5$ *Durchläufen terminiert, und man könnte hoffen, dass die Folge der gespeicherten Knotenpaare dann den gesuchten Isomorphismus aus* $\mathrm{Iso}(G,H)$ *ergäbe, also entweder* σ *oder* φ*. Dies ist jedoch nicht unbedingt der Fall. Nehmen wir etwa an, das Orakel* f *antwortet bei der ersten Frage nach* (G,H) *mit dem Knotenpaar* $(5,2)$*. Würde der Algorithmus aus Satz 2.18 nun einfach den Knoten* 5 *aus* G

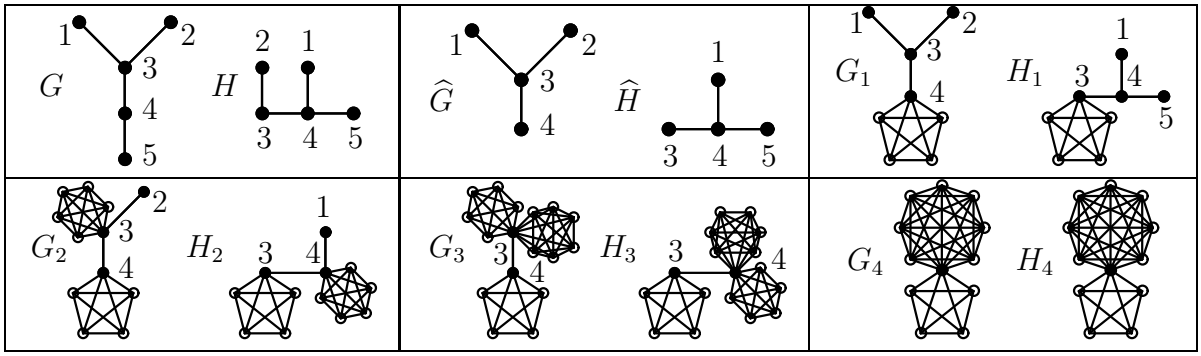


Abbildung 2.7: Beispiel für die Konstruktion aus Satz 2.18.

und den Knoten 2 *aus* H *(sowie die aus* 5 *bzw. aus* 2 *auslaufenden Kanten) löschen, so erhielte man die Graphen* $\widehat{G}$ *und* $\widehat{H}$ *aus Abbildung 2.7. Jedoch enthält* $\mathrm{Iso}(\widehat{G}, \widehat{H})$ *sechs Isomorphismen, von denen nur zwei mit dem gelöschten Paar* $(5, 2)$ kompatibel *sind, siehe Aufgabe 2.5.3. Das heißt, nur zwei der sechs Isomorphismen aus* $\mathrm{Iso}(\widehat{G}, \widehat{H})$ *sind partielle Isomorphismen von* σ *und* φ*. Dann aber könnte der Algorithmus im nächsten Schritt etwa das neue Paar* $(4, 5)$ *speichern, das weder zu* σ *noch zu* φ *gehört.*

Um solche Fälle auszuschließen, geht die DPOTM M *mit Orakel* f *aus Satz 2.18 anders vor. Sie löscht nicht einfach nur Knotenpaare, die sie von ihrem Orakel ermitteln lässt, sondern sie markiert die Nachbarknoten der gelöschten Knoten durch Cliquen hinreichender Größe. Eine* Clique *der Größe* k *ist der Graph, dessen* k *Knoten alle jeweils paarweise durch eine Kante verbunden sind. Im Beispiel wird also nach dem Löschen des ersten Knotenpaares* $(5, 2)$ *der Knoten* 4 *in* G *und der Knoten* 3 *in* H *jeweils durch eine Clique der Größe* 5 *markiert. Es ergibt sich das neue Paar* (G_1, H_1) *von Graphen, siehe Abbildung 2.7. Beachte, dass nun jeder Isomorphismus* $\pi \in \mathrm{Iso}(G_1, H_1)$ *mit dem Knotenpaar* $(5, 2)$ *aus den ursprünglichen Isomorphismen* σ *und* φ *aus* $\mathrm{Iso}(G, H)$ *kompatibel ist.*

Setzt M *dieses Verfahren sukzessive fort, so können sich für eine bestimmte Folge von Orakelantworten beispielsweise die Graphenpaare* (G_2, H_2)*,* (G_3, H_3) *und* (G_4, H_4) *aus Abbildung 2.7 ergeben. Das letzte Knotenpaar* $(4, 3)$ *ist bei* (G_4, H_4) *dann eindeutig bestimmt, und* M *hat in diesem Falle den totalen Isomorphismus* $\varphi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 4 & 3 & 2 \end{smallmatrix}\right)$ *aus* $\mathrm{Iso}(G, H)$ *konstruiert.*

Nun werden ausgehend vom Begriff der Orakel-Turingmaschine verschiedene Reduzierbarkeiten definiert. Alle hier betrachteten Reduzierbarkeiten sind effizient, also in Polynomialzeit berechenbar.

Definition 2.20 (Turing-Reduzierbarkeiten) *Sei* $\Sigma = \{0, 1\}$ *ein binäres Alphabet, seien* A *und* B *Mengen von Wörtern über* Σ*, und sei* $\mathcal{C}$ *eine Komplexitätsklasse. Die Klasse der Komplemente von Mengen in* $\mathcal{C}$ *ist definiert als* $\mathrm{co}\mathcal{C} = \{\overline{L} \mid L \in \mathcal{C}\}$*. Definiere die folgenden Reduzierbarkeiten:*

- Turing-Reduzierbarkeit: $A \leq_{\mathrm{T}}^{\mathrm{p}} B \iff A = L(M^B)$ *für eine DPOTM* M*.*
- Nichtdeterministische Turing-Reduzierbarkeit: $A \leq_{\mathrm{T}}^{\mathrm{NP}} B \iff A = L(M^B)$ *für eine NPOTM* M*.*
- Starke nichtdeterministische Turing-Reduzierbarkeit: $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B \iff A \in \mathrm{NP}^B \cap \mathrm{coNP}^B$*.*
- *Ist* $\leq_r$ *eine der oben definierten Reduzierbarkeiten, so nennen wir eine Menge* B *genau dann* $\leq_r$-hart für $\mathcal{C}$*, wenn* $A \leq_r B$ *für jede Menge* $A \in \mathcal{C}$ *gilt. Eine Menge* B *heißt genau dann* $\leq_r$-vollständig in $\mathcal{C}$*, wenn* $B \leq_r$*-hart für* $\mathcal{C}$ *ist und* $B \in \mathcal{C}$*.*
- $\mathrm{P}^{\mathcal{C}} = \{A \mid (\exists B \in \mathcal{C})\,[A \leq_{\mathrm{T}}^{\mathrm{p}} B]\}$ *ist der* Abschluss von $\mathcal{C}$ unter der $\leq_{\mathrm{T}}^{\mathrm{p}}$-Reduzierbarkeit.
- $\mathrm{NP}^{\mathcal{C}} = \{A \mid (\exists B \in \mathcal{C})\,[A \leq_{\mathrm{T}}^{\mathrm{NP}} B]\}$ *ist der* Abschluss von $\mathcal{C}$ unter der $\leq_{\mathrm{T}}^{\mathrm{NP}}$-Reduzierbarkeit.

Mit Hilfe der in Definition 2.20 eingeführten $\leq^{\mathrm{p}}_{\mathrm{T}}$- und $\leq^{\mathrm{NP}}_{\mathrm{T}}$-Reduzierbarkeit werden nun die Polynomialzeit-Hierarchie sowie die Low-Hierarchie und die High-Hierarchie in NP definiert.

