Praktische Grenzen der Berechenbarkeit

Skript zum Weiterbildungslehrgang X Informatik – Kurs 6
IFB–Nr. 721770501

Patrick Breuer

15.11.2007

Inhaltsverzeichnis

1 Der Komplexitätsbegriff **2**

2 Beschreibung der Zeitkomplexität **5**

3 Abschätzung der Zeitkomplexität **9**

4 Praktisch nicht anwendbare Algorithmen **13**

5 Praktisch unlösbare Probleme **16**

6 Nichtdeterministische Algorithmen, P und NP **22**

7 NP–Vollständigkeit **25**

8 Näherungslösungen **34**

1 Der Komplexitätsbegriff

Nachdem es den Pionieren der heutigen Informatik ein besonderes Anliegen war herauszufinden, welche Probleme prinzipiell unlösbar sind, interessierte man sich etwa ab 1960 dafür, welche Algorithmen auf real existierenden und zukünftigen Computern auch praktisch durchführbar sind. Das überraschende und erfreuliche Ergebnis dieser Überlegungen besteht in der Erkenntnis, dass auch die Klassifizierung nach *durchführbar* und *nicht durchführbar* von der Wahl einer konkreten Maschine unabhängig ist.

Unter *Komplexitäten* versteht man messbare oder berechenbare Merkmale von Algorithmen, im Wesentlichen die Laufzeit und den Speicherbedarf. Intuitiv ist klar, dass ein Algorithmus, der sparsam mit diesen Betriebsmitteln umgeht, als besser einzustufen ist im Vergleich zu anderen Algorithmen, die dieselbe Aufgabe lösen.

Als Beispiel betrachten wir (nach Mayr, ohne Datum) die folgende Aufgabe einer Aktienkursanalyse: Gegeben ist der Kursverlauf einer Aktie über 30 Tage (vgl. Tabelle 1). Gesucht ist der maximal mögliche Gewinn, der sich bei optimaler Wahl des Kauf– und Verkaufstages ergibt. Bezeichnet man den Gewinn am Tag t mit a_t, so ist also eine Teilfolge von $a_1, a_2, \ldots, a_n$ gesucht, deren Summe maximal ist. Da jedoch nicht nach dem optimalen Kauf– bzw. Verkaufstag gefragt ist, sondern nur nach der maximalen Teilsumme, lautet die Aufgabenstellung etwas allgemeiner ausgedrückt wie folgt:

> Gegeben ist eine Zahlenfolge $a_1, a_2, \ldots, a_n$. Gesucht ist eine Zahl s mit $s = \sum_{k=i}^{j} a_k$ $(1 \leq i \leq j \leq n)$, die unter allen möglichen Kombinationen von i und j maximal ist.

Ein erster Algorithmus (Abbildung 1) berechnet alle möglichen Teilsummen und bestimmt daraus das Maximum. Als Maß für den zeitlichen Aufwand wählen wir vereinfachend die Anzahl t_{a1} der Additionen in der innersten Schleife.

```
Algorithmus Kursanalyse 1
(Gegeben: Ein n-dimensionales Feld a)
  begin
    max←0
    for i←1 to n do
      for j←i to n do
        s←0
        for k←i to j do
          s←s+a[k]
        if s>max then max←s
  end
```

Abbildung 1: Algorithmus Kursanalyse, Version 1

Tag	Kurs	GuV	Tag	Kurs	GuV
1	150	1	16	169	4
2	153	3	17	160	-9
3	157	4	18	158	-2
4	149	-8	19	159	1
5	151	2	20	152	-7
6	148	-3	21	147	-5
7	155	7	22	147	0
8	160	5	23	148	1
9	159	-1	24	150	2
10	165	6	25	153	3
11	156	-9	26	152	-1
12	157	1	27	156	4
13	159	2	28	161	5
14	162	3	29	162	1
15	165	3	30	164	2

Tabelle 1: Kursverlauf einer Aktie über 30 Tage (Schlusskurse und Gewinne bzw. Verluste). Angenommener Anfangskurs am ersten Tag: 149

$$
\begin{aligned}
t_{a1}(n) &= \sum_{i=1}^{n} i \cdot (n+1-i) \\
&= \sum_{i=1}^{n} (i \cdot (n+1) - i^2) \\
&= (n+1) \cdot \sum_{i=1}^{n} i - \sum_{i=1}^{n} i^2 \\
&= (n+1) \cdot \frac{(n+1) \cdot n}{2} - \frac{n \cdot (n+1) \cdot (2n+1)}{6} \\
&= \frac{n \cdot (n+1) \cdot (n+2)}{6} \\
&= \frac{1}{6} n^3 + \frac{1}{2} n^2 + \frac{1}{3} n
\end{aligned}
$$

Der Berechnungsaufwand lässt sich bereits deutlich reduzieren, wenn man berücksichtigt, dass nicht alle Teilsummen einzeln bestimmt werden müssen, da ja $\sum_{k=i}^{j} a_k = \sum_{k=i}^{j-1} a_k + a_j$ gilt. Daraus ergibt sich die zweite Version des Algorithmus (Abbildung 2). Hier gilt für die Anzahl der Additionen

```
Algorithmus Kursanalyse 2
(Gegeben: Ein n-dimensionales Feld a)
  begin
    max←0
    for i←1 to n do
      s←0
      for j←i to n do
        s←s+a[j]
        if s>max then max←s
  end
```

Abbildung 2: Algorithmus Kursanalyse, Version 2

```
Algorithmus Kursanalyse 3
(Gegeben: Ein n-dimensionales Feld a)
  begin
    max←0
    s←0
      for i←1 to n do
        s←s+a[i]
        if s>max then max←s
        if s<0 then s←0
  end
```

Abbildung 3: Algorithmus Kursanalyse, Version 3

$$\begin{aligned} t_{a2}(n) &= \sum_{i=1}^{n} i \\ &= \frac{n \cdot (n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \,. \end{aligned}$$

Wenn man nun noch von der Tatsache Gebrauch macht, dass eine Teilfolge mit negativer Summe nie zu einer Teilfolge mit maximaler Summe erweitert werden kann, ergibt sich eine erneute Optimierung (Abbildung 3). Für die Anzahl der Additionen gilt hier

$$t_{a3}(n) = n \,.$$

Wie sehr die Anzahl der Rechenoperationen von einander abweichen, verdeutlicht Tabelle 2.

n	20	40	60	80	100
$t_{a1}(n)$	1540	11480	37820	88560	171700
$t_{a2}(n)$	210	820	1830	3240	5050
$t_{a3}(n)$	20	40	60	80	100

Tabelle 2: Aufwand der Teilsummen–Algorithmen im Vergleich

2 Beschreibung der Zeitkomplexität

Es ist klar, dass der Umfang der Eingabedaten entscheidend ist für die Laufzeit. Am Beispiel von Sortieralgorithmen wird deutlich, dass dabei oft auch die Struktur der Eingabedaten zu berücksichtigen ist. Offensichtlich wird dies auch am Beispiel der sequentiellen Suche in einem Feld. Wird das gesuchte Element schon an erster Stelle gefunden, ist der Aufwand sehr gering. Wird es an letzter Stelle oder gar nicht gefunden, ist er maximal. Im Mittel wird man einen Zeitbedarf erwarten, der zwischen diesen Extremwerten liegt. Dementsprechend unterscheidet man bei der Beschreibung von Komplexitäten drei Fälle:

1. *Best–Case–Analyse*: Die im günstigsten Fall erforderliche Laufzeit wird ermittelt.

