

Theoretische Informatik 3

Amin Coja-Oghlan*

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
coja@informatik.hu-berlin.de

1 Vorbemerkungen

Gegenstand der Vorlesung sind *Entwurf und Analyse effizienter Algorithmen*. Der Begriff *effizient* bezieht sich dabei auf den Bedarf an Berechnungsressourcen (Zeit- und Speicherbedarf) in Abhängigkeit der Größe n der Eingabe. Die Betrachtung ist grundsätzlich *asymptotisch*, d.h. wir sind am Verhalten von Speicher- und Zeitbedarf interessiert für $n \rightarrow \infty$, d.h. an „großen“ Eingaben. Diese Sichtweise, die zumeist zu praxisrelevanten Ergebnissen führt, ist zugleich die der *Komplexitätstheorie* (vgl. Theoretische Informatik 2).

Um den Speicher- und Zeitbedarf zu quantifizieren, nehmen wir bezug auf das *Registermaschinenmodell* (vgl. Theoretische Informatik 2 und [4, Abschnitt 2.6]). Jede Operation der Registermaschine zählen wir als einen Zeitschritt. Die arithmetischen Operationen, die der Registermaschine zur Verfügung stehen, sind dabei

- Addition,
- Subtraktion,
- Halbieren (abgerundet)

von beliebig großen ganzen Zahlen. Operationen für *Multiplikation* und *Division* stehen nicht unmittelbar zur Verfügung, können aber auf die obigen drei Operationen zurückgeführt werden. Um zwei n -stellige Zahlen zu multiplizieren, wird dann Zeit $O(n^2)$ benötigt (mit der „Schulmethode“). Entsprechend ist auch die Division mit Rest in Polynomzeit $n^{O(1)}$ möglich. Es ist jedoch wichtig, daß *weder Multiplikation noch Division* in konstanter Zeit $O(1)$ realisiert werden können.

Der Begriff „effizienter Algorithmus“ bezeichnet in der Regel Algorithmen, die auf einer Registermaschine *in Polynomzeit* n^k arbeiten, wobei n die Eingabelänge ist und $k = O(1)$. In dieser Vorlesung sind wir darüber hinaus daran interessiert, Algorithmen zu entwickeln, bei denen der Exponent k möglichst klein ausfällt. Die Algorithmen, die wir kennenlernen werden, sind fast ausnahmslos *praxistauglich*, was den hier gewählten Rahmen der theoretischen Untersuchung (nämlich die Registermaschine als Abbild von Assemblerprogrammierung) rechtfertigt. Freilich werden wir Algorithmen nicht als Registermaschinenprogramme formulieren, sondern eine intuitivere Darstellung bevorzugen, die sich *prinzipiell* in ein Registermaschinenprogramm umformulieren läßt.

Über das in [4] vorgestellte *deterministische* Registermaschinenmodell hinaus werden wir auch ein *randomisiertes* Modell betrachten. In diesem Modell steht eine weitere Anweisung **RANDOM** zur Verfügung, die den Inhalt des Registers r_0 („Akkumulator“) durch eine zufällige Zahl zwischen 0 und dem Registerinhalt von r_0 unmittelbar vor Ausführen der Anweisung ersetzt. Diese zufällige Zahl ist gleichverteilt, und die aufeinanderfolgenden Aufrufe von **RANDOM** erzeugen voneinander unabhängige Zufallszahlen.

Auch die randomisierte Version der Registermaschine hat sich als praxisrelevant erwiesen. Denn viele in der Praxis eingesetzte Algorithmen sind randomisiert, d.h. verwenden „Zufallszahlen“. Auf realen Maschinen ist es freilich unmöglich, zufällige Zahlen in ähnlich idealer Weise zu erzeugen wie mit der Anweisung **RANDOM**; stattdessen finden *Pseudo-Zufallszahlen* Verwendung. In dieser Vorlesung soll jedoch dieser Aspekt (wie Pseudo-Zufallszahlen generiert werden und wie sie sich zu „echten“ Zufallszahlen verhalten) nicht weiter vertieft werden. Wir begnügen uns mit der

* Datum: 29. Juni 2006.

Feststellung, daß praktisch alle randomisierten Algorithmen, die anhand des randomisierten Registermaschinenmodells entwickelt und analysiert werden können, praktisch mit Hilfe geeigneter Pseudo-Zufallszahlen so implementiert werden können, daß ihr reales Verhalten der theoretischen Analyse nahekommt.

Das Ergebnis der RANDOM-Aufrufe kann die *Ausgabe* des Algorithmus' und/oder dessen Laufzeit beeinflussen. Daher unterscheiden wir die folgenden zwei Typen von randomisierten Algorithmen.

Las-Vegas-Algorithmen. Die *Laufzeit* des Algorithmus' hängt vom Zufall ab, und die *erwartete* Laufzeit ist polynomiell. Die *Ausgabe* ist stets korrekt.

Monte-Carlo-Algorithmen. Die *Ausgabe* des Algorithmus' hängt vom Zufall ab, ist aber mit Wahrscheinlichkeit ≥ 0.99 korrekt. Hingegen ist die *Laufzeit* stets polynomiell.

Um randomisierte Algorithmen zu analysieren, sind die folgenden beiden Abschätzungen häufig hilfreich.

Markov-Ungleichung. Ist $X \geq 0$ eine reelle Zufallsgröße mit Erwartungswert $E(X)$, so gilt für alle $t > 0$:

$$P[X \geq t \cdot E(X)] \leq t^{-1}.$$

Chebyshev-Ungleichung. Ist X eine reelle Zufallsvariable mit Varianz

$$\text{Var}(X) = E[(X - E(X))^2] = E[X^2] - (E[X])^2,$$

so gilt für alle $t > 0$:

$$P[|X - E(X)| \geq t \cdot \sqrt{\text{Var}(X)}] \leq t^{-2}.$$

Diesem Skript liegt die folgende Literatur zugrunde.

Literatur

1. Emden-Weinert, T., Hougardy, S., Kreuter, B., Prömel, H.J., Steger, A.: Einführung in Graphen und Algorithmen. HU Berlin 1996.
2. Köbler, J.: Theoretische Informatik 3. HU Berlin 2005.
3. Lang, S.: Algebra. Addison Wesley 1994.
4. Papadimitriou, C.: Computational complexity. Addison Wesley 1994.
5. Schöning, U.: Algorithmik. Spektrum 2001.

2 Sortieren

Gegeben. Ganze Zahlen a_1, \dots, a_n .

Ziel. Sortiere diese Zahlen, so daß die ausgegebene Folge in aufsteigender Reihenfolge ist. D.h. gesucht ist eine Permutation (Bijektion) $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$, so daß

$$a_{\sigma(1)} \leq a_{\sigma(2)} \leq \dots \leq a_{\sigma(n)}.$$

Algorithmus 1. BubbleSort

Eingabe: Zahlen a_1, \dots, a_n . *Ausgabe:* Dieselben in aufsteigender Reihenfolge.

1. Setze $m = n - 1$.
2. Wiederhole folgendes
3. Setze $m' = 0$.
4. Für $i = 1, \dots, m$
5. Falls $a_i > a_{i+1}$, setze $m' = i - 1$ und vertausche a_i und a_{i+1} .
6. Setze $m = m'$.
7. bis $m = 0$.
8. Gebe die Folge a_1, \dots, a_n aus, die jetzt in aufsteigender Reihenfolge ist.

Lemma 2. *BubbleSort* gibt eine korrekt sortierte Folge aus.

Beweis. Induktion nach n . Für $n = 1$ ist nichts zu zeigen. Sei nun $n > 1$. Im ersten Durchlauf der Schritte 4–6 wird die größte Zahl ans Ende der Liste geschoben. Wenn die letzte Austauschaktion (Schritt 5) dabei zwischen a_i und a_{i+1} stattfand, ist der Teil a_{i+1}, \dots, a_n korrekt sortiert, und dies sind die größten Zahlen in der Menge $\{a_1, \dots, a_n\}$. In der folgenden Iteration der „Wiederhole“-Schleife wird nun dasselbe Verfahren angewendet auf die Liste a_1, \dots, a_i (weil $m = m' = i - 1$ gesetzt wurde). Daher zeigt die Induktionsannahme, daß auch der vordere Teil a_1, \dots, a_i korrekt sortiert wird. \square

Lemma 3. *BubbleSort* hat Laufzeit $O(n^2)$.

Beweis. Die Laufzeit wird dominiert durch die Zahl der *Vergleichsoperationen* (Schritt 5). In jeder Iteration der „Wiederhole“-Schleife werden m solche Vergleichsoperationen ausgeführt. Ferner sinkt m nach jeder Iteration um mindestens 1. Folglich gilt

$$\text{Gesamtzahl der Vergleiche} \leq \sum_{m=1}^{n-1} m = \frac{n(n-1)}{2} \leq \frac{1}{2}n^2.$$

Da je Vergleichsoperation Laufzeit $O(1)$ erforderlich ist, ist die Gesamtlaufzeit $O(n^2)$. \square

Die obige Analyse der Zahl der Vergleichsoperationen ist *scharf*: für die Folge

$$a_1 = n, a_2 = n - 1, \dots, a_n = 1$$

werden in der Tat $\binom{n}{2}$ Vergleiche ausgeführt.

Zwar hat *BubbleSort* eine polynomielle Laufzeit $O(n^2)$. Dennoch stellt sich die Frage, ob es effizientere Sortierverfahren gibt. Ein solcher Algorithmus ist das auf dem *divide and conquer*-Ansatz beruhende *MergeSort*.

Algorithmus 4. *MergeSort*

Eingabe: Zahlen a_1, \dots, a_n , wobei $n = 2^k$, $k \in \mathbb{N}$.

Ausgabe: Dieselben in aufsteigender Reihenfolge.

1. Falls $n = 1$, gebe einfach a_1 aus.
2. Sonst führe *MergeSort*($a_1, \dots, a_{\frac{n}{2}}$) sowie *MergeSort*($a_{1+\frac{n}{2}}, \dots, a_n$) aus.
Seien A', A'' die so berechneten Listen.
3. „Mische“ A' und A'' zu einer sortierten Liste A , die a_1, \dots, a_n enthält, und gebe A aus.

Lemma 5. *MergeSort* liefert stets eine korrekt sortierte Liste.

Beweis. Induktion über k . \square

Lemma 6. *MergeSort* führt $\leq n \log n$ Vergleiche aus.

Beweis. Mit $V(n)$ bezeichnen wir die Zahl der Vergleichsoperationen, die für eine Liste der Länge n benötigt werden. Dann erfüllt die Folge $V(n)$ die Rekursionsgleichung

$$V(1) = 0, \quad V(n) = 2V\left(\frac{n}{2}\right) + n - 1. \quad (1)$$

Dabei zählt $2V\left(\frac{n}{2}\right)$ die Zahl der Vergleichsoperationen in den beiden rekursiven Aufrufen (Schritt 2), während der zweite Summand $n - 1$ die Zahl der Vergleiche für das Mischen der zwei Listen in Schritt 3 ist. Das Gleichungssystem (1) wird von der Folge

$$F(n) = n \log(n) - n + 1$$

erfüllt. Denn

$$\begin{aligned} F(1) &= 1 \log(1) + 1 - 1 = 0, \\ 2F\left(\frac{n}{2}\right) + n - 1 &= 2\left(\frac{n}{2} \log(n/2) - \frac{n}{2} + 1\right) + n - 1 = n \log(n/2) + 1 = n \log(n) - n + 1 = F(n). \end{aligned}$$

Also folgt $V(n) = F(n) \leq n \log(n)$. \square

Satz 7. MergeSort sortiert eine Folge a_1, \dots, a_n in Zeit $O(n \log n)$.

Beweis. Dies folgt aus Lemmas 5 und 6 sowie der Tatsache, daß die Laufzeit von MergeSort durch die Zahl der Vergleichsoperationen dominiert ist. \square

Wir haben MergeSort lediglich für den Fall formuliert, daß $n = 2^k$ eine Potenz von 2 ist. Allerdings läßt sich der Algorithmus leicht verallgemeinern für beliebige Werte von n , z.B. indem in Schritt 2 bei der Division von n durch 2 geeignet gerundet wird.

Die Zahl $n \log n$ ist bereits für moderate Werte von n wesentlich kleiner als $\binom{n}{2}$. Etwa für $n = 10^6$ erhält man $\binom{n}{2} \approx \frac{1}{2} \cdot 10^{12}$, während $n \log n \approx 2 \cdot 10^7$.

Ein weiterer Sortieralgorithmus, der hier aufgrund seiner großen Praxisrelevanz diskutiert werden soll, ist QuickSort.

