

Skript zur Vorlesung Informatik III im WS 08/09
von Dr. Jörn Müller-Quade

Ausgearbeitet von Simon Stroh

26. Februar 2009

Zusammenfassung

Dieses Skript ist parallel zum zweiten Teil der Vorlesung Informatik 3 im WS 2008/2009 entstanden, es enthält also all das, was dort ab 2009 behandelt wurde. Die Themen davor sind durch das [Skript zur Informatik 3 Vorlesung von Prof. Prautzsch](#) abgedeckt.

Inhaltsverzeichnis

1	Komplexitätstheorie	3
1.1	Übersicht	3
1.1.1	Was ist das und wozu braucht man es?	3
1.1.2	Die Klasse P	4
1.1.3	Die Klasse NP	5
1.1.4	Ein Ausflug in andere Welten	7
1.1.5	NP Paradebeispiel: SAT	7
1.1.6	Polynomielle Reduzierbarkeit	7
1.1.7	Machine independence	8
1.2	NP -Vollständigkeit	9
1.2.1	Beispiele für NP -Probleme	9
1.2.2	NP Vollständigkeit	9
1.2.3	Satz von Cook	9
1.2.4	Der Schmale Grat	12
1.2.5	$P \stackrel{?}{=} NP$	12
1.3	NP -vollständige Probleme	13
1.3.1	NP -Vollständigkeit zeigen	13
1.3.2	3-SAT	13
1.3.3	Graph 3-Färbbarkeit	14
1.3.4	Vertex Cover	17
1.3.5	Weitere Aspekte	18
1.4	Probabilistische Komplexitätsklassen	19
1.4.1	Counting Classes	19
1.4.2	Probabilistische Klassen	20
1.4.3	Die Klasse R	20
1.4.4	Polynomprodukt Inäquivalenz	21
1.4.5	ZPP	22
1.4.6	PP und BPP	22
1.5	Ein bisschen Kryptographie	23
1.5.1	Randomisierte Reduktion	24
1.5.2	Das Rabin Kryptosystem	24
1.5.3	Brassards Theorem	26
2	Informationstheorie	28
2.1	Einführung	28
2.1.1	Was ist Information?	28
2.1.2	Anwendungen	29
2.1.3	Formale Definitionen	29

2.1.4	Der Entropiebegriff	30
2.1.5	Occams Razor	31
2.1.6	Entropie	31
2.1.7	Weitere Entropie-Zusammenhänge	32
2.1.8	Transinformation	33
2.2	Quellkodierungen	33
2.2.1	Verlustbehaftete Kompression	33
2.2.2	Kodierungen	33
2.2.3	Präfix-Codes	34
2.2.4	Erwartungswert	35
2.2.5	Shannon-Fano	35
2.2.6	Der Huffman-Code	36
2.2.7	Optimalität von Huffman	38
2.3	Kanalkodierungen	39
2.3.1	Hamming-Distanz	39
2.3.2	Maximum-likelihood-decoding	40
2.3.3	Rate	40
2.3.4	Shannons Theorem	40
2.4	Beispiele für Kanalkodierungen	43
2.4.1	Block-Codes	43
2.4.2	Parity-Code	46
2.4.3	Hamming-Codes	47
2.5	Kryptographie und Informationstheorie	48
2.5.1	Einfache Kryptographische Verfahren	48
2.5.2	Perfekte Sicherheit	49
2.5.3	Zusammenhang zur Entropie	51
2.5.4	Bit-Commitments	52
2.5.5	Bit Commitment-Verfahren	53
2.5.6	Code Obfuscation	54
2.6	Philosophisches: Die vier Grundfragen der Philosophie	55

Kapitel 1

Komplexitätstheorie

1.1 Übersicht

1.1.1 Was ist das und wozu braucht man es?

Früher fanden Mathematiker alle endlichen Probleme langweilig. Also Probleme, für die nur endlich viele Lösungen in Frage kommen. Es war ja, wenn auch lästig, möglich alle Möglichkeiten durchzuprobieren. Diese „brute-force“ Lösung ist jedoch eventuell sehr aufwendig. Heutzutage, mithilfe von Computern, ist so etwas auch eine praktikable Lösung, das hängt aber auch stark davon ab, wie kompliziert ein Problem ist. Damit setzt sich die Komplexitätstheorie auseinander, sie macht Aussagen über die „Kompliziertheit“ oder **Komplexität** von Problemen.

Anwendung: Kryptographie

Meistens ist es unangenehm festzustellen, dass ein Problem eine hohe Komplexität hat, das bedeutet ja, dass es schwer zu lösen ist und meist will man Probleme ja möglichst einfach und effizient lösen. Die einzige Anwendung, in der man tatsächlich nach schwierigen Problemen sucht und deren Schwierigkeit als positiv bewertet, ist die Kryptographie. Die Sicherheit aller modernen kryptographischen Verfahren beruht darauf, dass das Entschlüsseln ohne den Schlüssel zu kennen ein sehr kompliziertes, also komplexes, Problem ist.

Komplexität

Was ist also Komplexität? Zunächst unterscheidet man dabei zwischen zwei grundsätzlichen Arten von Komplexität, der **Raumkomplexität** und der **Zeitkomplexität**. Man untersucht dabei jeweils, wie viel Raum bzw. Zeit für die Berechnung benötigt wird. Dabei interessiert man sich nicht für einzelne Instanzen, das heißt konkrete Probleme für eine bestimmte Eingabe. Diese sind in einer festen Zeit lösbar. Man untersucht stattdessen die **Abhängigkeit von der Länge der Eingabe**. Hier muss man beachten, dass man diese Abhängigkeit **asymptotisch** betrachtet. Das heißt, man untersucht das Grenzverhalten für sehr große Eingaben. Das ist notwendig, da man ansonsten etwa einen Algorithmus entwerfen könnte, der eine große Tabelle hat, in der die Lösungen für alle

Probleme bis zu einer bestimmten Eingabelänge stehen. Für konkrete Zahlen hat dieser Algorithmus natürlich die kleinstmögliche Zeitkomplexität.

Beispiel: Multiplikation

Ein Problem, für das es verschiedene unterschiedlich effiziente Algorithmen gibt, ist zum Beispiel das Problem, zwei ganze Zahlen zu multiplizieren. Hier ist es so, dass die Antwort auf die Frage, welcher dieser Algorithmen am wenigsten Zeit benötigt um ein solches Problem zu lösen, von der Eingabelänge abhängig ist. Für kleine Zahlen ist die [Multiplikation, die man üblicherweise in der Grundschule lernt](#), ausreichend, ab einem bestimmten Punkt ist jedoch z.B. der [Karatsuba-Algorithmus](#) effizienter. Für wirklich sehr große Eingaben hingegen, gibt es einen noch effizienteren Algorithmus, den [Schönhage-Strassen-Algorithmus](#). Auch wenn dieser für die meisten praktischen Anwendungen langsamer als die anderen ist, da die Zahlen nicht groß genug sind, so ist er trotzdem im Sinne der Komplexitätstheorie von diesen der beste Algorithmus.

Zurück zur Komplexität

Man kann die Komplexität weiterhin noch differenzierter für alle Eingaben untersuchen. Dabei kann man etwa untersuchen:

- Den „worst case“, also der maximale Aufwand für eine Eingabe bestimmter Länge.
- Den „average case“, der Mittelwert über alle Eingaben. Das ist interessant, wenn etwa alle Eingaben bis auf wenige sehr effizient lösbar sind. Solche Zusammenhänge kann man beim „worst case“ nicht erkennen.
- „mindestens diesen Aufwand“, mit überwältigender Wahrscheinlichkeit.

In dieser Vorlesung werden wir uns dabei nur mit dem ersten Fall, „worst case“ auseinandersetzen. Es existieren weiterhin noch probabilistische Algorithmen, die manche Entscheidungen zufällig treffen.

Die Erkenntnisse über die Komplexität von Problemen nutzt man nun dazu, Problemen verschiedene Komplexitätsklassen zuzuordnen, die wichtigsten darunter sind die Klassen P und NP .

1.1.2 Die Klasse P

Definition 1.1.2.1. Entscheidungsprobleme sind Probleme der Form: Sei eine Sprache \mathcal{L} gegeben, sowie ein beliebiges x . Entscheide nun: $x \in \mathcal{L}$?

Definition 1.1.2.2. P ist die Menge der Sprachen, für die es eine deterministische Turingmaschine gibt, die in höchstens $p(|x|)$ Schritten entscheiden kann ob $x \in \mathcal{L}$. Dabei ist p ein beliebiges Polynom.

Beispiel 1.1.2.3. Ein Beispiel für eine solche Sprache sind etwa die Primzahlen.

Das heißt also, es existiert ein Algorithmus, der nach polynomial vielen Schritten entscheiden kann, ob eine Zahl eine Primzahl ist oder nicht. Dieser Algorithmus ([AKS-Methode](#)), bzw die Frage nach seiner Existenz, war dabei lange Zeit eines der schweren, ungelösten Probleme. Er wurde erst 2002 entdeckt.

Das ist interessant, da ein solcher Algorithmus zum Beispiel für das Faktorisieren von großen Zahlen noch nicht gefunden wurde. Will man etwa n faktorisieren, so könnte man zunächst meinen, dass man hier einfach eine Probedivision mit allen Zahlen kleiner n durchführen könnte. Deren Anzahl steigt ja linear (also auch polynomial) mit n . Das Problem ist, dass wir die *Länge* von n betrachten und sich mit jeder weiteren (binären) Ziffer die Anzahl der möglichen Zahlen kleiner n verdoppelt. Die Anzahl der Zahlen kleiner n wächst also exponentiell mit der Länge von n . Die Darstellung von Zahlen spielt also auch eine Rolle.

Kritik

An dieser Einteilung kann man kritisieren, dass sie recht grob ist. Das Polynom n^{17} ist eben auch ein Polynom und Funktionen mit einer Komplexität von n^3 sind in der Praxis meist schon zu ineffizient.

Verwechslungsgefahr

P ist nicht zu verwechseln mit der gleich benannten Klasse der Funktionsproblemen, die Funktionen enthält, die in polynomialer Zeit terminieren. Aussagen der Form „jener Algorithmus liegt in NP“ sind also nicht sinnvoll.

Weitere Klassen

Es gibt viele weitere Komplexitätsklassen, unter anderem etwa

- *EXPTIME*. So wie P , nur dass man verlangt, dass die Turingmaschine in $2^{p(|x|)}$ Schritten terminiert.
- *PSPACE*. Die Sprachen, die mit polynomialem Platzbedarf entscheidbar sind
- *IP*. Die Sprachen, für die ein interaktiver Beweis polynomialer Länge existiert

Dabei ist zum Beispiel schon gezeigt worden, dass $IP = PSPACE$. Es gibt unzählige weitere Beispiele, für die meisten ist noch nicht viel gezeigt worden, mehr dazu findet man im [Complexity Zoo](#).

1.1.3 Die Klasse \mathcal{NP}

Die meisten Leute haben sicher schon irgendwo die recht populäre Formel gesehen:

$$P \stackrel{?}{=} \mathcal{NP}$$

P kennen wir ja nun, was ist also \mathcal{NP} genau? Wie P ist es natürlich eine Menge an Sprachen. Dabei gibt es mehrere mögliche Definitionen, hier werden nun 3 verschiedene Varianten vorgestellt:

Definition 1.1.3.1. \mathcal{NP} ist die Klasse aller Sprachen, die von einer nichtdeterministischen Turingmaschine in polynomialer Zeit akzeptiert werden.

Das \mathcal{N} von \mathcal{NP} steht hier also für nichtdeterministische Turingmaschinen. Wichtig zu realisieren ist, dass für die Laufzeit nichts verlangt ist, wenn die TM nicht akzeptiert. Es wird nur verlangt, dass nur die Elemente der Sprache akzeptiert werden.

Orakelmaschinen

Für die nächste Definition brauchen wir noch einen neuen Begriff, den wir allerdings, da die Definition eher zur Veranschaulichung dient, nur informell formulieren wollen.

Definition 1.1.3.2. Eine **Orakelmaschine** ist eine Turingmaschine, die Informationen von einem „Orakel“ beziehen kann. Das Orakel schreibt dabei zunächst auf einen separaten Bandabschnitt, irgendeine Zeichenfolge endlicher Länge. Die Turingmaschine kann nun die Ausgabe des Orakels einlesen und diese als Lösung von Problemen annehmen. Da die Ausgabe des Orakels nichtdeterministisch ist, muss die Turingmaschine diese Lösung jedoch in der verfügbaren Rechenzeit überprüfen. Eine Orakelmaschine akzeptiert eine Eingabe dabei, wenn es eine Antwort des Orakels gibt, mit der die Maschine akzeptiert. Näheres, sowie eine formale Definition findet sich bei [GAREY, S. 30].

Damit kommen wir zur nächsten Definition:

Definition 1.1.3.3. \mathcal{NP} ist die Menge der Sprachen, die in polynomialer Zeit von Orakelmaschinen entschieden werden können.

Hier ist der nichtdeterministische Anteil der ersten Definition durch das Orakel repräsentiert.

Die letzte Definition schließlich, soll die sein mit der wir in Zukunft hauptsächlich arbeiten möchten.

Definition 1.1.3.4. Eine Sprache \mathcal{L} ist genau dann in \mathcal{NP} , wenn es eine Relation

$$R_{\mathcal{L}} \subseteq \mathcal{L} \times \{1, 0\}^*$$

gibt, wobei

$$R_{\mathcal{L}} \in P \text{ und } x \in \mathcal{L} \iff \exists y : (x, y) \in R_{\mathcal{L}}.$$

Weiterhin muss $|y|$ beschränkt sein durch $p(|x|)$, wobei wieder p ein beliebiges Polynom ist. Dabei heißt y Zeuge.

Das y , welches hier Zeuge genannt wird, ist dabei genau das, was das Orakel aus 1.1.3.3 errät.

Es ist weiterhin, wie schon erwähnt, wichtig zu verstehen, dass es bei diesem Begriff eine Asymmetrie gibt: (am Beispiel der Orakeldefinition:) Es wird nur verlangt, dass es möglich ist eine korrekte Ausgabe des Orakels in Polynomialer Zeit zu akzeptieren, nicht was bei falschen Ausgaben passiert. Für diese Fälle ist nichts verlangt, also auch keinerlei besonderes Verhalten für die TM

coNP = NP?

Man definiert $\text{coNP} = \{\mathcal{L} \mid \mathcal{L}^C \in \mathcal{NP}\}$. Also die Menge der Sprachen, deren Komplemente in \mathcal{NP} liegen. Ein weiteres interessantes und offenes Problem stellt die Frage dar, ob $\text{coNP} = \mathcal{NP}$. Allgemein wird angenommen, dass dies nicht der Fall ist. Es existiert auch ein Zusammenhang zu dem $P \stackrel{?}{=} \mathcal{NP}$ Problem: Sei \mathcal{L} eine Sprache, ist $\mathcal{L} \in P$, so ist auch $\mathcal{L}^C \in P$. Daher würde aus $P = \mathcal{NP} \Rightarrow \text{coNP} = \mathcal{NP}$ folgen.

1.1.4 Ein Ausflug in andere Welten

1995 hat Russel Impagliazzo fünf verschiedene „Welten“ vorgestellt, in denen wir leben könnten, abhängig davon, welche Antworten auf die noch ausstehenden Probleme sich als korrekt erweisen. Das hat vor allem Einfluss auf alle kryptographischen Ergebnisse. Diese Welten sind:

1. *Algorithmica* Hier gilt: $P = NP$
Daraus folgt vor allem, es existiert kein sicheres kryptographisches Verfahren, bei dem die Schlüssellänge kürzer als die Nachricht ist. (One Time Pad funktioniert, siehe 2.5.1)
2. *Heuristica* Hier gilt: $P \neq NP$ aber jedes NP Problem ist im „average case“ in polynomialer Zeit lösbar
Daraus folgt, ironischer Weise, dass das Finden schwerer Probleme selbst ein schweres Problem ist. Wollte man hier ein effizientes sicheres kryptographisches Verfahren finden, so müsste man zunächst ein solches Problem lösen.
3. *Pessiland* Hier gilt: $P \neq NP$ aber keine Einwegfunktionionen existieren
Hier kann man zwar leicht schwere Probleme finden, aber nicht leicht schwere Probleme finden zu denen man die Lösung kennt, was für Kryptographie notwendig ist.
4. *Minicrypt* Hier gilt: Einwegfunktionen existieren, aber keine Public-Key Kryptographie
Normale Kryptographie hingegen ist in dieser Variante möglich
5. *Cryptomania* Hier gilt: PK-Kryptographie existiert
Zur Zeit verhalten wir uns, als wären wir in dieser Welt, wir haben PK-Kryptographie, die bisher noch nicht gebrochen wurde. Die allgemeine Meinung ist, dass die Wahrscheinlichkeit recht hoch ist, dass wir uns in dieser Welt befinden.

1.1.5 NP Paradebeispiel: SAT

Ein Paradebeispiel für eine Sprache die in NP liegt, ist die Sprache der erfüllbaren booleschen Formeln, also diejenigen booleschen Formeln, für die man die Variablen so belegen kann, dass der gesamte Term **true** wird. Hat man einen solchen Ausdruck in KNF (konjunktiven Normalform) gegeben, sowie eine Belegung der Variablen, so dass der Term zu **true** ausgewertet wird, kann man in polynomialer Zeit überprüfen, ob die Belegung korrekt ist. Hingegen ist der Beweis dafür, dass ein Ausdruck nicht erfüllbar ist sehr schwer. Man muss für alle Belegungen zeigen, dass diese den Term nicht erfüllen.

Die Einschränkung auf Ausdrücke in KNF ist zulässig, da wir die „worst case“ Komplexität betrachten und es sich zeigen lässt, dass alle anderen Fälle einfacher sind.

1.1.6 Polynomielle Reduzierbarkeit

Definition 1.1.6.1. Eine Sprache \mathcal{A} (also ein Entscheidungsproblem) heißt *polynomiell reduzierbar* (oft auch engl.: *many-one*) auf eine Sprache \mathcal{B} , wenn

es eine Funktion $f : \mathcal{A} \rightarrow \mathcal{B}$ gibt, die in polynomialer Zeit berechenbar ist und wenn

$$x \in \mathcal{A} \iff f(x) \in \mathcal{B}$$

Man schreibt suggestiv $\mathcal{A} \leq \mathcal{B}$ (oder $\mathcal{B} \geq \mathcal{A}$) um auszudrücken, dass das leichtere Problem \mathcal{A} reduzierbar ist auf das schwerere Problem \mathcal{B} .

Läge nun \mathcal{B} in P , so wäre auch $\mathcal{A} \in P$, da es ja in polynomialer Zeit auf ein anderes Problem reduzierbar ist, das wiederum selbst in polynomialer Zeit entschieden werden kann. Desweiteren ist es wichtig anzumerken, dass diese Relation transitiv ist. Das heißt also aus $\mathcal{A} \leq \mathcal{B}$ sowie $\mathcal{B} \leq \mathcal{C}$ folgt $\mathcal{A} \leq \mathcal{C}$.