2. *Average–Case–Analyse*: Die im mittleren Fall (Durchschnitt, arithmetisches Mittel, Erwartungswert) erforderliche Laufzeit wird ermittelt.

3. *Worst–Case–Analyse*: Die im ungünstigsten Fall erforderliche Laufzeit wird ermittelt.

Die Laufzeit kann man sowohl durch Zeitmessungen als auch durch Berechnungen bestimmen. Zeitmessungen gelten aber immer nur für den speziellen Computer, auf dem der Algorithmus getestet wird, und sind somit nicht geeignet für allgemeine Aussagen über die Zeitkomplexität eines Algorithmus. Man bestimmt deshalb die Komplexität durch einen Term in Abhängigkeit von der Größe n der Eingabedaten. Dabei beschreibt n je nach Algorithmus z. B. die Anzahl der Datenelemente oder die Länge der Eingabe. Vervierfacht sich beispielsweise die Laufzeit bei einer Verdopplung des Umfangs der Eingabe, sagt man, der Algorithmus habe die Zeitkomplexität n^2.

In Tabelle 3 sind häufig auftretende Komplexitäten $k(n)$ mit grob gerundeten Funktionswerten angegeben (nach Gumm / Sommer 2002, 267). Für die weiteren Betrachtungen ist insbesondere der Vergleich der letzten beiden Zeilen von Bedeutung. Wir werden darauf noch näher eingehen.

Wie entscheidend sich die Zeitkomplexität auf die Laufzeit eines Algorithmus auswirkt, veranschaulicht Abbildung 4 an einigen ausgewählten Funktionen (nach Rechenberg 2000, 177).

Noch deutlicher wird der Zusammenhang, wenn man sich konkrete Zahlenbeispiele ansieht. Nehmen wir an, ein Rechenschritt werde in einer Millisekunde

$k(n)$	n	10	100	1000	10^4	10^5	10^6
konstant	1	1	1	1	1	1	1
logarithmisch	$\log_2 n$	3	7	10	13	17	20
linear	n	10	100	1000	10^4	10^5	10^6
log–linear	$n \cdot \log_2 n$	30	700	10^4	10^5	$2 \cdot 10^6$	$2 \cdot 10^7$
quadratisch	n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
kubisch	n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
exponentiell	2^n	1000	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{300000}

Tabelle 3: Häufig auftretende Komplexitäten und grob gerundete Funktionswerte

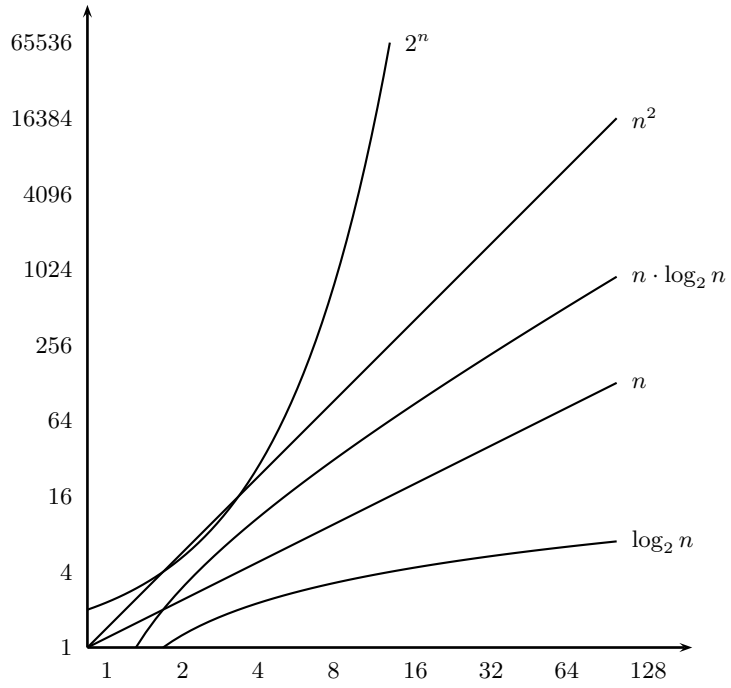

Abbildung 4: Wachstum einiger Funktionen zur Charakterisierung der Komplexität (in doppelt logarithmischer Darstellung)

bearbeitet, unabhängig davon, wie aufwändig dieser Schritt ist. Dann ergeben sich die in Tabelle 4 angegebenen Zeiten für die verschiedenen Algorithmen (nach Goldschlager / Lister 1990, 96).

Eingabegröße n	$\log_2 n$ ms	n ms	n^2 ms	2^n ms
10	0,003 s	0,01 s	0,1 s	1 s
100	0,007 s	0,1 s	10 s	$4 \cdot 10^{19}$ Jahre
1000	0,01 s	1 s	16,7 min	...
10000	0,013 s	10 s	1 T 3,8 h	...
100000	0,017 s	1,7 min	116 T	...

Tabelle 4: Zeitaufwand verschiedener Algorithmen

Wenn man sich nun überlegt, wie viele Rechenschritte jeweils in einer vorgegebenen Zeit möglich sind, wird das katastrophale Zeitverhalten exponentieller Algorithmen offensichtlich. Dies zeigt Tabelle 5 (nach Rechenberg 2000, 177).

	Max. Problemgröße n bei einer Laufzeit von		
Komplexität	1 s	1 min	1 h
$\log_2 n$	10^{301}	$6 \cdot 10^{18\,000}$	$10^{1\,000\,000}$
n	1000	60 000	3 600 000
n^2	31	244	1 897
2^n	9	15	21

Tabelle 5: Maximale Problemgröße in Abhängigkeit von Komplexität und Laufzeit

Während sich die Laufzeiten um den Faktor 60 unterscheiden, errechnet sich die jeweils nächste maximale Problemgröße
— bei der Komplexität $\log_2 n$ durch Potenzieren mit 60,
— bei der Komplexität n durch Multiplizieren mit 60,
— bei der Komplexität n^2 durch Multiplizieren mit $\sqrt{60}$ und
— bei der Komplexität 2^n durch *Addieren* von $\log_2 60$.
Bei der exponentiellen Komplexität 2^n lässt eine 3600–mal längere Laufzeit daher nur etwas mehr als die doppelte Problemgröße zu.