Algorithmus 8. QuickSort

Eingabe: Zahlen a_1, \dots, a_n . *Ausgabe:* Dieselben in aufsteigender Reihenfolge.

1. Finde einen Index $j \in \{1, \dots, n\}$, so daß (etwa) für die Hälfte der Indices i gilt $a_i \leq a_j$ (bzw. $a_i > a_j$); a_j heißt *Pivotelement*.
2. Vergleiche a_1, \dots, a_n mit a_j , um die zwei Listen

$$S = \{a_i : a_i < a_j\},$$

$$T = \{a_i : a_i > a_j\}$$

- zu bestimmen.
3. Sortiere S, T rekursiv mit QuickSort. Seien S', T' die so bestimmten sortierten Listen.
4. Gebe die Liste S', a_j, T' aus. (Falls a_j mehrfach in a_1, \dots, a_n auftritt, füge diese Zahl entsprechend oft zwischen S' und T' ein.)

Die Idee ist ähnlich wie bei MergeSort: die Listen S, T sind um einen konstanten Faktor (z.B. $\frac{1}{2}$) kleiner als die Ausgangsliste a_1, \dots, a_n . Daher wird das Problem, n Elemente zu sortieren, auf das Problem, zweimal $\frac{n}{2}$ Elemente zu sortieren, reduziert. Die dafür benötigte Laufzeit wird also $O(n \log n)$ sein. Hinzu kommt der Zeitbedarf, um das Pivotelement zu bestimmen.

Um ein geeignetes Pivotelement zu berechnen, beobachten wir zunächst, daß es nicht erforderlich ist, die Liste in jedem Schritt *genau* in der Mitte zu teilen. Denn um eine Laufzeit von $O(n \log n)$ zu erreichen, würde es genügen, daß in jedem Schritt $\#S, \#T \leq \frac{3}{4}n$. Die entscheidende Feststellung ist nun, daß es *viele* mögliche Pivotelemente a_j gibt, die dies sicherstellen. Dazu sei $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ eine Permutation, so daß

$$a_{\sigma(1)} \leq \dots \leq a_{\sigma(n)}.$$

Dann kommen als Pivotelemente gerade $a_{\sigma(n/4)}, \dots, a_{\sigma(3n/4)}$ in Frage, also *die Hälfte* aller Elemente. Daher implementieren wir den ersten Schritt einfach wie folgt:

Wähle $j \in \{1, \dots, n\}$ zufällig.

Wir erhalten so eine *Las Vegas*-Version von QuickSort.

Lemma 9. Die erwartete Zahl von Vergleichsoperationen, die QuickSort ausführt, ist $O(n \log n)$.

Beweis. Sei $X_{ij} = 1$, falls QuickSort zu irgendeinem Zeitpunkt $a_{\sigma(i)}$ und $a_{\sigma(j)}$ miteinander vergleicht, und sei andernfalls $X_{ij} = 0$ (X_{ij} ist also eine Zufallsvariable). Sei ferner

$$X = \sum_{1 \leq i < j \leq n} X_{ij}$$

die Gesamtzahl der Vergleiche (incl. rekursive Aufrufe). Sei schließlich $p_{ij} = E(X_{ij}) = P[X_{ij} = 1]$. Dann gilt für $i < j$

$$p_{ij} \leq \frac{2}{j - i + 1},$$

weshalb

$$\begin{aligned} E(X) &= \sum_{1 \leq i < j \leq n} E(X_{ij}) \leq \sum_{i=1}^n \sum_{i < j \leq n} \frac{2}{j-i+1} \\ &\leq 2 \sum_{i=1}^n \sum_{k=1}^n \frac{1}{k} = 2n \sum_{k=1}^n \frac{1}{k}. \end{aligned}$$

Weil $\sum_{k=1}^n \frac{1}{k} = \log(n) + O(1)$, folgt die Behauptung. \square

Wir haben also die folgenden Sortieralgorithmen kennengelernt:

- BubbleSort mit Laufzeit $O(n^2)$.
- MergeSort mit Laufzeit $O(n \log n)$.
- QuickSort mit *erwarteter* Laufzeit $O(n \log n)$.

Gibt es Sortieralgorithmen, die eine noch geringere Laufzeit als $O(n \log n)$ haben? Wir wollen diese Frage für eine umfassende Klasse von Algorithmen negativ beantworten. Dazu nennen wir einen Sortieralgorithmus *vergleichend*, falls er auf seine Eingabe a_1, \dots, a_n *ausschließlich* über Anfragen der Art

Ist $a_i < a_j$?

zugreift. Dies trifft auf die drei obigen Sortieralgorithmen zu.

Satz 10. *Zu jedem $n \in \mathbb{N}$ und zu jedem vergleichenden deterministischen Sortieralgorithmus \mathcal{A} gibt es eine Eingabe a_1, \dots, a_n , auf der \mathcal{A} Laufzeit $\Omega(n \log n)$ hat.*

Zum Beweis des Satzes benötigen wir das folgende Lemma.

Lemma 11. *Die Zahl der Permutationen von $\{1, \dots, n\}$ ist $n!$, und $\log(n!) = \Omega(n \log n)$.*

Beweis des Satzes. Anstatt die „schlechte“ Eingabe a_1, \dots, a_n für \mathcal{A} im Voraus zu konstruieren, definieren wir diese *während* der Algorithmus diese Zahlen sortiert. Genauer wird $a_i = \sigma(i)$ sein für eine gewisse Permutation σ von $\{1, \dots, n\}$. Ferner werden wir statt der *Gesamtlaufzeit* des Algorithmus’ nur die Zahl seiner Anfragen: „Ist $a_i < a_j$?“ abschätzen.

Die Definition von σ kann man sich als „Spiel“ zwischen \mathcal{A} und einem „Gegner“ vorstellen. In jedem Spielzug stellt \mathcal{A} eine Anfrage, die der Gegenspieler beantwortet. Das Ziel von \mathcal{A} ist, σ zu bestimmen (und damit a_1, \dots, a_n zu sortieren), während der Gegner versucht, σ möglichst lange zu „verbergen“.

Am Anfang ist \mathcal{A} lediglich bekannt, daß σ eine Permutation von $\{1, \dots, n\}$ ist. Ist also U_0 die Menge aller dieser Permutationen, so weiß \mathcal{A} nur, daß $\sigma \in U_0$. Im ersten Spielzug stellt \mathcal{A} nun eine Anfrage: „Ist $a_{i_1} < a_{j_1}$?“, die der Gegenspieler wie folgt beantwortet. Sei

$$\begin{aligned} U_{0,\text{ja}} &= \{\tau \in U_0 : \tau(i_1) < \tau(j_1)\}, \\ U_{0,\text{nein}} &= \{\tau \in U_0 : \tau(i_1) \geq \tau(j_1)\}. \end{aligned}$$

Dann ist die Antwort des Gegenspielers

- „ja“, falls $\#U_{0,\text{ja}} \geq \#U_{0,\text{nein}}$,
- „nein“, sonst.

Der Wissenstand von \mathcal{A} nach diesem ersten Spielzug ist folglich, daß $\sigma \in U_1$, wobei U_1 die größere der beiden Menge $U_{0,\text{ja}}$ und $\#U_{0,\text{nein}}$ ist.

Allgemein weiß \mathcal{A} zu Beginn des k -ten Zuges, daß $\sigma \in U_{k-1}$ und stellt im k -ten Zug eine Anfrage: „Ist $a_{i_k} < a_{j_k}$?“. Sei

$$\begin{aligned} U_{k-1,\text{ja}} &= \{\tau \in U_{k-1} : \tau(i_k) < \tau(j_k)\}, \\ U_{k-1,\text{nein}} &= \{\tau \in U_{k-1} : \tau(i_k) \geq \tau(j_k)\}. \end{aligned}$$

Dann lautet die Antwort des Gegners

- „ja“, falls $\#U_{k-1,\text{ja}} \geq \#U_{k-1,\text{nein}}$,
- „nein“, sonst,

so daß der neue Wissenstand des Algorithmus' ist, daß $\sigma \in U_k$, wobei U_k die größere der beiden Mengen $U_{k-1,\text{ja}}$, $U_{k-1,\text{nein}}$ bezeichnet.

Der Algorithmus hat σ in dem Moment berechnet, wenn die Menge U_k nur noch aus einem einzigen Element besteht, nämlich σ . Sei K dieser Zeitpunkt, d.h.

$$K = \min \{k : \#U_k \leq 1\}.$$

Wie groß ist K ? Weil der Gegner in jedem Schritt die *größere* der beiden Mengen $U_{k-1,\text{ja}}$, $U_{k-1,\text{nein}}$ „auswählt“, gilt für alle k

$$\#U_k \geq \frac{\#U_{k-1}}{2}.$$

Daraus folgt mit Hilfe von Lemma 11, daß

$$K \geq \log(\#U_0) = \log(n!) = \Omega(n \log n).$$

Die Laufzeit des Sortieralgorithmus' \mathcal{A} auf Eingabe σ ist also $\Omega(n \log n)$. □

Eine ähnliche untere Schranke kann für die Laufzeit randomisierter Sortieralgorithmen hergeleitet werden.

3 String-Matching

Gegeben. Ein *Text* $T = T_1 \dots T_n$ sowie ein *Muster* $P = P_1 \dots P_m$; T und P sind Wörter über einem Alphabet Σ .

Ziel. Finde *alle* Stellen, an denen P in T auftritt. Dabei sagt man, P tritt an der Stelle s in T auf, falls $T_{s+1} \dots T_{s+m} = P$.

Ein *naiver Algorithmus* für dieses Problem ist, für jedes $0 \leq s \leq n - m - 1$ zu testen, ob P in T an der Stelle s auftritt. Die Laufzeit dieses Verfahrens ist offenbar $\Theta(nm)$. In diesem Abschnitt werden wir deutlich effizientere Verfahren (mit der bestmöglichen Laufzeit $O(n + m)$) kennenlernen.

Ein erster Ansatz, um eine bessere Laufzeit zu erhalten als der naive Algorithmus, ist die Verwendung endlicher Automaten. Die Idee ist, aus P einen DFA $M = M(P)$ zu konstruieren, in den T zeichenweise eingegeben wird. Ein „Treffer“ tritt genau dann auf, wenn der Automat M in seinem Endzustand ist.

Der Automat M hat die Zustände $Z = \{0, 1, \dots, m\}$, wobei m der einzige Endzustand und 0 der Startzustand ist. Die Überföhrungsfunktion ist definiert durch

$$\delta(z, a) = \begin{cases} z + 1 & \text{falls } a = P_{z+1}, \\ \max\{y \leq z : P_1 \dots P_y \text{ ist} \\ \text{Endstück von } P_1 \dots P_z a\} & \text{falls die Menge nicht leer ist und } a \neq P_{z+1}, \\ 0 & \text{sonst.} \end{cases}$$

Die Funktion δ kann als Überföhrungstabelle mit $\#\Sigma \cdot m$ Einträgen gespeichert werden. Jeder Eintrag dieser Tabelle kann in Zeit $O(m^2)$ berechnet werden, so daß M in Zeit $O(\#\Sigma \cdot m^3)$ konstruiert werden kann.

Der Text T wird also in den Automaten M eingegeben. Für jedes Zeichen von T kann der Übergang von M (wenn dessen Überföhrungsfunktion δ explizit als Tabelle gegeben ist) in Zeit $O(1)$ realisiert werden. Die Laufzeit der Suche ist also $O(n)$. Als Gesamtlaufzeit für das String-Matching-Problem erhalten wir somit $O(n + \#\Sigma \cdot m^3)$. Falls m (und $\#\Sigma$) „klein“ ist im Vergleich zu n , so ist dies eine deutliche Verbesserung gegenüber dem naiven Algorithmus.

Eine noch bessere Laufzeit von $O(n+m)$ garantiert der *Knuth-Morris-Pratt-Algorithmus*. Die Idee ist, statt eines DFA nur eine „Präfix-Tabelle“ $\Pi = \Pi(P)$ mit m Einträgen als Hilfskonstruktion zu verwenden. Die Einträge Π_1, \dots, Π_m von Π sind definiert durch

$$\Pi_q = \max\{0 \leq k < q : P_1 \dots P_k = P_{q-k+1} \dots P_q\} \quad (2)$$

(„längstes Anfangsstück von P , das an der Stelle q endet“). Ist beispielsweise $P = \text{ananas}$, so ist $\Pi = (0, 0, 1, 2, 3, 0)$.

Unter Verwendung der Tabelle Π lautet der Algorithmus nun wie folgt.