1.1.7 Machine independence

Wir verwenden in der Komplexitätstheorie grundsätzlich die Turingmaschine als Rechnermodell. Die Frage stellt sich, ob diese Entscheidung gerechtfertigt ist. Könnte man nicht vielleicht statt Turingmaschinen etwa [Analog-](#) oder [Quantencomputer](#) verwenden, mit deren Hilfe Probleme, die in \mathcal{NP} liegen, in polynomialer Zeit entschieden werden könnten? Zumindest für Analogcomputer gibt es folgendes Gegenargument: Auch wenn Analogcomputer einen Vorgang genau simulieren können, so haben sie immer endliche Ablesepräzision. Man kann also einen Analogcomputer immer genauer mit einer Turingmaschine simulieren, als man Ergebnisse von ihm ablesen kann. Man stelle sich als Beispiel für einen Analogcomputer etwa ein kleines, mechanisches Modell eines Roboters vor. An diesem kann man nun etwa beobachten wie sich der echte Roboter bewegen lässt und ob er zum Beispiel durch ein Tür passt. Das Ergebnis kann man aber eben nicht beliebig genau ablesen. Interessanterweise ist unser Gehirn auch ein solcher Analogrechner.

Church-Turing-These

Von der Church-Turing-These gibt es zwei Varianten. Die einfache Church-Turing-These sagt aus

Theorem 1.1.7.1. Jedes Rechnermodell ist von einer Turingmaschine simulierbar.

Es wird allgemein angenommen, dass diese These gilt, die Gültigkeit dieser These rechtfertigt, dass wir unsere Betrachtungen auf Turingmaschinen beschränken. Es ist allerdings noch nichts über die Effizienz dieser Simulation gesagt. Das heißt vor allem, dass ein anderes Rechnermodell eventuell effizienter Probleme lösen könnte als die Turingmaschine. Das wäre schlecht, da Probleme, die wir als kompliziert ansehen, eventuell auf einem anderen Rechner nicht kompliziert sind. Das führt uns zur starken Church-Turing-These, diese besagt

Theorem 1.1.7.2. Jedes Rechnermodell ist in polynomialer Zeit von einer Turingmaschine simulierbar

Seit dem Aufkommen der Idee von Quantencomputern gibt es allerdings Kritik an dieser These. Es wird allerdings allgemein angenommen, dass sogar die starke Church-Turing-These gilt.

1.2 \mathcal{NP} -Vollständigkeit

1.2.1 Beispiele für \mathcal{NP} -Probleme

Zunächst ein paar Beispiele für Probleme, die in \mathcal{NP} liegen. Für alle diese Probleme erkennt man leicht die entsprechende Struktur: Hat man einen Zeugen gegeben, so ist ein einzelnes Entscheidungsproblem jeweils leicht verifizierbar.

1. SAT, siehe 1.1.5

2. Graph 3-Färbbarkeit

Ein Graph ist **3-Färbbar**, wenn man mit 3 Farben jedem Knoten eine Farbe zuordnen kann, so dass keine zwei benachbarten Knoten die selbe Farbe haben.

3. Faktorisierung

Hier ist es wichtig zu realisieren, dass das Faktorisieren selbst kein Entscheidungsproblem ist. Man muss erst das entsprechende Problem finden. In diesem Fall ist das zugehörige Entscheidungsproblem: Hat eine Zahl n einen Primfaktor $p \in \mathbb{N}$ mit $p \leq k$. Die Sicherheit des RSA-Verfahrens basiert darauf, dass dies ein schweres Problem ist.

1.2.2 \mathcal{NP} Vollständigkeit

Definition 1.2.2.1. Eine Sprache $\mathcal{L} \in \mathcal{NP}$ heißt **\mathcal{NP} -vollständig** (engl. \mathcal{NPC} , \mathcal{NP} -Complete), wenn für alle $\mathcal{L}' \in \mathcal{NP}$ gilt, $\mathcal{L}' \leq \mathcal{L}$

Definition 1.2.2.2. Eine beliebige Sprache die diese Bedingung erfüllt, aber nicht zwangsläufig selbst in \mathcal{NP} liegt, nennt man **\mathcal{NP} -hart** oder **\mathcal{NP} -hard**.

Das heißt also, um zu zeigen, dass ein Problem \mathcal{NP} -vollständig ist, muss man zeigen, dass *alle* Probleme in \mathcal{NP} auf dieses konkrete Problem polynomiell reduzierbar sind. Zunächst stellt sich die Frage ob solche Sprachen überhaupt existieren. Die Antwort darauf ist „ja“, es ist Stephan A. Cook 1971 gelungen das zu beweisen. Er zeigte folgenden Satz:

1.2.3 Satz von Cook

Satz 1.2.3.1. $\text{SAT} \in \mathcal{NPC}$

Beweis

Wir gehen davon aus das $\text{SAT} \in \mathcal{NP}$, siehe dazu auch 1.1.5.

Nun müssen wir für ein beliebiges Problem aus \mathcal{NP} zeigen, dass dieses in polynomialer Zeit auf SAT reduzierbar ist. Sei also \mathcal{L} eine beliebige Sprache mit $\mathcal{L} \in \mathcal{NP}$, sowie $M_{\mathcal{L}}$ die Orakelmaschine (1.1.3.2), die \mathcal{L} entscheiden kann. Wir möchten nun einen booleschen Ausdruck konstruieren, der überprüft ob ein einzelner Turingschritt gültig ist. Wir definieren uns Variablen der Form q_{i,t_j} . Dabei soll $q_{i,t_j} = 1$ sein, wenn sich $M_{\mathcal{L}}$ zum Zeitpunkt t_j im Zustand q_i befindet, ansonsten $q_{i,t_j} = 0$. Weiterhin definieren wir Variablen der Form s_{i,t_j} , um das Band zu beschreiben. Dabei ist s_{i,t_j} der Inhalt des Bandes an der Stelle i , zum Zeitpunkt t_j . Dabei ist wichtig, dass höchstens polynomial viele Bandfelder beschrieben werden können, da $M_{\mathcal{L}}$ nur polynomial viele Schritte hat

um das Band zu beschreiben und nur eine konstante Anzahl an Feldern vorher vom Orakel beschrieben wurden. Weiterhin kann es nach Definition auch nur polynomial viele Zeitpunkte geben, an denen $M_{\mathcal{L}}$ aktiv ist. Nun kann man das Akzeptieren von $M_{\mathcal{L}}$ folgendermaßen aufschreiben:

$$\begin{array}{c} q_{0,t_0}, q_{1,t_0}, \dots, q_{n,t_0}, s_{0,t_0}, \dots, s_{m,t_0} \\ \vdots \\ q_{1,t_s}, q_{2,t_s}, \dots, q_{n,t_s}, s_{1,t_s}, \dots, s_{m,t_s} \end{array}$$

Hier ist jede Zeile ein Zeitpunkt und jede Spalte ein Zustand, bzw. ein Bandfeld. Der Anfangszeitpunkt ist dabei t_0 , der Endzeitpunkt t_s . Nun können wir Ausdrücke formulieren, die mithilfe des Zustandes der Maschine zu einem bestimmten Zeitpunkt jeweils die Zustände zum nächsten Zeitpunkt beschreiben. Zunächst legen wir fest, dass $q_{0,t_0} = 1$ und $q_{i,t_1} = 0$ für $i \neq 0$. Also ist q_0 der Anfangszustand. Da die TM deterministisch ist, kann es nur einen geben. Weiterhin seien die Bandfelder $s_{0,t_0}, \dots, s_{j,t_0}$ mit der Eingabe belegt, sowie die Felder $s_{j+1,t_0}, \dots, s_{m,t_0}$ mit der Ausgabe des Orakels. Im weiteren möchten wir die Turingmaschine so modellieren, dass diese nur auf den Bandinhalt an der Stelle s_0 zugreift. Wir bewegen nicht den Schreibkopf, sondern rotieren stattdessen das Band, wenn der Kopf sich also nach links bewegen würde, verschieben wir stattdessen den Bandinhalt nach rechts. Weiterhin definieren wir die Variablen (c_0, c_1) , die für den „Befehl“ der Turingmaschine stehen, den Lese-/Schreibkopf zu bewegen. Dabei soll jeweils sein:

$$(c_0, c_1) = \begin{cases} (0, 0), & \text{heißt nicht bewegen} \\ (1, 0), & \text{heißt nach links bewegen} \\ (0, 1), & \text{heißt nach rechts bewegen} \end{cases}$$

Im weiteren möchten wir nun für jede der definierten Variablen ein Term erstellen, der überprüft ob ein Schritt der Turingmaschine gültig ist.

Das Band

Nun überlegen wir uns einen Term, der mit den bisher definierten Variablen zunächst für ein Feld auf dem Band überprüft, ob der Schritt der Turingmaschine gültig ist.

$$\begin{array}{l|l} (\overline{c_{0,t_j}} \vee \overline{c_{1,t_j}}) \wedge & \text{Entweder } c_0 \text{ oder } c_1 \text{ muss 0 sein} \\ ((\overline{c_{0,t_j}} \wedge \overline{c_{1,t_j}}) \Rightarrow (s_{i,t_{j+1}} \iff s_{i,t_j})) \wedge & \text{Kopf bewegt sich nicht} \\ ((c_{0,t_j} \wedge \overline{c_{1,t_j}}) \Rightarrow (s_{i,t_{j+1}} \iff s_{i-1,t_j})) \wedge & \text{Kopf bewegt sich nach links} \\ ((\overline{c_{0,t_j}} \wedge c_{1,t_j}) \Rightarrow (s_{i,t_{j+1}} \iff s_{i+1,t_j})) & \text{Kopf bewegt sich nach rechts} \end{array}$$

Wir brauchen also pro Feld auf dem Band einen Term mit sieben Klauseln um zu überprüfen, ob ein Schritt gültig ist.

Zustandsübergänge

Als nächstes überlegen wir uns aus der Zustandsübergangsfunktion Terme, die überprüfen, ob ein Schritt gültig ist. Allgemein wird dabei eine Regel der Form

$$q_i \xrightarrow{v|w,x} q_b, \text{ mit } v, w \in \{1, 0\} \text{ und } x \in \{N, L, R\}$$

umgewandelt zu einem Term der Form:

$$\begin{aligned} (q_{i,t_j} \wedge (s_{0,t_j} \iff v)) &\Rightarrow (s_{0,t_{j+1}} \iff w) \wedge \overline{c_{0,t_j}} \wedge \overline{c_{1,t_j}} \wedge q_{b,t_{j+1}} \text{ für } x = N \\ (q_{i,t_j} \wedge (s_{0,t_j} \iff v)) &\Rightarrow (s_{0,t_{j+1}} \iff w) \wedge c_{0,t_j} \wedge \overline{c_{1,t_j}} \wedge q_{b,t_{j+1}} \text{ für } x = L \\ (q_{i,t_j} \wedge (s_{0,t_j} \iff v)) &\Rightarrow (s_{0,t_{j+1}} \iff w) \wedge \overline{c_{0,t_j}} \wedge c_{1,t_j} \wedge q_{b,t_{j+1}} \text{ für } x = R \end{aligned}$$

Formt man diesen Ausdruck in eine KNF um, so erhält man 4 Klauseln pro Ableitungsregel.

Eindeutigkeit des Zustands und Finalzustände

Als Nächstes wollen wir sicherstellen, dass wir uns zu einem beliebigen Zeitpunkt immer nur in einem Zustand befinden. Dafür formulieren wir folgenden Term

$$(q_{0,t_j} \vee \dots \vee q_{n,t_j}) \wedge \bigwedge_{s \neq r} (\overline{q_{s,t_j}} \vee \overline{q_{r,t_j}})$$

Dieser Term enthält $n^2 + 1$ Klauseln

Als letztes wollen wir noch einen weiteren Term definieren der garantiert, dass wir uns am Ende des Akzeptiervorgangs in einem Endzustand befinden. Sei also \mathcal{F} die Menge der Endzustände, dann muss gelten:

$$\bigvee_{q_i \in \mathcal{F}} q_{i,t_s}$$

Aufwand

Wie aufwändig ist nun dieses ganze Verfahren?

	Benötigte Klauseln pro Zeitschritt
Für Zustände:	$k^2 + 1$
Für Funktionstabelle:	$8 \cdot k$
Für Band:	$8 \cdot s$
Für Endzustände:	1

Dabei gibt es s Zeitschritte. Alles in allem brauchen wir dann $m(k^2 + 8k + 7s + 2)$ Schritte. Dieser Term ist jedoch polynomiell von n abhängig. Daraus folgt: Die Reduktion ist polynomiell.

Ergebnis

Wir haben also ein *beliebiges* Entscheidungsproblem aus \mathcal{NP} polynomiell auf ein SAT Problem reduziert. Damit, zusammen mit $\text{SAT} \in \mathcal{NP}$, haben wir gezeigt, dass $\text{SAT} \in \mathcal{NP}$ -vollständig ist!

□

1.2.4 Der Schmale Grat

Oft sind sehr ähnlich aussehende Probleme in sehr unterschiedlichen Komplexitätsklassen. Darunter zum Beispiel:

Probleme in P	Probleme in NPC
<i>Existenz eines Eulerkreises</i> Ein Eulerkreis ist ein Zyklus in einem Graphen, der alle Kanten genau einmal enthält	<i>Existenz eines Hamiltonkreises</i> Ein Hamiltonkreis ist ein Zyklus in einem Graphen, der alle Knoten genau einmal enthält
<i>Existenz unärer Partitionen</i> Gegeben $A = \{a_1, \dots, a_n\} \subseteq \mathbb{N}$. jeweils unär kodiert. Gibt es ein $A' \subseteq A$ mit $\sum A' = \sum A \setminus A'$	<i>Existenz binärer Partitionen</i> Gegeben $A = \{a_1, \dots, a_n\} \subseteq \mathbb{N}$. jeweils binär kodiert. Gibt es ein $A' \subseteq A$ mit $\sum A' = \sum A \setminus A'$
<i>Graphen 2-Färbbarkeit</i> Ein Graph ist 2-Färbbar, wenn man jedem Knoten eine von zwei Farben zuordnen kann, so dass benachbarte Knoten nie dieselbe Farbe haben	<i>Graphen 3-Färbbarkeit</i> Ein Graph ist 3-Färbbar, wenn man jedem Knoten eine von drei Farben zuordnen kann, so dass benachbarte Knoten nie dieselbe Farbe haben

Viele weitere Probleme sind in NPC , eine lange Liste (mehr als 300 Probleme) findet sich bei [GAREY, S. 187]

1.2.5 $P \stackrel{?}{=} NP$

Sichtweisen

Es gibt verschiedene Standpunkte zu dem Thema $P \stackrel{?}{=} NP$. Man könnte etwa sagen, es gibt so viele Probleme, die in NPC liegen, dass es sicher leicht ist zu zeigen, dass irgend eins davon in P liegt. Andererseits kann man auch sagen, dass die Probleme in NPC aus der Sicht der Komplexitätstheorie alle so ähnlich sind, dass sie auch wahrscheinlich gleich schwer zu lösen sind. Allgemein wird aber angenommen, dass $P \neq NP$.

Schwierigkeiten

Warum ist es nun so schwer zu beweisen wie sich P zu NP verhält? Das liegt daran, dass ein Beweis für $P = NP$ oder $P \neq NP$ zwar nicht unmöglich ist, aber zumindest besondere Beweisverfahren erfordert. Man spricht hier von nicht relativierenden Beweisen. Ein Beweis, der eine Beziehung zwischen zwei Komplexitätsklassen zeigt, ist **relativierend**, wenn er weiterhin gültig ist, wenn man beiden Klassen ein beliebiges Orakel hinzufügt. Hätte man etwa für zwei Klassen K und L gezeigt, dass diese gleich sind, also dass $K = L$ gilt und der selbe Beweis zeigt auch $K^O = L^O$ für ein beliebiges Orakel O , so ist dies ein relativierender Beweis. Dabei ist A^O die Klasse A mit zusätzlichem O -Orakel. Es wurde aber gezeigt, dass es Orakel A und B gibt, für die jeweils gilt: $P^A = NP^A$ bzw. $P^B \neq NP^B$, das heißt, jeder relativierende Beweis würde zu einem Widerspruch führen, und diese Klasse an Beweisen lässt sich hier von vornherein nicht verwenden. Die meisten üblichen Beweistechniken, z.B. die Diagonalisierung,

sind leider relativierend: Man kann einer Turingmaschine ein beliebiges Orakel hinzufügen und diese kann ihr eigenes Halteproblem weiterhin nicht lösen.

1.3 \mathcal{NP} -vollständige Probleme

1.3.1 \mathcal{NP} -Vollständigkeit zeigen

Wie zeigt man für ein neues \mathcal{NP} -Problem, dass es \mathcal{NP} -vollständig ist? Allgemein reicht es aus zu zeigen, dass sich ein beliebiges \mathcal{NP} -vollständiges Problem auf dieses neue Problem reduzieren lässt. Natürlich gilt das nur für \mathcal{NP} -Probleme, damit ein Problem \mathcal{NP} -vollständig ist, muss zunächst auch gesichert sein, dass es überhaupt in \mathcal{NP} liegt. Da sich auf das \mathcal{NP} -vollständige Problem ja auch schon alle Probleme in \mathcal{NP} reduzieren lassen und Reduzierbarkeit transitiv ist, lassen sich auch alle Probleme aus \mathcal{NP} auf das neue Problem reduzieren. Einen Spezialfall bildet SAT. Hier kann man mit dem Satz von Cook (1.2.3) allgemein zeigen, dass sich alle Probleme aus \mathcal{NP} auf SAT reduzieren lassen. Da wir nun also schon gezeigt haben, dass dieses Problem in \mathcal{NPC} liegt, können wir nun SAT auf weitere Probleme reduzieren um zu zeigen, dass diese auch \mathcal{NP} -vollständig sind.

1.3.2 3-SAT

Als erstes wollen wir SAT auf 3-SAT reduzieren. 3-SAT ist dabei die Sprache der erfüllbaren booleschen Formeln in KNF, mit Klauseln, die höchstens 3 Variablen enthalten. Ein Beispiel für eine solche Formel ist:

$$\underbrace{(a \vee b \vee c)}_{\text{einzelne Klausel}} \wedge (\bar{a} \vee b \vee d)$$

Man sieht leicht, dass diese Sprache in \mathcal{NP} liegt, es muss also noch gezeigt werden, dass die Sprache \mathcal{NP} -Hart ist. Wir wollen also nun eine allgemeine boolesche Formel in KNF in eine umformen, die nur 3 Variablen pro Klausel hat und genau dann erfüllbar ist, wenn die allgemeine Formel auch erfüllbar ist. Gegeben sei also ein Term der Form:

$$(v_{1,1} \vee v_{1,2} \vee \dots \vee v_{1,n_1}) \wedge \dots \wedge (v_{m,1} \vee v_{m,2} \vee \dots \vee v_{m,n_m})$$

Wir definieren uns neue boolesche Variablen

$a_{1,1} \dots a_{1,n_1-3}, a_{2,1} \dots a_{2,n_2-3}, a_{3,1} \dots a_{m,n_m-3}$ und formen mit deren Hilfe die einzelnen Klauseln um. Wir formen um:

$$(v_{1,1} \vee v_{1,2} \vee \dots \vee v_{1,n_1})$$

wird zu

$$(v_{1,1} \vee v_{1,2} \vee a_{1,1}) \wedge (\bar{a}_{1,1} \vee v_{1,3} \vee a_{1,2}) \wedge \dots \wedge (\bar{a}_{1,n-3} \vee v_{1,n_1-1} \vee v_{1,n_1})$$

Dabei entstehen für eine Klausel mit n Variablen $n - 2$ neue Klauseln mit jeweils 3 Variablen und insgesamt $n - 3$ neuen Variablen. Zwei benachbarte Klauseln haben dabei jeweils eine der neuen Variablen gemeinsam, wobei sie bei einem der beiden negiert ist. Das heißt also, die neuen Variablen bewirken, dass jeweils

eine von zwei benachbarten Klauseln mit Sicherheit **true** ist. Da es jedoch $n - 2$ Klauseln gibt, aber nur $n - 3$ neue Variablen, kann keine Belegung der neuen Variablen den ganzen Term **true** werden lassen, wenn alle anderen Variablen **false** sind. Es muss mindestens eine der schon existierenden Variablen **true** sein. Dies ist aber genau die Bedingung dafür, dass die ursprüngliche Klausel **true** ist. Verfährt man so mit allen Klauseln, erhält man einen Term, der nur 3 Variablen pro Klausel hat und genau dann erfüllbar ist, wenn der ursprüngliche Term erfüllbar ist. Wir haben damit also ein beliebiges SAT-Problem auf ein 3-SAT Problem reduziert. Da wir für eine Klausel mit n Variablen $n - 2$ Klauseln mit $n - 3$ Variablen erzeugen müssen, ist diese Reduktion polynomiell. Daraus folgt, dass 3-SAT auch \mathcal{NP} -vollständig ist.