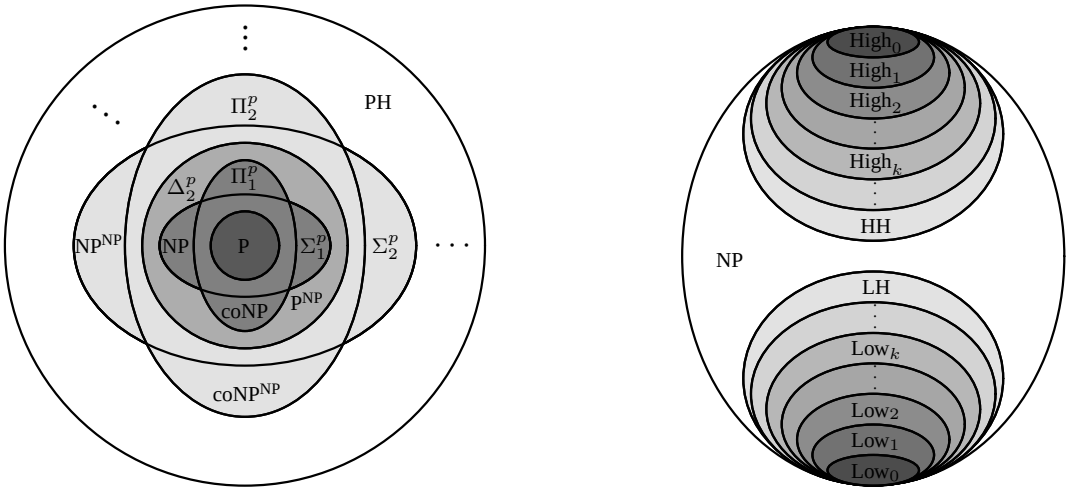


Abbildung 2.8: Die Polynomialzeit-, die Low- und die High-Hierarchie.

Definition 2.21 (Polynomialzeit-Hierarchie) *Die* Polynomialzeit-Hierarchie $\mathrm{PH} = \bigcup_{k\geq 0} \Sigma_k^p$ *ist definiert durch:* $\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathrm{P}$, $\Delta_{i+1}^p = \mathrm{P}^{\Sigma_i^p}$, $\Sigma_{i+1}^p = \mathrm{NP}^{\Sigma_i^p}$ *und* $\Pi_{i+1}^p = \mathrm{co}\Sigma_{i+1}^p$ *für* $i \geq 0$.

Insbesondere gilt: $\Delta_1^p = \mathrm{P}^{\Sigma_0^p} = \mathrm{P}^{\mathrm{P}} = \mathrm{P}$ und $\Sigma_1^p = \mathrm{NP}^{\Sigma_0^p} = \mathrm{NP}^{\mathrm{P}} = \mathrm{NP}$ und $\Pi_1^p = \mathrm{co}\Sigma_1^p = \mathrm{coNP}$. Der folgende Satz (ohne Beweis) gibt einige Eigenschaften dieser Hierarchien an, siehe Aufgabe 2.5.2.

Satz 2.22 (Meyer und Stockmeyer) *Für jedes* $i \geq 1$ *gilt:*

1. $\Sigma_{i-1}^p \cup \Pi_{i-1}^p \subseteq \Delta_i^p \subseteq \Sigma_i^p \cap \Pi_i^p$.
2. Σ_i^p, Π_i^p, Δ_i^p *und* PH *sind* $\leq^{\mathrm{p}}_{\mathrm{m}}$*-abgeschlossen.* Δ_i^p *ist sogar unter* $\leq^{\mathrm{p}}_{\mathrm{T}}$*-Reduktionen abgeschlossen.*
3. Σ_i^p *enthält genau die Mengen* A*, für die es eine Menge* $B \in \mathrm{P}$ *und ein Polynom* p *gibt, so dass für alle* $x \in \Sigma^*$ *gilt:* $x \in A \iff (\exists^p w_1)\,(\forall^p w_2) \cdots (\mathfrak{Q}^p w_i)\,[(x, w_1, w_2, \ldots, w_i) \in B]$*, wobei die Quantoren* $\exists^p$ *und* $\forall^p$ *polynomiell längenbeschränkt sind und* $\mathfrak{Q}^p = \exists^p$*, falls* i *ungerade ist, und* $\mathfrak{Q}^p = \forall^p$*, falls* i *gerade ist.*
4. *Ist* $\Sigma_{i-1}^p = \Sigma_i^p$*, so kollabiert die* PH *auf* $\Sigma_{i-1}^p = \Pi_{i-1}^p = \Delta_i^p = \Sigma_i^p = \Pi_i^p = \cdots = \mathrm{PH}$.
5. *Ist* $\Sigma_i^p = \Pi_i^p$*, so kollabiert die* PH *auf* $\Sigma_i^p = \Pi_i^p = \Delta_{i+1}^p = \Sigma_{i+1}^p = \Pi_{i+1}^p = \cdots = \mathrm{PH}$.
6. *In* Σ_i^p, Π_i^p *und* Δ_i^p *gibt es* $\leq^{\mathrm{p}}_{\mathrm{m}}$*-vollständige Probleme. Gibt es jedoch in* PH *ein* $\leq^{\mathrm{p}}_{\mathrm{m}}$*-vollständiges Problem, so kollabiert die* PH *auf eine endliche Stufe, d.h.,* $\mathrm{PH} = \Sigma_k^p = \Pi_k^p$ *für ein* k.

Definition 2.23 (Low-Hierarchie und High-Hierarchie in NP) *Definiere für* $k \geq 0$ *die* k*-te Stufe der*

- Low-Hierarchie $\mathrm{LH} = \bigcup_{k\geq 0} \mathrm{Low}_k$ in NP *durch* $\mathrm{Low}_k = \{L \in \mathrm{NP} \mid \Sigma_k^{p,L} \subseteq \Sigma_k^p\}$*;*
- High-Hierarchie $\mathrm{HH} = \bigcup_{k\geq 0} \mathrm{High}_k$ in NP *durch* $\mathrm{High}_k = \{H \in \mathrm{NP} \mid \Sigma_{k+1}^p \subseteq \Sigma_k^{p,H}\}$.

Eine Menge L ist also genau dann in Low_k, wenn sie als Orakel für eine Σ_k^p-Berechnung nutzlos ist. Alle Information, die L zu bieten hat, kann eine Σ_k^p-Maschine auch ohne Orakel selbst berechnen. Andererseits ist eine Menge H in High_k so reich an nützlicher Information, dass sie die Berechnungskraft einer Σ_k^p-Maschine um den für eine NP-Menge maximalen Betrag erhöht. Mit Hilfe des Orakels H aus High_k kann eine Σ_k^p-Maschine jede Σ_{k+1}^p-Berechnung simulieren. H leistet also für eine Σ_k^p-Maschine genauso viel wie eine NP-vollständige Menge. Die Inklusionsstruktur der oben definierten Hierarchien ist in Abbildung 2.8 dargestellt. Dunkler dargestellte Komplexitätsklassen sind dabei in heller dargestellten enthalten. Für keine dieser Inklusionen $\mathcal{C} \subseteq \mathcal{D}$ ist bekannt, ob sie *echt* ist, d.h., ob sogar $\mathcal{C} \neq \mathcal{D}$ gilt. Für $k = 0$ ist die Frage, ob $\Sigma_k^p \neq \Sigma_{k+1}^p$ gilt, gerade die P-versus-NP-Frage. Nun werden (wieder ohne Beweis) einige wichtige Eigenschaften dieser Hierarchien aufgelistet, siehe [54] und Aufgabe 2.5.2. Der Satz von Ladner (Satz 2.15) folgt sofort aus dem Spezialfall $n = 0$ der letzten Aussage von Satz 2.24.

Satz 2.24 (Schöning) *1.* $\text{Low}_0 = \text{P}$ *und* $\text{Low}_1 = \text{NP} \cap \text{coNP}$ *und* $\text{NP} \cap \text{coAM} \subseteq \text{Low}_2$.

2. $\text{High}_0 = \{H \mid H$ *ist* $\leq_{\text{T}}^{\text{p}}$*-vollständig in* NP$\}$ *und* $\text{High}_1 = \{H \mid H$ *ist* $\leq_{\text{sT}}^{\text{NP}}$*-vollständig in* NP$\}$.

3. $\text{Low}_0 \subseteq \text{Low}_1 \subseteq \cdots \subseteq \text{Low}_k \subseteq \cdots \subseteq \text{LH} \subseteq \text{NP}$.

4. $\text{High}_0 \subseteq \text{High}_1 \subseteq \cdots \subseteq \text{High}_k \subseteq \cdots \subseteq \text{HH} \subseteq \text{NP}$.

5. Für jedes $n \geq 0$ *ist* $\text{Low}_n \cap \text{High}_n$ *genau dann nicht leer, wenn* $\Sigma_n^p = \Sigma_{n+1}^p = \cdots = \text{PH}$.