Algorithmus 12. KMP-Matching

Eingabe: T , P und Π . *Ausgabe:* Alle Stellen in T , an denen P vorkommt.

1. Setze $q = 0$.
2. Für $i = 1, \dots, n$
3. Solange $q > 0$ und $P_{q+1} \neq T_i$, setze $q = \Pi_q$.
4. Wenn $P_{q+1} = T_i$, erhöhe q um 1.
5. Wenn $q = m$, gebe den Treffer $i - m$ aus und setze $q = \Pi_q$.

Lemma 13. *KMP-Matching gibt genau die Stellen aus, an denen P in T auftritt.*

Beweis. Bemerke, daß i die aktuell zu vergleichende Textposition ist und q die Zahl der Stellen, an denen bisher Übereinstimmung mit P festgestellt wurde. Die Behauptung folgt deshalb aus der Definition von Π . \square

Lemma 14. *Die Laufzeit von KMP-Matching ist $O(n)$.*

Beweis. Wir schätzen ab, wie oft sich der Wert von q ändern kann.

- Für jedes i kann Schritt 4 q um 1 erhöhen. Also kann q insgesamt $\leq n$ -mal um 1 wachsen.
- Schritt 3 erniedrigt jeweils q um ≥ 1 . Folglich wird Schritt 3 höchstens n mal durchlaufen.

\square

Lemma 15. *Auf Eingabe P kann Π in Zeit $O(m)$ berechnet werden.*

Beweis. Die Idee ist, P statt in T „in sich selbst“ zu suchen.

Algorithmus 16. KMP-Tabelle

Eingabe: P . *Ausgabe:* Die in (2) definierte Tabelle Π .

1. Setze $\Pi_1^* = k = 0$.
2. Für $q = 2, \dots, m$
3. Solange $k > 0$ und $P_{k+1} \neq P_q$, setze $k = \Pi_k^*$.
4. Wenn $P_{k+1} = P_q$, erhöhe k um 1.
5. Setze $\Pi_q^* = k$.
6. Gebe Π^* aus.

Per Induktion kann man zeigen, daß für alle q gilt $\Pi_q^* = \Pi_q$. Ferner zeigt ein ähnliches Argument wie im Beweis von Lemma 14, daß die Laufzeit von KMP-Tabelle $O(m)$ ist. \square

Kombiniert man die Algorithmen KMP-Matching und KMP-Tabelle miteinander, so erhält man das folgende Resultat.

Satz 17. *Das String-Matching-Problem kann in Zeit $O(n+m)$ gelöst werden.*

4 Graphen

Ist X eine Menge und $k \in \mathbf{N}$, so bezeichnet

$$\binom{X}{k} = \{S \subseteq X : \#S = k\}.$$

Proposition 18. *Ist $\#X = n$, so gilt $\#\binom{X}{k} = \binom{n}{k}$.*

Ein *Graph* ist ein Paar $G = (V, E)$ aus

- einer endlichen Menge V von *Knoten* und
- einer Menge $E \subseteq \binom{V}{2}$ von *Kanten*.

Mit K_n wird der *vollständige Graph* auf n Knoten bezeichnet:

$$K_n = (\{1, \dots, n\}, \binom{\{1, \dots, n\}}{2}).$$

Ferner bezeichnet C_n einen *Kreis* der Länge n und P_n einen *Pfad* der Länge n (d.h. mit $n + 1$ Knoten und n Kanten).

Ist $G = (V, E)$ ein Graph und $v \in V$ ein Knoten, so heißt

$$N(v) = N_G(v) = \{u \in V : \{u, v\} \in E\}$$

die *Nachbarschaft* von v in G , und die Knoten $u \in N(v)$ heißen *Nachbarn* von v ; man sagt auch, u und v sind *benachbart* oder *adjazent*. Ferner bezeichnet

$$\text{grad}(v) = \text{grad}_G(v) = \#N_G(v)$$

den *Grad* von v in G . Ist v ein Knoten und e eine Kante, so heißen v und e *inzident*, falls $v \in e$.

Proposition 19. *Für jeden Graphen $G = (V, E)$ gilt $\sum_{v \in V} \text{grad}(v) = 2\#E$.*

Beweis. Jede Kante trägt den Wert 2 sowohl zu der Summe $\sum_{v \in V} \text{grad}(v)$ als auch zu der rechten Seite $2\#E$ bei. □

Ein *Weg* der Länge l in G ist eine Folge $W = (v_0, \dots, v_l) \in V^{l+1}$, so daß $\{v_i, v_{i+1}\} \in E$ für alle $0 \leq i < l$. Ein Weg W heißt ein *Pfad*, falls v_0, \dots, v_l zusätzlich paarweise verschieden sind. Um Anfang und Ende des Weges hervorzuheben, spricht man auch von einem v_0 - v_l -*Weg/Pfad*. Die Knoten v_1, \dots, v_{l-1} nennt man entsprechend die *Inneren* Knoten von W . Ein *Kreis* der Länge $l \geq 3$ in G ist eine Folge $C = (v_1, \dots, v_l) \in V^l$, so daß (v_1, \dots, v_l) ein Pfad ist und außerdem $\{v_1, v_l\} \in E$.

Ein Graph $H = (V_H, E_H)$ heißt *Untergraph* von $G = (V, E)$, falls $V_H \subseteq V$ und $E_H \subseteq E$. Falls darüber hinaus $E_H = E \cap \binom{V_H}{2}$, nennen wir H einen *induzierten* Untergraphen. Ist $X \subseteq V$, so bezeichnet $G[X]$ den *auf X induzierten Untergraphen* von G :

$$G[X] = (X, \binom{X}{2} \cap E).$$

Für einen Graphen $G = (V, E)$ und eine Menge $S \subseteq V$ sei

$$G - S = G[V \setminus S].$$

Ist ferner $F \subseteq \binom{V}{2}$, so definieren wir

$$G - F = (V, E \setminus F), \quad G + F = (V, E \cup F).$$

Wir definieren eine Äquivalenzrelation \sim_G wie folgt: für $(v, w) \in V^2$ gelte

$$v \sim_G w \Leftrightarrow v = w \text{ oder es gibt einen } v\text{-}w\text{-Pfad in } G.$$

Wir sagen, w ist von v *erreichbar*, falls $v \sim_G w$. Die Äquivalenzklasse

$$[v]_G = \{w \in V : v \sim_G w\}$$

heißt die *Komponente* (oder auch *Zusammenhangskomponente*) von $v \in V$. Schließlich heißt G *zusammenhängend*, falls die Äquivalenzrelation Index 1 hat.

Proposition 20. *Jeder Graph $G = (V, E)$ hat $\geq \#V - \#E$ Komponenten.*

Beweis. Wir führen Induktion über $m = \#E$. Ist $m = 0$, so bildet jeder Knoten eine Komponente. Sei nun G ein Graph mit $m > 0$ Kanten. Wähle $e \in E$ beliebig und betrachte $G' = G - e$. Dann hat G' nach Induktion $\geq \#V - m + 1$ Komponenten. Weil durch Hinzufügen der Kante e zu G' die Zahl der Komponenten allenfalls um 1 sinken kann, folgt daraus, daß G nicht weniger als $\#V - m$ Komponenten hat. \square

Korollar 21. *Ein zusammenhängender Graph $G = (V, E)$ hat $\geq \#V - 1$ Kanten.*

Ein *Wald* ist ein Graph $G = (V, E)$, der keinen Kreis enthält. Ein zusammenhängender Wald heißt ein *Baum*. Ferner ist ein *Blatt* ein Knoten vom Grad 1.

Proposition 22. *Jeder Baum auf mindestens zwei Knoten enthält mindestens zwei Blätter.*

Beweis. Betrachte einen längsten Pfad in dem Baum. \square

Proposition 23. *Ein Baum $G = (V, E)$ hat genau $\#V - 1$ Kanten.*

Beweis. Induktion über $n = \#V$. Für $n = 1$ ist nichts zu zeigen. Angenommen $n \geq 2$; dann hat G nach Proposition 22 ein Blatt v . Da $G - v$ ein Baum ist, hat $G - v$ nach Induktion genau $n - 2$ Kanten, weshalb G genau $n - 1$ Kanten enthält. \square

Proposition 24. *Sei $G = (V, E)$ ein Baum und $v \in V$. Seien T_1, \dots, T_k die Komponenten von $G - v$. Dann sind die Untergraphen $G[T_i]$ für $i = 1, \dots, k$ Bäume, und $k = \text{grad}_G(v)$.*

Beweis. Daß die Untergraphen $G[T_i] = (T_i, E_i)$ Bäume sind, folgt direkt aus der Definition. Um zu zeigen, daß $k = \text{grad}_G(v)$, beobachten wir, daß

$$\#V = 1 + \sum_{i=1}^k \#T_i, \quad (3)$$

$$\#E = \text{grad}_G(v) + \sum_{i=1}^k \#E_i. \quad (4)$$

Da $\#E_i = \#V_i - 1$ und $\#E = \#V - 1$ aufgrund von Proposition 23, folgt aus (3) und (4), daß

$$\#V - 1 = \text{grad}_G(v) - k + \sum_{i=1}^k \#T_i = \text{grad}_G(v) - k + \#V - 1,$$

weshalb $k = \text{grad}_G(v)$. \square

Lemma 25. *Sei $G = (V, E)$ ein zusammenhängender Graph, der einen Kreis C enthält. Sei ferner e eine Kante des Kreises C . Dann ist $G - e$ zusammenhängend.*

Beweis. Wäre $G - e$ unzusammenhängend, so lägen die beiden Endknoten u, v der Kante e in verschiedenen Komponenten von $G - e$. Weil aber e auf dem Kreis C liegt, gibt es in G einen u - v -Pfad P , der die Kante e nicht benutzt. Da P aber auch in $G - e$ enthalten ist, liegen u und v in derselben Komponente von $G - e$, das ist ein Widerspruch. Folglich war die Annahme, daß $G - e$ unzusammenhängend ist, falsch. \square

Ein Untergraph $T = (U, F)$ des Graphen $G = (V, E)$ heißt ein *spannender Baum* von G , falls $U = V$ und T ein Baum ist.

Proposition 26. *Jeder zusammenhängende Graph hat einen spannenden Baum.*

Beweis. Sei $G = (V, E)$ zusammenhängend und $m = \#E$. Wir führen Induktion über m . Im Fall $m = 0$ ist $\#V \leq 1$, so daß G ein Baum ist. Ist $m > 0$ und G selbst kein Baum, so enthält G einen Kreis C . Sei e eine Kante des Kreises C . Dann ist $G - e$ zusammenhängend (nach Lemma 25) und enthält folglich nach Induktion einen spannenden Baum. \square

Satz 27. *Der vollständige Graph K_n hat genau n^{n-2} spannende Bäume ($n \geq 2$).*

Beweis. Sei $V = \{1, \dots, n\}$. Das folgende Codierungsverfahren ordnet einem spannenden Baum des K_n ein Codewort $(t_1, \dots, t_{n-2}) \in V^{n-2}$ zu.

Algorithmus 28. Codieren

Eingabe: Ein spannender Baum $T = (V, E)$ des K_n .

Ausgabe: Ein Codewort (t_1, \dots, t_{n-2}) .

1. Für $i = 1, \dots, n - 2$:
2. Bestimme das kleinste Blatt v von T ; t_i ist der Nachbar von v in T .
3. Setze $T = T - v$.

Das entsprechende Verfahren zur Decodierung lautet:

Algorithmus 29. Decodieren

Eingabe: Ein Codewort $(t_1, \dots, t_{n-2}) \in V^{n-2}$.

Ausgabe: Ein spannender Baum $T = (V, E)$ des K_n .

1. Setze $S = \emptyset$ und $T = (V, \emptyset)$.
2. Für $i = 1, \dots, n - 2$:
3. Wähle $s_i \in V \setminus (S \cup \{t_i, \dots, t_{n-2}\})$ minimal.
4. Füge die Kante $e_i = \{s_i, t_i\}$ zu T hinzu.
5. Setze $S = S \cup \{s_i\}$.
6. Füge die Kante $e_{n-1} = V \setminus S$ zu T hinzu.

Die beiden Verfahren sind zueinander invers, d.h. $\text{Decodieren}(\text{Codieren}(T)) = T$ und

$$\text{Codieren}(\text{Decodieren}(t_1, \dots, t_{n-2})) = (t_1, \dots, t_{n-2}).$$

Daraus folgt, daß die Menge der spannenden Bäume des K_n in Bijektion steht zu der Menge V^{n-2} , deren Kardinalität gerade n^{n-2} ist. \square

5 Breiten- und Tiefensuche

Es geht um die folgenden algorithmischen Probleme:

- Bestimmen der *Zusammenhangskomponenten* eines gegebenen Graphen.
- Berechnen eines spannenden Baumes in einem zusammenhängenden Graphen.
- Berechnen eines kürzesten Weges zwischen zwei Knoten.