1.3.3 Graph 3-Färbbarkeit

Graph 3-Färbbarkeit ist das Entscheidungsproblem, ob es eine Belegung der Knoten eines Graphen mit 3 Farben gibt, so dass mit einer Kante verbundene Knoten unterschiedliche Farben haben. Um zu zeigen, dass dieses Problem \mathcal{NP} -vollständig ist wollen wir zeigen, dass 3-SAT darauf reduzierbar ist, dass es selbst in \mathcal{NP} liegt ist wieder leicht zu erkennen. Das heißt also: wir wollen aus einem gegebenen 3-SAT Problem ein Graph 3-Färbbarkeits Problem erstellen, so dass der Graph genau dann 3-Färbbar ist, wenn der Term des 3-SAT Problems erfüllbar ist. Um das zu erreichen überlegen wir uns ein paar Strukturen, die wir in dem Graph verwenden möchten.

Als erstes legen wir uns auf die drei Farben, die wir für den Graphen, den wir konstruieren möchten, verwenden möchten, fest. Diese sind grundsätzlich auch vertauschbar, allerdings dient diese Wahl der Übersicht. Die drei Farben sollen sein: $\{T, F, O\}$, was kurz für **true**, **false** und **other** steht.

Hilfsgraph

Sei also ein 3-SAT Problem gegeben, welches die Variablen x_1, x_2, \dots, x_n verwendet. Wir konstruieren zunächst den Graphen, der unter Abb. 1.1 dargestellt wird. Dieser hat drei verbundene Knoten, die mit O, T sowie F beschriftet sind und als Dreieck angeordnet sind. Damit dieser Graph dreigefärbt werden kann, müssen diese drei Knoten jeweils unterschiedliche Farben haben. Desweiteren erstellen wir für jede Variable x_i , die in dem 3-SAT Problem vorkommt, zwei weitere verbundene Knoten, die mit x_i bzw. \bar{x}_i beschriftet sind, sowie beide mit dem mit O beschrifteten Knoten verbunden sind. Dabei ist die Einfärbung der Knoten O, T und F jeweils zwar noch nicht vorgegeben, wir können jedoch o.B.d.A. annehmen sie wäre so wie hier dargestellt, da sich die 3-Färbbarkeit des Graphen nicht ändert, wenn man diese vertauscht.

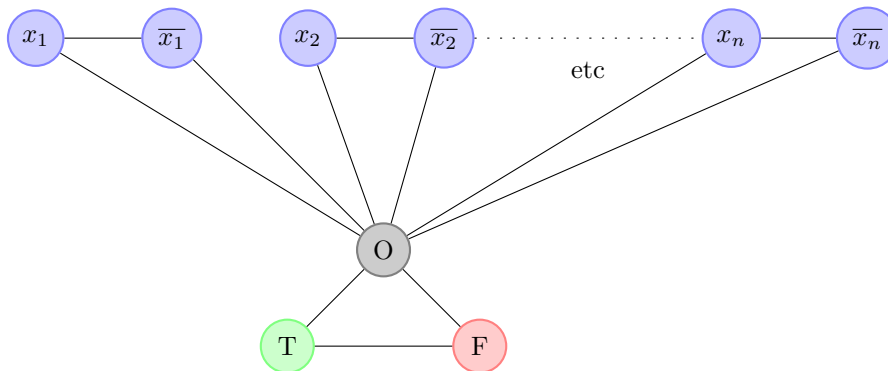


Abbildung 1.1: Hilfsgraph

Dreieck und Propeller

Die nächste Struktur, die wir betrachten wollen, ist ein einfaches Dreieck, das an zwei Ecken noch jeweils mit einem der Knoten (hier x und y) verbunden ist.

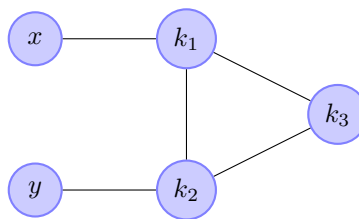


Abbildung 1.2: Dreieck mit verbundenen Ecken

Angenommen wir färben nun zunächst x und y ein. Dann erkennen wir, dass die Farbe von k_3 genau dann eindeutig bestimmt ist, wenn x die gleiche Farbe hat wie y . Die nächste Struktur die wir betrachten wollen, ist ein „Propeller“, der aus zwei solchen verbundenen Dreiecken aufgebaut ist.

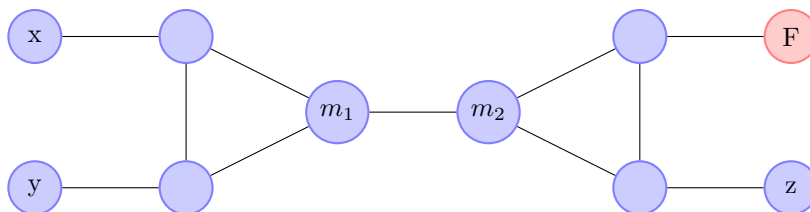


Abbildung 1.3: „Propeller“

In diesem Graphen sollen nun wieder die Knoten x, y und z jeweils entweder mit T oder F eingefärbt werden. Für ein einzelnes Dreieck gilt, dass der Knoten, der mit dem anderen Dreieck verbunden ist, genau dann in seiner Farbe festgelegt ist, wenn die zwei Knoten die an dieses Dreieck angefügt wurden, die gleiche Farbe haben. Sind also etwa x, y mit T eingefärbt, so müssen die beiden Knoten die mit x und y verbunden sind jeweils mit F und O eingefärbt sein und demnach m_1

auch mit T . Da aber ein Eck des anderen Dreieck mit einem Knoten verbunden ist, der mit F eingefärbt ist, ist m_2 nur dann festgelegt, wenn z auch mit F eingefärbt ist. Nicht dreifärbbar ist dieses Konstrukt also nur in einem Fall: Wenn alle verbundenen Knoten mit F eingefärbt sind.

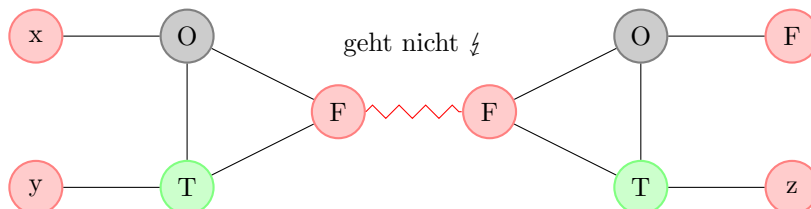


Abbildung 1.4: „Propeller“, Ecken verbunden und ungültig eingefärbt

Dieses Konstrukt, der „Propeller“, ist also genau dann 3-Färbbar, wenn mindestens eine der verbundenen Knoten nicht F ist. Der „Propeller“ soll nun also eine einzelne Klausel eines 3-SAT Problems repräsentieren. Damit wir die entsprechenden Bedingungen für die Knoten, die die Variablen darstellen, erfüllen (sie dürfen z.B. nicht mit O eingefärbt werden), sowie mehrere dieser Teilgraphen kombinieren können, verbinden wir beliebig viele „Propeller“ mit dem zuvor konstruierten Hilfsgraphen. In Abb. 1.3.3 ist dabei der Graph mit dem „Propeller“ der Klausel $(x_1 \vee \overline{x_2} \vee x_n)$ dargestellt

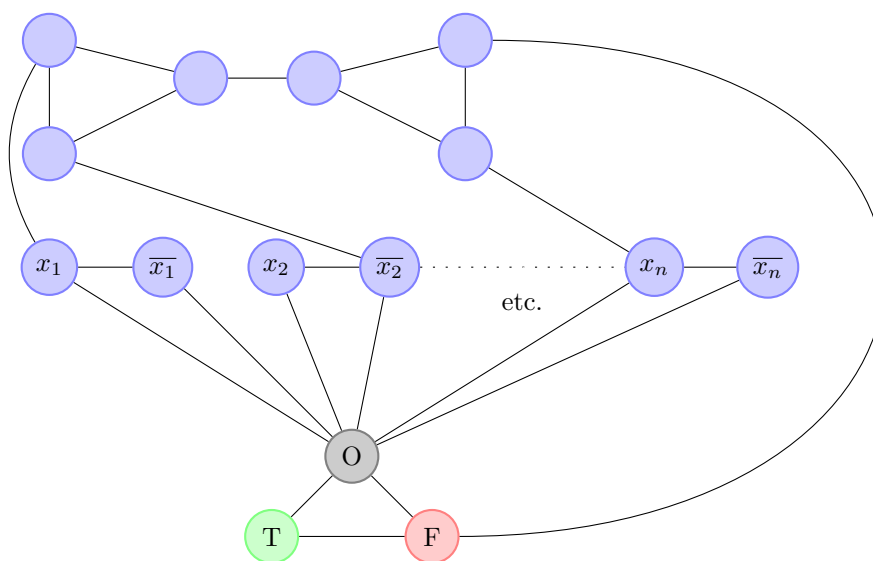


Abbildung 1.5: Hilfsgraph mit einem „Propeller“

Man sieht nun, dass man so beliebig viele weitere solche Propeller anfügen kann. So kann man also für jedes 3-SAT Problem einen Graphen finden, der genau dann 3-färbbar ist, wenn das 3-SAT Problem erfüllbar ist. Da wir für jede Klausel einen solchen Teilgraphen konstruieren müssen, sieht man leicht ein,

dass diese Reduktion polynomiell ist. Es gilt also $\text{SAT} \leq 3\text{-SAT} \leq 3\text{-Färbbarkeit}$. Das heißt also, Graphen 3-Färbbarkeit ist auch \mathcal{NP} -vollständig.

1.3.4 Vertex Cover

Motivation: Angenommen, man hat ein Kommunikationsnetz, dargestellt durch einen Graphen. Dabei sind die Kommunikationspartner die Knoten. Die Kanten stellen nun dar, wer wen des Lügens beschuldigt. Also, wenn etwa eine Kante zwischen „Alice“ und „Bob“ existiert, so heißt dies, dass die beiden jeweils behaupten der Andere wäre nicht vertrauenswürdig. Man nimmt nun an es gibt eine Verschwörung, an der ein gewisser Teil der Kommunikationsteilnehmer beteiligt ist, so dass sich dadurch alle Konflikte erklären lassen. Gegeben eines solchen Graphen also, sucht man eine Menge an Knoten, so dass für jede Kante, mindestens einer der durch sie verbundenen Knoten in dieser Menge liegt. Eine solche Menge heißt dann Vertex Cover.

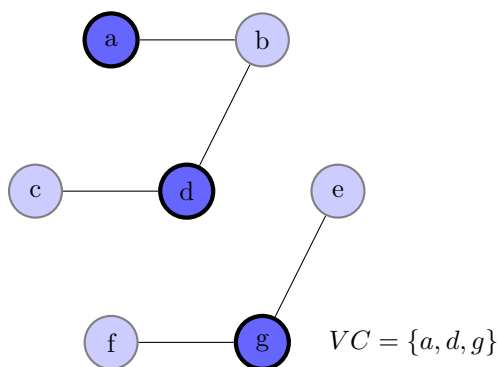


Abbildung 1.6: Beispiel für ein Vertex Cover

Interessant ist dabei vor allem auch ein minimal Vertex Cover, also ein Vertex Cover das eine möglichst geringe Anzahl an Knoten enthält. In der Praxis ist das etwa für Rechnernetze interessant, bei denen einzelne Rechner unterschiedliche Ergebnisse liefern, also manche der Rechner defekt sind. Hier ist es interessant herauszufinden was die Erklärung für die Probleme ist, die am wenigsten Rechner als defekt markiert. Diese Überdeckung muss jedoch nicht eindeutig sein. Ein Graph, der genau aus zwei verbundenen Knoten besteht, hat schon zwei Möglichkeiten. Das zugehörige Entscheidungsproblem, das wir betrachten wollen, lautet nun: Gegeben einen Graphen, sowie $k \in \mathbb{N}$, gibt es ein Vertex Cover mit weniger als k Knoten? Auch bei diesem Problem sieht man gleich ein, dass es in \mathcal{NP} liegt. Wir wollen wieder 3-SAT auf das Vertex Cover Problem reduzieren.

Wir nehmen also wieder an, wir haben einen beliebigen booleschen Term in KNF mit 3 Variablen pro Klausel, und versuchen daraus einen Graphen zu konstruieren, der genau dann ein Vertex Cover mit weniger Knoten als ein bestimmtes $k \in \mathbb{N}$ hat, wenn die Formel erfüllbar ist. Wir bilden zunächst wieder einen Hilfsgraphen, der alle Variablen enthält, dabei seien x_1, x_2, \dots, x_n wieder

die booleschen Variablen.

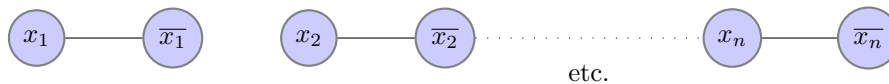


Abbildung 1.7: Hilfsgraph

Hier sieht man leicht, dass jede Vertex Cover dieser Menge, die höchstens n Knoten enthält, für jedes $x_i \in \{x_1 \dots x_n\}$ nur entweder den Knoten für x_i oder den Knoten für \bar{x}_i enthält. Nun betrachten wir wieder die einzelnen Klauseln des 3-SAT Problems. Für jede Klausel erstellen wir einen neuen Teilgraphen, bestehend aus einem Dreieck. Die drei Ecken des Dreiecks verbinden wir mit den Knoten, die den Variablen der Klausel entsprechen. In Abb. 1.3.4 ist dies wieder am Beispiel der Klausel $(x_1 \vee \bar{x}_2 \vee x_n)$ vorgeführt.

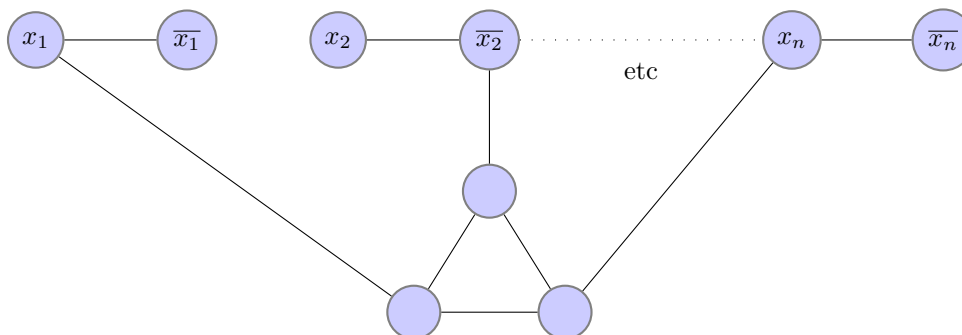


Abbildung 1.8: Hilfsgraph mit einem Dreieck

Man fragt nun nach der Existenz einer Vertex Cover mit höchstens $n + 2m$ Knoten. Dabei ist n die Anzahl der Variablen und m die Anzahl der Klauseln (also im Graphen die Anzahl der Dreiecke). Um ein Vertex Cover für ein einfaches Dreieck zu finden benötigt man 2 Knoten. Da die Ecken der Dreiecke aber noch mit den Variablen verbunden sind, findet man nur dann eine Überdeckung für ein Dreieck, wenn mindestens eine der verbundenen Variablen auch Teil des Vertex Covers sind. Das heißt, man muss sich für jede Variable x_i entscheiden ob x_i oder \bar{x}_i Teil des Vertex Covers sind, und man findet nur dann ein entsprechendes Vertex Cover, wenn man für jede Klausel mindestens eine Variable ausgewählt hat, die darin vorkommt. Also gibt es ein solches Vertex Cover genau dann, wenn der Term erfüllbar ist. Damit haben wir also ein beliebiges 3-SAT Problem auf ein Vertex Cover Problem reduziert und somit zusammen mit $3\text{-SAT} \in \mathcal{NP}$ gezeigt, dass auch dieses Problem \mathcal{NP} -vollständig ist.

1.3.5 Weitere Aspekte

Fixed parameter tractability

Oft hängen Entscheidungsprobleme von einem Parameter ab. Bei diesem Beispiel ist es so, dass ein 3-SAT Problem auf ein Vertex Cover Problem reduziert wird,

so dass dieses neue Problem einen Parameter k hat, der von dem 3-SAT Problem abhängt. Würde man diesen Parameter fest wählen, so wäre das Vertex Cover Problem wiederum einfach zu lösen, und zwar mit vollständiger Suche (engl. „brute force“). Da es eine maximale Anzahl an Knoten gibt, ist eine solche Suche also in polynomieller Zeit möglich. In einem solchen Fall spricht man von „fixed parameter tractability“

Self reducibility

Bei manchen Problemen könnte die Schwierigkeit zwar asymptotisch steigen, aber es könnten dennoch einzelne Instanzen mit langer Eingabelänge leicht sein. Das heißt also, die asymptotische Betrachtungsweise kann unvollständig sein. Aber: 3-SAT mit n Variablen lässt sich polynomiell auf 3-SAT mit $n - 1$ Variablen reduzieren, indem man einfach eine Variable des Terms fest wählt, einmal **true** und einmal **false**. Diese Reduktion ist natürlich polynomiell, da man genau 2 neue Terme bildet. Daraus folgt wiederum, dass die „worst case“ complexity bei 3-SAT nicht sinkt.

Random self reductability

Manchmal reicht auch asymptotische Schwierigkeit nicht, in der Kryptographie will man etwa, dass ein zufälliges Problem schon eine schwere Instanz ist. Hier versucht man ein Problem auf eine zufällige Instanz gleicher Länge zu reduzieren. Ist das möglich, so spricht man von „random self reductability“. Ein Beispiel dafür ist der diskrete Logarithmus. Sei p eine Primzahl, sowie $g \in \mathbb{N}$ mit $g < p$. Weiterhin sein $r \in \mathbb{N}$. Das Problem ist nun, wenn $g^r \bmod p$ sowie g gegeben sind, den Wert von r zu bestimmen. Dieses Problem lässt sich nun wie folgt auf eine zufällige Instanz reduzieren: Wähle zufällig ein $r' \in \mathbb{N}$, berechne $g^{r'} \cdot g^r = g^{r+r'}$. Dies ist eine zufällige Instanz. Bestimme also $r + r'$ und berechne mit r' das gesuchte r .

1.4 Probabilistische Komplexitätsklassen

Die nächste Gruppe von Klassen, die wir betrachten möchten, ist die der probabilistischen Komplexitätsklassen. Die Probleme, die in diesen Klassen liegen, entsprechen am ehesten denen, die man zur Zeit als sinnvoll Berechenbar ansieht.