6. Für jedes $n \geq 0$ *gibt es genau dann Mengen in* NP*, die weder in* Low_n *noch in* High_n *sind, wenn* $\Sigma_n^p \neq \Sigma_{n+1}^p$*. Es gibt genau dann Mengen in* NP*, die weder in* LH *noch in* HH *sind, wenn die* PH *echt unendlich ist, also nicht auf eine endliche Stufe kollabiert.*

2.5.2 Graphisomorphie ist in der Low-Hierarchie

Nun nehmen wir den Beweis des angekündigten Resultates in Angriff, dass `GI` in Low_2 liegt. Dieses Ergebnis ist ein starkes Indiz gegen die NP-Vollständigkeit von `GI`. Denn wäre `GI` NP-vollständig, so wäre es in $\text{High}_0 \subseteq \text{High}_2$, weil High_0 nach Satz 2.24 genau die $\leq_{\text{T}}^{\text{p}}$-vollständigen Mengen aus NP enthält, insbesondere also die $\leq_{\text{m}}^{\text{p}}$-vollständigen Mengen aus NP. Ebenfalls nach Satz 2.24 ist aber $\text{Low}_2 \cap \text{High}_2$ genau dann nicht leer, wenn die PH auf Σ_2^p kollabiert, was als sehr unwahrscheinlich gilt.

Für den Beweis dieses Resultats benötigen wir noch das so genannte Hashing-Lemma, siehe Lemma 2.26. Hashing ist eine Methode zur dynamischen Verwaltung von Daten. Jedem Datensatz ist dabei ein Schlüssel zugeordnet, der diesen eindeutig identifiziert. Die Menge U der potenziellen Schlüssel ist sehr groß und wird als *Universum* bezeichnet, während die Menge $V \subseteq U$ der tatsächlich verwendeten Schlüssel viel kleiner sein kann. Es geht nun darum, die Elemente von U mittels einer *Hash-Funktion* $h : U \to T$ in eine *Hash-Tabelle* $T = \{0, 1, \ldots, k-1\}$ einzutragen, wobei mehrere Schlüssel aus U dieselbe Adresse in T haben können. Nach Möglichkeit sollen dabei jedoch je zwei verschiedene Schlüssel aus V verschiedene Adressen in T erhalten, d.h., *Kollisionen* für tatsächlich verwendete Schlüssel sollen vermieden werden und h soll auf V injektiv sein.

Unter den verschiedenen bekannten Varianten von Hash-Verfahren ist für unsere Zwecke das *Universal Hashing* von besonderem Interesse, das 1979 von Carter und Wegman [8] eingeführt wurde. Hier besteht die Idee darin, aus einer geeigneten Familie von Hash-Funktionen eine Hash-Funktion *zufällig* auszuwählen. Dieses Hash-Verfahren ist in dem Sinn universell, dass es nicht mehr von einer bestimmten Menge V abhängt, sondern auf *allen* hinreichend kleinen Mengen V mit großer Wahrscheinlichkeit Kollisionen vermeidet. Die Wahrscheinlichkeit bezieht sich dabei auf die zufällige Wahl der Hash-Funktion. Wir denken uns im Folgenden Schlüssel als Wörter über dem Alphabet $\Sigma = \{0, 1\}$ codiert. Mit Σ^n bezeichnen wir die Menge aller Wörter der Länge n in Σ^*.

Definition 2.25 (Hashing) *Sei $\Sigma = \{0,1\}$, und seien m und t natürliche Zahlen mit $t > m$. Eine* Hash-Funktion *$h : \Sigma^t \to \Sigma^m$ ist eine lineare Abbildung, die durch eine boolesche $(t \times m)$-Matrix $B_h = (b_{i,j})_{i,j}$ mit $b_{i,j} \in \{0,1\}$ gegeben ist. Für $x \in \Sigma^t$ und $1 \leq j \leq m$ ergibt sich das j-te Bit von $y = h(x) \in \Sigma^m$ als $y_j = (b_{1,j} \wedge x_1) \oplus (b_{2,j} \wedge x_2) \oplus \cdots \oplus (b_{t,j} \wedge x_t)$, wobei $\oplus$ die logische Paritätsoperation bezeichnet, d.h., $a_1 \oplus a_2 \oplus \cdots \oplus a_n = 1 \iff |\{i \mid a_i = 1\}| \equiv 1 \bmod 2$.*

Sei $\mathcal{H}_{t,m} = \{h : \Sigma^t \to \Sigma^m \mid B_h$ ist eine boolesche $(t \times m)$-Matrix$\}$ eine Familie von Hash-Funktionen für die Parameter t und m. Auf $\mathcal{H}_{t,m}$ nehmen wir die Gleichverteilung an: Eine Hash-Funktion h wird aus $\mathcal{H}_{t,m}$ gezogen, indem die $b_{i,j}$ in B_h unabhängig und gleichverteilt gewählt werden.

Sei $V \subseteq \Sigma^t$. Für eine Teilfamilie $\widehat{\mathcal{H}}$ von $\mathcal{H}_{t,m}$ gibt es auf V eine Kollision*, falls gilt:*

$$(\exists v \in V)\,(\forall h \in \widehat{\mathcal{H}})\,(\exists x \in V)\,[v \neq x \wedge h(v) = h(x)].$$

Andernfalls ist $\widehat{\mathcal{H}}$ auf V kollisionsfrei.

Eine Kollision auf V bedeutet also, dass die Injektivität einer jeden Hash-Funktion aus der Teilfamilie $\widehat{\mathcal{H}}$ auf V gestört ist. Das folgende Lemma sagt, dass auf jeder hinreichend kleinen Mengen V eine zufällig gewählte Teilfamilie von $\mathcal{H}_{t,m}$ kollisionsfrei ist. Ist V jedoch zu groß, so ist eine Kollision unvermeidlich. Auf den Beweis von Lemma 2.26 verzichten wir.

Lemma 2.26 (Hashing-Lemma) *Seien $t, m \in \mathbf{N}$ Parameter, $V \subseteq \Sigma^t$ und $\widehat{\mathcal{H}} = (h_1, h_2, \ldots, h_{m+1})$ eine zufällig unter Gleichverteilung gewählte Familie von Hash-Funktionen aus $\mathcal{H}_{t,m}$. Sei*

$$K(V) = \{\widehat{\mathcal{H}} \mid (\exists v \in V)\,(\forall h \in \widehat{\mathcal{H}})\,(\exists x \in V)\,[v \neq x \wedge h(v) = h(x)]\}$$

das Ereignis, dass für $\widehat{\mathcal{H}}$ auf V eine Kollision stattfindet. Dann gilt:

1. *Ist $|V| \leq 2^{m-1}$, so tritt $K(V)$ mit Wahrscheinlichkeit höchstens $1/4$ ein.*
2. *Ist $|V| > (m+1)2^m$, so tritt $K(V)$ mit Wahrscheinlichkeit 1 ein.*

In Abschnitt 1.6 wurde die Arthur-Merlin-Hierarchie definiert und erwähnt, dass diese Hierarchie auf ihre zweite Stufe kollabiert. Hier sind wir an der Klasse coAM interessiert, siehe Definition 1.26.

Satz 2.27 (Schöning) `GI` *ist in* Low_2.

Beweis. Nach Satz 2.24 ist jede NP-Menge aus coAM in Low_2. Um zu beweisen, dass `GI` in Low_2 liegt, genügt es also zu zeigen, dass `GI` in coAM ist. Seien G und H zwei Graphen mit jeweils n Knoten. Wir wollen das Hashing-Lemma anwenden. Es liegt nahe, die in Lemma 1.18 definierte Menge

$$A(G,H) \quad = \quad \{(F,\varphi) \mid F \cong G \text{ und } \varphi \in \mathrm{Aut}(F)\} \cup \{(F,\varphi) \mid F \cong H \text{ und } \varphi \in \mathrm{Aut}(F)\}$$

dabei die Rolle von V aus Lemma 2.26 spielen zu lassen. Nach Lemma 1.18 ist $|A(G,H)| = n!$, falls $G \cong H$, und $|A(G,H)| = 2n!$, falls $G \ncong H$. Damit die zu konstruierende coAM-Maschine für `GI` in Polynomialzeit arbeitet, müssen die Parameter t und m aus dem Hashing-Lemma Polynome in n sein. Um dieses Lemma anwenden zu können, müssten wir das Polynom $m = m(n)$ so wählen, dass gilt:

$$n! \leq 2^{m-1} < (m+1)2^m < 2n!, \tag{2.4}$$

denn dann wäre die Menge $V = A(G,H)$ groß genug, mit hinreichend großer Wahrscheinlichkeit zwei isomorphe Graphen G und H von zwei nicht isomorphen Graphen zu unterscheiden. Leider ist es nicht möglich, ein Polynom m zu finden, dass der Ungleichung (2.4) genügt. Stattdessen wählen wir eine andere Menge V, um die Lücke zwischen unterer und oberer Schranke groß genug zu machen.