Wir gehen stets davon aus, daß der Eingabegraph in Form einer *Adjazenzliste* gegeben ist. Der Graphen $G = (V, E)$ mit Knotenmenge $V = \{v_1, \dots, v_n\}$ und Kantenmenge $E = \{e_1, \dots, e_m\}$ wird beschrieben durch die folgende Datenstruktur:

- Jedem Knoten v_i entspricht eine verkettete Liste L_i , die seine Nachbarn enthält. Die Ordnung, in der die Nachbarn von v_i in der Liste L_i auftreten, ist *beliebig*.
- Ferner wird ein Feld der Größe n gespeichert, das zu jedem Index i einen Zeiger auf die Liste L_i enthält.

Die gesamte Datenstruktur besteht also aus $\Theta(n + m)$ natürlichen Zahlen, die jeweils $O(n^2)$ sind.

Mit Hilfe der Adjazenzliste können wir beispielsweise die folgenden elementaren Operationen ausführen:

- für zwei Knoten v, w testen, ob $\{v, w\} \in E$; Laufzeit: $O(\min\{\text{grad}_G(v), \text{grad}_G(w)\})$.
- die Nachbarn eines Knotens v aufzählen; Laufzeit: $O(\text{grad}_G(v))$.
- eine Kante $\{v, w\} \in \binom{V}{2} \setminus E$ zu G hinzufügen; Laufzeit: $O(1)$.

Wir lernen im folgenden zwei einfache aber vielfach verwendbare Algorithmen kennen, um die Knoten und Kanten eines Graphen zu durchlaufen, nämlich Breiten- und Tiefensuche. Mit diesen Algorithmen können die eingangs genannten Probleme in linearer Zeit $O(n + m)$ gelöst werden.

Für die Breitenuche benötigen wir eine *Warteschlange*. Dies ist eine Datenstruktur, die folgende Operationen zuläßt:

Anlegen. Eine neue Warteschlange kann in Zeit $O(1)$ angelegt werden. Sie enthält am Anfang keine Einträge.

Testen. In Zeit $O(1)$ kann abgefragt werden, ob die Warteschlange leer ist.

Hinzufügen. In eine existierende Warteschlange Q kann ein neues Element v in Zeit $O(1)$ eingefügt werden.

Entnehmen. Aus einer Warteschlange kann in Zeit $O(1)$ ein Element entnommen werden, wenn diese nicht leer ist. Werden die Elemente in der Reihenfolge v_1, \dots, v_k eingefügt, so werden sie auch in *derselben Reihenfolge* wieder entnommen („fiffo“).

Der Algorithmus Breitenuche („BFS“ für „breadth first search“) lautet wie folgt.

Algorithmus 30. BFS

Eingabe: Ein Graph $G = (V, E)$ und ein Startknoten $s \in V$.

Ausgabe: Listen $(d_v)_{v \in V}$, $(p_v)_{v \in V}$.

1. Setze anfangs $d_v = \infty$ und $p_v = \emptyset$ für alle $v \in V$.
Setze anschließend $d_s = 0$.
2. Lege eine Warteschlange Q an und füge s in Q ein.
3. Solange Q nicht leer ist
4. Entnehme ein Element v aus Q .
 Für alle $u \in N_G(v)$
5. Falls $d_u = \infty$
 setze $d_u = d_v + 1$, $p_u = v$ und füge u in Q ein.

Satz 31. *Hat $G = (V, E)$ n Knoten und m Kanten, so läuft BFS in Zeit $O(n + m)$. Ferner gilt:*

1. d_v ist die Länge eines kürzesten s - v -Pfades in G (bzw. ∞ , falls kein solcher Pfad existiert).
2. Wenn G zusammenhängend ist, ist

$$T = (V, \{\{v, p_v\} : v \in V \setminus \{s\}\})$$

ein spannender Baum von G .

Beweis. Die Initialisierung in Schritt 1 kostet Zeit $O(n)$. Ferner wird Schritt 5 insgesamt höchstens

$$\sum_{v \in V} \text{grad}_G(v) = 2\#E \quad (\text{vgl. Proposition 19})$$

mal durchlaufen, so daß die Laufzeit der Schritte 3–5 $O(m)$ ist. Insgesamt erhalten wir $O(n+m)$.

Um 1. zu beweisen, bemerken wir, daß nach Konstruktion (Schritt 5) für alle $v \in V$ mit $p_v \neq \emptyset$ gilt, daß

$$d_v = d_{p_v} + 1 \geq 1.$$

Folgt man daher dem Pfad

$$v, p_v, p_{p_v}, \dots \quad (5)$$

so nimmt in jedem Schritt der „ d -Wert“ um 1 ab. Weil s der einzige Knoten „ d -Wert“ 0 ist, endet der Pfad in s und hat Länge d_v .

Andererseits haben wir zu zeigen, daß es keinen kürzeren v - s -Pfad als (5) gibt. Um dies zu zeigen, stellen wir fest, daß die „ d -Werte“ der Knoten in Q sich zu jedem Zeitpunkt um höchstens eins unterscheiden, wobei die Knoten mit den kleineren „ d -Werten“ vorn stehen. Daraus folgt, daß

$$\forall \{x, y\} \in E : d_x \leq d_y + 1. \quad (6)$$

Sei nun $s = u_0, \dots, u_k = v$ ist irgendein s - v -Pfad. Dann folgt aus (6), daß

$$d_v = d_{u_k} \leq d_{u_{k-1}} + 1 \leq d_{u_{k-2}} + 2 \dots \leq d_{u_0} + k = d_s + k = k,$$

weshalb der Pfad (5) der Länge d_v ein kürzester v - s -Pfad ist.

Die zweite Aussage ist eine unmittelbare Konsequenz der Tatsache, daß (5) ein v - s -Pfad ist. \square

Korollar 32. Die Zusammenhangskomponenten eines gegebenen Graphen $G = (V, E)$ können in Zeit $O(n+m)$ berechnet werden.

Beweis. Modifiziere BFS wie folgt:

- Beginne von einem beliebigen Startknoten s und führe die Schritte 1–5 aus. Die Knoten v mit $d_v < \infty$ bilden eine Zusammenhangskomponente.
- Markiere anschließend alle Knoten v mit $d_v < \infty$ als „tot“ und setze $d_v = \infty$ und $p_v = \emptyset$.
- Wähle einen neuen, noch nicht als „tot“ markierten Startknoten und wiederhole die Schritte 2–5, wobei die als „tot“ markierten Knoten ignoriert werden. Wiederum bilden die Knoten v mit $d_v < \infty$ eine Zusammenhangskomponente.
- Wiederhole dies, bis alle Knoten des Graphen als „tot“ markiert sind.

\square

Eine weitere Art, einen Graphen zu Durchlaufen, ist Tiefensuche („DFS“ für „depth first search“). Die notwendige Datenstruktur ist ein *Keller*, der folgende Operationen zuläßt:

Anlegen. Ein neuer Keller kann in Zeit $O(1)$ angelegt werden. Er enthält am Anfang keine Einträge.

Testen. In Zeit $O(1)$ kann abgefragt werden, ob der Keller leer ist.

Hinzufügen. In einen existierenden Keller K kann ein neues Element v in Zeit $O(1)$ eingefügt werden.

Entnehmen. Aus einem Keller kann in Zeit $O(1)$ ein Element entnommen werden, wenn dieser nicht leer ist. Werden die Elemente in der Reihenfolge v_1, \dots, v_k eingefügt, so werden sie in der umgekehrten Reihenfolge v_k, \dots, v_1 wieder entnommen („lifo“).

Algorithmus 33. DFS

Eingabe: Ein Graph $G = (V, E)$ und ein Startknoten $s \in V$.

Ausgabe: Eine Liste $(p_v)_{v \in V}$.

1. Setze anfangs $p_v = \emptyset$ für alle $v \in V$.
2. Lege einen (leeren) Keller S an.
3. Setze $v = s$.
Wiederhole
4. Wenn es ein $u \in N_G(v) \setminus \{s\}$ mit $p_u = \emptyset$ gibt, füge v zu S hinzu, setze $p_u = v$ und $v = u$; andernfalls prüfe, ob S nicht leer ist, entnehme in diesem Fall ein Element w aus S und setze $v = w$. Sonst setze $v = \emptyset$.
5. bis $v = \emptyset$.

Satz 34. Ist $G = (V, E)$ ein Graph mit n Knoten und m Kanten, so hat DFS Laufzeit $O(n + m)$. Falls G zusammenhängend ist, ist

$$T = (V, \{\{v, p_v\} : v \in V \setminus \{s\}\})$$

ein spannender Baum von G .

6 Minimal spannende Bäume

Gegeben: Ein zusammenhängender Graph $G = (V, E)$ mit Kantengewichten $w : E \rightarrow \mathbf{N}$.

Gesucht: Ein spannender Baum $T = (V, F)$ von G , dessen Gewicht $w(T) = \sum_{e \in F} w(e)$ minimal ist.

Algorithmus 35. Kruskal

Eingabe: Ein zusammenhängender Graph $G = (V, E)$ und eine Funktion $w : E \rightarrow \mathbf{N}$.

Ausgabe: Ein spannender Baum T .

1. Sortiere die Kanten E von G nach aufsteigenden Gewichten:

$$w(e_1) \leq w(e_2) \leq \dots \leq w(e_m).$$

2. Initialisiere $T = (V, \emptyset)$.
3. Für $i = 1, \dots, m$
4. Wenn $T + e_i$ kreisfrei ist, füge die Kante e_i zu T hinzu.

Lemma 36. *Kruskal* berechnet einen minimal spannenden Baum von (G, w) .

Beweis. Sei T die Ausgabe von *Kruskal*. Die Konstruktion stellt sicher, daß T kreisfrei ist. Weil ferner G zusammenhängend ist, trifft dasselbe auf T zu, so daß T ein spannender Baum ist.

Für einen beliebigen spannenden Baum T' von G definieren wir

$$\mu(T') = \min \{1 \leq i \leq m : e_i \text{ liegt in } T \text{ aber nicht in } T'\}$$

(mit der Konvention, daß $\mu(T) = \infty$). Ferner sei T^* ein minimal spannender Baum von G , für den der Wert $\mu(T^*)$ größtmöglich ist. Falls $T = T^*$, ist T ein minimal spannender Baum.

Wir nehmen nun für einen Widerspruchsbeweis an, daß $T \neq T^*$. Sei $1 \leq i \leq m$ minimal mit der Eigenschaft, daß e_i zu T aber nicht zu T^* gehört. Dann enthält $T^* + e_i$ einen Kreis C , der wiederum eine Kante e_j enthält, welche nicht zu T gehört. Es gilt $j > i$, weil T^* alle Kanten aus der Menge $\{e_1, \dots, e_{i-1}\}$ enthält, welche zu T gehören. Dies impliziert, daß $w(e_j) \geq w(e_i)$. Daher ist $T^{**} = T^* - e_j + e_i$ ebenfalls ein minimal spannender Baum von G . Aber da $\mu(T^{**}) > i = \mu(T^*)$, erhalten wir einen Widerspruch zur Wahl von T^* . Dieser zeigt, daß $T = T^*$. \square

Satz 37. Sei G ein Graph mit n Knoten und m Kanten. *Kruskal* bestimmt in Zeit $O(m \log n)$ einen minimal spannenden Baum.

Beweis. Daß die Ausgabe von *Kruskal* ein minimal spannender Baum ist, zeigt Lemma 36. In Bezug auf die Laufzeit bemerken wir, daß $n - 1 \leq m$, weil G zusammenhängend ist. Die Sortieroperation im ersten Schritt kann mit *MergeSort* in Zeit

$$O(m \log m) \leq O\left(m \log \binom{n}{2}\right) = O(m \log n)$$

ausgeführt werden.

Die Laufzeit der Schritte 2–3 ist ebenfalls $O(m \log n)$, wenn man den Test auf Kreisfreiheit wie folgt implementiert: zu jedem Knoten wird ein Eintrag angelegt, der die Nummer seiner Zusammenhangskomponente in dem aktuellen Wald T enthält; anfangs hat T n Komponenten, die von $1, \dots, n$ nummeriert sind. Jedesmal, wenn eine Kante $e = \{u, v\}$ zu T hinzugefügt wird, werden die Komponenten von u und v vereinigt. Die Komponentennummern werden entsprechend aktualisiert, indem diejenigen in der *kleineren* Komponente durch die Nummer der größeren Komponente ersetzt werden. Auf diese Art ändert sich für jeden Knoten höchstens $\log n$ -mal die Komponentennummer, was auf die gewünschte Laufzeit $O(m \log n)$ führt. \square

7 Kürzeste Pfade

Gegeben: Ein zusammenhängender Graph $G = (V, E)$ mit *Kantengewichten* $\ell : E \rightarrow \mathbf{N}$. Zwei Knoten s, t .