Nun könnte man einwenden, dass es nur eine Frage der Zeit ist, bis Rechner gebaut werden, die schnell genug arbeiten, um mit exponentiellen Algorithmen fertig zu werden. Doch Nachrechnen entkräftet dieses Argument. Nehmen wir an, es wäre möglich, einen Computer zu bauen, der 100–mal schneller arbeitet als in unserem Beispiel. Dann wächst die maximale Problemgröße bei einer Laufzeit von einer Stunde nur um $\log_2 100$ ($\approx 6,64$) auf 27. Um ein Problem der Größe $n = 100$ bearbeiten zu können, wäre ein Computer notwendig, der um den Faktor $6 \cdot 10^{23}$ schneller ist. Die Zeitkomplexität beschreibt also das Verhalten von Algorithmen

einigermaßen maschinenunabhängig.

Bei den angegebenen Komplexitäten handelt es sich um ausgewählte Sonderfälle. Um Algorithmen sinnvoll klassifizieren zu können, ist eine viel allgemeinere Unterteilung notwendig. Zunächst betrachtet man nur die *asymptotische Komplexität*, d. h. den Zeitbedarf für *große* Werte des Parameters n.

Darüber hinaus fasst man Komplexitäten zu Klassen zusammen, wenn sie sich nur um einen konstanten Faktor unterscheiden.

Definition 1. Die *asymptotische Ordnung* $O\left(g(n)\right)$ (gesprochen „groß O“) einer Funktion $g : \mathbb{N} \to \mathbb{R}_0^+$ ist die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}_0^+$, die *für hinreichend große Werte von* n nach oben durch ein positives reelles Vielfaches von g beschränkt sind. $O(g(n))$ enthält also alle Funktionen f, für die ein $r \in \mathbb{R}$ existiert, so dass für große Werte von n $f(n) \leq r \cdot g(n)$ gilt. Oder formal:

$$O\left(g(n)\right) = \left\{ f : \mathbb{N} \to \mathbb{R}_0^+ \mid \quad \exists r \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N}_0 \; \forall n \geq n_0 : \quad f(n) \leq r \cdot g(n) \right\}$$

Bei dem Buchstaben O handelt es sich eigentlich um ein großes Omikron. Es gibt andere Definitionen einer Ordnung, die mit Ω (Omega) bzw. Θ (Theta) bezeichnet werden.

Beispiel 2. Die Funktionen f_1 und f_2 mit

$$f_1(n) = 3n^2 + 3n - 3$$

und

$$f_2(n) = 4,5n^3 + 6n^2$$

sind Elemente der Ordnung $O(n^3)$, denn es gilt z. B.:
Für alle $n \geq 4$ ist $f_1(n) \leq n^3$ ($r = 1$ und $n_0 = 4$).
Für alle $n \geq 12$ ist $f_2(n) \leq 5n^3$ ($r = 5$ und $n_0 = 12$).

Die O–Notation ermöglicht es, Algorithmen in Komplexitätsklassen einzuteilen. Für die konkrete Laufzeit eines Programms ist jedoch auch entscheidend, wie die Grundoperationen vom Compiler in Maschinensprache übersetzt werden. Die Laufzeit einer Grundoperation wird aber durch die Übersetzung nur um einen konstanten Faktor gegenüber der Abschätzung verändert. Weil diese Veränderung keinen Einfluss auf die Zuordnung des Algorithmus zu einer Komplexitätsklasse hat, können wir auf eine Unterscheidung zwischen Programmiersprache und Maschinensprache verzichten.

Beim Gebrauch der O–Notation hat sich eine nicht korrekte Schreibweise durchgesetzt: Um beispielsweise auszudrücken, dass ein Algorithmus mit dem Zeitaufwand $f(n)$ von der Ordnung $O(n^2)$ ist, schreibt man oft $f = O(n^2)$, obwohl es richtig $f \in O(n^2)$ heißen müsste. Die Schreibweise ist problematisch, weil z. B. mit $f \in O(n^2)$ auch $f \in O(n^3)$ gilt, aber aus $f = O(n^2)$ und $f = O(n^3)$

wegen der Symmetrie und Transitivität der Gleichheitsrelation fälschlicherweise $O(n^2) = O(n^3)$ folgte.

Aus der Definition und den Beispielen wird deutlich, dass die asymptotische Ordnung nur eine obere Schranke für das Laufzeitverhalten eines Algorithmus darstellt. Aussagekräftige Abschätzungen der Zeitkomplexität bestehen daher immer in der Bestimmung einer möglichst kleinen oberen Schranke. Dies soll im folgenden Abschnitt an einzelnen Beispielen verdeutlicht werden.

3 Abschätzung der Zeitkomplexität

Wir haben in Abschnitt 1 bereits gesehen, dass die Laufzeit eines Algorithmus neben dem Umfang auch von der Struktur der Eingabedaten abhängig ist. Eine Analyse der Laufzeit im günstigsten Fall geht meist von Voraussetzungen aus, die in der Praxis einen seltenen Sonderfall darstellen. Sie ist daher wenig aussagekräftig. Die rechnerische Analyse der Zeitkomplexität im *average case* setzt Methoden der Wahrscheinlichkeitsrechnung ein, wodurch die Berechnungen häufig sehr komplex werden. Außerdem verfügt man meist nicht über präzise Informationen über die Wahrscheinlichkeitsverteilung der Eingabedaten. Wir beschränken uns daher auf die Analyse im *worst case* an einfachen Beispielen.

Betrachten wir zunächst die sequentielle Suche in einem sortierten Feld ganzer Zahlen (Abbildung 5). In der einfachsten Version vergleicht der Algorithmus der Reihe nach jedes Feldelement mit der gesuchten Zahl bis eine Übereinstimmung gefunden wird oder das letzte Feldelement erreicht ist.

```
Algorithmus Sequentielle Suche
(Gegeben: Ein sortiertes n-dimensionales Feld a
und ein Suchschlüssel s)
  begin
    gefunden←false
    i←1
    repeat
      if a[i]=s then
        gefunden←true
      i←i+1
    until gefunden or i>n
  end
```

Abbildung 5: Algorithmus *Sequentielle Suche*

Unabhängig von den Eingabedaten sind zunächst zwei Wertzuweisungen erforderlich. Innerhalb der *Repeat*–Schleife, die im ungünstigsten Fall n–mal durchlaufen wird, sind ein Vergleich und eine Wertzuweisung zu berücksichtigen. Die Anweisung `gefunden←true` braucht nicht gezählt zu werden, weil es ja nur um

den ungünstigsten Fall geht, in dem die gesuchte Zahl nicht gefunden wird. Die Abbruchbedingung der Schleife geht jedoch mit zwei Vergleichen in die Berechnung ein. Daraus ergibt sich als Zeitkomplexität $t_{seq}(n)$ der sequentiellen Suche im ungünstigsten Fall

$$\begin{aligned} t_{seq}(n) &= 2 + n \cdot (1 + 1 + 2) \\ &= 4n + 2 \\ t_{seq} &\in O(n) \end{aligned}$$