Definiere $V = A(G,H)^n = \underbrace{A(G,H) \times A(G,H) \times \cdots \times A(G,H)}_{n\text{-mal}}$. Nun wird (2.4) zu:

$$(n!)^n \leq 2^{m-1} < (m+1)2^m < (2n!)^n, \tag{2.5}$$

und diese neue Ungleichung kann durch die Wahl von $m = m(n) = 1 + \lceil n \log n! \rceil$ erfüllt werden.

Konstruiere eine coAM-Maschine M für GI wie folgt. Bei Eingabe der Graphen G und H mit n Knoten berechnet M zunächst den Parameter m. Die Menge $V = A(G,H)^n$ enthält n-Tupel von Paaren der Form (F, φ), wobei F ein Graph mit n Knoten ist und $\varphi \in \mathrm{Aut}(F)$. Die Elemente von V seien geeignet als Wörter der Länge $t(n)$ über dem Alphabet $\Sigma = \{0,1\}$ codiert, wobei t ein geeignetes Polynom ist. Dann rät M eine zufällig unter Gleichverteilung gewählte Familie $\widehat{\mathcal{H}} = (h_1, h_2, \ldots, h_{m+1})$ von Hash-Funktionen aus $\mathcal{H}_{t,m}$. Dies entspricht Arthurs Zug. Jede Hash-Funktion $h_i \in \widehat{\mathcal{H}}$ wird durch eine boolesche $(t \times m)$-Matrix repräsentiert. Daher lassen sich die $m+1$ Hash-Funktionen h_i in $\widehat{\mathcal{H}}$ als ein Wort $z_{\widehat{\mathcal{H}}} \in \Sigma^*$ der Länge $p(n)$ darstellen, wobei p ein geeignetes Polynom ist. Modifiziere das Kollisionsprädikat $K(V)$ aus dem Hashing-Lemma nun wie folgt:

$$B = \{(G, H, z_{\widehat{\mathcal{H}}}) \mid (\exists v \in V)\,(\forall i : 1 \leq i \leq m+1)\,(\exists x \in V)\,[v \neq x \wedge h_i(v) = h_i(x)]\}.$$

Da der $\forall$-Quantor in B nur über polynomiell viele i quantifiziert und daher deterministisch in Polynomialzeit ausgewertet werden kann, können die beiden $\exists$-Quantoren in B zu *einem* polynomiell längenbeschränkten $\exists$-Quantor zusammengefasst werden. Nach Satz 2.22 ist somit B eine Menge in $\Sigma_1^p = \mathrm{NP}$. Sei N eine NPTM für B. Für das geratene Wort $z_{\widehat{\mathcal{H}}}$, das $m+1$ unabhängig und gleichverteilt gewählte Hash-Funktionen aus $\mathcal{H}_{t,m}$ repräsentiert, simuliert M nun die Rechnung von $N(G, H, z_{\widehat{\mathcal{H}}})$. Dies entspricht Merlins Zug. M akzeptiert ihre Eingabe (G,H) genau dann, wenn $N(G, H, z_{\widehat{\mathcal{H}}})$ akzeptiert.

Wir schätzen nun die Wahrscheinlichkeit ab (über die zufällige Wahl der in $z_{\widehat{\mathcal{H}}}$ codierten Hash-Funktionen), dass M ihre Eingabe (G,H) akzeptiert. Sind G und H isomorph, so ist $|A(G,H)| = n!$ nach Lemma 1.18. Aus Ungleichung (2.5) folgt $|V| = (n!)^n \leq 2^{m-1}$. Nach Lemma 2.26 ist die Wahrscheinlichkeit, dass $(G, H, z_{\widehat{\mathcal{H}}})$ in B ist und $M(G,H)$ somit akzeptiert, höchstens $1/4$. Sind G und H jedoch nicht isomorph, so folgt $|A(G,H)| = 2n!$ aus Lemma 1.18. Aus Ungleichung (2.5) ergibt sich nun $|V| = (2n!)^n > (m+1)2^m$. Nach Lemma 2.26 ist in diesem Fall die Wahrscheinlichkeit, dass $(G, H, z_{\widehat{\mathcal{H}}})$ in B ist und $M(G,H)$ somit akzeptiert, gleich 1. Es folgt, dass GI in coAM liegt. ∎

2.5.3 Graphisomorphie ist in SPP

Die probabilistische Klasse RP wurde in Definition 1.23 in Abschnitt 1.4.1 eingeführt. In diesem Abschnitt spielen zwei andere wichtige probabilistische Klassen eine Rolle, die wir nun definieren: PP und SPP, Akronyme für *P*robabilistische *P*olynomialzeit und *S*toische *P*robabilistische *P*olynomialzeit.

Definition 2.28 (PP und SPP) *Die Klasse* PP *enthält genau die Probleme A, für die es eine NPTM M gibt, so dass für jede Eingabe x gilt: Ist $x \in A$, so akzeptiert $M(x)$ mit einer Wahrscheinlichkeit $\geq 1/2$, und ist $x \notin A$, so akzeptiert $M(x)$ mit einer Wahrscheinlichkeit $< 1/2$.*

Für eine NPTM M mit Eingabe x bezeichne $\mathrm{acc}_M(x)$ *die Anzahl der akzeptierenden Pfade von $M(x)$ und* $\mathrm{rej}_M(x)$ *die Anzahl der ablehnenden Pfade von $M(x)$. Definiere* $\mathrm{gap}_M(x) = \mathrm{acc}_M(x) - \mathrm{rej}_M(x)$.

Die Klasse SPP *enthält genau die Probleme A, für die es eine NPTM M gibt, so dass für alle x gilt:* $(x \in A \implies \mathrm{gap}_M(x) = 1)$ *und* $(x \notin A \implies \mathrm{gap}_M(x) = 0)$.

Eine SPP-Maschine ist also "stoisch" in dem Sinn, dass ihr "gap" – also die Differenz von akzeptierenden und ablehnenden Pfaden – stets nur zwei aus einer exponentiellen Anzahl von möglichen Werten

annehmen kann. Im Gegensatz zu PP ist SPP damit eine so genannte "*promise*"-Klasse, denn eine SPP-Maschine M gibt das "Versprechen", dass $\text{gap}_M(x) \in \{0, 1\}$ für alle x gilt; siehe auch Aufgabe 2.5.4.

Der Begriff der Lowness kann für jede beliebige relativierbare Komplexitätsklasse $\mathcal{C}$ definiert werden: Eine Menge A heißt genau dann $\mathcal{C}$*-low*, wenn $\mathcal{C}^A = \mathcal{C}$ gilt. Insbesondere enthält für jedes k die Stufe Low_k der Low-Hierarchie aus Definition 2.23 gerade die NP-Mengen, die Σ_k^p-low sind. Alle in der oben definierten Klasse SPP enthaltenen Mengen sind PP-low. Diese und andere nützliche Eigenschaften von SPP werden ohne Beweis im folgenden Satz zusammengefasst, siehe auch [33, 34, 13].

Satz 2.29 *1.* SPP *ist* PP*-low, d.h.,* $\text{PP}^{\text{SPP}} = \text{PP}$.

2. SPP *ist selbst-low, d.h.,* $\text{SPP}^{\text{SPP}} = \text{SPP}$.

3. Seien $A \in$ NP *vermittels einer NPTM* N *und* $L \in \text{SPP}^A$ *vermittels einer NPOTM* M*, so dass* $M^A(x)$ *für alle Eingaben* x *nur Fragen* q *stellt, für die* $\text{acc}_N(q) \leq 1$ *gilt. Dann ist* L *in* SPP.

4. Seien $A \in$ NP *vermittels einer NPTM* N *und* $f \in \text{FP}^A$ *vermittels einer DPOTM* M*, so dass* $M^A(x)$ *für alle Eingaben* x *nur Fragen* q *stellt, für die* $\text{acc}_N(q) \leq 1$ *gilt. Dann ist* f *in* FP^{SPP}.

Der folgende Satz sagt, dass die lexikographisch kleinste Permutation in einer rechten co-Menge effizient berechnet werden kann. Die lexikographische Ordnung auf $\mathfrak{S}_n$ ist in Beispiel 2.17 definiert.