Gesucht: Ein Pfad $P = (v_0, \dots, v_k)$ von $s = v_0$ nach $t = v_k$, dessen *Länge*

$$\ell(P) = \sum_{i=1}^k \ell(\{v_{i-1}, v_i\})$$

minimal ist.

Für zwei Knoten $s, t \in V$ sei $D(s, t)$ die Länge eines kürzesten s - t -Pfades.

Algorithmus 38. Dijkstra

Eingabe: G, ℓ, s wie oben.

Ausgabe: Eine Abbildung $d : V \rightarrow \mathbf{N}$.

1. Setze anfangs $d(v) = \infty$ für alle $v \in V$.
Setze anschließend $d(s) = 0$ und $Q = V$.
2. Solange $Q \neq \emptyset$
3. Wähle $x \in Q$ so, daß $d(x) = \min_{v \in Q} d(v)$.
Setze $Q = Q \setminus \{x\}$.
4. Für alle $y \in N_G(x)$ setze $d(y) = \min \{d(y), d(x) + \ell(\{x, y\})\}$.

Lemma 39. Für die Ausgabe $d : V \rightarrow \mathbf{N}$ von *Dijkstra* gilt $d(t) = D(s, t)$ für alle $t \in V$.

Beweis. Wir zeigen, daß für jeden Knoten t zu dem Zeitpunkt, wenn er aus Q entfernt wird, gilt, daß $d(t) = D(s, t)$. Dazu führen wir Induktion über die Zahl der bereits aus Q entfernten Knoten. Weil s als erster Knoten aus Q entfernt wird, ist der Induktionsanfang klar. Nehme nun an, daß der Knoten x aus Q entfernt wird. Dann gibt es (aufgrund der Konstruktion in Schritt 4) einen Pfad von s nach x , dessen Länge $\leq d(x)$ ist. Also folgt $d(x) \geq D(s, x)$. Um die umgekehrte Ungleichung zu beweisen, betrachten wir einen s - x -Pfad

$$s = v_1, \dots, v_k = x$$

der Länge $D(s, x)$. Sei i der größte Index auf diesem Pfad, so daß v_i vor x aus Q entfernt wurde. Nach Induktion gilt

$$d(v_i) = D(s, v_i). \tag{7}$$

Weil v_{i+1} *nicht* vor x aus Q entfernt wird, haben wir

$$d(x) \leq d(v_{i+1}) \leq d(v_i) + \ell(\{v_i, v_{i+1}\}) \stackrel{(7)}{=} D(s, v_i) + \ell(\{v_i, v_{i+1}\}) \leq \ell(v_1, \dots, v_k) = D(s, x).$$

Folglich erhalten wir $d(x) = D(s, x)$. \square

Satz 40. Sei $n = \#V$. *Dijkstra* berechnet in Zeit $O(n^2)$ die Funktion $d(t) = D(s, t)$.

Beweis. Lediglich die Behauptung über die Laufzeit bleibt zu zeigen. Dazu bemerken wir, daß in jeder Iteration der Schritte 3–4 die Zahl $\#Q$ um 1 sinkt. Daher werden insgesamt n Iterationen durchgeführt. Die Berechnung des Minimums in Schritt 3 nimmt je Iteration Zeit $O(n)$ in Anspruch, insgesamt also Zeit $O(n^2)$. Schließlich benötigt Schritt 4 insgesamt Laufzeit $O(\sum_{v \in V} \text{grad}_G(v)) = O(m) = O(n^2)$. \square

Die oben dargestellte Version von Dijkstra bestimmt lediglich die *Abstände* von s zu allen anderen Knoten, aber nicht die entsprechenden *kürzesten Pfade*. Diese können jedoch generiert werden; dazu führt man ein weiteres Feld $(p_v)_{v \in V}$ ein, das zu jedem Knoten v seinen „Vorgänger“ auf einem kürzesten Weg zu s speichert.

Mit Hilfe von *Fibonacci-Heaps* kann die Laufzeit des Dijkstra-Algorithmus' auf $O(m + n \log n)$ reduziert werden.

8 Gruppen

In diesem und dem folgenden Abschnitt geht es um *Primzahlen* und um Algorithmen, die entscheiden, ob eine gegebene Zahl prim ist. Zur Erinnerung: $x \in \mathbf{Z}$ teilt $y \in \mathbf{Z}$, falls es ein $z \in \mathbf{Z}$ gibt, so daß $xz = y$; man schreibt $x|y$ für „ x teilt y “ und $x \nmid y$ für „ x teilt y nicht“. Eine Zahl $n \geq 2$ heißt eine *Primzahl*, wenn es keine Zahl $1 < x < n$ gibt, die n teilt. Der wichtigste Satz über Primzahlen ist der folgende sog. „Fundamentalsatz der Arithmetik“.

Satz 41. *Jede natürliche Zahl $n \geq 2$ kann eindeutig als Produkt $n = \prod_{i=1}^k p_i^{e_i}$ dargestellt werden, wobei $k \in \mathbf{N}$, $p_1 < p_2 < \dots < p_k$ Primzahlen sind, und $e_1, \dots, e_k \geq 1$ natürliche Zahlen.*

Satz 41 impliziert, daß eine Zahl p genau dann prim ist, wenn folgendes gilt:

$$\forall x, y \in \mathbf{Z} : (p|x \cdot y) \rightarrow (p|x \vee p|y).$$

Um algorithmisch zu testen, ob eine gegebene Zahl prim ist, benötigen wir einige elementare Fakten über endliche abelsche Gruppen. Eine *abelsche Gruppe* ist eine Menge G zusammen mit einer Abbildung $*$: $G \times G \rightarrow G$, $(a, b) \mapsto a * b$, die die folgenden Bedingungen erfüllt.

Kommutativität. Für alle $a, b \in G$ gilt $a * b = b * a$.

Assoziativität. Für alle $a, b, c \in G$ gilt $(a * b) * c = a * (b * c)$.

Neutrales Element. Es gibt ein $e \in G$, so daß für alle $a \in G$ gilt $e * a = a$.

Inverses Element. Zu jedem $a \in G$ gibt es ein $b \in G$, so daß $ab = e$.

Ein Beispiel für eine Gruppe ist $G = \mathbf{Z}$ mit $*$ = +. Von besonderer algorithmischer Bedeutung sind allerdings *endliche* Gruppen, die aus \mathbf{Z} konstruiert werden können. Sei dazu $n \geq 2$ eine ganze Zahl. Wir definieren eine Äquivalenzrelation $\equiv \pmod n$ („Kongruenz modulo n “) auf \mathbf{Z} wie folgt:

Für $x, y \in \mathbf{Z}$ gilt $x \equiv y \pmod n$ genau dann, wenn $n|y - x$.

Die Äquivalenzklasse der Zahl x bezeichnen wir mit $x \pmod n$, d.h.

$$x \pmod n = \{y \in \mathbf{Z} : n|y - x\}.$$

Da man jede Zahl $x \in \mathbf{Z}$ durch n mit Rest dividieren kann (d.h. x kann in der Form $x = qn + r$ mit $0 \leq r < n$ dargestellt werden), hat die Äquivalenzrelation $\equiv \pmod n$ Index n , und die verschiedenen Äquivalenzklassen sind

$$0 \pmod n, 1 \pmod n, \dots, n - 1 \pmod n.$$

Mit

$$\mathbf{Z}_n = \{k \pmod n : k \in \mathbf{Z}\}$$

bezeichnen wir die Menge aller n Äquivalenzklassen.

Die Kongruenzrelation ist mit der Addition und Multiplikation in folgendem Sinne verträglich: für alle $x, y, x', y' \in \mathbf{Z}$ gilt:

$$\begin{aligned}(x \equiv x' \pmod{n}) \wedge (y \equiv y' \pmod{n}) &\Rightarrow x + y \equiv x' + y' \pmod{n}, \\ (x \equiv x' \pmod{n}) \wedge (y \equiv y' \pmod{n}) &\Rightarrow x \cdot y \equiv x' \cdot y' \pmod{n}.\end{aligned}$$

Daher kann man auf \mathbf{Z}_n ebenfalls eine „Addition“ sowie eine „Multiplikation“ definieren:

$$\begin{aligned}+ : \mathbf{Z}_n \times \mathbf{Z}_n &\rightarrow \mathbf{Z}_n, (x \pmod{n}, y \pmod{n}) \mapsto (x + y) \pmod{n}, \\ \cdot : \mathbf{Z}_n \times \mathbf{Z}_n &\rightarrow \mathbf{Z}_n, (x \pmod{n}, y \pmod{n}) \mapsto (x \cdot y) \pmod{n}.\end{aligned}$$

Mit anderen Worten,

$$(x \pmod{n}) + (y \pmod{n}) = (x + y) \pmod{n}, \quad (x \pmod{n}) \cdot (y \pmod{n}) = (x \cdot y) \pmod{n}.$$

Proposition 42. \mathbf{Z}_n ist bezüglich der oben definierten Addition eine abelsche Gruppe.

Der sog. „Chinesische Restsatz“ charakterisiert, wie sich für eine Zahl $n = n_1 \cdot n_2$ die Gruppen $\mathbf{Z}_n, \mathbf{Z}_{n_1}, \mathbf{Z}_{n_2}$ zueinander verhalten.

Satz 43. Seien $n_1, n_2 \geq 2$ Zahlen, deren größter gemeinsamer Teiler 1 ist. Sei $n = n_1 \cdot n_2$. Dann ist die Abbildung

$$\mathbf{Z}_n \rightarrow \mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2}, \quad x \pmod{n} \mapsto (x \pmod{n_1}, x \pmod{n_2}) \tag{8}$$

bijektiv. Zu je zwei Zahlen $y, z \in \mathbf{Z}$ gibt es also eine Zahl $x \in \mathbf{Z}$, so daß

$$x \equiv y \pmod{n_1} \quad \text{und} \quad x \equiv z \pmod{n_2}.$$

Beweis. Wir zeigen zunächst, daß die Abbildung (8) injektiv ist. Denn angenommen, $x, y \in \mathbf{Z}$ sind so, daß

$$x \equiv y \pmod{n_1} \quad \text{und} \quad x \equiv y \pmod{n_2}.$$

Dann gilt $n_1|x - y$ und $n_2|x - y$. Weil n_1 und n_2 teilerfremd sind, folgt aus Satz 41, daß $n = n_1 n_2|x - y$, so daß $x \equiv y \pmod{n}$ und folglich $x \pmod{n} = y \pmod{n}$.

Schließlich bemerken wir, daß

$$\#\mathbf{Z}_n = n = n_1 n_2 = \#\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2}.$$

Aus der Injektivität der Abbildung (8) folgt also die Surjektivität derselben. \square

\mathbf{Z}_n ist mit der Multiplikation *keine* Gruppe. Der Grund ist, daß $0 \pmod{n}$ kein inverses Element hat. Daher definieren wir eine Teilmenge

$$\mathbf{Z}_n^* = \{k \pmod{n} : k \in \mathbf{Z} \text{ und } \text{ggT}(k, n) = 1\}.$$

Proposition 44. \mathbf{Z}_n^* ist bezüglich der oben definierten Multiplikation eine abelsche Gruppe.

Der Beweis des folgenden Satzes findet sich etwa in [3].

Satz 45. Wenn n eine Primzahl ist, gibt es eine Zahl $g \in \mathbf{Z}$, so daß

$$\mathbf{Z}_n^* = \{g^i \pmod{n} : 0 \leq i < n - 1\}.$$

Eine *Untergruppe* einer abelschen Gruppe $G, *$ eine eine Teilmenge H von G , die folgende Eigenschaften hat:

Neutrales Element. Das neutrale Element e von G liegt in H .

Abgeschlossenheit. Für alle $a, b \in H$ gilt $a * b \in H$.

Inverses Element. Zu jedem $a \in H$ gibt es ein $c \in H$, so daß $a * c = e$.

Daher ist H zusammen mit $*$ selbst wiederum eine Gruppe.