Die Funktionsweise der binären Suche (Abbildung 6), die wir als nächstes untersuchen, wird oft am Beispiel der Suche in einem Telefonbuch verdeutlicht, obwohl der Vergleich hinkt. Es wird zunächst der Index des mittleren Feldelementes bestimmt. Wird die gesuchte Zahl an dieser Stelle gefunden, ist die Suche beendet. Andernfalls wird auf die gleiche Art in der linken bzw. rechten Hälfte weiter gesucht, je nach Größe der zuletzt mit dem Suchschlüssel verglichenen Zahl. Ist die Anzahl der Elemente im aktuell zu durchsuchenden Feldbereich gerade, wird der größte Feldindex der linken Hälfte als Mittelwert verwendet.

```
Algorithmus Binäre Suche
(Gegeben: Ein sortiertes n-dimensionales Feld a
und ein Suchschlüssel s)
  begin
    gefunden←false
    l←1
    r←n
    repeat
      m←(l+r) div 2
      if a[m]=s then
        gefunden←true
      if a[m]>s then
        r←m-1
      if a[m]<s then
        l←m+1
    until gefunden or l>r
  end
```

Abbildung 6: Algorithmus *Binäre Suche*

Dadurch ergeben sich maximal $\lceil \log_2 n \rceil$ Durchläufe der *Repeat*–Schleife (Hier gibt $\lceil \log_2 n \rceil$ den Logarithmus der auf n folgenden Zweierpotenz an, falls $\log_2 n$ nicht ganzzahlig ist). Innerhalb der Schleife sind jeweils drei Vergleiche und zwei Wertzuweisungen zu bearbeiten. Zusammen mit den drei Zuweisungen am Anfang, die lediglich der Initialisierung der Variablen dienen, ergibt sich als Zeit-

komplexität $t_{bin}(n)$ der binären Suche im ungünstigsten Fall

$$\begin{aligned} t_{bin}(n) &= 3 + \lceil \log_2 n \rceil \cdot (3 + 2) \\ &= 5 \cdot \lceil \log_2 n \rceil + 3 \\ t_{bin} &\in O(\log_2 n) \end{aligned}$$

Auch hier mache man sich die Unterschiede der beiden Algorithmen mit Zahlen deutlich — an dieser Stelle auch am Beispiel des Telefonbuchs:

Beispiel 3. Sucht man in einem 1000–seitigen Telefonbuch sequentiell, so muss man unter (ungünstigen) Umständen alle 1000 Seiten der Reihe nach aufschlagen. Mit binärer Suche hat man nach spätestens zehn Versuchen die passende Seite gefunden. Für die berechneten Komplexitäten der angegebenen Algorithmen erhält man folgende Werte:

Sequentielle Suche:	$t_{seq}(1000) = 4002$
Binäre Suche:	$t_{bin}(1000) = 53$.

Ausführliche Beschreibungen verschiedener Sortier– und Suchalgorithmen sowie präzise Berechnungen des jeweiligen Aufwandes hat Knuth zusammengetragen (Knuth 1973).

Zur Abschätzung der Zeitkomplexität eines rekursiven Algorithmus betrachten wir das Problem der *Türme von Hanoi*: In der vereinfachten Version gibt es drei Stäbe A, B und C, auf die drei unterschiedlich große runde Scheiben gesteckt werden können. Die Ausgangssituation besteht darin, dass alle drei Scheiben der Größe nach geordnet — die größte ganz unten — auf Stab A liegen. Man soll nun die Scheiben in derselben Anordnung auf B platzieren, wobei folgende Regeln zu beachten sind:
— Es darf immer nur eine Scheibe bewegt werden.
— Es darf nie eine Scheibe auf einer kleineren liegen.
Eine Lösung dieser Aufgabe ist schnell gefunden:

$$A \to B, A \to C, B \to C, A \to B, C \to A, C \to B, A \to B \ .$$

In der ursprünglichen Version des Problems ging es aber um 64 Scheiben. Eine Lösung für diese Aufgabe aufzuschreiben, ist schlicht unmöglich. In obiger Notation wäre die Zeile $1,8 \cdot 10^{14}$ km lang — knapp das Fünfmilliardenfache des Erdumfangs. Dennoch ist es möglich, allgemein für n Scheiben eine einfache rekursive Strategie anzugeben, für deren Anwendung wir den Zeitbedarf bestimmen können:

— Bewege $n - 1$ Scheiben von A nach C.

— Bewege eine Scheibe von A nach B.

— Bewege $n - 1$ Scheiben von C nach B.

Damit wird die Problemgröße reduziert, und die Teilprobleme im ersten und dritten Schritt lassen sich analog lösen. Es ergibt sich der in Abbildung 7 dargestellte Algorithmus.

```
Algorithmus Hanoi(n,von,nach,ueber)
(von: Ausgangsstab,
nach: Zielstab
ueber: Hilfsstab)
  begin
    if n=1 then
      Ausgabe: von -> nach
    else
      Hanoi(n-1,von,ueber,nach)
      Ausgabe: von -> nach
      Hanoi(n-1,ueber,nach,von)
  end
```

Abbildung 7: Rekursiver Algorithmus *Türme von Hanoi*

In jedem Durchlauf des Algorithmus sind zunächst ein Vergleich und eine Ausgabeanweisung zu berücksichtigen. Hinzu kommt der Aufwand für die beiden rekursiven Aufrufe, falls $n > 1$ ist. Bezeichnen wir die Zeitkomplexität mit $t_{han}(n)$, so ergibt sich also für den Gesamtaufwand

$$t_{han}(n) = \begin{cases} 2 & \text{für } n = 1 \\ 2 + 2 \cdot t_{han}(n-1) & \text{für } n > 1 \; . \end{cases}$$

Aus der Zinsrechnung wissen wir, dass man diesen rekursiven Term umschreiben kann (Kapitalberechnung bei nachschüssiger Einzahlung (*postnumerando*) und einem Zinssatz von 100%):

$$\begin{aligned} t_{han}(n) &= 2 \cdot (2^n - 1) \\ &= 2^{n+1} - 2 \; . \end{aligned}$$

Es gilt also

$$t_{han} \in O(2^n) \; .$$

Leider hat man nicht immer eine Formel zur Verfügung, mit der sich eine Rekursionsgleichung in eine geschlossene Form bringen lässt. Oft bleibt dann nur die Möglichkeit, eine Wertetabelle zu erstellen, den Funktionsterm zu raten und anschließend dessen Korrektheit mit Hilfe vollständiger Induktion zu beweisen. In manchen Fällen liefern aber auch Computer–Algebra–Systeme brauchbare Terme, die zumindest geeignet sind, die Funktion einer asymptotischen Ordnung zuzuordnen.

Um das oben gestellte Problem der Türme von Hanoi mit 64 Scheiben zu lösen, beträgt die Anzahl der Grundoperationen

$$t_{han}(64) = 36.893.488.147.419.103.230 \quad \text{(etwa 37 Trillionen)} .$$

Verdoppelt man den Zeitaufwand, so lässt sich das Problem lediglich für *eine* weitere Scheibe lösen. Gleiches gilt natürlich bei einer Verdopplung der Arbeitsgeschwindigkeit unter Beibehaltung der Rechenzeit. In Abschnitt 2 und insbesondere in Tabelle 4 wurde schon deutlich, dass Algorithmen der Ordnung $O(2^n)$ schon bei relativ kleinem Umfang der Eingabedaten praktisch nicht mehr anwendbar sind. Um diese Problematik geht es im folgenden Abschnitt.