Satz 2.30 *Seien* $\mathfrak{G} \leq \mathfrak{S}_n$ *eine Permutationsgruppe mit* $\mathfrak{G} = \langle G \rangle$ *und* $\pi \in \mathfrak{S}_n$ *eine Permutation. Es gibt einen Algorithmus, der bei Eingabe* (G, π) *in Polynomialzeit die lexikographisch kleinste Permutation der rechten co-Menge* $\mathfrak{G}\pi$ *von* $\mathfrak{G}$ *in* $\mathfrak{S}_n$ *bestimmt.*

Beweis. Abbildung 2.9 zeigt den Algorithmus LERC zur Berechnung der lexikographisch kleinsten Permutation in der rechten co-Menge $\mathfrak{G}\pi$ von $\mathfrak{G}$ in $\mathfrak{S}_n$, wobei die Permutationsgruppe $\mathfrak{G}$ durch einen Generator G gegeben ist, siehe Definition 1.12 in Abschnitt 1.2.4.

```
LERC(G, π) {
  Berechne den Turm 𝔊^(n) ≤ 𝔊^(n−1) ≤ ··· ≤ 𝔊^(1) ≤ 𝔊^(0) von Stabilisatoren in 𝔊;
    φ_0 = π;
    for (i = 0, 1, ..., n − 1) {
      x := i + 1;
      Berechne das Element y im Orbit 𝔊^(i)(x), für das φ_i(y) minimal ist;
      Bestimme eine Permutation τ_i in 𝔊^(i) mit τ_i(x) = y;
      φ_{i+1} := τ_i φ_i;
    }
    return φ_n;
}
```

Abbildung 2.9: Algorithmus LERC bestimmt das kleinste Element der rechten co-Menge $\mathfrak{G}\pi$.

Nach Satz 1.13 kann der Turm $\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}$ der Stabilisatoren von $\mathfrak{G}$ in Polynomialzeit berechnet werden. Genauer gesagt, werden für jedes i mit $1 \leq i \leq n$ die vollständigen rechten Transversalen T_i von $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$ und somit ein starker Generator $S = \bigcup_{i=1}^{n-1} T_i$ von $\mathfrak{G}$ bestimmt. Da $\varphi_0 = \pi$ und $\mathfrak{G}^{(n-1)} = \mathfrak{G}^{(n)} = \mathbf{id}$ gilt, genügt es, zum Nachweis der Korrektheit des Algorithmus LERC zu zeigen, dass für jedes i mit $0 \leq i \leq n-1$ die lexikographisch kleinste Permutation von $\mathfrak{G}^{(i)}\varphi_i$ in $\mathfrak{G}^{(i+1)}\varphi_{i+1}$ enthalten ist. Daraus folgt mit Induktion, dass $\mathfrak{G}^{(n)}\varphi_n = \{\varphi_n\}$ die

lexikographisch kleinste Permutation von $\mathfrak{G}\pi = \mathfrak{G}^{(0)}\varphi_0$ enthält. Folglich gibt der Algorithmus LERC tatsächlich die lexikographisch kleinste Permutation φ_n von $\mathfrak{G}\pi$ aus.

Um die obige Behauptung zu beweisen, bezeichnen wir mit $\mathfrak{H}(x)$ den Orbit des Elements $x \in [n]$ in einer Permutationsgruppe $\mathfrak{H} \leq \mathfrak{S}_n$. Sei τ_i die Permutation in $\mathfrak{G}^{(i)}$, die $i+1$ auf das Element y im Orbit $\mathfrak{G}^{(i)}(i+1)$ abbildet, für das $\varphi_i(y) = x$ das minimale Element in der Menge $\{\varphi_i(z) \mid z \in \mathfrak{G}^{(i)}(i+1)\}$ ist. Nach Satz 1.13 kann der Orbit $\mathfrak{G}^{(i)}(i+1)$ in Polyomialzeit berechnet werden, und da $\mathfrak{G}^{(i)}(i+1)$ höchstens $n-i$ Elemente enthält, kann y effizient bestimmt werden. Nach Definition des Algorithmus gilt $\varphi_{i+1} = \tau_i\varphi_i$. Da jede Permutation in $\mathfrak{G}^{(i)}$ jedes Element von $[i]$ auf sich selbst abbildet und da $\tau_i \in \mathfrak{G}^{(i)}$, gilt für jedes j mit $1 \leq j \leq i$, für jedes $\tau \in \mathfrak{G}^{(i)}$ und jedes $\sigma \in \mathfrak{G}^{(i+1)}$:

$$(\sigma\varphi_{i+1})(j) = \varphi_{i+1}(j) = (\tau_i\varphi_i)(j) = \varphi_i(j) = (\tau\varphi_i)(j).$$

Insbesondere folgt daraus, dass für die lexikographisch kleinste Permutation μ in $\mathfrak{G}^{(i)}\varphi_i$ gilt, dass jede Permutation aus $\mathfrak{G}^{(i+1)}\varphi_{i+1}$ mit μ auf den ersten i Elementen, also auf $[i]$, übereinstimmen muss.

Außerdem gilt für jedes $\sigma \in \mathfrak{G}^{(i+1)}$ und für das oben definierte Element $x = \varphi_i(y)$:

$$(\sigma\varphi_{i+1})(i+1) = \varphi_{i+1}(i+1) = (\tau_i\varphi_i)(i+1) = x.$$

Natürlich ist $\mathfrak{G}^{(i+1)}\varphi_{i+1} = \{\varphi \in \mathfrak{G}^{(i)}\varphi_i \mid \varphi(i+1) = x\}$. Die Behauptung folgt nun aus der Tatsache, dass $\mu(i+1) = x$ für die lexikographisch kleinste Permutation μ von $\mathfrak{G}^{(i)}\varphi_i$ gilt.

Somit ist gezeigt, dass der Algorithmus LERC effizient und korrekt arbeitet. ∎

Theorem 2.30 kann leicht zu Korollar 2.31 erweitert werden, siehe Aufgabe 2.5.3. Anschließend beweisen wir Satz 2.32, das Hauptresultat dieses Abschnitts.

Korollar 2.31 *Sei $\mathfrak{G} \leq \mathfrak{S}_n$ eine Permutationsgruppe mit $\mathfrak{G} = \langle G \rangle$, und seien π und ψ zwei Permutationen in $\mathfrak{S}_n$. Es gibt einen Algorithmus, der bei Eingabe (G, π, ψ) in Polynomialzeit die lexikographisch kleinste Permutation von $\psi\mathfrak{G}\pi$ bestimmt.*

Satz 2.32 (Arvind und Kurur) GI *ist in* SPP.

Beweis. Das funktionale Problem AUTO ist so definiert: Berechne für einen gegebenen Graphen G einen starken Generator der Automorphismengruppe $\mathrm{Aut}(G)$; siehe Definition 1.12 und den nachfolgenden Absatz sowie Definition 1.14 für diese Begriffe. Nach dem Resultat von Mathon [38] sind die Probleme AUTO und GI Turing-äquivalent (siehe auch [34]), d.h., $\mathtt{AUTO} \in \mathrm{FP}^{\mathtt{GI}}$ und $\mathtt{GI} \in \mathrm{P}^{\mathtt{AUTO}}$. Daher genügt es zu zeigen, dass AUTO in $\mathrm{FP}^{\mathrm{SPP}}$ liegt. Denn mit der Selbst-Lowness von SPP aus Satz 2.29 folgt dann, dass GI in $\mathrm{P}^{\mathtt{AUTO}} \subseteq \mathrm{SPP}^{\mathrm{SPP}} \subseteq \mathrm{SPP}$ liegt, und der Satz ist bewiesen.