Proposition 46. *Wenn G eine abelsche Gruppe und $H \subseteq G$ eine endliche Teilmenge ist, die die beiden obigen Eigenschaften „Neutrales Element“ und „Abgeschlossenheit“ hat, so ist H eine Untergruppe von G .*

Beweis. Zu jedem Element $a \in H$ können wir aufgrund der Abgeschlossenheit eine Abbildung

$$f_a : H \rightarrow H, \quad x \mapsto a * x$$

definieren. Diese Abbildung ist injektiv; denn sei $b \in G$ das inverse Element von a in der Gruppe G , und seien $x, y \in H$ so, daß

$$a * x = f_a(x) = f_a(y) = a * y.$$

Dann folgt $x = e * x = (b * a) * x = b * (a * x) = b * (a * y) = (b * a) * y = e * y = y$.

Weil H eine endliche Menge ist, folgt aus der Injektivität von f_a die Surjektivität. Da ferner $e \in H$, gibt es ein $c \in H$, so daß $a * c = e$. Also hat H die Eigenschaft „Inverses Element“ und ist somit eine Untergruppe. \square

Satz 47. *Wenn G eine endliche abelsche Gruppe und $H \subseteq G$ eine Untergruppe ist, so gilt $\#H \mid \#G$.*

Beweis. Für $x \in G$ definieren wir $H_x = \{x * h : h \in H\}$. Ferner definieren wir eine Äquivalenzrelation \sim auf G durch

$$x \sim y \Leftrightarrow H_x = H_y.$$

Dann ist die Äquivalenzklasse von x genau die Menge

$$[x] = \{y \in G : y \sim x\} = \{y \in G : \bar{y} * x \in H\},$$

wobei \bar{y} das zu y inverse Element bezeichnet. Für jedes $x \in G$ ist die Abbildung

$$[x] \rightarrow H = [e], \quad y \mapsto y * \bar{x}$$

eine Bijektion. Folglich gilt für je zwei Elemente $x, x' \in G$, daß $\#[x] = \#[x'] = \#H$, so daß

$$\#G = \#H \cdot \text{Index der Relation } \sim.$$

Insbesondere haben wir $\#H \mid \#G$. \square

Korollar 48. *Für jede Primzahl n und jede Zahl $1 \leq a < n$ gilt $a^{n-1} \equiv 1 \pmod{n}$.*

Beweis. Die Menge $H = \{a^i \pmod{n} : i \in \mathbf{N}\}$ ist nach Proposition 46 eine Untergruppe von \mathbf{Z}_n^* . Weil H endlich ist, gibt es eine Zahl $j > 0$, so daß $a^j \equiv 1 \pmod{n}$. Sei

$$k = \min\{j > 0 : a^j \equiv 1 \pmod{n}\}.$$

Dann ist $H = \{a^j \pmod{n} : 0 \leq j < k\}$, so daß $\#H = k$. Nach Satz 47 gilt daher $k \mid n - 1$. Sei also $q \in \mathbf{N}$ so, daß $kq = n - 1$. Dann erhalten wir

$$a^{n-1} = a^{kq} = (a^k)^q \equiv 1^q = 1 \pmod{n},$$

wie behauptet. \square

9 Primzahlen

Gegeben: Eine Zahl $n \in \mathbf{N}$.

Frage: Ist n eine Primzahl?

Ein *naiver Algorithmus* für dieses Problem ist, für jede natürliche Zahl $2 \leq k \leq \sqrt{n}$ zu testen, ob $k|n$. Weil jedoch die Zahl der Tests \sqrt{n} ist, während die *Größe der Eingabe* $\log n$ ist, ist die Laufzeit dieses naiven Verfahrens *exponentiell* und der Algorithmus daher nicht effizient. Das Ziel in diesem Abschnitt ist, effizientere (Monte-Carlo-)Primzahltests mit polynomieller Laufzeit $\log^{O(1)} n$ zu entwickeln.

Ein erster Schritt zu einem solchen Algorithmus ist der *Fermat-Test*. Sei $n \in \mathbf{N}$. Wir nennen eine Zahl $1 \leq a < n$ einen *F-Zeugen* für n , falls $a^{n-1} \not\equiv 1 \pmod n$. Nach Korollar 48 hat n *keinen* F-Zeugen, wenn n eine Primzahl ist. Ist n andererseits ungerade aber keine Primzahl, so nennen wir a einen *F-Lügner*, falls $a^{n-1} \equiv 1 \pmod n$; offenbar sind 1 und $n-1$ stets F-Lügner.

Lemma 49. *Sei $n \geq 2$ eine natürliche Zahl.*

1. *Falls $1 \leq a < n$ die Eigenschaft $a^r \equiv 1 \pmod n$ hat für ein $r \geq 1$, so gilt $a \in \mathbf{Z}_n^*$.*
2. *Gilt $a^{n-1} \equiv 1 \pmod n$ für alle $1 \leq a < n$, so ist n eine Primzahl.*

Beweis. zu 1.: Wir haben $a \cdot a^{r-1} = a^r \equiv 1 \pmod n$, weshalb $a \in \mathbf{Z}_n^*$.

zu 2.: Nach 1. gilt $\mathbf{Z}_n^* = \{1, \dots, n-1\}$. Wären jedoch $1 < x \leq y < n$ so, daß $xy = n$, so hätten wir $xy \equiv 0 \pmod n$. Weil $x \in \mathbf{Z}_n^*$, gibt es ein z , so daß $xz \equiv 1 \pmod n$. Folglich gilt

$$0 = 0 \cdot z \equiv (xy)z = y(xz) \equiv y \pmod n,$$

im Widerspruch zur Annahme $1 < y < n$. Also hat n keine Zerlegung in echte Teiler und ist daher eine Primzahl. \square

Korollar 50. *Jede zusammengesetzte Zahl n hat einen F-Zeugen.*

Algorithmus 51. Fermat

Eingabe: Eine ungerade Zahl $n \geq 5$.

Ausgabe: 0 („vielleicht Primzahl“) oder 1 („bestimmt zusammengesetzt“).

1. Wähle $a \in \{2, \dots, n-2\}$ zufällig.
2. Falls $a^{n-1} \not\equiv 1 \pmod n$, antworte 1, sonst 0.

Der Algorithmus kann so implementiert werden, daß seine Laufzeit $\log^{O(1)} n$ (also polynomiell in der Größe der Eingabe) ist; dieser Punkt wird weiter unten genauer diskutiert.

Nach Korollar 48 wird **Fermat** *stets* eine 0 ausgeben, wenn n eine Primzahl ist. Unser Ziel ist also, eine Aussage zu treffen über die Wahrscheinlichkeit, daß die Antwort 0 ist, obwohl n zusammengesetzt ist. Zu diesem Zweck ist es wichtig, die *Anzahl* der F-Zeugen einer zusammengesetzten Zahl zu untersuchen.

Satz 52. *Für jede ungerade zusammengesetzte Zahl $n \geq 3$, die einen F-Zeugen in \mathbf{Z}_n^* hat, gilt*

$$P[\text{Fermat antwortet 1}] \geq \frac{1}{2}.$$

Beweis. Die Menge

$$L = \{a \pmod n : 1 \leq a < n \wedge a^{n-1} \equiv 1 \pmod n\}$$

der F-Lügner ist nach Lemma 49 in \mathbf{Z}_n^* enthalten. Ferner hat L die beiden folgenden Eigenschaften:

1. Es gilt $1 \pmod n \in L$.
2. Sind $a, b \in L$, so folgt $ab \in L$.

Nach Proposition 46 ist also L eine Untergruppe von \mathbf{Z}_n^* . Weil L echt in \mathbf{Z}_n^* enthalten ist, folgt aus Satz 47, daß $\#L \leq \frac{1}{2}\#\mathbf{Z}_n^*$. Da außerdem für alle $a \in \mathbf{Z}_n \setminus \mathbf{Z}_n^*$ gilt $a^{n-1} \not\equiv 1 \pmod{n}$, folgt die Behauptung. \square

Wiederholt man den Fermat-Test l mal (mit unabhängigen Zufallszahlen), so folgt also, daß

- für eine Primzahl n die Ausgabe in allen l Versuchen 0 lautet;
- für eine zusammengesetzte Zahl $n \geq 3$, die einen F-Zeugen in \mathbf{Z}_n^* hat, die Wahrscheinlichkeit, daß in allen l Versuchen 0 ausgegeben wird, $\leq 2^{-l}$ ist.

Wählt man etwa $l = 100$, so ist die Wahrscheinlichkeit, n fälschlicherweise für eine Primzahl zu halten, $\leq 10^{-30}$.

Allerdings hat *nicht* jede zusammengesetzte Zahl $n \geq 3$ einen F-Zeugen in \mathbf{Z}_n^* ; solche Zahlen n nennet man *Carmichael-Zahlen*. Für diese Zahlen ist der Fermat-Test *kein* zufriedenstellender Primzahltest (die Fehlerwahrscheinlichkeit kann sehr nahe bei 1 liegen). Um zu einem besseren Primzahltest zu kommen, benötigen wir die folgende Aussage über Carmichael-Zahlen.

Lemma 53. *Eine Carmichael-Zahl n wird von mindestens drei verschiedenen Primzahlen geteilt.*

Beweis. Wir zeigen, daß eine Zahl, die höchstens zwei Primteiler hat, keine Carmichael-Zahl ist. Der erste Fall ist, daß es eine Primzahl $p \geq 3$ gibt, so daß $p^2|n$. Stelle n dar als $n = p^k m$, $p \nmid m$, $k \geq 2$. Wenn $m = 1$, setzen wir $a = p + 1$; anderenfalls gibt es nach Satz 43 eine Zahl $1 \leq a \leq n$, so daß

$$a \equiv 1 + p \pmod{p^2} \quad \text{und} \quad a \equiv 1 \pmod{m}. \quad (9)$$

Dann gilt $a \in \mathbf{Z}_n^*$; denn $p^2|a - (1 + p)$, so daß $p \nmid a$. Im Fall $m > 1$ gilt außerdem $m \nmid a$, weil $m|a - 1$. Also sind a und n teilerfremd, so daß $a \in \mathbf{Z}_n^*$. Wäre nun $a^{n-1} \equiv 1 \pmod{n}$, so erhielten wir einerseits

$$a^{n-1} \equiv 1 \pmod{p^2}, \quad (10)$$

weil $p^2|n$. Andererseits gilt nach (9)

$$a^{n-1} \equiv (1 + p)^{n-1} = 1 + (n-1)p + \sum_{i=2}^{n-1} \binom{n-1}{i} p^i \equiv 1 + (n-1)p \pmod{p^2}. \quad (11)$$

Kombiniert man (10) und (11), so ergibt dies $(n-1)p \equiv 0 \pmod{p^2}$, so daß $p|n-1$; dies ist aber unmöglich, weil $p|n$. Damit ist der erste Fall zum Widerspruch geführt.

Der zweite Fall ist, daß $n = pq$ für zwei verschiedene Primzahlen $p > q \geq 3$. In diesem Fall gibt es nach Satz 45 eine Zahl $1 \leq g \leq p-1$, so daß

$$\mathbf{Z}_p^* = \{g^i \pmod{p} : 1 \leq i \leq p-1\}. \quad (12)$$

Nach Satz 43 gibt es ferner eine Zahl $1 \leq a < n$, so daß

$$a \equiv g \pmod{p} \quad \text{und} \quad a \equiv 1 \pmod{q}.$$

Weil weder p noch q ein Teiler von a ist, folgt $a \in \mathbf{Z}_n^*$. Nehmen wir nun an, daß $a^{n-1} \equiv 1 \pmod{n}$. Weil $p|n$, folgt daraus

$$g^{n-1} \equiv a^{n-1} \equiv 1 \pmod{p}.$$

Dies impliziert aufgrund von (12), daß

$$p-1|n-1 = pq-1 = (p-1)q + q-1.$$

Also $p-1|q-1$, im Widerspruch zur Annahme $p > q$. \square

Sei $1 \leq a < n$. Wir nennen a eine *Quadratwurzel der 1 modulo n* , falls $a^2 \equiv 1 \pmod{n}$. Die *trivialen* Quadratwurzeln der 1 sind $a = 1$ und $a = n-1$.

Lemma 54. Wenn p eine Primzahl ist, sind 1 und $p - 1$ die einzigen Quadratwurzeln der 1 .