4 Praktisch nicht anwendbare Algorithmen

Die Beispiele in Abschnitt 2 zeigen, dass zwischen den Komplexitäten n^2 und 2^n die Grenze zwischen *praktisch anwendbar* und *praktisch nicht anwendbar* zu liegen scheint. Tatsächlich ist es im Allgemeinen sinnvoll, solche Algorithmen als *anwendbar* zu bezeichnen, deren Komplexität sich durch ein Polynom beschreiben lässt. Algorithmen mit exponentieller Zeitkomplexität sind dagegen *praktisch nicht anwendbar*. Wir unterscheiden für die folgenden Betrachtungen also zunächst nur noch zwischen *polynomialen* und *nicht polynomialen* Algorithmen.

Definition 4. Ein Algorithmus heißt *polynomial* (von polynomialer Ordnung), wenn seine Zeitkomplexität durch eine Funktion $f(n)$ beschrieben wird, für die ein $k \in \mathbb{N}$ existiert, so dass $f \in O(n^k)$ gilt. Ein Algorithmus heißt *exponentiell* (von exponentieller Ordnung), wenn er nicht polynomial ist.

Aus der Definition ergibt sich, dass auch Algorithmen mit logarithmischer Zeitkomplexität als *polynomial* bezeichnet werden, obwohl der entsprechende Term für die Komplexität kein Polynom ist. Es geht hier also vorrangig um eine Abgrenzung zu exponentiellen Algorithmen. Mathematisch ist diese Sichtweise in Ordnung, weil nach der Definition der asymptotischen Ordnung $\log n \in O(n)$ und $n \log n \in O(n^2)$ gilt.

Für Polynome $f(n)$ ist k der Grad des Polynoms, also der größte Exponent. Es gilt nämlich:

$$\begin{aligned} c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \cdots + c_1 \cdot n + c_0 \\ \leq c_k \cdot n^k + c_{k-1} \cdot n^k + \cdots + c_1 \cdot n^k + c_0 \cdot n^k \end{aligned}$$

und somit

$$O(c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \cdots + c_1 \cdot n + c_0) \subseteq O(n^k) .$$

Aus dieser Feststellung und den beiden folgenden Sätzen ergeben sich weitere Rechtfertigungen für die Wahl der „Zweiteilung“ von Komplexitätsklassen.

Satz 5. Werden zwei polynomiale Algorithmen nacheinander ausgeführt, ist der resultierende Gesamtalgorithmus polynomial.

Beweis. Seien $f(n)$ und $g(n)$ die Komplexitäten der beiden Algorithmen mit

$$\begin{aligned} f(n) &= c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \cdots + c_1 \cdot n + c_0 \in O(n^k) \ , \\ g(n) &= d_l \cdot n^l + d_{l-1} \cdot n^{l-1} + \cdots + d_1 \cdot n + d_0 \in O(n^l) \ . \end{aligned}$$

Ohne Beschränkung der Allgemeinheit gelte $l \geq k$. Setzt man nun $c_i := 0$ für $i = k+1, \ldots, l$, so ergibt sich für den Gesamtalgorithmus die Komplexität

$$\begin{aligned} f(n) + g(n) = (c_l + d_l) \cdot n^l + (c_{l-1} + d_{l-1}) \cdot n^{l-1} \\ + \cdots + (c_1 + d_1) \cdot n + c_0 + d_0 \in O(n^l) \ . \end{aligned}$$

□

Satz 6. Wird ein Teil eines polynomialen Algorithmus durch ein Modul ersetzt, das selbst einen polynomialen Algorithmus enthält, ist der resultierende Gesamtalgorithmus polynomial.

Beweis. Es sei $f(n)$ die Komplexität des Algorithmus, in dem ein Teil durch ein Modul mit der Komplexität $g(n)$ ersetzt wird.

$$\begin{aligned} f(n) &= c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \cdots + c_1 \cdot n + c_0 \in O(n^k) \ , \\ g(n) &= d_l \cdot n^l + d_{l-1} \cdot n^{l-1} + \cdots + d_1 \cdot n + d_0 \in O(n^l) \ . \end{aligned}$$

Die Komplexität $h(n)$ des resultierenden Algorithmus hängt davon ab, in welchem Umfang die ersetzten Anweisungen in die Berechnung eingehen. In jedem Fall gilt jedoch

$$h(n) \leq f(g(n))$$

mit

$$\begin{aligned} f(g(n)) = c_k \cdot (d_l \cdot n^l + d_{l-1} \cdot n^{l-1} + \cdots + d_1 \cdot n + d_0)^k \\ + c_{k-1} \cdot (d_l \cdot n^l + d_{l-1} \cdot n^{l-1} + \cdots + d_1 \cdot n + d_0)^{k-1} \\ + \cdots + c_1 \cdot (d_l \cdot n^l + d_{l-1} \cdot n^{l-1} + \cdots + d_1 \cdot n + d_0) + c_0 \end{aligned}$$

Demnach hat der Gesamtalgorithmus maximal die Komplexität n^{k+l}.

□

Die Klasse der polynomialen Algorithmen ist also abgeschlossen bezüglich der Hintereinanderausführung und der Ersetzung. Dies bildet eine wichtige Grundlage für die folgende

Definition 7. Ein Algorithmus heißt *praktisch anwendbar* (*durchführbar, handhabbar,* engl. *tractable*), wenn er polynomial ist, andernfalls *praktisch nicht anwendbar* (*nicht durchführbar, nicht handhabbar,* engl. *intractable*).

Zusammen mit den vorangegangenen Sätzen sind durch diese Definition grundlegende Forderungen erfüllt, die man intuitiv an anwendbare Algorithmen stellt:

— Werden zwei anwendbare Algorithmen nacheinander ausgeführt, so soll auch der resultierende Algorithmus als anwendbar bezeichnet werden.

— Wird in einem anwendbaren Algorithmus ein Teil durch ein Modul ersetzt, das einen anwendbaren Algorithmus enthält, so soll auch der neue Gesamtalgorithmus als anwendbar bezeichnet werden.

In Abschnitt 2 wurde bereits anhand von Beispielen argumentiert, dass auch eine weitere wichtige Forderung erfüllt ist: Nicht anwendbare Algorithmen werden auch auf zukünftigen Computern nicht anwendbar sein. Diesen Sachverhalt gilt es nun etwas formaler zu fassen. Durch die Church–Turing–These wurde der Begriff der Berechenbarkeit auf die Existenz einer Turingmaschine zurückgeführt. Analog dazu lässt sich auch ein Algorithmus als polynomial bezeichnen, wenn er auf einer Turingmaschine polynomiale Komplexität besitzt. Dies ist möglich, weil die Transformation einer beliebigen Beschreibung eines Algorithmus in eine äquivalente Turingtafel stets mit polynomialem Aufwand durchführbar ist. Wenn auch nicht absehbar ist, welche Computer uns in der Zukunft zur Verfügung stehen werden, rechtfertigt dies doch die folgende These:

Satz 8 (*These der sequentiellen Berechenbarkeit*)**.** Alle sequentiellen Computer besitzen ähnliche polynomiale Berechnungszeiten.