Unser Ziel ist es also, einen $\mathrm{FP}^{\mathrm{SPP}}$-Algorithmus für AUTO zu finden. Für einen gegebenen Graphen G soll dieser Algorithmus einen starken Generator $S = \bigcup_{i=1}^{n-1} T_i$ für $\mathrm{Aut}(G)$ berechnen, wobei

$$\mathbf{id} = \mathrm{Aut}(G)^{(n)} \leq \mathrm{Aut}(G)^{(n-1)} \leq \cdots \leq \mathrm{Aut}(G)^{(1)} \leq \mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$$

der Turm von Stabilisatoren von $\mathrm{Aut}(G)$ und T_i, $1 \leq i \leq n$, eine vollständige rechte Transversale von $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$ ist. Beginnend mit dem trivialen Fall $\mathrm{Aut}(G)^{(n)} = \mathbf{id}$ bauen wir Schritt für Schritt einen starken Generator für $\mathrm{Aut}(G)^{(i)}$ auf, für fallendes i. Schließlich erhalten wir so einen starken Generator für $\mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$. Nehmen wir also an, dass ein starker Generator $S_i = \bigcup_{j=i}^{n-1} T_j$ für $\mathrm{Aut}(G)^{(i)}$ bereits gefunden ist. Wir beschreiben nun, wie der $\mathrm{FP}^{\mathrm{SPP}}$-Algorithmus eine vollständige rechte Transversale T_{i-1} von $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$ bestimmt. Dazu definieren wir die Menge

$$A = \left\{ (G, S, i, j, \pi) \;\middle|\; \begin{array}{l} S \subseteq \mathrm{Aut}(G) \text{ und } \langle S \rangle \text{ ist ein punktweiser Stabilisator von } [i] \text{ in } \mathrm{Aut}(G), \pi \text{ ist} \\ \text{eine partielle Permutation, die punktweise } [i-1] \text{ stabilisiert, und } \pi(i) = j, \\ \text{und es gibt ein } \tau \in \mathrm{Aut}(G)^{(i-1)} \text{ mit } \tau(i) = j \text{ und } \mathrm{LERC}(S, \tau) \text{ erweitert } \pi \end{array} \right\}.$$

Nach Satz 2.30 kann die lexikographisch kleinste Permutation $\mathrm{LERC}(S,\tau)$ der rechten co-Menge $\langle S\rangle\tau$ durch den Algorithmus aus Abbildung 2.9 in Polynomialzeit bestimmt werden. Die partielle Permutation π ist Teil der Eingabeinstanz (G,S,i,j,π), da wir die Menge A als Orakel benutzen wollen, um durch Präfixsuche die lexikographisch kleinste Permutation $\tau \in \mathrm{Aut}(G)^{(i-1)}$ mit $\tau(i)=j$ zu ermitteln; vgl. auch Abbildung 2.6 in Beispiel 2.17.

```
N(G,S,i,j,π) {
  Verifiziere, dass S ⊆ Aut(G)^(i);
  Rate nichtdeterministisch eine Permutation τ ∈ 𝔖_n;        // G hat n Knoten
  if (τ ∈ Aut(G)^(i-1) und τ(i) = j und τ erweitert π und τ = LERC(S,τ))
    akzeptiere und halte;
    else lehne ab und halte;
}
```

Abbildung 2.10: NP-Maschine N für Orakel A.

Abbildung 2.10 gibt eine NPTM N für das Orakel A an. Somit ist A in NP. Entscheidend ist, dass wenn $\tau(i)=j$ gilt, dann ist $\sigma(i)=j$ für jede Permutation σ in der rechten co-Menge $\langle S\rangle\tau$.

Wir zeigen nun, dass die Anzahl der akzeptierenden Pfade von N bei Eingabe (G,S,i,j,π) entweder 0 oder 1 ist, falls $\langle S\rangle = \mathrm{Aut}(G)^{(i)}$ gilt. Allgemein gilt $\mathrm{acc}_N(G,S,i,j,\pi) \in \{0, |\mathrm{Aut}(G)^{(i)}|/|\langle S\rangle|\}$.

Angenommen, (G,S,i,j,π) ist in A und $\langle S\rangle = \mathrm{Aut}(G)^{(i)}$. Gilt $\tau(i)=j$ für ein $\tau \in \mathrm{Aut}(G)^{(i-1)}$ und $j>i$, so besteht die rechte co-Menge $\langle S\rangle\tau$ aus genau den Permutationen in $\mathrm{Aut}(G)^{(i-1)}$, die i auf j abbilden. Folglich entspricht der einzige akzeptierende Pfad von $N(G,S,i,j,\pi)$ der eindeutig bestimmten lexikographisch kleinsten Permutation $\tau = \mathrm{LERC}(S,\tau)$. Ist andererseits $\langle S\rangle$ eine echte Untergruppe von $\mathrm{Aut}(G)^{(i)}$, dann kann $\mathrm{Aut}(G)^{(i)}\tau$ als die disjunkte Vereinigung von $k = |\mathrm{Aut}(G)^{(i)}|/|\langle S\rangle|$ vielen rechten co-Mengen von $\langle S\rangle$ dargestellt werden. Im Allgemeinen hat $N(G,S,i,j,\pi)$ also k akzeptierende Pfade, falls (G,S,i,j,π) in A ist, und keinen akzeptierenden Pfad sonst.

Abbildung 2.11 zeigt den FP^A-Algorithmus M^A für AUTO. Die DPOTM M stellt dabei ihrem Orakel A nur solche Fragen $q=(G,S_i,i,j,\pi)$, für die $\langle S_i\rangle = \mathrm{Aut}(G)^{(i)}$ gilt. Folglich gilt $\mathrm{acc}_N(q) \le 1$ für jede wirklich gestellte Frage q. Mit Teil 4 von Satz 2.29 folgt dann $\mathtt{AUTO} \in \mathrm{FP}^{\mathrm{SPP}}$.

Die Behauptung, dass die Ausgabe S_0 von $M^A(G)$ ein starker Generator für $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$ ist, wird durch Induktion über n gezeigt. Der Induktionsanfang ist $n-1$, und $S_{n-1} = \{\mathrm{id}\}$ erzeugt natürlich $\mathrm{Aut}(G)^{(n-1)} = \mathbf{id}$. Für den Induktionsschritt nehmen wir an, dass am Anfang der i-ten Iteration ein starker Generator S_i für $\mathrm{Aut}(G)^{(i)}$ bereits gefunden ist. Wir zeigen, dass am Ende der i-ten Iteration die Menge $S_{i-1} = S_i \cup T_{i-1}$ ein starker Generator für $\mathrm{Aut}(G)^{(i-1)}$ ist. Für jedes j mit $i+1 \le j \le n$ überprüft die Frage "$(G,S_i,i,j,\hat\pi) \in A$?", ob es in $\mathrm{Aut}(G)^{(i-1)}$ eine Permutation gibt, die i auf j abbildet. Die anschließende Präfixsuche konstruiert durch geeignete Fragen an das Orakel A die lexikographisch kleinste Permutation $\hat\pi$ in $\mathrm{Aut}(G)^{(i-1)}$ mit $\hat\pi(i)=j$. Wie oben behauptet, werden dabei nur Fragen q mit $\mathrm{acc}_N(q) \le 1$ an A gestellt, weil S_i ein starker Generator für $\mathrm{Aut}(G)^{(i)}$ ist, also $\langle S_i\rangle = \mathrm{Aut}(G)^{(i)}$ gilt. Nach Konstruktion ist am Ende der i-ten Iteration T_{i-1} eine vollständige rechte Transversale von $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$. Folglich ist $S_{i-1} = S_i \cup T_{i-1}$ ein starker Generator für $\mathrm{Aut}(G)^{(i-1)}$. Schließlich ist nach n Iterationen ein starker Generator S_0 für $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$ gefunden. ∎

Aus den ersten beiden Aussagen von Satz 2.29 ergibt sich sofort Korollar 2.33.

Korollar 2.33 GI *ist low für* SPP *und für* PP, *d.h.,* $\mathrm{SPP}^{\mathtt{GI}} = \mathrm{SPP}$ *und* $\mathrm{PP}^{\mathtt{GI}} = \mathrm{PP}$.

```
M^A(G) {
  Setze T_i := {id} für alle i, 0 ≤ i ≤ n − 2;                    // G hat n Knoten
        // T_i wird eine vollständige rechte Transversale von Aut(G)^(i+1) in Aut(G)^(i) sein
  Setze S_i := ∅ für alle i, 0 ≤ i ≤ n − 2;
  Setze S_{n−1} := {id};                          // S_i wird ein starker Generator für Aut(G)^(i) sein
  for (i = n − 1, n − 2, ..., 1) {
        // am Anfang der i-ten Iteration ist S_i bereits gefunden und S_{i−1} wird nun berechnet
    Sei π : [i − 1] → [n] die partielle Permutation mit π(a) = a für alle a ∈ [i − 1]
                                  // für i = 1 ist π die nirgends definierte partielle Permutation
    for (j = i + 1, i + 2, ..., n) {
      Setze π̂ := πj, d.h., π̂ erweitert π um das Paar (i, j) mit π̂(i) = j;
      if ((G, S_i, i, j, π̂) ∈ A) {
       // Präfixsuche konstruiert die kleinste Permutation in Aut(G)^(i−1), die i auf j abbildet
        for (k = i + 1, i + 2, ..., n) {
          Finde das Element ℓ außerhalb des Bildes von π̂ mit (G, S_i, i, j, π̂ℓ) ∈ A;
          π̂ := π̂ℓ;
        }                                          // jetzt ist π̂ eine totale Permutation in S_n
        T_{i−1} := T_{i−1} ∪ π̂;
      }
    }     // jetzt ist T_{i−1} eine vollständige rechte Transversale von Aut(G)^(i) in Aut(G)^(i−1)
    S_{i−1} := S_i ∪ T_{i−1};
  }
  return S_0;                          // S_0 ist ein starker Generator für Aut(G) = Aut(G)^(0)
}
```

Abbildung 2.11: FP$^{\text{SPP}}$-Algorithmus M^A für AUTO.