Beweis. Angenommen $a^2 \equiv 1 \pmod{p}$. Dann gilt $p|(a+1)(a-1)$ und folglich $p|a+1$ oder $p|a-1$. \square

Sei $n \geq 3$ ungerade und $n - 1 = 2^k u$ für eine Zahl $k \geq 1$ und eine ungerade Zahl u . Wir nennen $1 \leq a < n$ einen *A-Zeugen* für n , falls

$$a^u \not\equiv 1 \pmod{n} \quad \text{und} \quad a^{u \cdot 2^i} \not\equiv -1 \pmod{n} \quad \text{für alle } 0 \leq i < k.$$

Offenbar ist jeder F-Zeuge für n auch ein A-Zeuge. Wenn n zusammengesetzt ist und a kein A-Zeuge für n ist, nennen wir a einen *A-Lügner* für n .

Der Begriff des A-Zeugen führt auf einen effizienten Monte-Carlo-Algorithmus für das Primzahlproblem.

Algorithmus 55. Miller-Rabin

Eingabe: Eine ungerade Zahl $n \geq 5$.

Ausgabe: 0 („vielleicht Primzahl“) oder 1 („bestimmt zusammengesetzt“).

1. Wähle $a \in \{2, \dots, n - 2\}$ zufällig.
2. Wenn a ein A-Zeuge für n ist, antworte 1 , sonst 0 .

Lemma 56. Wenn n einen A-Zeugen a hat, so ist n zusammengesetzt.

Beweis. Setze $b_i = a^{u \cdot 2^i}$ für $0 \leq i \leq k$. Wenn $b_k \not\equiv 1 \pmod{n}$, so ist a ein F-Zeuge für n , und die Behauptung folgt aus Korollar 48. Sonst sei $i \geq 1$ minimal mit der Eigenschaft, daß $b_i \equiv 1 \pmod{n}$. Weil a ein A-Zeuge ist, folgt $b_{i-1} \not\equiv -1 \pmod{n}$, so daß b_{i-1} eine nichttriviale Quadratwurzel der 1 modulo n ist und die Behauptung aus Lemma 54 folgt. \square

Den Beweis der folgenden Aussage geben wir am Ende dieses Abschnittes.

Lemma 57. Wenn n zusammengesetzt ist, so gilt $P[\text{Miller-Rabin antwortet } 0] \leq \frac{1}{2}$.

Satz 58. Sei $n \geq 3$ ungerade.

1. Falls n eine Primzahl ist, antwortet *Miller-Rabin* stets 1 .
2. Falls n zusammengesetzt ist, gilt $P[\text{Miller-Rabin antwortet } 0] \leq \frac{1}{2}$.

Die Laufzeit von *Miller-Rabin* ist $O(\log^4 n)$, also polynomiell in der Eingabelänge.

Beweis. Die ersten beiden Teile des Satzes folgen aus Lemma 56 und Lemma 57. Es bleibt zu zeigen, daß der Test in Schritt 2 in Zeit $O(\log^4 n)$ durchgeführt werden kann. Das Verfahren dazu lautet wie folgt:

- Zerlege $n - 1 = u \cdot 2^k$ mit $2 \nmid u$ und $k \geq 1$.
- Berechne die Zahl $1 \leq b_0 \leq n$, so daß $b_0 \equiv a^u \pmod{n}$; ist $b \in \{1, n - 1\}$, antworte 0 .
- Bestimme dann iterativ b_i so, daß $b_i \equiv b_{i-1}^2 \pmod{n}$ ($1 \leq i < k$). Falls $b_i = n - 1$, antworte 0 , und wenn $b_i = 1$, antworte 1 .

Der „kritische“ Teil ist die Berechnung von b_0 . Diese Berechnung erfolgt mit der Methode des *schnellen Potenzierens*:

Algorithmus 59. Potenzieren

Eingabe: Zahlen a , n und u .

Ausgabe: Eine Zahl $0 \leq b < n$, so daß $b \equiv a^u \pmod{n}$.

1. Berechne die Binärdarstellung $u = \sum_{i=0}^l u_i 2^i$.
2. Setze $b_0 = a$.
Setze $b = b_0$, wenn $u_0 = 1$, und sonst $b = 1$. Für $i = 1, \dots, l$
3. Berechne b_{i-1}^2 ; teile diese Zahl durch n und bezeichne mit $0 \leq b_i < n$ den Divisionsrest.
Falls $u_i = 1$, setze $b' = b_i b$, berechne den Divisionsrest $0 \leq b'' < n$ von b durch n und setze $b = b''$.

Diesem Algorithmus liegt die Beobachtung zugrunde, daß $b^u = \prod_{1 \leq i \leq l: u_i=1} b^{2^i}$; die Potenzen b^{2^i} werden dabei durch iteriertes (d.h. l -maliges) Quadrieren berechnet. \square

Beweis von Lemma 57. Aufgrund von Satz 52 dürfen wir annehmen, daß n eine Carmichael-Zahl ist. Das Ziel ist, eine Untergruppe $B \subset \mathbf{Z}_n^*$ zu konstruieren, die alle A-Lügner enthält. Die Behauptung folgt dann unmittelbar aus Satz 47.

Sei $n - 1 = 2^k u$ mit $2 \nmid u$, und definiere

$$i_0 = \max\{i \geq 0 : \exists \text{ A-Lügner } \alpha \text{ mit } \alpha^{u2^i} \equiv -1 \pmod{n}\}; \quad (13)$$

die Menge auf der rechten Seite ist nicht leer, weil $(-1)^u = -1$. Sei entsprechend $0 \leq a_0 < n$ ein A-Lügner, so daß

$$a_0^{u2^{i_0}} \equiv -1 \pmod{n}. \quad (14)$$

Weil n eine Carmichael-Zahl ist, ist a_0 ein F-Lügner, so daß

$$a_0^{u2^k} = a_0^{n-1} \equiv 1 \pmod{n}.$$

Wir definieren nun die Menge

$$B = \{a \pmod{n} : 0 < a < n \wedge a^{u2^{i_0}} \equiv \pm 1 \pmod{n}\}. \quad (15)$$

Im folgenden zeigen wir, daß B die drei folgenden Eigenschaften hat.

1. Die Menge L der A-Lügner ist in B enthalten.
2. B ist eine Untergruppe von \mathbf{Z}_n^* .
3. $B \neq \mathbf{Z}_n^*$.

zu 1.: Sei a ein A-Lügner. Wenn $a^u \equiv 1 \pmod{n}$, so folgt $a^{u2^{i_0}} \equiv 1 \pmod{n}$, weshalb $a \in B$. Wenn andererseits $a^{u2^i} \equiv -1 \pmod{n}$ für ein i , so gilt nach (13) $0 \leq i \leq i_0$. Falls $i = i_0$, so stellt (15) sicher, daß $a \in B$; und wenn $i < i_0$, haben wir

$$a^{u2^{i_0}} = (a^{u2^i})^{2^{i_0-i}} \equiv 1 \pmod{n}.$$

Also gilt in jedem Fall $a \in B$.

zu 2.: Offenbar gilt $1 \in B$. Wenn ferner $a, b \in B$, so haben wir

$$a^{u2^{i_0}} \equiv \pm 1 \pmod{n}, \quad b^{u2^{i_0}} \equiv \pm 1 \pmod{n}.$$

Daraus folgt, daß

$$(ab)^{u2^{i_0}} = a^{u2^{i_0}} b^{u2^{i_0}} \equiv (\pm 1)(\pm 1) = \pm 1 \pmod{n},$$

weshalb $ab \in B$. Daher zeigt Proposition 46, daß B eine Untergruppe ist von \mathbf{Z}_n^* ist.

zu 3.: Weil n nach Lemma 53 mindestens drei verschiedene Primteiler hat, können wir n darstellen als Produkt $n = n_1 n_2$, wobei 1 der größte gemeinsame Teiler der ungeraden Zahlen n_1 und n_2 ist. Nach Satz 43 gibt es eine Zahl $0 \leq a < n$, so daß

$$a \equiv a_0 \pmod{n_1} \quad \text{und} \quad a \equiv 1 \pmod{n_2}. \quad (16)$$

Wir behaupten, daß

$$a \in \mathbf{Z}_n^* \setminus B. \quad (17)$$

Wegen (16) und der Wahl (14) von a_0 haben wir

$$a^{u2^{i_0}} \equiv -1 \pmod{n_1} \quad \text{und} \quad a^{u2^{i_0}} \equiv 1 \pmod{n_2}. \quad (18)$$

Dies impliziert, daß

$$a^{u2^{i_0}} \not\equiv 1 \pmod{n} \quad \text{und} \quad a^{u2^{i_0}} \not\equiv -1 \pmod{n},$$

weshalb $a \notin B$. Andererseits zeigt (18)

$$a^{u2^{i_0+1}} \equiv 1 \pmod{n},$$

so daß $a \in \mathbf{Z}_n^*$. Somit haben wir (17) und damit 3. bewiesen. \square

10 Das RSA-Kryptosystem

Ein Sender *Bob* möchte eine Nachricht an eine Empfängerin *Alice* schicken. Allerdings ist der Kanal von Bob zu Alice nicht „abhörsicher“, so daß Bob seine Nachricht verschlüsseln möchte. Bezeichnen wir die Nachricht mit M , so benötigt Bob also eine Verschlüsselungsprozedur P sowie Alice eine Entschlüsselungsprozedur S , so daß

$$S(P(M)) = M. \quad (19)$$

Außerdem sollte die Nachricht M für einen Dritten „schwer“ zu entschlüsseln sein, wenn dieser nur $P(M)$ kennt. In einem *public key*-Kryptosystem ist nun die effiziente Prozedur P öffentlich bekannt (also auch dem „neugierigen Dritten“), während nur Alice S kennt. Jeder Nutzer (Bob eingeschlossen) kann also Alice eine verschlüsselte Nachricht schicken, die nur Alice mit vertretbarem Aufwand dechiffrieren kann. Für jeden anderen Nutzer sollte der Bedarf an Rechenzeit, der zum Entziffern von $P(M)$ nötig ist, extrem groß sein.

Das RSA-Kryptosystem (benannt nach seinen Entwicklern Rivest, Shamir und Adleman) beruht auf der Tatsache, daß derzeit kein effizienter Algorithmus bekannt ist, um eine gegebene Zahl n in Primfaktoren zu zerlegen.

1. Alice wählt zwei „große“ Primzahlen p und q (typischerweise aus mehreren 100 Ziffern) und bestimmt $n = pq$.
2. Ferner wählt Alice eine Zahl $0 < e < n$, so daß $\text{ggT}(e, (p-1)(q-1)) = 1$. Zu dieser Zahl e berechnet Alice eine Zahl $0 < d < n$, so daß $ed \equiv 1 \pmod{(p-1)(q-1)}$.
3. Der *öffentliche Schlüssel*, den Alice allen Nutzern bekanntgibt, lautet (e, n) .
4. Der *geheime Schlüssel*, den Alice für sich behält, ist (d, n) . (Die Primzahlen p, q braucht sich Alice forthin nicht zu merken.)

Wir nehmen an, daß die *Nachrichten*, die Bob an Alice schicken möchte, einfach Zahlen zwischen 0 und $n-1$ sind, d.h. Elemente von \mathbf{Z}_n^* . Die Prozedur P lautet dann

$$P : \mathbf{Z}_n^* \rightarrow \mathbf{Z}_n^*, \quad M \bmod n \mapsto M^e \bmod n.$$

Um den Funktionswert zu berechnen, genügt also die Kenntnis des öffentlichen Schlüssels (e, n) . Die Prozedur S zur Entschlüsselung lautet

$$S : \mathbf{Z}_n^* \rightarrow \mathbf{Z}_n^*, \quad C \bmod n \mapsto C^d \bmod n;$$

um $S(C)$ zu bestimmen, ist also (anscheinend) die Kenntnis des geheimen Schlüssels (d, n) nötig.

Satz 60. *Die beiden Funktionen P und S erfüllen (19).*

Beweis. Sei $ed = 1 + k(p-1)(q-1)$. Dann gilt für jede Zahl $0 \leq M < n$

$$\begin{aligned} S(P(M)) &= S(M^e \bmod n) = M^{ed} \bmod n = M^{1+k(p-1)(q-1)} \bmod n \\ &= (M \bmod n) \cdot (M^{k(p-1)(q-1)} \bmod n). \end{aligned}$$

Es genügt also zu zeigen, daß

$$M^{k(p-1)(q-1)} \equiv 1 \pmod n. \quad (20)$$

Dazu beweisen wir, daß

$$M^{k(p-1)(q-1)} \equiv 1 \pmod p \quad \text{und} \quad M^{k(p-1)(q-1)} \equiv 1 \pmod q; \quad (21)$$

dann folgt (20) unmittelbar aus Satz 43. Um (21) zu zeigen, bemerken wir, daß aufgrund von Satz 48

$$M^{k(p-1)(q-1)} = (M^{k(q-1)})^{p-1} \equiv 1 \pmod p.$$

Entsprechend gilt $M^{k(p-1)(q-1)} = (M^{k(p-1)})^{q-1} \equiv 1 \pmod q$, woraus (21) und somit die Behauptung folgt. \square

Ein ähnlicher Beweis zeigt, daß auch $P(S(M)) = M$. Dieser Zusammenhang kann für „digitale Unterschriften“ verwendet werden.