1.4.1 Counting Classes

Bei probabilistischen Komplexitätsklassen werden wir Maschinen betrachten, die nur mit einer gewissen Wahrscheinlichkeit akzeptieren. Diese Wahrscheinlichkeit hängt dabei von der Anzahl der akzeptierenden Pfade ab. Das heißt also, wir möchten akzeptierende Pfade zählen, dazu befassen wir uns zunächst mit den sogenannten Counting Classes.

Definition 1.4.1.1. Eine **Zählende TM** (auch **CTM** für engl. **Counting TM**) ist eine nichtdeterministische Turingmaschine, deren Output die Anzahl akzeptierender Berechnungspfade ist.

Definition 1.4.1.2. Die Klasse $\#P$ (gesprochen: sharp P) besteht aus den Funktionen, die durch polynomiell zeitbeschränkte Counting Turingmaschinen berechenbar sind.

Man sieht sofort ein, dass diese Klasse „stärker“ ist als \mathcal{NP} . Eine NDTM, die eine Sprache aus \mathcal{NP} akzeptiert, kann nur entscheiden ob es einen akzeptierenden Pfad gibt. Die CTM hingegen kann sogar die Anzahl der akzeptierenden Pfade ausgeben und somit kann man mit ihrer Hilfe die NDTM leicht simulieren. Es ist dabei wichtig zu realisieren, dass dies eine Klasse von Funktionen ist, und nicht eine Klasse von Entscheidungsproblemen. So etwas wie $\mathcal{NP} \subseteq \#P$ zu schreiben wäre also Unsinn. Sinnvoller wäre eine Aussage der Form: $P^{\mathcal{NP}} = P^{\#P}$. Eine andere Aussage die man über diese Klasse machen kann, ist das $\#P \subseteq \text{PSPACE}$, wobei hier PSPACE die Menge der Funktionen ist, zu deren Berechnung nur polynomiell viel Platz benötigt wird. Interessant ist auch, dass es ein $\#P$ -vollständiges Problem gibt: $\#SAT$, welches einer booleschen Formel die Anzahl der erfüllenden Belegungen zuordnet.

1.4.2 Probabilistische Klassen

Die Grundidee, auf der die probabilistischen Klassen aufbauen, ist nun, anstelle nichtdeterministischer Verzweigungen probabilistische Verzweigungen zu verwenden. Es wird also jede der Entscheidungen zufällig getroffen. Eine solche Maschine akzeptiert dann mit einer Wahrscheinlichkeit entsprechend der Anzahl der akzeptierenden Pfade, geteilt durch die Anzahl die insgesamt möglichen Pfade, sofern sich die Maschine in einer entsprechenden Normalform befindet: Die Pfade müssen alle gleichlang sein und alle Verzweigungen müssen genau den Grad zwei haben.

1.4.3 Die Klasse R

Definition 1.4.3.1. Eine **random TM** (auch **RTM**) ist eine nichtdeterministische Turingmaschine, für die für jede Eingabe entweder keine akzeptierende Berechnung existieren, oder mindestens die Hälfte aller Berechnungen akzeptieren.

Das heißt also, eine RTM, die akzeptiert, bestätigt, dass sich ein Wort tatsächlich in der Sprache befindet. Das Ergebnis stimmt in diesem Fall mit Sicherheit. Wenn sie ablehnt, so heißt das nur, dass das Wort mit einer gewissen Wahrscheinlichkeit nicht in der Sprache ist. Diese Wahrscheinlichkeit lässt sich allerdings beliebig klein machen, indem man den Vorgang mehrmals wiederholt.

Definition 1.4.3.2. Die Klasse R besteht nun aus den Entscheidungsproblemen, die eine RTM in polynomieller Zeit lösen kann. Dabei heißt hier lösen, dass es in der Maschine einen akzeptierenden Pfad gibt.

Bei dieser Definition ist zu beachten, dass wenn die RTM einen Pfad enthält der akzeptiert, nach Definition mindestens die Hälfte ihrer Pfade akzeptieren. Es gilt: $P \subseteq R \subseteq \mathcal{NP}$, wobei man hier jeweils vermutet, dass diese Inklusionen echt sind.

Asymmetrie

Ähnlich wie bei der Klasse \mathcal{NP} gibt es auch hier ein Asymmetrie. Die Entscheidung ist lediglich dann mit Sicherheit richtig, wenn die Antwort „Ja“ ist, ist die Antwort „Nein“, so schließt dies nicht aus, dass sich das Wort in der Sprache befindet.

1.4.4 Polynomprodukt Inäquivalenz

Ein gutes Beispiel für ein Problem, für das ein solcher Algorithmus existiert, ist das Problem der Polynomäquivalenz. Das Problem lässt sich folgendermaßen formulieren: Gegeben zwei Multimengen $\{P_1, \dots, P_n\}$ und $\{Q_1, \dots, Q_n\}$ von multivariablen Polynome, entscheide, ob folgendes gilt:

$$\prod_{i=1}^m P_i \neq \prod_{i=1}^m Q_i$$

Das heißt also, die Antwort auf dieses Problem ist „Ja“, wenn die Polynome ungleich sind. Das Problem bei dem naiven Ansatz die Polynome einfach auszumultiplizieren ist, dass dabei die Anzahl der **Monome** zu schnell steigen kann. Auch ein etwas überdachterer Ansatz, die Polynome in ihre **Primfaktoren** zu zerlegen schlägt fehl: Auch hier gibt es das Problem, die Anzahl der Monome kann immernoch zu schnell steigen. Als Beispiel hierfür betrachte man etwa die Faktorisierung von $(x^n - 1)$.

Probabilistischer Ansatz

Wie kann man dieses Problem nun probabilistisch angehen? Die Grundidee besteht darin, eine zufällige Zahl in die Polynome einzusetzen, und zu überprüfen, ob auf beiden Seiten der Gleichung das selbe Ergebnis herauskommt. Wenn nicht, so kann man mit Sicherheit sagen, dass die Polynome ungleich sind. Wenn das selbe Ergebnis herauskommt, so gibt der Algorithmus „falsch“ zurück, da die Polynome nicht unbedingt ungleich sind. Es lässt sich dadurch jedoch noch nicht sicher sagen, ob die Polynome ungleich sind oder nicht. Die Möglichkeit besteht noch, dass der Algorithmus gerade einen Schnittpunkt der beiden Polynome erraten hat. Es ist wiederum die Asymmetrie der Klasse R zu betonen.

$\text{co}R$

Für die Klasse $\text{co}R$, also die Klasse der Komplemente aller Sprachen aus R , ist diese Asymmetrie gerade invertiert. Betrachtet man etwa die Polynomäquivalenz (im Gegensatz zur Polynom**in**äquivalenz, die wir betrachtet haben), so kann man mit dem selben Algorithmus nur mit Sicherheit „Nein“ – also die Polynome sind ungleich – antworten. Es existieren also Zeugen für die Ungleichheit von Polynomen, aber man erkennt keine einfachen Zeugen für deren Gleichheit. Daher wird allgemein auch angenommen, dass $R \neq \text{co}R$, aber auch diese Frage ist noch offen.

1.4.5 ZPP

Definition 1.4.5.1. $ZPP = R \cap coR$

Das heißt also, für Probleme aus der Klasse ZPP gibt es beide Arten von Algorithmen, welche die sicher sind, wenn sie „Ja“ sagen, sowie welche die sicher sind, wenn sie „Nein“ sagen. Das heißt, diese Klasse an Problemen ist in erwarteter polynomieller Zeit entscheidbar. Dabei ist die Antwort auch in beide Richtungen definitiv korrekt.

Kasinos

In Anlehnung daran, dass man auch dort ein Risiko eingeht (auch wenn dieses erheblich höher ist), werden randomisierten Algorithmen im Sinne von 1.4.3.1 als „Monte Carlo Algorithmen“ bezeichnet. Diese Klasse von Algorithmen, die Probleme aus ZPP lösen, also die Algorithmen, die in erwartet polynomieller Zeit laufen und, wenn sie terminieren, auch mit Sicherheit das richtige Ergebnis liefern, heißen entsprechend „Las Vegas Algorithmen“.

1.4.6 PP und BPP

Die Klassen, die wir nun erreichen, sind die eingangs erwähnten Probleme, die heutzutage als sinnvoll berechenbar gelten. Ob das so bleibt ist natürlich nicht sicher, da ja auch die Frage ob $P \stackrel{?}{=} \mathcal{NP}$ noch nicht geklärt ist.

Normalform

Wie schon erwähnt müssen die nichtdeterministischen Turingmaschinen, die wir im folgenden betrachten, noch normalisiert werden. Diese Normalisierung garantiert, dass alle Verzweigungen den Verzweigungsgrad zwei haben und alle Pfade gleichlang sind. Im Wesentlichen sorgt dies also dafür, dass die Maschine als vollständiger binärer Baum dargestellt werden kann. So können wir auch statt von Wahrscheinlichkeiten, von Pfadanzahlen reden.

PP

Definition 1.4.6.1. Eine **Probabilistische Turingmaschine (PTM)** ist eine nichtdeterministische Turingmaschine mit Output „Ja“, wenn die Mehrheit der Pfade akzeptiert. Falls die Mehrheit der Pfade nicht akzeptiert, hat sie den Output „Nein“. Bei exakt gleich vielen akzeptierenden und nicht akzeptierenden Pfaden hat sie den Output „weiß nicht“.

Definition 1.4.6.2. Die Klasse PP (kurz für probabilistisch polynomiell) enthält die Menge aller Entscheidungsprobleme, die von einer PTM in polynomieller Zeit gelöst werden.

Auch wenn diese Klasse schon eher das ist, was wir haben möchten, so können wir sie noch nicht direkt verwenden. Man betrachte folgende Überlegung: Sei F eine boolesche Formel, also ein SAT Problem. Nun führen wir eine weitere Variable ein: x_{neu} . Das Entscheidungsproblem $F \vee x_{neu}$ liegt nun in PP . Damit ist also $\mathcal{NP} \subseteq PP$. Wir suchen hingegen ja realistisch berechenbare Funktionen, \mathcal{NP} ist uns jedoch schon „zu schwer“. Wir haben also nichts gewonnen. Problematisch

ist dabei, dass wir lediglich verlangen, dass die Mehrheit der Pfade akzeptiert. Diese Aussage ist zu schwach, sind immer lediglich minimal mehr akzeptierende Pfade als nicht akzeptierende vorhanden, so wird deren Mehrheit mit steigender Pfadzahl immer unbedeutender.

BPP

Die Klasse, auf die wir eigentlich hinaus wollten, ist also nicht *PP*. Stattdessen definieren wir die Klasse *BPP*. Dabei steht „B“ für „bounded“, also beschränkt. Dabei wird die Anzahl der akzeptierenden Pfade nach unten beschränkt.

Definition 1.4.6.3. Definition: *BPP* besteht aus den Entscheidungsproblemen, die durch eine PTM lösbar sind, die mit einer Wahrscheinlichkeit $\frac{1}{2} + \delta$ mit $\delta > 0$ die richtige Antwort gibt.

Hier wählen wir also eine Konstante δ , die dafür sorgt, dass die Anzahl der akzeptierenden Pfade mit der Gesamtzahl der Pfade hinreichend stark steigt. Erkenntnisse über probabilistische Algorithmen der PTM aus dieser Klasse erlauben es nun endlich sinnvolle Algorithmen zu finden.

R, NP und BPP

Wie verhält sich nun *BPP* zu den bisherigen Komplexitätsklassen, bzw. speziell zu *NP* sowie *R*? Klar ist, dass z.B. *P* komplett in *BPP* liegt, da die Klasse *BPP* die probabilistische Funktionalität nicht verwenden muss. Indizien sprechen dafür, dass sowohl $BPP \not\subseteq NP$ als auch $BPP \not\supseteq NP$ gilt. Weiterhin ist es wahrscheinlich, dass $R \neq coR$. Die Klassen verhalten sich also vermutlich wie in Abb. 1.9 dargestellt.

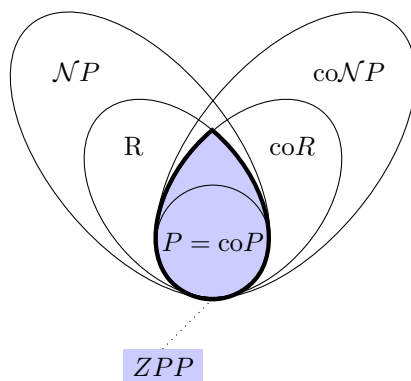


Abbildung 1.9: *ZPP* mit *NP*, *R* und *P* in der Übersicht

1.5 Ein bisschen Kryptographie

Wie bereits erwähnt ist eines der wichtigsten Anwendungsgebiete der Komplexitätstheorie die Kryptographie. Hier ist es wichtig zu untersuchen, wie schwer etwa das Brechen eines kryptographischen Systems ist. Wir möchten uns im Folgenden mit zwei wichtigen Fragen aus der Sicht der Komplexitätstheorie befassen:

- Was ist beweisbare Sicherheit ?
- Kann es Public-Key Verfahren geben, für die deren Brechen einem \mathcal{NP} -vollständigen Problem entspricht?

Man liest häufig etwa: „Die Sicherheit des RSA-Verfahrens beruht auf der Schwierigkeit des Problems lange Zahlen zu faktorisieren“. Auch wenn das durchaus stimmen könnte, so ist es noch nicht gezeigt worden. Was müsste man zeigen, um diese Aussage zu rechtfertigen? Nun, man müsste zeigen, dass das Brechen von RSA impliziert, dass man Zahlen faktorisieren kann. Damit wäre klar: RSA zu brechen, muss so schwer sein wie das Faktorisieren, da man dieses Problem ja damit lösen könnte.

1.5.1 Randomisierte Reduktion

Zunächst müssen wir unseren bisherigen Reduktionsbegriff erweitern. Ein Kryptoverfahren mit einer gewissen Wahrscheinlichkeit zu brechen, würde uns schon ausreichen. Wir möchten, dass wir ein Kryptoverfahren als „unsicher“ bewerten, wenn es sich mit einer gewissen Wahrscheinlichkeit brechen lässt. „worst case“ reicht also nicht. Hierzu führen wir die probabilistische Turingreduktion ein:

Definition 1.5.1.1. Ein Problem A lässt sich probabilistisch auf ein Problem B reduzieren, wenn eine Orakelmaschine mit Zugriff auf ein B -Orakel die Probleme aus A in polynomialer Zeit lösen kann.

In anderen Worten: Man muss mithilfe eines gelösten B Problems auch A lösen können. Das heißt also, relativ zum Orakel B liegt A in BPP , also $A \in BPP^B$. Diese Reduktionsart unterscheidet sich von der many-one-Reduktion. Sie ist probabilistisch, A muss also mit hoher Wahrscheinlichkeit entscheidbar sein.

1.5.2 Das Rabin Kryptosystem

Nun lernen wir ein Kryptosystem kennen, für das sich wirklich zeigen lässt, dass es so schwer ist mit unbekanntem Schlüssel zu entschlüsseln wie das Faktorisieren einer zusammengesetzten Zahl. Dabei ist es wichtig, dass hier das Wort „entschlüsseln“ und nicht „brechen“ verwendet wurde. Ein kryptographisches System wird schon bei geringeren Problemen als „gebrochen“ angesehen, wenn z.B. die gleiche Nachricht verschlüsselt immer identisch ist. Man könnte ja zum Beispiel bei einer Kommunikation die Nachrichten mit älteren vergleichen, und so Rückschlüsse über den Inhalt ziehen. Zunächst aber wollen wir einführen, was ein Public-Key Kryptosystem eigentlich ist:

Definition 1.5.2.1. Ein Public-Key Verfahren besteht aus drei Teilen.

- **KeyGeneration** — Ein Verfahren um Schlüssel zu erzeugen.
- **Verschlüsselung** — Ein Verfahren um mit Schlüssel und Klartext eine verschlüsselte Nachricht zu erzeugen.
- **Entschlüsselung** — Ein Verfahren um mit Schlüssel und verschlüsselter Nachricht den Klartext zurückzugewinnen.

Dabei hat die `KeyGeneration` einen Sicherheitsparameter und wird folgendermaßen verwendet: `KeyGeneration(1k)`.

Der übergebene Parameter ist also die Zahl k , unär kodiert. Die unäre Kodierung ist notwendig damit diese Funktion in polynomialer Zeit läuft. Betrachten wir nun diese drei Elemente bei dem Rabin Verfahren.

KeyGeneration

Beim Rabin Kryptosystem generiert die `KeyGeneration` Funktion zwei zufällige Primzahlen p und q und berechnet dann $n = p \cdot q$. Der öffentliche Schlüssel (public key) ist dann n , der Private (p, q). Man sieht sofort: Der private Schlüssel lässt sich aus dem Öffentlichen nur durch das Faktorisieren von n gewinnen. Aus Effizienzgründen sollte man dabei $p \equiv q \equiv 3 \pmod{4}$ wählen. Der Sicherheitsparameter bestimmt dabei der Länge von n .

Verschlüsselung

Gegeben einer Zahl m , mit $m \leq n$ kann man diese nun mit folgender Operation verschlüsseln:

$$c = m^2 \pmod{n}$$

c ist dann m verschlüsselt mit dem Schlüssel n . Das heißt, man bildet das Quadrat im [Restklassenring](#) modulo n .

Entschlüsselung

Das Entschlüsseln entspricht dann demnach dem Ziehen von Quadratwurzeln in $\mathbb{Z}/n\mathbb{Z}$ — das ist ein Ring im algebraischen Sinn, der die Zahlen $\{0, \dots, n-1\}$ enthält. Die Sicherheit des Rabin-Verfahrens liegt nun darin, dass es keine effiziente Methode gibt diese Wurzel zu ziehen, falls n keine Primzahl ist. Es ist allerdings sehr einfach die Wurzeln — es sind bis zu zwei — in $\mathbb{Z}/p\mathbb{Z}$ mit p prim und $p \equiv 3 \pmod{4}$ zu ziehen. Desweiteren kann man zeigen, dass $\mathbb{Z}/n\mathbb{Z}$ isomorph ist zu $(\mathbb{Z}/p\mathbb{Z}, \mathbb{Z}/q\mathbb{Z})$, was wiederum bedeutet, dass man die Rechnungen auch in $(\mathbb{Z}/p\mathbb{Z}, \mathbb{Z}/q\mathbb{Z})$ machen kann, sofern man die Zahlen p und q kennt. Hier ist das Ziehen der Quadratwurzeln sehr einfach, da ja p und q prim sind. Weiterhin ist es wichtig das es eine explizite, einfache Möglichkeit gibt das Ergebnis aus $(\mathbb{Z}/p\mathbb{Z}, \mathbb{Z}/q\mathbb{Z})$ wieder nach $\mathbb{Z}/n\mathbb{Z}$ umzuwandeln: Den [Chinesischen Restsatz](#). Ein Problem dabei ist, dass es bis zu vier gültige Lösungen für die Wurzel in $\mathbb{Z}/n\mathbb{Z}$ gibt und man nicht unterscheiden kann, welches die korrekte Entschlüsselung ist. Um diesem Problem entgegenzuwirken, kann man festgelegte Muster verschlüsseln, die der **Empfänger** erkennen kann, sodass er weiß welche der vier Möglichkeiten korrekt ist. Dabei muss jedoch beachtet werden, das dann der folgende Beweis nicht mehr funktioniert.