Übungsaufgaben

Aufgabe 2.5.1 Nach Definition 2.20 gilt: $A \leq_{\mathrm{T}}^{\mathrm{p}} B \iff A \in \mathrm{P}^B$. Zeige: $A \leq_{\mathrm{T}}^{\mathrm{p}} B \iff \mathrm{P}^A \subseteq \mathrm{P}^B$.

Aufgabe 2.5.2 Zeige, dass die in Beispiel 2.17 definierte Menge Pre-Iso in NP liegt und dass die in Abbildung 2.6 dargestellte Maschine N eine DPOTM ist, also in Polynomialzeit arbeitet.

Aufgabe 2.5.3 Bestimme die Menge der Isomorphismen $\mathrm{Iso}(\widehat{G}, \widehat{H})$ für die in Beispiel 2.19 definierten Graphen $\widehat{G}$ und $\widehat{H}$.

Aufgabe 2.5.4 Welche der folgenden Klassen sind "*promise*"-Klassen: NP und coNP, RP und coRP, AM und coAM, MA und coMA? Haben "*promise*"-Klassen vollständige Probleme? Begründe deine Antwort.

Probleme

Problem 2.1 Eine *starke NPOTM* ist eine NPOTM mit drei Typen von Endzuständen, d.h., die Menge F der Endzustände von M wird zerlegt in F_a, F_r und $F_?$, so dass gilt: Ist $x \in A$, so hat $M^B(x)$ einen Pfad, der mit einem Zustand aus F_a endet, und keinen Pfad, der mit einem Zustand aus F_r endet. Ist

jedoch $x \notin A$, so hat $M^B(x)$ hat einen Pfad, der mit einem Zustand aus F_r endet, und keinen Pfad, der mit einem Zustand aus F_a endet. In beiden Fällen darf $M^B(x)$ auf gewissen Pfaden mit einem Zustand aus $F_?$ enden, der der Antwort "weiß ich nicht" entspricht. Starke NPOTM sind also Maschinen, die niemals lügen. Zeige die folgenden beiden Aussagen:

(a) $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B \iff$ es gibt eine starke NPOTM M mit $A = L(M^B)$.

(b) $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B \iff \mathrm{NP}^A \subseteq \mathrm{NP}^B$.

Hinweis: Verallgemeinere Aufgabe 2.5.1.

Problem 2.2 Beweise die Aussagen der Sätze 2.22 und 2.24. (Vorsicht: Einige sind schwierig!)

Problem 2.3 Modifiziere den Beweis von Satz 2.30 so, dass sich Korollar 2.31 ergibt.

Notizen zum Kapitel

Teile der Kapitel 1 und 2 beruhen auf dem Buch [52]. Was hier nur in stark komprimierter Form dargestellt werden konnte, findet man dort umfassend und in allen technischen Details beschrieben, mit umfangreicheren Beispielen, größeren Zahlen, schöneren und zahlreicheren Abbildungen, genaueren Erläuterungen und detaillierteren Beweisen. So findet man in [52] die Beweise, auf die hier verzichtet wurde, etwa für die Sätze 2.22, 2.24 und 2.29 und für Lemma 2.26. Der Vorteil der vorliegenden Kapitel 1 und 2 liegt dagegen darin, kurz, knapp und kompakt und dennoch klar und korrekt zu sein.

Mehr Hintergrund zur Komplexitätstheorie findet man z.B. in den Büchern [43, 25, 69, 70]. Eine wertvolle Quelle für die Theorie der NP-Vollständigkeit ist noch immer der Klassiker [15] von Garey und Johnson. Der Beweis von Satz 2.14 findet sich auch in anderen Büchern, z.B. in [15] und in [43]. Die $\leq_{\mathrm{T}}^{\mathrm{p}}$-Reduzierbarkeit wurde von Cook [9] und die $\leq_{\mathrm{m}}^{\mathrm{p}}$-Reduzierbarkeit von Karp [31] eingeführt. Ladner, Lynch und Selman [36] leisteten eine umfassende und tiefgehende Arbeit beim Studium komplexitätsbeschränkter Reduzierbarkeiten. Aufgabe 2.5.1 sowie Problem 2.1 gehen auf Selman [60] zurück.

Dantsin et al. [11] erzielten die bisher beste obere Schranke $O(1.481^n)$ der deterministischen Zeitkomplexität von k-SAT für $k \geq 3$. Der hier vorgestellte probabilistische Algorithmus von Schöning beruht auf der Idee einer „*eingeschränkten lokalen Suche mit Wiederholung*" [56]. Für k-SAT mit $k \geq 4$ ist der Algorithmus von Paturi et al. [44] noch etwas besser. Der derzeit beste probabilistische Algorithmus für 3-SAT und 4-SAT geht auf Iwama und Tamaki [30] zurück. Ihr Algorithmus kombiniert geschickt die Algorithmen von Paturi et al. [44] und Schöning [56] und hat eine Laufzeit von $O(1.324^n)$. Für k-SAT mit $k \geq 5$ ist ihr Algorithmus nicht besser als der von Paturi et al. [44].

Tabelle 2.5 gibt eine Übersicht über die hier besprochenen und einige weitere Algorithmen für das Erfüllbarkeitsproblem. Die derzeit besten Resultate sind fett gedruckt.

Algorithmus	Typ	3-SAT	4-SAT	5-SAT	6-SAT
Backtracking	det.	$O(1.913^n)$	$O(1.968^n)$	$O(1.987^n)$	$O(1.995^n)$
Monien und Speckenmeyer [42]	det.	$O(1.618^n)$	$O(1.839^n)$	$O(1.928^n)$	$O(1.966^n)$
Dantsin et al. [11]	det.	$\boldsymbol{O(1.481^n)}$	$\boldsymbol{O(1.6^n)}$	$\boldsymbol{O(1.667^n)}$	$\boldsymbol{O(1.75^n)}$
Paturi et al. [44]	prob.	$O(1.362^n)$	$O(1.476^n)$	$\boldsymbol{O(1.569^n)}$	$\boldsymbol{O(1.637^n)}$
Schöning [56]	prob.	$O(1.334^n)$	$O(1.5^n)$	$O(1.6^n)$	$O(1.667^n)$
Iwama und Tamaki [30]	prob.	$\boldsymbol{O(1.324^n)}$	$\boldsymbol{O(1.474^n)}$	—	—

Tabelle 2.5: Laufzeiten einiger Algorithmen für das Erfüllbarkeitsproblem.

Das Graphisomorphieproblem wird umfassend im Buch von Köbler, Schöning und Torán [34] behandelt, besonders in komplexitätstheoretischer Hinsicht. Hoffman [29] untersucht gruppentheoretische

Algorithmen für GI. Gál et al. [14] zeigten, dass man aus einem partiellen Isomorphismus der Größe $O(\log n)$ für zwei isomorphe Graphen mit je n Knoten einen totalen Isomorphismus konstruieren kann. Dieses Resultat wird in Satz 2.18 optimal verbessert, der sagt, dass dafür bereits ein partieller Isomorphismus der Größe 1 genügt. Dieses Ergebnis sowie Beispiel 2.19 gehen auf die Arbeit [22] von Große, Rothe und Wechsung zurück. Die Polynomialzeit-Hierarchie wurde von Meyer und Stockmeyer [39, 66] eingeführt, die unter anderem die Aussagen von Satz 2.22 bewiesen. Schöning führte die Low- und die High-Hierarchie ein [54]. Er bewies die Aussagen von Satz 2.24 in [54, 55] und zeigte in [55], dass GI in Low_2 liegt. Köbler et al. [33, 32] erzielten die ersten Resultate hinsichtlich der Lowness von GI für probabilistische Klassen wie PP. Ihre Ergebnisse wurden von Arvind und Kurur [2] verbessert, die bewiesen, dass GI sogar in SPP liegt. Lemma 2.26 geht auf Carter und Wegman [8] zurück. SPP verallgemeinert die von Valiant [68] eingeführte Klasse UP. Diese und andere "*promise*"-Klassen wurden in einer Reihe von Arbeiten intensiv untersucht, z.B. in [23, 33, 32, 13, 48, 50, 27, 7, 2].