11 Der MST-Algorithmus für TSP

Im Travelling-Salesman-Problem ist eine symmetrische Matrix $M = (m_{ij})_{1 \leq i, j \leq n}$ gegeben, deren Einträge positive ganze Zahlen sind. Das Ziel ist, eine Permutation σ von $\{1, \dots, n\}$ zu bestimmen, so daß

$$w(\sigma) = m_{\sigma(1)\sigma(n)} + \sum_{i=1}^{n-1} m_{\sigma(i)\sigma(i+1)}$$

minimal ist. Die Interpretation ist bekanntlich, daß m_{ij} den "Abstand" zwischen den "Städten" i und j angibt, und daß das optimale σ eine "Tour" durch die n "Städte" ist. Den Wert, den der Ausdruck $w(\sigma)$ für dieses optimale σ annimmt, bezeichnen wir mit OPT ; d.h. $OPT = \min_{\sigma} w(\sigma)$.

Weil TSP ein NP-schweres Problem ist, kennen wir keinen effizienten Algorithmus, der stets die optimale Tour σ bestimmt. Daher wollen wir uns hier mit der Frage, ob wir zumindest effizient stets eine "gute" Tour τ bestimmen können, d.h. eine Tour, die "nicht sehr viel schlechter" ist als OPT . Einen Algorithmus, der dies leistet, nennt man einen *Approximationsalgorithmus*.

Der folgende Satz zeigt, daß das TSP keinen guten Approximationsalgorithmus hat.

Satz 61. *Sei $r > 1$ eine beliebig große, aber fest gewählte Zahl. Wenn es einen polynomiellen Algorithmus \mathcal{A} gibt, der für jede TSP-Instanz eine Tour τ bestimmt, so daß $w(\tau) \leq r \cdot OPT$, so gilt $P = NP$.*

Beweis. Wir zeigen, daß \mathcal{A} verwendet werden kann, um das NP-schwere Hamiltonkreisproblem zu lösen. Sei dazu $G = (V, E)$ ein Graph mit Knotenmenge $V = \{1, \dots, n\}$. Wir definieren für $v, w \in V$

$$m_{vw} = \begin{cases} 1 & \text{falls } \{v, w\} \in E, \\ rn + 1 & \text{sonst.} \end{cases} \quad (22)$$

Falls nun der Graph G einen Hamiltonkreis hat, gilt für die Tour τ , die diesem folgt, daß $w(\tau) = n$. Wenn andererseits G keinen Hamiltonkreis hat, gilt für jede Permutation τ , daß $w(\tau) > rn$. \square

Stellt man sich die Matrixeinträge m_{ij} als "Abstände" zwischen "Städten" vor, so sollten diese die *Dreiecksungleichung*

$$m_{ik} \leq m_{ij} + m_{jk} \text{ für je drei verschiedene } i, j, k \quad (23)$$

erfüllen. Jedoch ist leicht zu sehen, daß die in (22) definierte Matrix (23) im allgemeinen nicht erfüllt. Daher kann man fragen, ob es einen vernünftigen Approximationsalgorithmus für das TSP gibt, wenn M in der Tat (23) erfüllt.

Algorithmus 62. MST-Approx

Eingabe: Eine matrix M , die (23) erfüllt.

Ausgabe: Eine Tour τ durch die "Städte" $1, \dots, n$.

1. Sei $G = K_n$ und $\ell(v, w) = m_{vw}$ für $v, w \in V = \{1, \dots, n\}$.
Berechne mit Kruskal einen minimal spannenden Baum $T = (V, F)$ in G .
2. Bestimme eine Folge $t_1, \dots, t_l \in V$, so daß jede Kante von F genau *zweimal* in der Kantensfolge

$$\{t_1, t_2\}, \dots, \{t_{l-1}, t_l\}, \{t_l, t_1\}$$

3. vorkommt.
Die gewünschte Tour durch die Städte $1, \dots, n$ erhält man, indem man in der Folge t_1, \dots, t_l jeden Knoten an den Stellen streicht, wo er zum $1 < i$ -ten Mal vorkommt.

Satz 63. *Sei τ die Tour, die MST-Approx ausgibt. Dann gilt $w(\tau) \leq 2 \cdot OPT$.*

Beweis. Sei σ eine Tour, so daß $w(\sigma) = OPT$. Dann ist $T^* = (V, F^*)$ mit

$$F^* = \{\{\sigma_i, \sigma_{i+1}\} : 1 \leq i < n\}$$

ein spannender Baum von (G, ℓ) . Folglich gilt

$$OPT = w(\sigma) \geq \sum_{i=1}^{n-1} m_{\sigma(i) \sigma(i+1)} \geq \sum_{\{v,w\} \in F} m_{vw}.$$

Ferner gilt aufgrund der Dreiecksungleichung (23), daß $w(\tau) \leq 2 \sum_{\{v,w\} \in F} m_{vw}$. \square

12 Der Irrfahrt-Algorithmus für 3SAT

In diesem Abschnitt lernen wir eine weitere Möglichkeit kennen, mit NP-schweren Problemen kennenzulernen: “schnelle” exponentielle Algorithmen.

Das 3-SAT-Problem ist bekanntlich NP-vollständig. Um zu entscheiden, ob eine gegebene 3-SAT-Formel $\phi = \phi_1 \wedge \dots \wedge \phi_m$ mit Klauseln ϕ_1, \dots, ϕ_m über Variablen x_1, \dots, x_n eine erfüllende Belegung besitzt, könnte man alle 2^n Belegungen der Variablen durchprobieren. Der Zeitbedarf dieses Algorithmus’ ist offenbar $(2 + o(1))^n$.

Unser Ziel ist, einen Algorithmus für 3-SAT mit (erwarteter) Laufzeit $(\frac{4}{3} + o(1))^n$ anzugeben.

Algorithmus 64. Irrfahrt

Eingabe: Eine 3-SAT-Formel ϕ wie oben.

Ausgabe: Entweder eine erfüllende Belegung oder “Fehler”.

1. Wiederhole folgendes $M = \lceil n(4/3)^n \rceil$ mal.
2. Wähle Belegung $A : \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$ der Variablen gleichverteilt unter allen 2^n möglichen Belegungen aus.
3. Wiederhole folgendes $3n$ mal.
4. Falls A eine erfüllende Belegung ist, gebe A aus und terminiere.
5. Sonst wähle eine nicht von A erfüllte Klausel $\phi_i = l_{1,i} \vee l_{2,i} \vee l_{3,i}$ beliebig aus. Wähle dann eines der drei Literale $l_{1,i}, l_{2,i}, l_{3,i}$ zufällig gleichverteilt aus und ändere den Wahrheitswert der entsprechenden Variablen.
6. Gebe “Fehler” aus.

Offenbar ist die Laufzeit von **Irrfahrt** beschränkt durch $(\frac{4}{3} + o(1))^n$. Wir zeigen, daß **Irrfahrt** ein guter Monte-Carlo-Algorithmus für 3-SAT ist. Genauer gilt folgendes.

Satz 65. *Wenn ϕ erfüllbar und n hinreichend groß ist, dann findet **Irrfahrt** mit Wahrscheinlichkeit ≥ 0.99 eine erfüllende Belegung.*

Wir nehmen also an, daß ϕ eine erfüllende Belegung A^* hat. Die Analyse des Algorithmus’ (und damit der Beweis des Satzes) besteht aus zwei Komponenten.

1. Auf wie vielen Variablen stimmt die in Schritt 2 zufällig gewählte Belegung A mit A^* überein?
2. Angenommen A und A^* unterscheiden sich in j Variablen – was ist dann die Wahrscheinlichkeit, daß die Schritte 3–5 eine erfüllende Belegung finden?

Die erste Frage ist relativ leicht zu beantworten. Weil die Belegung A zufällig gleichverteilt ausgewählt wird und damit $A(x_i)$ für jedes i unabhängig mit Wahrscheinlichkeit $\frac{1}{2}$ den Wert “wahr” bzw. “falsch” annimmt, gilt

$$p_j = P[A \text{ und } A^* \text{ unterscheiden sich in genau } j \text{ Variablen}] = \binom{n}{j} 2^{-n}. \quad (24)$$

Zur Beantwortung der zweiten Frage betrachten wir die Zahl Y der Variablen, in denen sich A und A^* unterscheiden. Anfangs (nach Schritt 2) gelte $Y = j$. Jedesmal, wenn die Schritte 4–5 ausgeführt werden, ändert sich die Zufallsgröße Y . Weil A^* die Klausel ϕ_i erfüllt, belegt A^* mindestens eines der drei Literale $l_{1,i}, l_{2,i}, l_{3,i}$ mit “wahr”. Daher können wir beobachten, daß

- mit Wahrscheinlichkeit $\geq \frac{1}{3}$ der Wert Y um 1 sinkt, und
- Y mit Wahrscheinlichkeit $\leq \frac{2}{3}$ um 1 steigt.

Die Zufallsvariable Y führt also eine sog. “Irrfahrt” auf der Zahlengeraden aus (was den Namen des Algorithmus’ erklärt). Unser Ziel ist, die Wahrscheinlichkeit abzuschätzen, daß Y nach höchstens $3n$ Schritten auf 0 sinkt (oder der Algorithmus vorher abbricht, weil schon eine andere erfüllende Belegung als A^* gefunden wurde).

Wir bezeichnen mit q_{jk} die Wahrscheinlichkeit, daß die Anfangsbelegung aus Schritt 2 sich von A^* an genau j Stellen unterscheidet, und daß die Schritte 3–5 innerhalb von k “Runden” eine erfüllende Belegung finden. Es gilt aufgrund der obigen Überlegungen

$$q_{jk} \geq \binom{j+2k}{k} (2/3)^k (1/3)^{j+k}.$$

Bezeichnen wir ferner mit q_j die Wahrscheinlichkeit, daß die Schritte 3–5 *überhaupt* eine erfüllende Belegung finden, wenn die Anfangsbelegung aus Schritt 2 sich von A^* an genau k Stellen unterscheidet, so gilt

$$q_j \geq \max_{1 \leq k \leq j} q_{jk} \geq q_{jj} \geq \binom{3j}{j} (2/3)^j (1/3)^{2j}.$$

Lemma 66. *Es gilt $q_j \geq \Omega(j^{-1/2}2^{-j})$.*

Beweis. Mit Hilfe der Stirling-Formel

$$j! \sim \sqrt{2\pi j} j^j \exp(-j)$$

erhalten wir

$$\binom{3j}{j} = \frac{(3j)!}{j!(2j)!} = \Omega(j^{-1/2})(3j)^{3j} (2j)^{-2j} j^{-j} = \Omega(j^{-1/2})(27/4)^j.$$

Folglich gilt $q_j \geq \binom{3j}{j} (2/3)^j (1/3)^{2j} \geq \Omega(j^{-1/2}2^{-j})$, wie behauptet. \square

Wir können mit Hilfe von Lemma 66 und (24) die Wahrscheinlichkeit P , daß in den Schritten 2–5 von **Irrfahrt** eine erfüllende Belegung gefunden wird, wie folgt abschätzen:

$$\begin{aligned} P &\geq \sum_{j=0}^n p_j q_j \geq \sum_{j=1}^n \binom{n}{j} 2^{-n} \Omega(j^{-1/2}2^{-j}) = \Omega(n^{-1/2}2^{-n}) \sum_{j=1}^n \binom{n}{j} 2^{-j} \\ &= \Omega(n^{-1/2}2^{-n})(3/2)^n = \Omega(n^{-1/2}(3/4)^n). \end{aligned} \quad (25)$$

Beweis von Satz 65. **Irrfahrt** versucht $M = \lceil n(4/3)^n \rceil$ mal, von einer zufälligen Startbelegung aus eine erfüllende Belegung zu finden. Für jeden Versuch ist die Erfolgswahrscheinlichkeit P . Da die Versuche unabhängig voneinander sind, erhalten wir aufgrund von (25)

$$P[\text{Irrfahrt antwortet “Fehler”}] \leq (1 - P)^M \leq \exp(-MP) \leq \exp(-\Omega(\sqrt{n})) = o(1).$$

Ist n hinreichend groß, so ist die rechte Seite ≤ 0.01 , woraus die Behauptung folgt. \square