Sicherheit

Warum ist also das Wurzelziehen so schwer? Wie angekündigt kann man zeigen, das dieses Problem mindestens so schwer ist, wie das Faktorisieren einer Zahl. Wir wollen also zeigen: Wer Rabin entschlüsseln kann, kann auch Zahlen faktorisieren. Sei also O ein Orakel, welches zufällig eine der vier Wurzeln einer Zahl bestimmen

kann. Gegeben einer Zahl $n = p \cdot q$, mit p und q unbekannt, kann man nun folgendermaßen vorgehen:

1. Wähle ein $m < n$ zufällig und gleichverteilt.
2. Berechne m^2 .
3. Frage O nach der Wurzel $x = \sqrt{m^2}$.
4. Überprüfe: Ist $x = m$ oder $x = n - m$? Wenn ja, wiederhole die beiden vorigen Schritte.
5. Ansonsten: Haben wir x mit $x \neq m$, $x \neq n - m$ sowie $x^2 = m^2$.
6. Es gilt nun (nach der binomischen Formel): $(m + x)(m - x) \equiv m^2 - x^2 \equiv 0 \pmod{n}$.

Damit ist also $(m + x)(m - x)$ ein Vielfaches von n . Ein Faktor von n kann man nun also mit $\text{ggT}((m + x), n)$ oder $\text{ggT}((m - x), n)$ bestimmen. Rabin zu Entschlüsseln ist also genauso schwer wie das Faktorisieren von n .

Sicherheit?

Es ist wichtig zu bemerken, dass Rabin in dieser Form dennoch nicht sicher ist. Brechen \neq Entschlüsseln, das System weißt in der vorgestellten Form einige Schwächen auf, etwa das vorhin beschriebene Problem: Die gleiche Nachricht wird hier etwa immer gleich verschlüsselt. Bei einer kleinen Auswahl an möglichen Nachrichten gibt es entsprechen wenige verschlüsselte Nachrichten. Eine einfache Möglichkeit dieses Problem zu umgehen ist z.B. einen zufälligen Schlüssel mit Rabin zu verschlüsseln, und diesen verwenden um ein anderes [symmetrisches](#) Verfahren wie [AES](#) sicher für die weitere Kommunikation zu verwenden.

1.5.3 Brassards Theorem

\mathcal{NP} -vollständige Kryptographie?

Die Sicherheit kryptographischer Verfahren beruht ja mittlerweile bekanntermaßen auf der Komplexität von gewissen Problemen. Die Frage liegt nahe, ob es ein Verfahren gibt, dessen Sicherheit auf einem \mathcal{NP} -vollständigen Problem beruht. Diese Klasse an Problemen ist ja bekanntermaßen schwer zu lösen. Im Gegensatz dazu ist bei der Faktorisierung z.B. noch nicht klar, ob diese nicht etwa doch einfach lösbar ist. Für Quantencomputer gibt es z.B. den [Shor-Algorithmus](#) und die Möglichkeit, dass Quantencomputer durch Turingmaschinen effizient simulierbar sind, besteht auch. Andererseits gibt es für \mathcal{NP} -vollständige Probleme bisher keinen Algorithmus — weder für klassische noch für Quantencomputer — der effizient ist. Ein Verfahren, das auf ein solches Problem aufbaut, wäre also sehr vorteilhaft. Leider gibt es bisher kein solches Verfahren, und eine ganze Klasse von Verfahren wird von folgendem Theorem ausgeschlossen:

Brassards Theorem

Gegeben ein Public-Key Kryptosystem, für das gilt

- Es gibt einen eindeutigen Private-Key für jeden Public-Key.

- Das Entschlüsselungsverfahren ist deterministisch.

Betrachte folgendes Entscheidungsproblem: Gegeben eine polynomial entscheidbare Menge S auf der Menge der Klartexte, entscheide, ob ein gegebener String zu einem Klartext aus S entschlüsselt wird. Das Theorem sagt nun aus: Dieses Problem liegt in $\mathcal{NP} \cap \text{co}\mathcal{NP}$. Das impliziert vor allem, dass wenn $\mathcal{NP} \neq \text{co}\mathcal{NP}$ (was allgemein angenommen wird), dass dann das Problem nicht in \mathcal{NPC} liegt. Dabei sieht man leicht ein, dass das Problem in \mathcal{NP} liegt, dabei ist der Zeuge der Schlüssel. Etwas schwerer, aber im Grunde mit der selben Idee ist der Beweis für $\text{co}\mathcal{NP}$: Auch hier wird der Private-Key vom Orakel erraten und es wird einfach überprüft ob das Ergebnis nicht in S liegt. Dabei muss man allerdings noch beachten, dass es hier möglich ist, dass die Eingabe gar kein Chiffre ist. In diesem Fall gibt die Maschine alles, was sie gemacht hat, aus, da sie nur Polynomial viel Zeit hat, sind das höchstens polynomial viel Schritte, also ein polynomial langer Beweis. Es ist also sehr schwer ein Public-Key-Verfahren zu finden, das auf einem \mathcal{NP} -vollständigen Problem aufbaut.

Kapitel 2

Informationstheorie

2.1 Einführung

2.1.1 Was ist Information?

Die Informationstheorie befasst sich mit Informationen. Aber was sind Informationen genau? [Claude Shannon](#) führte zuerst den Informationsbegriff ein, zunächst als „mathematical theory of communication“, also als Theorie der Kommunikation.

Shannonscher Informationsbegriff

Betrachten wir zunächst eine einfache Situation um den Begriff zu illustrieren: Man stelle sich eine Zufallsquelle vor, die Nachrichten verschickt, diese ist der **Sender**. Die Zufallsquelle könnte z.B. einfach eine Münze sein, die immer wieder geworfen wird. Weiterhin gibt es einen Beobachter, den **Empfänger**, der die Nachrichten des Senders beobachtet.

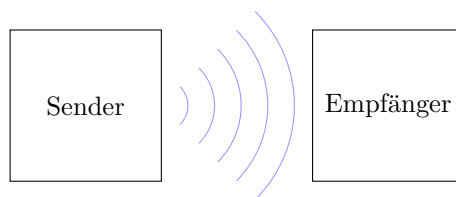


Abbildung 2.1: Sender und Empfänger

Der Shannonsche Informationsbegriff ist nun ein Maß der Überraschung, die der Empfänger beim Erhalt der Nachrichten des Senders hat. Das heißt, vorhersehbare Nachrichten haben wenig Information, während Nachrichten, die unerwartet sind, also eine geringe Auftrittswahrscheinlichkeit haben, einen hohen Informationsgehalt haben. Übertragen auf das Fernsehprogramm hat also eine Werbesendung, die immer wieder läuft, wenig Informationen. Die Lottozahlen hingegen, die unvorhersehbar sind, beinhalten viel Informationen.

Kritik

Auch wenn dieser Begriff sehr gut geeignet ist für die Anwendungen, die wir haben, gibt es auch hier wieder Kritik. So enthält nach dieser Definition das Rauschen im Fernsehen, welches sich sehr unvorhersehbar verhält, mehr Informationen als etwa die Tagesschau, bei der z.B. der Hintergrund immer gleich bleibt.

2.1.2 Anwendungen

Zunächst eine kurze Übersicht über ein paar mögliche Anwendungen der Informationstheorie, auf die wir dann noch näher eingehen wollen.

Quellkodierung

Bei einer Quellkodierung werden die Signale, die der Sender verschickt, kodiert, üblicherweise mit dem Ziel diese zu komprimieren, also die Informationsdichte zu erhöhen. Beispiele dafür sind **ZIP** oder **GZIP**.

Kanalkodierung

Verschickt man Signale über einen Kanal, so hat man oft das Problem, dass die Daten, die man durch den Kanal schickt, verfälscht werden. Das heißt, einzelne Bits werden invertiert. Um mit diesem Problem umzugehen, verwendet man Kanalkodierungen. Das sind Kodierungen, die die Informationsdichte verringern, so dass selbst wenn die Daten verändert werden, die Informationen erhalten bleiben. Eine sehr einfache Kanalkodierung ist z.B. der „triple repetition code“, dabei wird jedes Zeichen einfach dreimal verschickt. Wird nun höchstens eine der drei Wiederholungen verändert, so kann die Veränderung einfach festgestellt werden.

Kryptographie

Auch in der Kryptographie spielt die Informationstheorie eine Rolle. Man sucht dabei nach Verschlüsselungen, bei der das Chiffre ohne bekannten Schlüssel möglichst wenig Informationen über den Klartext enthält. So könnte ein Angreifer mit großer Wahrscheinlichkeit nichts über den Klartext in Erfahrung bringen, auch mit aller Rechenleistung der Welt.

2.1.3 Formale Definitionen

Wir möchten nun die Information I definieren, die ein Zeichen enthält, das mit der Wahrscheinlichkeit p verschickt wird. Dazu überlegen wir uns zunächst ein paar Bedingungen, die dieser Begriff erfüllen sollte.

1. Information soll nicht negativ sein. Also sollte gelten $I_p \geq 0$.
2. Ein sicheres Ereignis (also wenn $p = 1$) soll keine Informationen liefern.
3. Für zwei unabhängige Ereignisse soll sich die Information summieren. Also:
$$I_{(p_1 \cdot p_2)} = I_{p_1} + I_{p_2}.$$
4. Die Information soll stetig sein, also kleine Änderungen an der Wahrscheinlichkeit sollen nur kleine Änderungen an der Information bewirken.

Wenn wir nun überlegen, welche Formel diese Regeln erfüllen kann, so ergibt sich folgender Begriff:

Definition 2.1.3.1. $I_p = \log_b\left(\frac{1}{p}\right) [= -\log_b(p)]$

Dabei verwenden wir im Folgenden immer die Basis $b = 2$.

Beispiel

Das Werfen einer Münze ergibt jeweils mit der Wahrscheinlichkeit $p = \frac{1}{2}$ Kopf oder Zahl. Dabei ist es egal, welches von beiden wir betrachten, da nur die Wahrscheinlichkeit eine Rolle spielt. Betrachten wir nun die Information, die ein konkreter String der Länge n enthält, der durch mehrfachen Münzwurf erzeugt wurde, so ergibt sich $p = \frac{1}{2^n}$, und damit $I = -\log_2\left(\frac{1}{2^n}\right) = n$.

2.1.4 Der Entropiebegriff

Als nächstes führen wir den Entropiebegriff ein. Die Entropie ist dabei eine Länge, unter die ein String nicht komprimiert werden kann. Eng zusammenhängend damit ist der Begriff der **Kolmogorov Komplexität**. Diese ist für einen gegebenen String, die Länge des kürzesten Programms das diesen String ausgibt. Somit ist die Entropie natürlich eine untere Schranke für die Kolmogorov Komplexität.

Exkurs: Ursprung des Entropiebegriffs

Der Entropiebegriff hat seinen Ursprung in der Physik. Der informationstheoretische Entropiebegriff ist eng mit dem physikalischen verwandt, die Definition ist identisch. Man betrachte etwa ein sich ausbreitendes Gas:

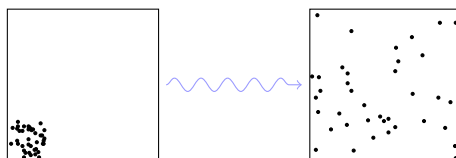


Abbildung 2.2: Entropie nimmt zu

Hier sagt der **zweite Hauptsatz der Thermodynamik** aus, dass der Pfeil in Abb. 2.2 nur nach rechts geht, also die Entropie nur zunehmen kann.

Maxwells Dämon

Diesen Satz könnte man dadurch verletzen, dass man einen gasgefüllten Raum mit zwei Kammern konstruiert, zwischen denen eine reibungsfreie Schiebetür ist, die ein kleiner Dämon immer dann aufmacht, wenn ein schnelles Teilchen von links, oder ein langsames Teilchen von rechts kommt. Damit sortiert er die Teilchen nach ihrer **Geschwindigkeit, also Temperatur**.

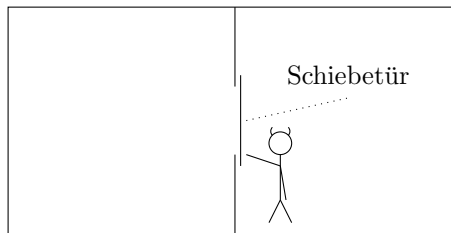


Abbildung 2.3: Maxwells Dämon

Man könnte nun aus der gewonnenen Temperaturdifferenz Energie gewinnen (z.B. mit einem [Stirling-Motor](#)), scheinbar „kostenlos“. Eine mögliche Folgerung daraus ist, dass Datenverarbeitung Energie benötigt.

2.1.5 Occams Razor

[Occams Razor](#) ist ein Prinzip, das sagt, dass wenn mehrere Theorien zur Verfügung stehen, dass dann die Einfachere zu bevorzugen ist. Was dabei „einfach“ heißt, ist allerdings nicht genauer spezifiziert.

2.1.6 Entropie

Wie ist also nun im Sinne der Informationstheorie Entropie definiert?

Definition 2.1.6.1. Die Entropie einer diskreten Zufallsvariable ist (analog zum physikalischem Begriff) definiert durch

$$H(X) = \sum_{x \in X} p(x) \log\left(\frac{1}{p(x)}\right) = \mathbb{E}I(x)$$

dabei gelten die folgenden Konventionen

$$0 \cdot \log 0 := 0, \quad 0 \cdot \log \frac{0}{0} := 0, \quad a \cdot \log \frac{a}{0} := \infty$$

Dabei gilt immer $H(X) \geq 0$.

Basiswechsel

Für eine andere Basis $b \neq 2$ ergibt sich

$$H_b = \log_b 2 \cdot H(X)$$

Die Entropie wird also bei einem Basiswechsel nur mit einem konstanten Faktor multipliziert. **Vorsicht**, H_α bezeichnet hingegen oft die [Rényi-Entropie](#)

Beispiel

Sei $X = 1$ mit Wahrscheinlichkeit p und $X = 0$ mit Wahrscheinlichkeit $(1 - p)$. Dann ist $H(x) = -p \log(p) - (1 - p) \log(1 - p)$

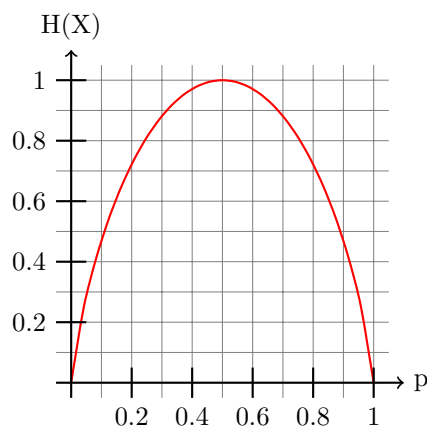


Abbildung 2.4: Entropie

2.1.7 Weitere Entropie-Zusammenhänge

Definition 2.1.7.1. Die **gemeinsame Entropie** der Zufallsvariablen X, Y mit der gemeinsamen Verteilung $p(x, y)$ ist definiert durch

$$H(X, Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \frac{1}{p(x, y)}$$

Definition 2.1.7.2. Die **bedingte Entropie** der Zufallsvariable Y in Abhängigkeit von X mit gemeinsamer Verteilung $p(x, y)$ ist

$$\begin{aligned} H(Y|X) &= \sum_{x \in X} p(x) H(Y|X = x) \\ &= - \sum_{x \in X} p(x) \sum_{y \in Y} p(y|x) \log(p(y|x)) \\ &= - \sum_{x \in X, y \in Y} p(x, y) \log(p(y|x)) \end{aligned}$$

Satz 2.1.7.3. Kettenregel Es gilt

$$H(X, Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

Beweis. Es gilt $\log p(x, y) = \log p(x) + \log p(x|y)$ wegen $\frac{p(x, y)}{p(y)} = p(x|y)$ \square

Daraus kann man folgern

$$H((X, Y)|Z) = H(X|Z) + H(Y|X, Z)$$

Vorsicht. Es kann $H(X|Y) \neq H(Y|X)$

2.1.8 Transinformation

Definition 2.1.8.1. Es ist

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \end{aligned}$$

Visualisiert sieht das ganze im Überblick so aus:

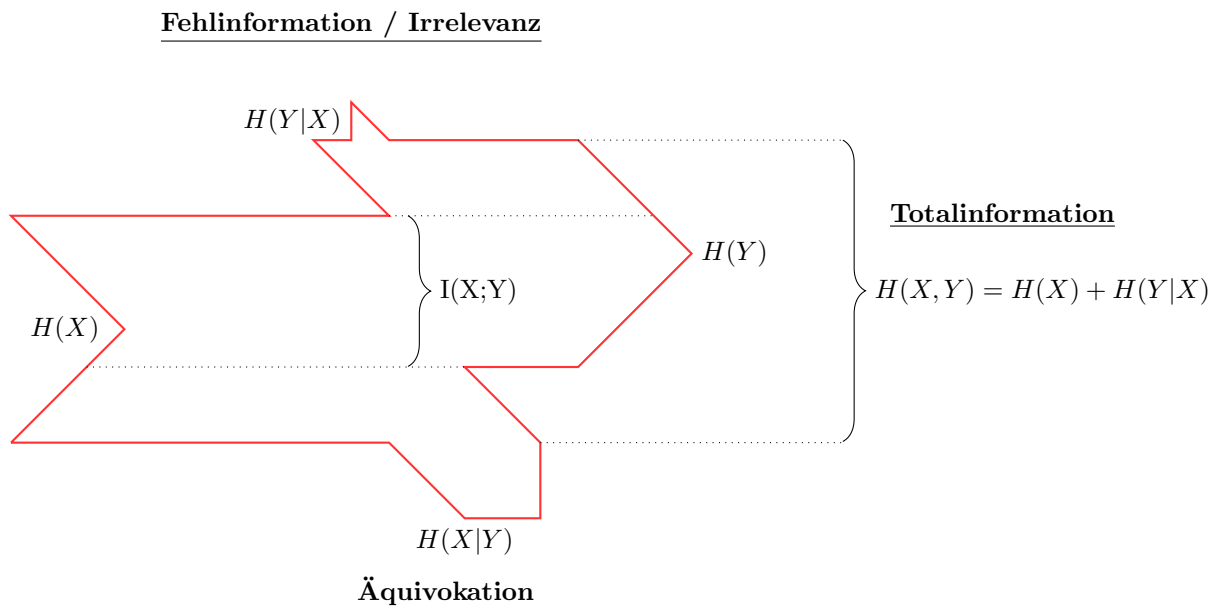


Abbildung 2.5: Transinformation (frei nach [Wikipedia](#))

2.2 Quellkodierungen

2.2.1 Verlustbehaftete Kompression

Bei Quellkodierungen, also Kompressionsverfahren, unterscheidet man grundsätzlich zwischen zwei Typen. Verlustbehaftete und verlustfreie Kompression. Verlustbehaftete Kompressionsverfahren sind dabei, wie der Name schon sagt, Kompressionsverfahren, bei denen echt Information verloren gehen. Typische Beispiele für solche Verfahren sind etwa [JPEG 2000](#) oder [MP3](#). Ein solches Verfahren lässt dabei üblicherweise unwichtige Informationen weg und übernimmt nur die wesentlichen Merkmale. Wir möchten uns im folgenden allerdings mit verlustfreier Kompression auseinandersetzen.

2.2.2 Kodierungen

Grundsätzlich ist es nicht optimal alle Symbole, die in einer Nachricht verwendet werden, in Bitstrings der gleichen Länge zu kodieren, wie es zum Beispiel bei

ASCII der Fall ist. Man möchte stattdessen Symbole, die oft vorkommen, auf kürzere Bitstrings abbilden. Dabei muss natürlich darauf geachtet werden, dass die Dekodierung dennoch eindeutig bleibt. Man betrachte etwa den folgenden Code:

Zeichen	A	B	C	D
Kodierung	0	01	10	11

Hier ist $ADA \cong 0110$ aber auch $BC \cong 0110$. Also auch wenn A ein häufig vorkommendes Zeichen ist, so hilft uns diese Kodierung nicht weiter, da sie nicht immer eindeutig dekodiert werden kann.