Dadurch ist gewährleistet, dass unsere Definition der Anwendbarkeit von Algorithmen maschinenunabhängig ist. Zwar lässt sich die These nicht beweisen, doch sprechen alle bisherigen Erkenntnisse über sequentielle Computer für ihre Gültigkeit.

In der Praxis ist bei der Klassifizierung eines Algorithmus als *nicht durchführbar* dennoch Vorsicht geboten. Man muss stets berücksichtigen, dass es sich bei allen bisherigen Betrachtungen um die asymptotische Komplexität handelt, also um eine Abschätzung der Laufzeit bei sehr großem Umfang der Eingabedaten. Es können daher auch exponentielle Algorithmen praktisch brauchbar sein, wenn der Exponent einen sehr kleinen Vorfaktor hat und zu hohe Laufzeiten aufgrund der konkret gegebenen Daten nicht vorkommen können. Auch haben exponentielle Algorithmen ihre praktische Relevanz, wenn es darum geht, eine angemessene Zeit nach einer optimalen Lösung eines Problems zu suchen, bevor man sich mit einer Näherungslösung zufrieden gibt. Umgekehrt können auch polynomiale Algorithmen unbrauchbar sein und sogar ungünstiger als exponentielle.

Beispiel 9. Ein exponentieller Algorithmus mit der Komplexität $t_e(n) = 0,001 \cdot 2^{0,001n}$ ist für $n = 100000$ noch schneller als ein polynomialer Algorithmus mit der Komplexität $t_p(n) = 1000 \cdot n^5$.

Für die prinzipiellen Betrachtungen der folgenden Abschnitte wollen wir dennoch bei der oben festgelegten (und begründeten) Unterscheidung bleiben: Wir bezeichnen einen Algorithmus als nicht anwendbar, wenn er nicht polynomial ist.

5 Praktisch unlösbare Probleme

Wir wollen nun analog zu den prinzipiell unlösbaren Problemen auch diejenigen Probleme klassifizieren, die in der Praxis unlösbar sind. Nach den bisherigen Erkenntnissen kann die Lösung eines Problems daran scheitern, dass der angewandte Algorithmus eine zu hohe (d. h. exponentielle) Zeitkomplexität besitzt. Die praktische Unlösbarkeit eines Problems über die Komplexität eines bestimmten geeigneten Algorithmus zu definieren, ist jedoch ein unbrauchbarer Ansatz. Schließlich ist es möglich, jeden beliebigen Algorithmus so zu erweitern, dass er dasselbe Problem löst, jedoch exponentielle Zeitkomplexität besitzt. Man verwendet deshalb die *Existenz* eines polynomialen Algorithmus als Grundlage für die Definition der praktischen Lösbarkeit.

Definition 10. Ein Problem heißt *praktisch lösbar*, wenn es einen polynomialen Lösungsalgorithmus für dieses Problem gibt, andernfalls *praktisch unlösbar*.

Es gibt viele Probleme, zu deren Lösung bisher kein polynomialer Algorithmus gefunden wurde, für die aber auch noch nicht bewiesen werden konnte, dass ein solcher Algorithmus nicht existiert. Hier zeigt sich ein grundlegender Unterschied zur Definition eines anwendbaren Algorithmus: Die praktische Unlösbarkeit eines Problems gilt im Allgemeinen nur für den aktuellen Stand der Forschung. Vielfach kann nicht ausgeschlossen werden, dass zu einem späteren Zeitpunkt ein schnellerer Algorithmus gefunden wird. Besonders interessant sind in diesem Zusammenhang solche Probleme, für die sich ein potentieller polynomialer Algorithmus schon seit sehr langer Zeit hartnäckig seiner Entdeckung entzieht.

Wir sehen uns jedoch zunächst ein Problem an, das polynomial lösbar ist. Eine scheinbar unwesentliche Veränderung des Problems macht es dann praktisch unlösbar.

Beispiel 11 (*Königsberger Brückenproblem*)**.** Das ursprüngliche Problem wird häufig folgendermaßen zitiert: Im Königsberg des 18. Jahrhunderts fragte man sich, ob es einen Rundweg durch die Stadt gibt, bei dem jede der sieben Brücken genau einmal benutzt wird (vgl. Abbildung 8). Dass dies nicht möglich ist, ist leicht einzusehen: Startet man beispielsweise am Ufer B, so verlässt man es über eine der Brücken a, b oder f. Über eine der beiden verbleibenden Brücken kehrt man wieder zurück und verlässt B dann über die letzte Brücke. Man hat keine

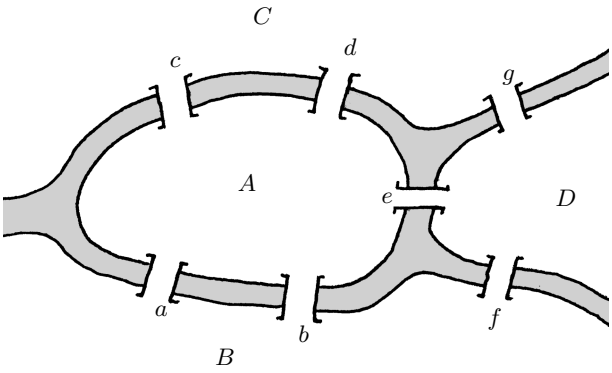

Abbildung 8: Skizze zum *Königsberger Brückenproblem*

Möglichkeit, nach B zurückzukehren. Das gleiche gilt *mutatis mutandis* für die Startpunkte A, C und D, sowie für A, B, C und D als Zwischenstationen.

Leonhard Euler verallgemeinerte die Fragestellung auf ungerichtete Graphen. Identifiziert man die Brücken mit Kanten und die durch sie verbundenen Orte mit Knoten, lautet das Problem wie folgt:

> Gibt es in einem ungerichteten Graphen einen Weg, der jede Kante genau einmal enthält und zum Ausgangsknoten zurückführt?

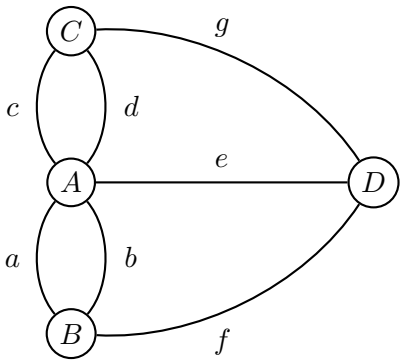

Abbildung 9: Graph zum Königsberger Brückenproblem

Abbildung 9 zeigt den Graphen zum Königsberger Brückenproblem. Um Eulers Lösung des Problems verstehen zu können, ist eine Begriffsklärung notwendig.

Definition 12. In einem ungerichteten Graphen heißt ein Knoten *Nachbar* eines Knotens v, wenn er mit v durch eine Kante verbunden ist. Die Anzahl der vom Knoten v ausgehenden Kanten bezeichnet man als dessen *Grad*. In einem Graphen *ohne* Mehrfachkanten zwischen zwei Knoten entspricht der Grad der Anzahl der Nachbarn.