Der Autor dankt Uwe Schöning für seine hilfreichen Kommentare zu einer früheren Version dieses Kapitels. Insbesondere beruht die Wahrscheinlichkeitsanalyse des Algorithmus RANDOM-SAT aus Abschnitt 2.4, die die ursprüngliche Argumentation vereinfacht, auf Vortragsfolien von Uwe Schöning; eine ausführlichere Analyse kann in [57] nachgelesen werden. Außerdem sei Dietrich Stoyan, Sigurd Assing und Holger Spakowski für das Korrekturlesen früherer Versionen der Kapitel 1 und 2 und Gábor Erdélyi und Robert Stoyan für ihre Hilfe bei der Übersetzung herzlich gedankt. Die Deutsche Forschungsgemeinschaft (DFG) unterstützte den Autor unter Kennzeichen RO 1202/9-1.

Literaturverzeichnis

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. Unpublished manuscript, August 2002.

[2] V. Arvind and P. Kurur. Graph isomorphism is in SPP. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science*, pages 743–750. IEEE Computer Society Press, November 2002.

[3] L. Babai. Trading group theory for randomness. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 421–429. ACM Press, April 1985.

[4] L. Babai and S. Moran. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254–276, 1988.

[5] A. Beygelzimer, L. Hemaspaandra, C. Homan, and J. Rothe. One-way functions in worst-case cryptography: Algebraic and security properties are on the house. *SIGACT News*, 30(4):25–40, December 1999.

[6] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, February 1999.

[7] B. Borchert, L. Hemaspaandra, and J. Rothe. Restrictive acceptance suffices for equivalence problems. *London Mathematical Society Journal of Computation and Mathematics*, 3:86–95, March 2000.

[8] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[9] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, 1971.

[10] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[11] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, October 2002.

[12] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[13] S. Fenner, L. Fortnow, and S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48(1):116–148, 1994.

[14] A. Gál, S. Halevi, R. Lipton, and E. Petrank. Computing from partial solutions. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity*, pages 34–45. IEEE Computer Society Press, May 1999.

[15] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[16] O. Goldreich. Randomness, interactive proofs, and zero-knowledge—A survey. In R. Herken, editor, *The Universal Turing Machine: A Half-Century Survey*, pages 377–405. Oxford University Press, Oxford, 1988.

[17] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001.

[18] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, July 1991.

[19] S. Goldwasser. Interactive proof systems. In J. Hartmanis, editor, *Computational Complexity Theory*, pages 108–128. AMS Short Course Lecture Notes: Introductory Survey Lectures, Proceedings of Symposia in Applied Mathematics, Volume 38, American Mathematical Society, 1989.

[20] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, February 1989.

[21] S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. In S. Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, pages 73–90. JAI Press, Greenwich, 1989. A preliminary version appeared in *Proc. 18th Ann. ACM Symp. on Theory of Computing*, 1986, pp. 59–68.

[22] A. Große, J. Rothe, and G. Wechsung. Computing complete graph isomorphisms and hamiltonian cycles from partial ones. *Theory of Computing Systems*, 35(1):81–93, February 2002.

[23] J. Hartmanis and L. Hemachandra. Complexity classes without machines: On complete languages for UP. *Theoretical Computer Science*, 58(1–3):129–142, 1988.

[24] J. Håstad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, April 1988. Special issue on cryptography.

[25] L. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 2002.

[26] L. Hemaspaandra, K. Pasanen, and J. Rothe. If P $\neq$ NP then some strongly noninvertible functions are invertible. In *Proceedings of the 13th International Symposium on Fundamentals of Computation Theory*, pages 162–171. Springer-Verlag *Lecture Notes in Computer Science #2138*, August 2001.

[27] L. Hemaspaandra and J. Rothe. Unambiguous computation: Boolean hierarchies and sparse Turing-complete sets. *SIAM Journal on Computing*, 26(3):634–653, June 1997.

[28] L. Hemaspaandra and J. Rothe. Creating strong, total, commutative, associative one-way functions from any one-way function in complexity theory. *Journal of Computer and System Sciences*, 58(3):648–659, June 1999.

[29] C. Hoffman. *Group-Theoretic Algorithms and Graph Isomorphism*. *Lecture Notes in Computer Science #136*. Springer-Verlag, 1982.

[30] K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. Technical Report TR03-053, Electronic Colloquium on Computational Complexity, July 2003. 3 pages.

[31] R. Karp. Reducibilities among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, 1972.

[32] J. Köbler, U. Schöning, S. Toda, and J. Torán. Turing machines with few accepting computations and low sets for PP. *Journal of Computer and System Sciences*, 44(2):272–286, 1992.

[33] J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2:301–330, 1992.

[34] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhäuser, 1993.

[35] R. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[36] R. Ladner, N. Lynch, and A. Selman. A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1(2):103–124, 1975.

[37] A. Lenstra and H. Lenstra, Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, 1993.

[38] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979.

[39] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*, pages 125–129, 1972.

[40] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, March 2002.

[41] G. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.

[42] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[43] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[44] R. Paturi, P. Pudlák, M. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, pages 628–637. IEEE Computer Society Press, November 1998.

[45] J. Pollard. Theorems on factorization and primality testing. *Proc. Cambridge Philos. Soc.*, 76:521–528, 1974.

[46] M. Rabi and A. Sherman. An observation on associative one-way functions in complexity theory. *Information Processing Letters*, 64(5):239–244, 1997.

[47] M. Rabin. Probabilistic algorithms for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[48] R. Rao, J. Rothe, and O. Watanabe. Upward separation for FewP and related classes. *Information Processing Letters*, 52(4):175–180, April 1994. Corrigendum appears in the same journal, 74(1–2):89, 2000.

[49] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[50] J. Rothe. A promise class at least as hard as the polynomial hierarchy. *Journal of Computing and Information*, 1(1):92–107, April 1995. Special Issue: *Proceedings of the Sixth International Conference on Computing and Information*, CD-ROM ISSN 1201-8511, Trent University Press.

[51] J. Rothe. Some facets of complexity theory and cryptography: A five-lecture tutorial. *ACM Computing Surveys*, 34(4):504–549, December 2002.

[52] J. Rothe. *Complexity Theory and Cryptology. An Introduction to Cryptocomplexity*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 2004. To appear.

[53] A. Salomaa. *Public-Key Cryptography*, volume 23 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, second edition, 1996.

[54] U. Schöning. A low and a high hierarchy within NP. *Journal of Computer and System Sciences*, 27:14–28, 1983.

[55] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1987.

[56] U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 410–414. IEEE Computer Society Press, October 1999.

[57] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001. In German.

[58] U. Schöning. *Ideen der Informatik*. Oldenbourg Verlag, München, Wien, 2002. In German.

[59] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. *Theoretical Computer Science*, 32(4):615–623, 2002.

[60] A. Selman. Polynomial time enumeration reducibility. *SIAM Journal on Computing*, 7(4):440–457, 1978.

[61] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992.

[62] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):657–715, 1949.

[63] S. Singh. *The Code Book. The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Fourth Estate, London, 1999.

[64] R. Solovay and V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977. Erratum appears in the same journal, 7(1):118, 1978.

[65] D. Stinson. *Cryptography Theory and Practice*. CRC Press, Boca Raton, 1995.

[66] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1977.

[67] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, ser. 2*, 42:230–265, 1936. Correction, *ibid*, vol. 43, pp. 544–546, 1937.

[68] L. Valiant. The relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20–23, 1976.

[69] K. Wagner and G. Wechsung. *Computational Complexity*. D. Reidel Publishing Company, 1986. Distributors for the U.S.A. and Canada: Kluwer Academic Publishers.

[70] G. Wechsung. *Vorlesungen zur Komplexitätstheorie*, volume 32 of *Teubner-Texte zur Informatik*. Teubner, Stuttgart, 2000. In German.

[71] S. Zachos and H. Heller. A decisive characterization of BPP. *Information and Control*, 69:125–135, 1986.