2.2.3 Präfix-Codes

Eine Möglichkeit eine eindeutige Kodierung zu erreichen ist die Verwendung von Präfix-Codes.

Definition 2.2.3.1. Ein **Präfix-Code** ist eine Kodierung, so dass für jedes Codewort bestehend aus c_1, \dots, c_n gilt, dass für ein $k < n$, das Wort c_1, \dots, c_k nicht selbst wieder ein Codewort ist.

Das heißt also, dass ein Codewort nicht ein Anfangsteilstück hat, das selbst wieder ein Codewort ist. Eine solche Kodierung kann man, wenn man sie von links nach rechts liest, eindeutig dekodieren, da ja eindeutig ist wenn ein Zeichen endet. Ein Präfix Code wird durch einen Baum definiert: Wenn man die Symbole binär kodieren will, so verwendet man einen binären Baum. Nun verwendet man als Blätter die zu kodierenden Symbole. Die Äste an jeder Verzweigung werden dann mit 0 und 1 beschriftet. Nun bestimmt der Pfad von der Wurzel zu einem Zeichen dessen Kodierung. Ein Präfix Code für die Zeichen $\{\sigma_1, \dots, \sigma_7\}$ lässt sich z.B. durch den Baum in Abb. 2.6 darstellen

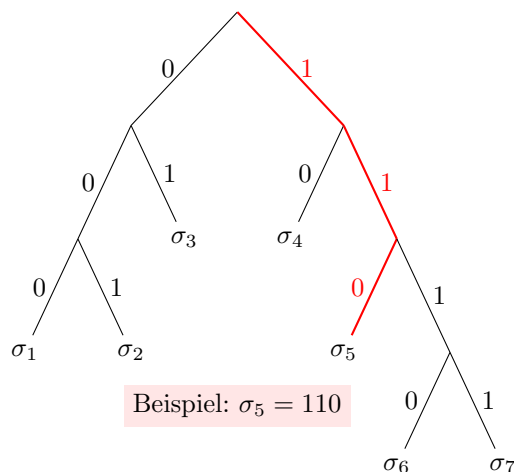


Abbildung 2.6: Präfix Code

Nun sucht man den Baum, der zu einer gegebenen Verteilung auf den Symbolen die kürzeste erwartete Kodierung ergibt. Dabei findet man ein **optimales**

Verfahren für **gedächtnislose** Quellen. Dabei ist eine gedächtnislose Quelle eine Quelle, bei der keine Abhängigkeiten zwischen den Zeichen auftreten. Das heißt, die Wahrscheinlichkeit, dass ein Zeichen auftritt, ist unabhängig davon, welche Zeichen bisher aufgetreten sind.

Quellen mit Gedächtnis

Die meisten Quellen sind keine gedächtnislosen Quellen. Wählt man etwa zufällige deutsche Wörter, so ist die Wahrscheinlichkeit, dass nach einem „q“ ein „u“ folgt, sehr hoch.

2.2.4 Erwartungswert

Betrachten wir nun die folgenden vier Zeichen und ihre Auftretswahrscheinlichkeit:

Symbol	00	01	10	11
Auftretswahrscheinlichkeit	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{3}{16}$	$\frac{9}{16}$

Jedes Zeichen ist hier mit 2 Bit kodiert, der Erwartungswert für die kodierte Länge eines Zeichens ist also demnach 2. Man findet aber bessere Kodierungen, etwa:

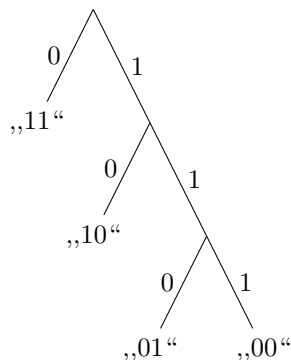


Abbildung 2.7: Bessere Kodierung

Unter Berücksichtigung der neuen Kodierungslängen der Zeichen berechnet sich der Erwartungswert für die Kodierungslänge eines einzelnen Zeichens nun als

$$\frac{1}{16} \cdot 3 + \frac{3}{16} \cdot 3 + \frac{3}{16} \cdot 2 + \frac{9}{16} \cdot 1 = \frac{27}{16} \leq 2$$

So haben wir als eine Kodierung gefunden, die die gleichen Informationen mit weniger Zeichen übertragen kann. Die Frage stellt sich, wie man nun eine möglichst effiziente Kodierung findet. Hierzu gibt es mehrere Verfahren.

2.2.5 Shannon-Fano

Das erste Verfahren, das wir untersuchen möchten, ist die Kodierung von Shannon und Fano. Dabei geht man wie folgt vor:

1. Sortiere die vorkommenden Symbole nach ihrer Häufigkeit.
2. Bestimme nun den Punkt an dem die Reihe aus Symbolen aufgeteilt werden muss, so dass die aufsummierten Wahrscheinlichkeiten der beiden entstehenden Gruppen möglichst gleich sind.
3. Hänge nun die beiden entstehenden Gruppen als Blätter an eine neue Wurzel.
4. Verfahre nun rekursiv: Ersetze die Gruppen jeweils durch den Baum der beim Anwenden des Verfahrens auf sie jeweils entsteht, solange, bis alle Blätter einzelne Symbole sind.
5. Der resultierende Baum ist die Kodierung.

Beispiel

Wir wollen nun anhand der folgenden Verteilung eine Kodierung mittels des Shannon-Fano-Verfahrens bestimmen.

Symbol	A	B	C	D	E
Relative Häufigkeit	15	7	6	6	5

Es ergibt sich folgender Baum:

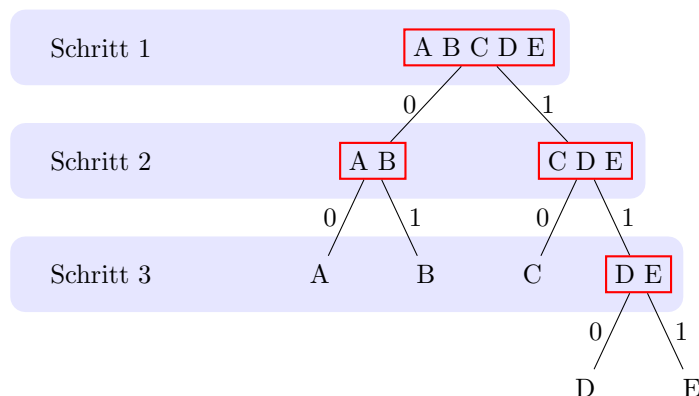


Abbildung 2.8: Shannon-Fano

2.2.6 Der Huffman-Code

Shannon-Fano liefert aber leider nicht immer die optimale Kodierung. Als besser erweist sich die Huffman-Kodierung, wie wir sehen werden, kodiert diese immer optimal. Anstatt die Symbolmenge immer wieder zu zerteilen, arbeitet diese bottom-up. Dabei geht man wie folgt vor:

1. Erstelle aus jedem der Symbole einen Baum, der genau dieses Symbol als einzigen Knoten enthält. Diese bilden einen **Wald**.
2. Wiederhole die folgenden Schritte solange, bis der Wald nur noch ein einziger Baum ist:

- Suche zwei Bäume, so dass die summierte Auftrittswahrscheinlichkeit aller enthaltenen Blätter minimal wird.
- Erstelle einen neuen Baum, der eine neue Wurzel hat, an die die beiden ausgewählten Bäume gehängt werden.

Beispiel

Betrachten wir wieder die Verteilung von vorhin, wie sieht nun die Kodierung aus?

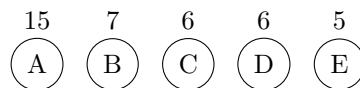


Abbildung 2.9: Huffman, Ausgangslage

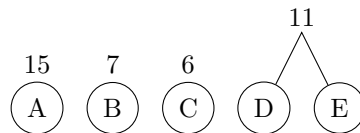


Abbildung 2.10: Huffman, Schritt 1

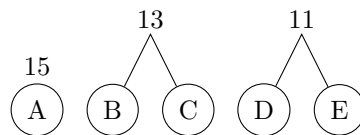


Abbildung 2.11: Huffman, Schritt 2

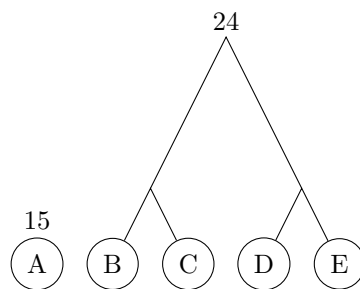


Abbildung 2.12: Huffman, Schritt 3

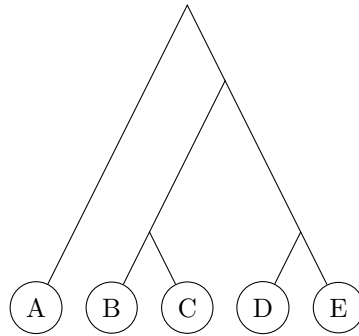


Abbildung 2.13: Huffman, Ergebnis

2.2.7 Optimalität von Huffman

Nun wollen wir zeigen, dass die Huffman-Kodierung optimal ist, also der Erwartungswert für die Zeichenlänge minimal. Dafür führen wir zunächst noch ein paar Begrifflichkeiten ein. Σ soll das Alphabet sein, für den der Code erstellt werden soll. Es enthält $|\Sigma| = n$ Zeichen. Weiterhin seien die Zeichen in Σ : $\sigma_0, \dots, \sigma_{n-1}$. Die Auftretswahrscheinlichkeit eines Zeichens σ bezeichnen wir mit $p(\sigma)$, die Länge des Pfades zu σ , also die Kodierungslänge mit $l(\sigma)$. Nun gilt es folgenden Term zu minimieren:

$$\mathbb{E} = \sum_{\sigma \in \Sigma} p(\sigma) \cdot l(\sigma)$$

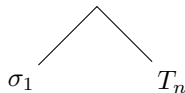
Um zu zeigen, dass dieser Term bei einem Huffman-Baum minimal ist, verwenden wir zwei Lemmas

Lemma 2.2.7.1. Jeder innere Knoten hat in einem optimalen Baum zwei Kind-Knoten.

Beweis. Angenommen ein innerer Knoten hat nur einen Kind-Knoten, dann kann man den inneren Knoten einfach durch den Kind-Knoten ersetzen und verringert damit insgesamt den Erwartungswert \mathbb{E} , dass ist jedoch ein Widerspruch zur Annahme der Baum sei optimal. ζ □

Lemma 2.2.7.2. Wenn σ_1 und σ_2 die beiden unwahrscheinlichsten Symbole sind, so haben sie in einem optimalen Baum den selben Vater-Knoten.

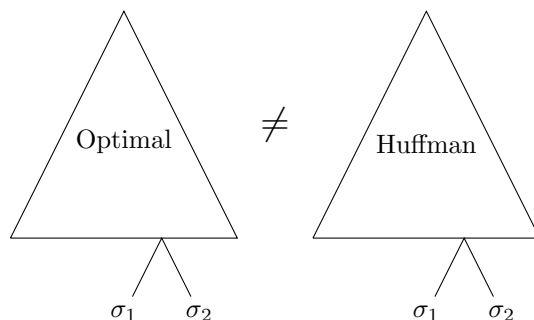
Beweis. Wir wollen wieder einen Widerspruch erzeugen: σ_1 sei das Symbol mit größerer Tiefe (als σ_2) in dem Baum und habe einen Nachbarbaum T_n . also:



Die Summe der Wahrscheinlichkeiten von T_n ist definitionsgemäß größer als die von σ_2 . Vertauscht man nun einfach T_n mit σ_2 , so ist E für der resultierenden Baum kleiner als das bisherige E . Der alte Baum kann also nicht der optimale gewesen sein. ζ □

Satz 2.2.7.3. Der Huffman-Baum ist der optimale Baum

Beweis. Seien wieder σ_1, σ_2 die unwahrscheinlichsten Zeichen. Nehmen wir also an der optimale Baum wäre nicht aus dem oben beschriebenen Verfahren hervorgekommen. Untersuchen wir dann das kleinste Gegenbeispiel.



Nun ersetzen wir den Teilbaum $\sigma_1 \wedge \sigma_2$ in beiden Bäumen durch ein σ' mit $p(\sigma') = p(\sigma_1) + p(\sigma_2)$. Der Teilbaum $\sigma_1 \wedge \sigma_2$ existiert, da σ_1 und σ_2 nicht auf unterschiedlicher Höhe hängen dürfen, wenn sie aber auf gleicher Höhe hängen und nicht benachbart sind, so kann man ohne Beeinflussung von E den Baum so umhängen, dass sie benachbart sind. Wir haben nun unter der Annahme, dass das oben das kleinste Gegenbeispiel war, ein kleineres konstruiert (da Huffman *und* optimaler Baum sich nicht ändern bei der Ersetzung). Das widerspricht der Annahme, dass es überhaupt ein Gegenbeispiel gibt. ζ

□

2.3 Kanalkodierungen

Nun wollen wir uns mit Kanalkodierungen befassen. Das sind im Allgemeinen fehlererkennende oder korrigierende Codes, also Codes, die man verwenden kann, wenn der Kanal, über den das Signal geschickt wird, dieses zufällig verändert, wie es zum Beispiel bei Funkübertragung der Fall sein kann.

2.3.1 Hamming-Distanz

Definition 2.3.1.1 (Hamming Distanz). Für $x, y \in \{0, 1\}^n$ ist

$$d(x, y) := \sum_{i=1}^n (1 - \delta_{x_i, y_i}) = \#\{i | i = 1, \dots, n, x_i \neq y_i\}$$

die Hamming-Distanz zwischen x und y .

Damit ist die Hamming Distanz zwischen x und y also die Anzahl der Zeichen in x , die sich von denen in y unterscheiden.

Definition 2.3.1.2. Weiterhin sei $B_\rho(x)$ die Menge aller Worte y mit $d(x, y) \leq \rho$.

B_ρ ist dabei eine Kugel (die Hamming-Kugel) um x mit Radius ρ .

2.3.2 Maximum-likelihood-decoding

Sei eine Kodierung gegeben, dann beschreibt maximum-likelihood-decoding die Idee, wenn ein Zeichen y empfangen wird, dieses als dasjenige Codewort x zu dekodieren, für das $d(x, y)$ minimal wird.

2.3.3 Rate

Definition 2.3.3.1. Für einen Code mit M verschiedenen Worten der Länge n ist die Rate

$$R = \frac{\log_2 M}{n}$$

Ist die Rate also kleiner als 1, heißt dies, dass die Kodierung mehr Zeichen zum kodieren der Worte als mindestens notwendig verwendet.

2.3.4 Shannons Theorem

Betrachten wir nun einen Code C mit M Worten der Länge n . Seien x_1, \dots, x_M die Codeworte. Desweiteren benutzen wir maximum-likelihood-decoding. Nun sei P_i die Wahrscheinlichkeit dafür, dass falsch dekodiert wird, wenn das Wort x_i gesendet wurde. Die durchschnittliche Wahrscheinlichkeit einer falschen Dekodierung ist dann

$$P_C = \frac{1}{M} \sum_{i=1}^M P_i$$

Binärer symmetrischer Kanal

Nun betrachten wir einen Spezialfall eines Kanals: Den binären, symmetrischen Kanal (engl.: binary symmetric channel, BSC). Das ist ein Kanal, der nur zwei mögliche Zeichen verschicken kann, 0 und 1, und dabei die Wahrscheinlichkeit für einen Fehler unabhängig davon ist, welches Zeichen verschickt wurde. Das heißt also, dass die Wahrscheinlichkeit, dass 1 geschickt wird und 0 empfangen, genauso groß ist, wie die Wahrscheinlichkeit, dass 0 geschickt und 1 empfangen wird.

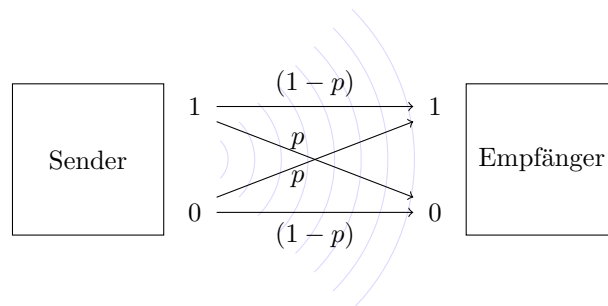


Abbildung 2.14: BSC - Binärer Symmetrischer Kanal

Nun wissen wir, dass es nur eine Fehlerwahrscheinlichkeit p gibt. Wir definieren $q = (1 - p)$. Nun definieren wir unter Berücksichtigung der Bedingungen

und allen möglichen Codes

$$P^*(M, n, p) = \text{das minimale } P_c$$

Theorem 2.3.4.1 (Shannon). Für

$$0 < R < \underbrace{1 + p \log p + q \log q}_{\text{Kanalkapazität}}$$

sowie

$$M_n = 2^{\lfloor R \cdot n \rfloor}$$

gilt

$$P^*(M_n, n, p) \rightarrow 0 \text{ wenn } n \rightarrow \infty$$

Das heißt, für hinreichend lange Worte wird die durchschnittliche Fehlerwahrscheinlichkeit beliebig klein.

Vorbemerkungen zum Beweis

Die Wahrscheinlichkeit dafür, dass in einem Wort w Fehler übertragen werden, ist $p^w \cdot q^{n-w}$, hängt also nur von w ab. Damit ergibt sich für die Anzahl der Fehler in einem Wort der Länge n als Erwartungswert $n \cdot p$ sowie die Varianz $np(1-p)$. Setze nun

$$b = \sqrt{\frac{n \cdot p \cdot (1-p)}{\frac{\varepsilon}{2}}}$$

Nun folgt mit der **Tschebyscheff-Ungleichung**

$$P(W > n \cdot p + b) \leq \frac{\varepsilon}{2}$$

O.B.d.A ist $p < \frac{1}{2}$ (Für $p > \frac{1}{2}$ kann man die Symbole umdefinieren). Damit folgt $\rho = \lfloor np + b \rfloor < \frac{n}{2}$ für genügend große n . Nun ist

$$|B_\rho(x)| = \sum_{i \leq \rho} \binom{n}{i} < \frac{n}{2} \binom{n}{\rho} \leq \frac{n}{2} \cdot \frac{n^n}{\rho^\rho (n-\rho)^{n-\rho}}$$

Wir wollen weiterhin folgende Abschätzungen für $n \rightarrow \infty$ verwenden:

$$\frac{\rho}{n} \log \frac{\rho}{n} = \frac{\lfloor n \cdot p + b \rfloor}{n} \log \frac{\lfloor n \cdot p + b \rfloor}{n} = p \log p + O\left(\frac{1}{\sqrt{n}}\right),$$

$$\left(1 - \frac{\rho}{n}\right) \log \left(1 - \frac{\rho}{n}\right) = q \log q + O\left(\frac{1}{\sqrt{n}}\right)$$

Nun wollen wir noch zwei Funktionen einführen, die eine Rolle im Beweis spielen. Seien $u, v \in \{0, 1\}^*$, dann ist

$$f(u, v) := \begin{cases} 0, & \text{für } d(u, v) > \rho \\ 1, & \text{für } d(u, v) \leq \rho \end{cases}$$

Sei nun x_i ein Codewort und $y \in \{0, 1\}^n$, dann ist

$$g_i(y) := 1 - f(y, x_i) + \sum_{j \neq i} f(y, x_j)$$

Das heißt also, $g_i(y)$ ist genau dann 0, wenn x_i das einzige Codewort ist mit $d(x_i, y) \leq \rho$. Andernfalls ist $g_i(y) \geq 1$.