Ein Rundweg durch den Graphen, bei dem jede Kante genau einmal benutzt wird, kann nur dann gelingen, wenn jeder Knoten genau so oft erreicht wird wie er wieder verlassen wird. Daran scheitert jeder Lösungsversuch im Brückenproblem, wie wir bereits oben gesehen haben. Eulers Antwort auf die Frage nach der Existenz eines solchen Rundweges — der für Graphen auch *Eulerkreis* genannt wird — lautet deshalb:

> In einem ungerichteten Graphen existiert genau dann ein Weg, der jede Kante genau einmal enthält und zum Ausgangsknoten zurückführt, wenn der Grad jedes Knotens gerade ist.

```
Algorithmus Eulerkreis
(Eingabe: Ein Feld g der Dimension n x n, das die Kanten des
Graphen festlegt
Ausgabe: 'ja', falls der Graph einen Eulerkreis enthält, sonst 'nein')
  begin
    euler←true
    for i←1 to n do
      for j←1 to n do
        grad[i]←grad[i]+g[i,j]
    for i←1 to n do
      if grad[i] mod 2 = 1 then
        euler←false
    if euler=true then
      Ausgabe: ja
    else
      Ausgabe: nein
  end
```

Abbildung 10: Algorithmus *Eulerkreis*

Ein Algorithmus, der zu einem gegebenen Graphen mit n Knoten entscheidet, ob er einen Eulerkreis enthält, ist in Abbildung 10 angegeben. Als Eingabe wird hier eine $n \times n$–Matrix G verwendet, für die gilt:

$$g_{ij} = \text{Anzahl der Kanten zwischen Knoten } i \text{ und Knoten } j\,.$$

Dabei bezeichnet g_{ij} den Wert in der i–ten Zeile und j–ten Spalte, wenn man G als rechteckiges Schema notiert.

Wegen der geschachtelten Zählschleife liegt die Zeitkomplexität des Algorithmus in der Klasse $O(n^2)$. Das Problem ist also mit polynomialem Zeitaufwand lösbar.

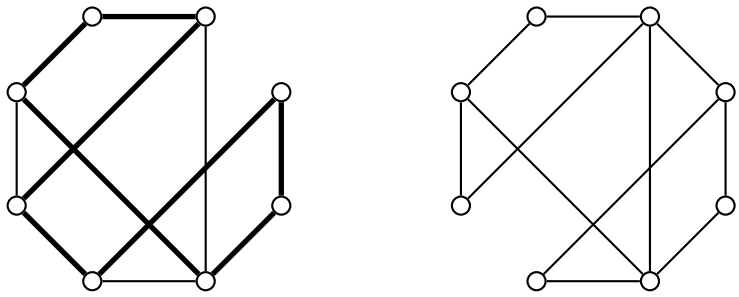

Abbildung 11: Graphen mit bzw. ohne Hamilton–Zyklus

Beispiel 13 (*Hamilton–Zyklus*, engl. *Hamiltonian circuit*)**.** Unter einem *Hamilton–Zyklus* versteht man in einem ungerichteten Graphen einen Rundweg, der jeden *Knoten* genau einmal enthält (vgl. Abbildung 11).

Die Frage nach der Existenz eines Hamilton–Zyklus in einem gegebenen Graphen ähnelt sehr dem zuvor besprochenen Problem. Während es dort um einen Rundweg ging, der alle Kanten genau einmal enthält, wird hier nach einem Rundweg gefragt, der jeden Knoten genau einmal enthält. Doch bislang gibt es keinen polynomialen Algorithmus, der das Problem löst. Die einzige bekannte Möglichkeit, einen Graphen auf die Existenz eines Hamilton–Zyklus hin zu untersuchen, besteht in der Überprüfung aller theoretisch möglichen Knotenfolgen. Weil es bei einem Rundweg keine Rolle spielt, mit welchem Knoten man beginnt, müssen bei einem Graphen mit n Knoten $(n-1)!$ Permutationen überprüft werden. Das Problem *Hamilton–Zyklus* ist also praktisch unlösbar.

Beispiel 14 (*Problem des Handlungsreisenden*, engl. *travelling salesman problem*)**.** Von praktischer Relevanz ist eine Verallgemeinerung des Hamilton–Problems: Jeder Kante wird eine Zahl zugeordnet, der *Kostenwert*. Das Problem besteht nun darin zu entscheiden, ob es einen Hamilton–Zyklus gibt, bei dem die Summe der Kostenwerte aller benutzten Kanten eine Kostengrenze k nicht übersteigt. Wählt man als Kosten beispielsweise die zu erwartende Fahrzeit zwischen zwei Orten, entspricht das Problem der Frage, ob eine Rundreise durch vorgegebene Orte innerhalb einer festgelegten Zeit möglich ist (wobei jeder Ort genau einmal besucht wird). Abbildung 12 zeigt einen solchen Graphen. Das Problem des Handlungsreisenden stellt eine Verallgemeinerung des Hamilton–Problems dar, weil dieses als Sonderfall aufgefasst werden kann: Alle Kanten haben den Kostenwert 1, und die Kostengrenze ist durch die Anzahl der Knoten gegeben.

Für das Problem des Handlungsreisenden existiert kein polynomialer Algorithmus. Es ist praktisch unlösbar.

Die Fragestellung, über die das Problem hier eingeführt wurde, wird häufig als *Entscheidungsvariante* des Problems bezeichnet. Es sind jedoch auch andere Fragestellungen möglich, die in Tabelle 6 angegeben sind. Auch die Zahlvariante und die Optimierungsvariante sind bislang nicht polynomial lösbar. Man kann

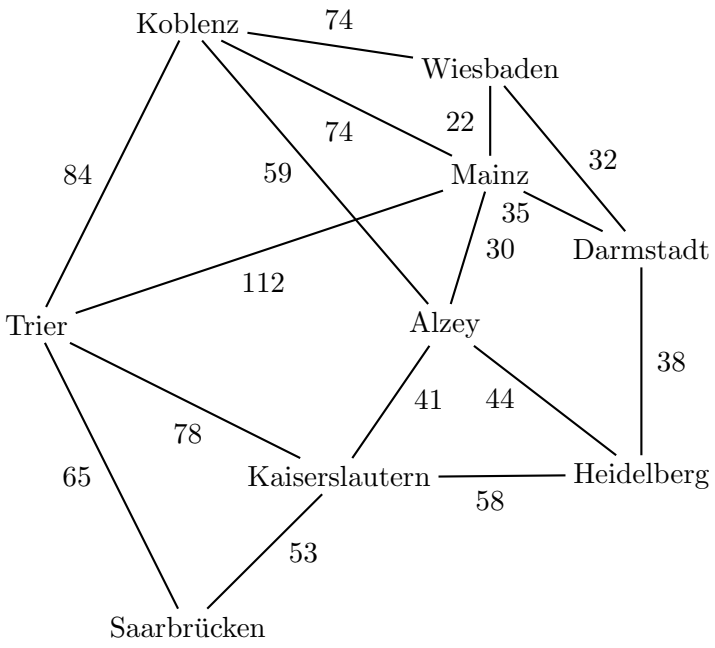

Abbildung 12: Graph zum *Problem des Handlungsreisenden*

sogar nachweisen, dass alle drei Varianten äquivalent sind. Sollte also für eine Variante ein polynomialer Algorithmus gefunden werden, sind auch die anderen Varianten polynomial lösbar.