Beweis. Wir wählen zunächst die Codeworte x_1, \dots, x_M zufällig und unabhängig aus. Nun dekodieren wir wie folgt: Wird y empfangen und x_i ist das einzige Codewort mit $d(y, x_i) \leq \rho$, dann dekodieren wir y zu x_i . Andernfalls geben wir „Fehler“ zurück. Mit P_i von oben gilt

$$P_i \leq \sum_{y \in \{0,1\}^n} P(y|x_i) g_i(y) = \sum_y P(y|x_i) (1 - f(y, x_i)) + \sum_y \sum_{j \neq i} P(y|x_i) f(y, x_i)$$

Die Wahrscheinlichkeit außerhalb einer Kugel $B_\rho(x_i)$ zu liegen ist kleiner als $\frac{\varepsilon}{2}$ (da ja $P(W > n \cdot p + b) \leq \frac{\varepsilon}{2}$), also können wir den linken Term durch $\frac{\varepsilon}{2}$ abschätzen, es folgt

$$P_C \leq \frac{\varepsilon}{2} + \frac{1}{M} \sum_{i=1}^M \sum_y \sum_{j \neq i} P(y|x_i) f(y, x_j)$$

Sei $\mathcal{E}(P_C)$ nun der Durchschnitt über alle P_C . Dann gilt natürlich

$$P^*(M, n, p) \leq \mathcal{E}(P_C)$$

da ja P^* das Optimum war. Eingesetzt ergibt sich

$$\begin{aligned} P^*(M, n, p) &\leq \frac{\varepsilon}{2} + \frac{1}{M} \sum_{i=1}^M \sum_y \sum_{j \neq i} \mathcal{E}(P(y|x_i)) \underbrace{\mathcal{E}(f(y, x_j))}_{\frac{|B_\rho|}{2^n}} \\ &= \frac{\varepsilon}{2} + \frac{1}{M} \sum_{i=1}^M \underbrace{\sum_y \mathcal{E}(P(y|x_i))}_1 (M-1) \frac{|B_\rho|}{2^n} \\ &= \frac{\varepsilon}{2} + (M-1) \frac{|B_\rho|}{2^n} \end{aligned}$$

Nun ziehen wir den Logarithmus und teilen durch n

$$\frac{\log(P^*(M, n, p) - \frac{\varepsilon}{2})}{n} \leq \frac{\log M - (1 + p \log p + q \log q) + O\left(\frac{1}{\sqrt{n}}\right)}{n}$$

Ersetze nun M durch M_n und benutze die Beschränkung von R , es ergibt sich

$$\frac{\log(P^*(M_n, n, p) - \frac{\varepsilon}{2})}{n} < -\beta < 0$$

für hinreichend große n , damit gilt

$$P^*(M_n, n, p) < \frac{\varepsilon}{2} + s^{-\beta n}$$

Somit haben wir das Theorem gezeigt. \square

Dieses Theorem ist allerdings nur ein Existenzbeweis und gibt uns leider keine konkrete Möglichkeit eine solche Kodierung zu konstruieren. Im Folgenden möchten wir uns also mit möglichen Kanalkodierungen befassen.

2.4 Beispiele für Kanalkodierungen

Wie ist ein Code genau definiert? Zunächst unterscheidet man zwischen zwei verschiedenen Arten von Codes:

- Block-Codes: Hier hat man Codeworte fester Länge. Aufeinanderfolgende Blöcke werden unabhängig voneinander kodiert.
- Faltungscodes: Codeworte können beliebig lang sein. Die Zeichen *sind* vom Vorgeschehen abhängig.

Wir möchten uns nun mit Block-Codes befassen.

2.4.1 Block-Codes

Ein Block-Code hat folgende Eigenschaften:

- Es ist immer ein Code über ein endliches Alphabet \mathcal{Q} , wobei vor allem $\mathcal{Q} = \{0, 1\}$ interessant ist.
- Ein Block-Code ist eine Teilmenge $C \subseteq \mathcal{Q}^n$ für ein $n \in \mathbb{N}$, d.h. alle Blöcke haben die selbe Länge.

Weiterhin gilt, falls $\#C = 1$, so heißt C trivial, da es nur ein Codewort gibt. Für $\#\mathcal{Q} = 2$ heißt C binär, für $\#\mathcal{Q} = 3$ ternär, etc.

Definition 2.4.1.1. Die Minimaldistanz eines nichttrivialen Block-Codes C ist

$$m(C) := \min_{c_1, c_2 \in C, c_1 \neq c_2} d(c_1, c_2)$$

Dabei ist hier $d(c_1, c_2)$ die Hamming-Distanz (2.3.1.1) Nun möchten wir den Begriff der Rate aus 2.3.3.1 noch verallgemeinern:

Definition 2.4.1.2. Für einen Code $C \subseteq \mathcal{Q}^n$ heißt

$$R(C) := \frac{\log(\#C)}{\log((\#\mathcal{Q})^n)} = \frac{\log(\#C)}{n \cdot \log(\#\mathcal{Q})}$$

die (Informations-) Rate von C

Man sieht hier, dass sich für $\#\mathcal{Q} = 2$ der Spezialfall aus 2.3.3.1 ergibt. Wir führen nun einen weiteren Begriff ein, den perfekten Code.

Definition 2.4.1.3. Ein Code C mit ungerader Minimaldistanz $m(C)$ heißt **perfekt**, falls es für jedes $x \in \mathcal{Q}^n$ genau ein $c \in C$ gibt, sodass $d(x, c) \leq \frac{m(C)-1}{2}$

Das heißt also, man kann mit maximum-likelihood-decoding jedes empfangene Wort eindeutig einem Codewort zuordnen.

Lemma 2.4.1.4. Ein Block-Code C mit Minimaldistanz $m(C) = d$ kann entweder bis zu $d - 1$ Fehler erkennen oder bis zu $\lfloor \frac{d-1}{2} \rfloor$ Fehler korrigieren.

Beweisskizze. Siehe Abb 2.15 □

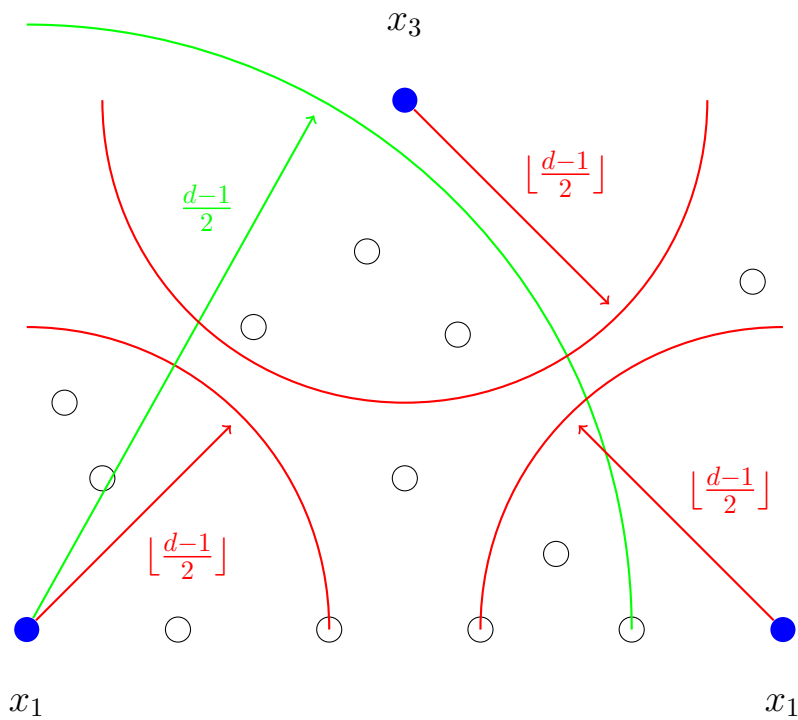


Abbildung 2.15: Beweisskizze

Wir sind nun also daran interessiert möglichst gute Kodierungen zu finden. Was macht aber eine gute Kodierung aus? Man findet folgende Eigenschaften:

1. Möglichst hohe Rate, also möglichst viele Bits pro Symbol bzw. pro Codewort
2. Möglichst hohe Minimaldistanz, um eine möglichst hohe Korrekturleistung zu erzielen
3. Effizientes Decodieren

In Abb. 2.16 erkennt man dabei, dass 1 und 2 konkurrierende Ziele sind.

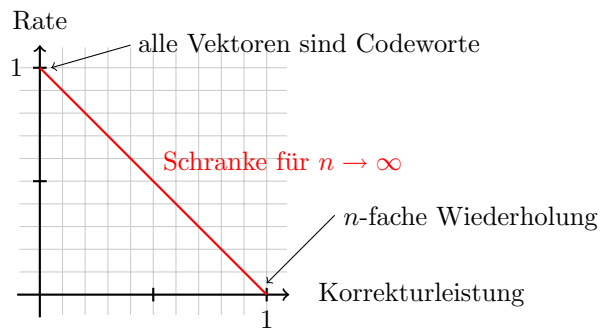


Abbildung 2.16: Rate gegen Korrekturleistung

Betrachten wir nun den eindeutigen **endlichen Körper** \mathbb{F}_q . Dieser hat $q = p^n$ Elemente, wobei p eine Primzahl ist.

Definition 2.4.1.5. Ein linearer $[n, k]$ -Block Code C ist ein Untervektorraum von \mathbb{F}_q^n der Dimension k

Dabei betrachten wir im Folgenden nur $\mathbb{F}_q = \mathbb{F}_2 = \{0, 1\}$. Sei also $\mathbb{F} = \mathbb{F}_2$. Man sieht, dass \mathbb{F}_2^n gerade die Bitvektoren, also Binären Worte der Länge n enthält. Für lineare $[n, k]$ -Codes C gilt dann $R(C) = \frac{k}{n}$.

Definition 2.4.1.6. Für $x \in \mathbb{F}_q^n$ definieren wir die Hamming-Metrik (Hamming Gewicht, L_0 -Norm)

$$\text{wgt}(x) := d(x, 0) = \sum_{i=1}^n (1 - \delta_{x_i, 0}) = \#\{i | i = 1, \dots, n, x_i \neq 0\}$$

Lemma 2.4.1.7. Für Block-Codes gilt $d(x, y) = \text{wgt}(x - y)$

Beweis.

$$\begin{aligned} d(x, y) &= \#\{i | i = 1, \dots, n, x_i \neq y_i\} \\ &= \#\{i | i = 1, \dots, n, x_i - y_i \neq 0\} = \text{wgt}(x - y) \end{aligned}$$

□

Da aber für lineare Block-Codes mit $x, y \in C$ auch $x - y \in C$, entspricht die Minimaldistanz also dem kürzesten Vektor $c \in C$. Es gibt nun zwei Möglichkeiten einen solchen Code zu beschreiben:

- Ist C eine linearer $[n, k]$ -Code, so können wir C als Kern einer $\mathbb{F}^{(n-k) \times n}$ -Matrix H angeben.

$$C = \text{Ker}(H) = \{x \in \mathbb{F}^n | H \cdot x = 0\}$$

Dabei heißt H Prüfmatrix oder Parity-Check-Matrix.

- Beschreibung über Codierungsabbildung: Für $[n, k]$ -Code C können wir $\mathbb{F}^{n \times k}$ -Matrix G angeben sodass

$$C = \text{Bild}(G) = \{y \in \mathbb{F}^n | \exists x \in \mathbb{F}^k : y = G \cdot x\}$$

G bildet Informationsworte auf Codeworte ab.

Dabei ist die Parity-Check-Matrix die wichtigere Beschreibungsart, vor allem hinsichtlich Fehlererkennung und Fehlerkorrektur.

Weiterhin gilt für gegebene G, H , dass $H \cdot G = 0$, dass heißt, jede Spalte von G ist ein gültiges Codewort

Definition 2.4.1.8. Für $x \in \mathbb{F}^n$ heißt $s = H \cdot x$ das Fehlersyndrom von x .

Dabei hängt s nur von einem additiven Fehler ab, nicht aber vom Codewort selbst: Ist $x = c + e$, so ist

$$H \cdot x = H \cdot (c + e) = H \cdot c + H \cdot e = H \cdot e = s$$

Definition 2.4.1.9. Für gegebenes s heißt (falls eindeutig) das $e \in \mathbb{F}^n$ mit $\text{wgt}(e) = \min\{\text{wgt}(x) | x \in \mathbb{F}^n, x \neq 0\}$ der **Coset-Leader** von s .

2.4.2 Parity-Code

Betrachten wir Parity-Codes, in diesem Fall even-parity. Die Idee dahinter ist einfach: Man fügt einem Informationswort ein Bit hinzu das beschreibt ob die Quersumme des Informationswortes gerade oder ungerade ist, um so einfache Bitfehler zu erkennen.

Beispiel 2.4.2.1. Wähle mit den Bezeichnungen von oben also etwa

$$G = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ 1 & 1 & 1 & 1 & \end{pmatrix} \quad H = (1 \ 1 \ 1 \ 1 \ 1)$$

sowie

$$x = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Dann ist

$$c = G \cdot x = (1 \ 0 \ 1 \ 0 \ 0)$$

Wobei hier das letzte Bit das Parity Bit ist, welches aussagt, dass die Quersumme der vorigen Bits gerade ist. Es ist natürlich $H \cdot c^T = 0$.

Parity-Codes sind $[n, n - 1]$ -Codes, kodieren also $n - 1$ Informationsbits in Wörter der Länge n .

Rate und Minimaldistanz

Die Rate eines Parity-Codes ergibt sich als:

$$R(C) = \frac{n-1}{n} = 1 - \frac{1}{n}$$

Das heißt also, $R(C) \rightarrow 1$ für $n \rightarrow \infty$.

Für die Minimaldistanz ergibt sich:

$$m(C) = 2$$

Daraus folgt, Parity Codes können einen einzelnen Bitfehler erkennen, aber keine Fehler korrigieren.

Allgemeine Konstruktion

Ist C ein $[n, k]$ Code mit ungerader Minimaldistanz d und Prüfmatrix H , so können wir den parity-erweiterten Code \bar{C} definieren durch die Prüfmatrix \bar{H}

$$\bar{H} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ & & & 0 \\ & H & & \vdots \\ & & & 0 \end{pmatrix}$$

\bar{C} hat Minimaldistanz $d + 1$, da alle Gewichte in \bar{C} gerade sind.

2.4.3 Hamming-Codes

Hamming-Codes sind eine Familie von dual-projektiven Codes.

Definition 2.4.3.1. Hamming-Codes sind $[2^k - 1, 2^k - k - 1]$ -Codes, für die je zwei Spalten der Prüfmatrix linear unabhängig sind

Beispiel 2.4.3.2. Der $[7, 4]$ -Hamming Code hat folgende Matrizen

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \quad G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rate und Minimaldistanz

Als Rate für Hamming-Codes ergibt sich

$$R(C) = \frac{2^k - k - 1}{2^k - 1} = 1 - \frac{k}{2^k - 1}$$

Und als Minimaldistanz

$$m(C) = 3$$

Theorem 2.4.3.3. Hamming Codes sind perfekt

Beweis. Sei C ein $[2^k - 1, 2^k - k - 1]$ Hamming Code. Betrachte Kugeln $B_1(c)$, also diejenigen mit Radius 1 um die einzelnen Codeworte $c \in C$. Dann

$$\#B_1(c) = 1 + 2^k - 1 = 2^k = B$$

Der C hat Dimension $2^k - k - 1$, also gibt es $\#C = 2^{2^k - k - 1}$ Codeworte. Für $c_1 \neq c_2$ sind $B(c_1)$ und $B(c_2)$ disjunkt. Die Kugelpackung hat $B \cdot \#C$ Elemente

$$B \cdot \#C = 2^k \cdot 2^{2^k - k - 1} = 2^{2^k - 1} = \#\mathbb{F}^{2^k - 1}$$

□

Bei Hamming Codes ist nun ein effizientes Dekodieren von 1-Bit fehlerbehafteten Wörtern möglich.

Problem: COSET-WEIGHTS

Gegeben: Prüfmatrix H und ein Syndrom s und Zahl k Frage: Gibt es ein e mit $\text{wgt}(e) \leq k$ sodass $H \cdot e = s$?

Das zugehörige Suchproblem ist **COSET-LEADER**

Theorem 2.4.3.4 (Berlekamp, McEliece, van Tilborg). COSET-WEIGHTS ist \mathcal{NP} -vollständig

Beweisführung: Reduktionen

$$3\text{SAT} \leq 3\text{DM} \leq \text{COSET-WEIGHTS}$$

in [BMT78] Berlekamp, McEliece, van Tilborg: On the Inherent Intractability of Certain Coding Problems Mehr noch: Sogar das Finden guter Codes ist \mathcal{NP} -vollständig

Problem: SUBSPACE-WEIGHTS

Gegeben: Prüfmatrix H und Zahl k

Frage: Gibt es ein c mit $\text{wgt}(c) = k$ sodass $H \cdot c = 0$?

Auch [BMT78]: SUBSPACE-WEIGHTS ist \mathcal{NP} -vollständig

Weitere Verfahren

Es gibt noch viele weitere Block-Codierungsverfahren die effizientes Decodieren ermöglichen:

- [Repetition-Codes](#)
- Fourier und Polynom-Codes: [Reed-Solomon](#), [BCH](#)
- Algebraische Geometrie Codes: [Goppa Codes](#)
- d-reguläre Expander-Graphen: [LDPC Codes](#)

2.5 Kryptographie und Informationstheorie

Wie angekündigt kann auch Informationstheorie auf die Kryptographie angewendet werden. Dabei versucht man kryptographische Verfahren zu finden, bei denen der Chiffretext keine, bzw. möglichst wenig Informationen über den Klartext enthält.

2.5.1 Einfache Kryptographische Verfahren

Zunächst ein paar einfache, unsichere Kryptographische Verfahren.

Cäsar Verschlüsselung

Bei der Cäsar Verschlüsselung wird jeder Buchstabe des Klartextes ersetzt durch den, der eine konstante Zahl später im Alphabet kommt. Das Alphabet wird also „verschoben“, man spricht daher auch von [Verschiebechiffren](#).

Klartext	...	A	B	C	E	F	G	H	...
Chiffretext	...	W	X	Y	Z	A	B	C	...

Diese Art von Verschlüsselungen sind unsicher, sie enthalten noch sehr viel Information über den Klartext: Zum Beispiel ändert sich die Häufigkeit der auftretenden Buchstaben nicht. Das heißt, man kann, wenn man die Verteilung der Buchstaben kennt (z.B. die Verteilung der Buchstaben in deutschen Wörtern),

die wahrscheinlichen Schlüssel schnell ermitteln. Natürlich ist dieses Verfahren auch deshalb unsicher, weil es nur 25 mögliche Schlüssel gibt. Es ist allerdings allgemeiner auch eine [Monoalphabetische Substitutionchiffre](#), für die man im Allgemeinen eine Häufigkeitsanalyse benötigt.

Vigenère-Verschlüsselung

Die Vigenère-Verschlüsselung, welche lange Zeit als „Le Chiffre indéchiffrable“ (dt. „Die unentzifferbare Verschlüsselung“) galt, funktioniert ähnlich: Hier gibt es ein Schlüsselwort das die Verschlüsselung bestimmt. Ordnet man den Buchstaben nun die Zahlen 0 bis 25 zu, so kann man den Klartext verschlüsseln, indem man die Zahlen des ersten Buchstabens des Klartextes und des ersten Buchstaben des Schlüsselwortes addiert und modulo 26 rechnet. Als nächsten nimmer man jeweils den nächsten Buchstaben, am Ende des Schlüssels angekommen, nimmt man wieder dessen ersten Buchstaben. In Abb. 2.17 sieht man welcher Buchstabe jeweils als Schlüssel verwendet wird, wenn man den Text „DIESERTEXTIST-HOCHKRITISCH“ mit dem Schlüssel „ONYX“ verschlüsselt.

Klartext	DIESERTEXTISTHOCHKRITISCH
Schlüssel	ONYXONYXONYXONYXONYXONYXO

Abbildung 2.17: Beispiel mit Schlüssel „ONYX“

Aber auch dieses Verfahren ist unsicher: für einen Schlüssel der Länge n , wird jeder n -te Buchstabe des Klartextes mit dem selben Schlüssel verschlüsselt, kennt man also die Länge des Schlüssels, so gibt es die selben Probleme die es bei einer Cäsar-Verschlüsselung gibt.

Vernam One Time Pads: Ein sicheres Kryptosystem

Nach [Gilbert Vernam](#). Hier geht man nun wie bei der Vigenère-Verschlüsselung vor, aber verwendet einen zufälligen Schlüssel der genauso lange ist wie der Klartext und verwendet diesen Schlüssel nur ein einziges mal. Dieses Verfahren heißt dann „One-Time-Pad“. Wie wir sehen werden, ist dies ein sicheres Verfahren, sofern man sich an die erwähnten Restriktionen hält. Verwendet man etwa die Schlüssel doch mehrfach, so entspricht diese Verschlüsselung der Vigenère-Verschlüsselung, mit den gleichen Probleme, siehe auch das [VENONA-Projekt](#). Moderne Varianten verwenden technische Maßnahmen um sicherzustellen das ein Schlüssel nur ein einziges mal verwendet wird.

2.5.2 Perfekte Sicherheit

Was macht also informationstheoretisch ein sicheres kryptographisches Verfahren aus? Wir möchten nun den Begriff der perfekten Sicherheit einführen. Dabei ist anzumerken das selbst ein Verfahren das diese Bedingung erfüllt nicht automatisch sicher ist. Gegen Anwendungsfehler etwa schützt auch perfekte Sicherheit nicht.

Definition 2.5.2.1. Sei M ein Klartext, sowie für ein Verschlüsselungsverfahren E der zugehörige Chiffretext. Sei nun $p(M)$ die Wahrscheinlichkeit den Klartext zu erraten, sowie $p(M|E)$ die Wahrscheinlichkeit den Klartext zu erraten, wenn

das Chiffretext bekannt ist. Das Verschlüsselungsverfahren heißt perfekt sicher, wenn gilt

$$p(M|E) = p(M)$$

Das heißt, wenn man den Chiffretext kennt, kann man den Klartext nicht eher erraten als ohne ihn zu kennen. Mit dem **Satz von Bayes** folgt außerdem:

$$p(M|E) = \frac{p(M) \cdot p(E|M)}{p(E)} \iff p(E|M) = p(E)$$

Man kann bei einem solchen Verfahren also auch mit bekanntem Klartext keine Aussage über den Chiffretext machen. Um dies bei endlichen Nachrichten zu erreichen muss die Anzahl der verwendeten Schlüssel größer gleich der Anzahl der Nachrichten sein (Wie das auch bei Vernam One-Time-Pads der Fall ist).

Shannon

Shannon hat perfekte Sicherheit folgendermaßen beschrieben: Wenn man einen Graph erstellt, dessen Knoten die Klar- und Chiffretexte und dessen Kanten mögliche Klar-/Chiffretext Paare darstellen, so ist das Verfahren genau dann perfekt sicher, wenn dieser Graph vollständig und bipartit ist. Natürlich müssen die Paare dabei alle gleichwahrscheinlich sein.

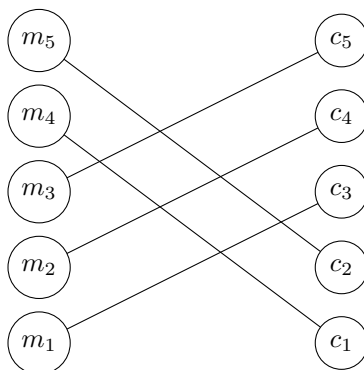


Abbildung 2.18: Graph für eine Cäsar-Verschlüsselung

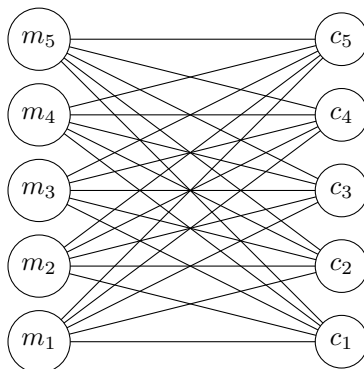


Abbildung 2.19: Vollständiger Bipartiter Graph

2.5.3 Zusammenhang zur Entropie

Betrachten wir die Nachricht M und den Schlüssel K jeweils als Zufallsquellen, dann existieren also auch $H(M)$ und $H(K)$. Diese Betrachtungsweise erlaubt es uns nun also auch hier Zusammenhänge zu untersuchen.

Äquivokation als Sicherheitsmaß.

Man erkennt zum Beispiel, dass man die Äquivokation nun als Sicherheitsmaß betrachten kann. Interessant sind dabei jeweils

$$H(K|E) = - \sum_{E,K} p(E, K) \log p(K|E)$$

$$H(M|E) = - \sum_{E,M} p(E, M) \log p(M|E)$$

und wie sich diese Begriffe zu $H(K)$ bzw. $H(M)$ verhalten.

Satz 2.5.3.1. Für perfekte Sicherheit gilt $H(M|E) = H(M)$.

Beweis. Es ist

$$p(E|M) = \frac{p(E, M)}{p(M)}$$

Mit der Definition von perfekter Sicherheit, $p(E|M) = p(E)$, sieht man

$$\Rightarrow p(E, M) = p(E) \cdot p(M)$$

Das heißt also, dass $p(E)$ und $p(M)$ stochastisch unabhängig sind. Untersuchen wir also nun $H(M|E)$, so folgt:

$$\begin{aligned} H(M|E) &= - \sum_{E,M} p(E, M) \log p(M|E) \\ &= - \sum_{E,M} p(E) \cdot p(M) \log p(M) \\ &= - \underbrace{\sum_E p(E)}_1 \cdot \underbrace{\sum_M p(M) \log p(M)}_{-H(M)} = H(M) \end{aligned}$$

Umgekehrt also auch:

$$H(M|E) = H(M) \Rightarrow p(M|E) = p(M)$$

□

Auch die Transinformation, also gegenseitige Information, verhält sich wie erwartet:

$$I(M; E) = H(M) - H(M|E) = 0$$

2.5.4 Bit-Commitments

Gerechter Münzwurf via Telefon

Ein interessantes kryptographisches Problem ist folgendes: Kann man über eine Telefon-Verbindung gerecht eine Münze werfen, also ein zufälliges Bit ermitteln, so dass keiner der beiden Gesprächspartner das Ergebnis bestimmen kann? Die triviale Möglichkeit, dass einer der beiden Gesprächsteilnehmer eine Münze wirft reicht offenbar nicht aus, er könnte ja behaupten ein beliebiges Ergebnis wäre herausgekommen. Manuel Blum hat sich hierfür 1982 eine Vorgehensweise überlegt:

1. Alice steckt ein zufälliges Bit b in einen Tresor — wie dieser funktioniert sehen wir gleich.
2. Alice schickt Bob den Tresor, dieser kann ihn aber ohne den Schlüssel nicht öffnen.
3. Bob sendet Alice ein zufälliges Bit b' .
4. Alice sendet Bob den Schlüssel des Tresors.
5. Nun ist $b \oplus b'$ das Ergebnis des Münzwurfes.

Dabei legen sich beide auf ihr jeweiliges Bit fest, bevor sie das des anderen kennen. Außerdem ist $b \oplus b'$ schon dann zufällig, wenn einer der beiden ehrlich ist, also sein Ergebnis echt zufällig wählt.

Bit Commitment

Wie funktioniert nun der Tresor? Der „Tresor“ ist offenbar das zentrale Element in diesem Algorithmus. Ein Verfahren das einen solchen „Tresor“ heißt Bit-Commitment. Dieses besteht dabei aus zwei Phasen:

- `commit(b)` — Das Bit b wird festgelegt ohne jedoch den Wert zu offenbaren.
- `unveil` — Deckt auf, was b ist, entspricht also dem öffnen des Tresores.

Für ein solches Verfahren gibt es nun zwei Sicherheitseigenschaften:

- „binding“ — Damit ist gemeint, dass das b das im `unveil` Aufruf offenbart wird, das selbe ist das, das im `commit` aufruf festgelegt wird.
- „hiding“ — Die Eigenschaft, dass der Empfänger vor dem `unveil` nichts über b erfährt.

Man spricht von informationstheoretischem binding, wenn selbst ein in jeder Form unbeschränkter Sender das Bit nicht anders unvielen kann, als es committed wurde.

Informationstheoretisches hiding heißt, dass die Nachricht in `commit(b)` statistisch unabhängig von b ist, also keinerlei Informationen enthält über b .

Man sieht, dass diese beiden Kriterien sich leider ausschließen: Wenn eine solches Verfahren im Informationstheoretischen Sinne binding ist, so muss die Nachricht von `commit` also eindeutig zuordenbar sein, damit ist sie jedoch nicht statistisch unabhängig von b . Andererseits, wenn die Nachricht die `commit`

erzeugt statistisch unabhängig von b ist, müssen mit der gleichen Argumentation **unviel** Aufrufe existieren, die ein beliebiges Ergebnis haben, was allerdings nicht binding ist. Also kann im informationstheoretischen Sinne ein solches Verfahren nicht gleichzeitig binding und hiding sein.

2.5.5 Bit Commitment-Verfahren

Endliche Körper und Dlog

Wie bereits erwähnt bildet $\mathbb{Z}/p\mathbb{Z}$ mit p prim einen Körper, auch als \mathbb{F}_p (veraltet auch $GF(p)$) bezeichnet. Dabei interessiert uns im Folgenden vor allem $\mathbb{F}_p \setminus \{0\}$, also die multiplikative Gruppe des Körpers \mathbb{F}_p . Diese wird im folgenden als \mathbb{F}_p^\times bezeichnet. Diese Gruppe ist *zyklisch*, das heißt, es gibt ein einzelnes Element g , welches die Gruppe erzeugt, geschrieben $\langle g \rangle = \mathbb{F}_p^\times$. Das wiederum bedeutet, für jedes $h \in \mathbb{F}_p^\times$ gibt es ein k mit $g^k = h$. Dann heißt k Dlog (diskreter Logarithmus) von h zu g . Den Dlog zu bestimmen ist ein schwieriges Problem, wie nehmen an, der Dlog ist nicht in Polyzeit berechenbar. Damit haben wir mit $x \rightarrow g^x$ eine injektive Einwegfunktion, also eine Funktion die schwer zu invertieren ist.

Definition 2.5.5.1. Gegeben eine Einwegfunktion und ein Prädikat r auf der Urbildmenge, nennen wir ρ ein „hard-core-bit“, falls $\rho(r)$ wenn nur $f(r)$ gegeben ist, schwer zu berechnen ist.

Nun gilt: Das innere Produkt mit einem zufälligen Vektor ist mit an Sicherheit grenzender Wahrscheinlichkeit ein hard-core-bit. Gegeben einer injektiven Einwegfunktion (siehe Dlog) kann man sich nun auf ein Bit b festlegen, indem man zufällige Vektoren x und y mit $b = \langle x, r \rangle$ wählt, und dann als $\text{commit}(b)$ zurückgibt: $(f(x), r)$. Als **unviel** verwendet man einfach (b, x, r) , womit der Empfänger einfach überprüfen kann ob $b = \langle x, r \rangle$. Dabei hat dieses Verfahren informationstheoretisches binding, aber nur komplexitätstheoretisches hiding.

Pederson-Commitments

Pederson-Commitments sind nun informationstheoretisch hiding, und komplexitätstheoretisch binding. Hier geht man folgendermaßen vor: Seien g und h mit $\langle g \rangle = \langle h \rangle = \mathbb{F}_p^\times$ gegeben, und der gegenseitige Dlog von g und h unbekannt (das kann man etwa dadurch erreichen, dass der Empfänger diese festlegt). Nun gibt $\text{commit}(m)$ für den Bitstring m , zusammen mit einem zufälligen Bitstring r zurück: $g^m \cdot h^r$. **unviel** offenbart dann einfach m und r . Dieses Verfahren ist informationstheoretisch hiding, da sich das r im Exponent wie ein One-Time-Pad verhält. Es ist allerdings nur komplexitätstheoretisches binding, da man wenn man den Dlog berechnen kann beliebig unvielen kann.

Satz 2.5.5.2. Pederson Commitments sind komplexitätstheoretisch binding

Beweis. Angenommen der Sender kann unviel zu m_1, r_1 sowie m_2, r_2 . Es sei nun

α der Dlog von h bezüglich g , also $g^\alpha = h$. Dann gilt:

$$\begin{aligned} g^{m_1} \cdot h^{r_1} &= g^{m_2} \cdot h^{r_2} \\ \iff g^{m_1 - m_2} &= h^{r_2 - r_1} = (g^\alpha)^{r_2 - r_1} = g^{\alpha \cdot (r_2 - r_1)} \\ \iff \alpha \cdot (r_2 - r_1) &= m_1 - m_2 \\ \iff \alpha &= \frac{m_1 - m_2}{r_2 - r_1} \end{aligned}$$

Damit wäre aber der Dlog gebrochen ζ . Bei Pederson Commitments die binding Eigenschaft zu brechen ist also mindestens so schwer wie den Dlog zu berechnen. \square

2.5.6 Code Obfuscation

Nun wollen wir uns mit einer weiteren interessanten Fragestellung befassen auf die man die Komplexitätstheorie anwenden kann: Können wir eine exakt spezifizierte Maschine so bauen, dass niemand außer dem Erbauer die Maschine versteht? Also, wenn wir das Verhalten einer Maschine genau spezifiziert haben, gelingt es uns diese so nachzubauen, dass man nicht verstehen kann wie die Maschine funktioniert? Ähnlich vielleicht, wie die Maschinen die [Tinguely](#) gebaut und [Rube Goldberg](#) erdacht hat.

Hätte man eine solche Maschine, könnte man damit viele Dinge tun, die ansonsten nicht möglich wären:

- DRM — Software ließe sich ganz einfach dadurch schützen, dass man den Code Obfusziert.
- Seitenkanalfreie Chipkarten
- Public Key Encryption aus symmetrischen Chiffren — Gegeben ein Symmetrisches Verschlüsselungsverfahren, könnte man einfach den obfuszierten Verschlüsselungsalgorithmus (mitsamt Schlüssel) veröffentlichen. Damit kann jeder etwas verschlüsseln, aber nur wer den Schlüssel kennt entschlüsseln.
- Rechnen auf Chiffreten — Man könnte mitsamt den Chiffreten eine Maschine veröffentlichen die Operationen auf Chiffrete erlaubt indem sie die Chiffrete einfach entschlüsselt, die Operation anwendet und dann wieder verschlüsselt. Da diese Maschine unmöglich zu verstehen ist, ergibt sich kein Problem.

Was erwarten wir also nun von einem solchen Verfahren genau? Wir möchten, dass jemand der den Code unserer Maschine kennt, genausowenig darüber aussagen kann wie jemand der eine Black Box hat die die selbe Funktionalität bietet. Reverse engineering soll also unmöglich sein. Das heißt: Zu jeder Polybeschränkten TM A mit Zugang zu zum Code C gibt es eine polybeschränkte TM auf von C realisierte Funktionalität, so dass

$$P[A(C) \rightarrow 1] - P[\rho^\sigma \rightarrow 1] \leq \varepsilon$$

Wobei ε vernachlässigbar ist.

Definition 2.5.6.1. Diese Eigenschaft nennen wir Black Box Obfuscation

Satz 2.5.6.2. Black Box Obfuscation ist unmöglich

Beweis. Wir verwenden ein Diagonalisierungsargument indem wir ein Programm konstruieren für das Black Box Obfuscation unmöglich ist. Sei also C ein Programm das durch zwei zufällige Passwörter α und β sowie eine Wahrscheinlichkeit p bestimmt ist. Dieses Programm gibt bei Eingabe des Passwortes α das andere Passwort, β , aus. Bei einer anderen Eingabe testet C ob das zur Eingabe gehörende Turingprogramm bei Eingabe von α auch β zurückgibt. Falls ja, gibt es mit einer Wahrscheinlichkeit von p eine 1 zurück. Dieses Programm kann nicht obfusziert werden. \square

2.6 Philosophisches: Die vier Grundfragen der Philosophie

Immanuel Kant hat sich vier Grundfragen der Philosophie aufgestellt und versucht zu beantworten. Interessanterweise können wir mit dem Wissen das wir aus Informatik III mitgenommen haben, auch Aussagen zu diesen Fragen machen.

Was kann ich wissen?

Mit dieser Frage beschäftigt sich die Erkenntnistheorie

Unter dem was im Rahmen dieser Vorlesung behandelt wurde, ist zu dieser Frage natürlich sofort der Gödelsche Unvollständigkeitssatz zu nennen: Wir können aus einem formalen System entweder nicht alles Korreket beweisen, oder das System ist nicht widerspruchsfrei. Das ist auch der nächste Punkt: Die Widerspruchsfreiheit der Mathematik ist auch nicht beweisbar. Wie wir gesehen haben sind die Implikationen aus diesen Sätzen enorm.

Schließlich kann man natürlich auch den Standpunkt des Solipsismusses vertreten: Nur das Ich existiert, die Außenwelt ist nur Bewusstseinsinhalt ohne eigene Existenz. Die eigene Existenz ist dabei nach Descartes berühmten Grundsatz „Cogito ergo sum“ — „Ich denke also bin ich“ — das einzige über das man wirklich sicher sein kann.

Was soll ich tun?

Diese Frage wird von der Ethik behandelt. Auf diese Frage hat Kant selbst eine recht formale Antwort gegeben, seinen Kategorischen Imperativ, dieser sagt aus:

„Handle nur nach derjenigen Maxime, durch die du zugleich wollen kannst, dass sie ein allgemeines Gesetz werde.“

Man kann nun auch überlegen ob man diese Frage mit den mitteln der Informatik beantworten kann. Könnte man etwa durch Simulation die bestmögliche Handlung ermitteln? Was ist diese überhaupt, also was wäre die Zielfunktion? All dies hängt auch noch von der vierten Frage, von dem Menschenbild ab.

Was darf ich hoffen?

Mit dieser Frage beschäftigt sich die Religionsphilosophie.

Darunter fallen auch Fragen wie: Existiert ein Persönlicher Gott? Wie entsteht Bewusstsein? Gibt es hier einen Zusammenhang zum kategorischen Imperativ?

Was ist der Mensch?

Mit dieser Frage setzt sich die [Anthropologie](#) auseinander.

Hier stellt sich vor allem die Frage nach den Grenzen: wo fängt ein Mensch an zu leben? Wann genau ist er tot? Wann begann die Menschheit als Spezies? Auch hier bietet die Informatik vielleicht wieder die Möglichkeit durch Simulation diese Fragen zumindest teilweise zu beantworten.

Literaturverzeichnis

- [GOLDREICH] Oded Goldreich: *Foundations of Cryptography - Basic Tools*. Cambridge: Cambridge University Press
- [GAREY] Michael R. Garey; David S. Johnson: *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories, Inc.
- [LINT] J.H. van Lint: *Introduction to Coding Theory*. Berlin: Springer-Verlag.