Entscheidungsvariante:	Gibt es zu einem gegebenen Kostenwert k eine Rundreise, deren Kosten k nicht übersteigen?
Zahlvariante:	Was ist der kleinste Kostenwert k, für den eine Rundreise existiert?
Optimierungsvariante:	Welches ist die kostengünstigste Rundreise?

Tabelle 6: Varianten des Problems des Handlungsreisenden

Beispiel 15 (*Verpackungsproblem*, engl. *bin–packing problem*)**.** Gegeben sind k Behälter einer festen Größe G und n Gegenstände mit den Größen $g_1, \ldots, g_n$. Gesucht ist eine Verteilung der Gegenstände auf die Behälter, bei der die jeweilige Summe der Größen der Gegenstände die Größe G der Behälter nicht überschreitet. Es lassen sich wieder drei Varianten unterscheiden (Tabelle 7).

Entscheidungsvariante:	Gibt es eine zulässige Verteilung der Gegenstände auf die Behälter?
Zahlvariante:	Was ist die kleinste Anzahl von Behältern, so dass alle Gegenstände auf zulässige Art verteilt werden können?
Optimierungsvariante:	Welche zulässige Verteilung kommt mit der kleinsten Anzahl von Behältern aus?

Tabelle 7: Varianten des Verpackungsproblems

Bezüglich der praktischen Lösbarkeit sind wiederum alle drei Varianten äquivalent, und es existiert bislang kein polynomialer Lösungsalgorithmus. Das Verpackungsproblem ist praktisch unlösbar.

Beispiel 16 (*Stundenplanproblem*, engl. *time–tabling problem*)**.** Die Erstellung eines Stundenplans lässt sich als Anwendung des Verpackungsproblems ansehen: Die Behälter sind die zur Verfügung stehenden Raum–Zeit–Kombinationen, auf die alle zu erteilenden Unterrichtsstunden, also die Fach–Lerngruppe–Kombinationen, zu verteilen sind. Die Kombinationen aus Fach und Lerngruppe müssen dabei entsprechend der Anzahl vorgeschriebener Wochenstunden berücksichtigt werden. Die Anzahl n der Gegenstände entspricht also der Gesamtzahl der wöchentlichen Unterrichtsstunden. Für die Verpackungsgröße gilt hier $G = 1$, weil pro Raum und Unterrichtsstunde nur *eine* Lerngruppe in *einem* Fach unterrichtet werden kann.

Hinzu kommen zahlreiche Randbedingungen, die bei der Suche nach einer Lösung zu beachten sind. Beispielsweise muss die Raumgröße entsprechend der Größe der Lerngruppe berücksichtigt werden, bestimmte Unterrichtstunden müssen in speziellen Fachräumen stattfinden, Lehrpersonen können nicht gleichzeitig in verschiedenen Räumen unterrichten, eine Lerngruppe kann nicht gleichzeitig in verschiedenen Fächern unterrichtet werden usw. Auch das Stundenplanproblem ist also praktisch unlösbar. Wie es dazu kommt, dass es trotzdem Stundenpläne gibt, sehen wir uns in Abschnitt 8 an.

Beispiel 17 (*Faktorisierungsproblem*)**.** Gegeben ist eine n–stellige natürliche Zahl, gesucht ist ihre Zerlegung in Primfaktoren. Alle Versuche, einen Algorithmus mit akzeptabler Komplexität zu finden, sind bisher gescheitert. Es ist stets ein exponentieller Aufwand in Abhängigkeit von n erforderlich. Damit ist auch das Faktorisierungsproblem praktisch unlösbar, woraus seine grundlegende Bedeutung im Zusammenhang mit Verschlüsselungstechniken resultiert.

Beispiel 18 (*Türme von Hanoi*)**.** Ein Lösungsalgorithmus für das Problem der *Türme von Hanoi* wurde in Abschnitt 3 angegeben. Er hat die Zeitkomplexität $O(2^n)$, und ein schnellerer Algorithmus ist nicht möglich. Das Problem *Türme von Hanoi* ist praktisch unlösbar.

Unser Ziel ist es nun, diejenigen Probleme zu klassifizieren, für die *möglicherweise* ein schneller Algorithmus existiert.

6 Nichtdeterministische Algorithmen, P und NP

Wir wenden uns noch einmal dem Problem *Hamilton–Zyklus* zu. Wie oben gesehen besteht eine Möglichkeit zur Lösung des Problems darin, ausgehend von einem Startknoten der Reihe nach alle $(n-1)!$ Permutationen der übrigen Knoten daraufhin zu überprüfen, ob sie einen Rundweg im Graphen darstellen. Dazu kann man sich die Knoten in einem Auswahlbaum angeordnet vorstellen, wie dies Abbildung 14 für den in Abbildung 13 gezeigten Graphen verkürzt darstellt.

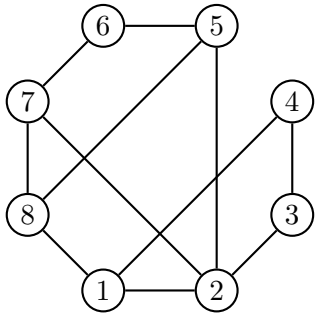

Abbildung 13: Graph zum Problem *Hamilton–Zyklus*

Wir betrachten nun den folgenden „Lösungsalgorithmus“:

— Errate einen Pfad im Auswahlbaum.

— Überprüfe, ob es sich um einen Hamilton–Zyklus handelt.

Wenn eine bestimmte Knotenfolge vorgegeben ist, lässt sich leicht überprüfen, ob es sich um einen Hamilton–Zyklus handelt: Die Knoten müssen der Reihe nach durch Kanten verbunden sein. Auch das Auswählen eines Pfades erfordert keinen großen Rechenaufwand. Beide Schritte sind mit polynomialem Zeitaufwand durchführbar. Dennoch wird man das Verfahren nicht ohne weiteres als *Lösungsalgorithmus* bezeichnen. Was diesen Algorithmus von bisherigen unterscheidet, ist das Raten der richtigen Lösung. Hier ist nicht festgelegt, auf welche Art diese Lösung bestimmt wird. Einen solchen Algorithmus, in dem es mehrere Möglichkeiten für den konkreten Ablauf der Bearbeitung gibt, bezeichnet man als *nichtdeterministisch*.

Nichtdeterministische Algorithmen bestehen also aus einer *Ratephase*, in der Lösungskandidaten bestimmt werden, und einer *Prüfphase*, in der die Kandidaten daraufhin getestet werden, ob sie tatsächlich eine Lösung des Problems darstellen. Theoretisch gesehen erzeugt ein nichtdeterministischer Algorithmus für alle möglichen Lösungskandidaten eine Kopie von sich selbst. In der Praxis können

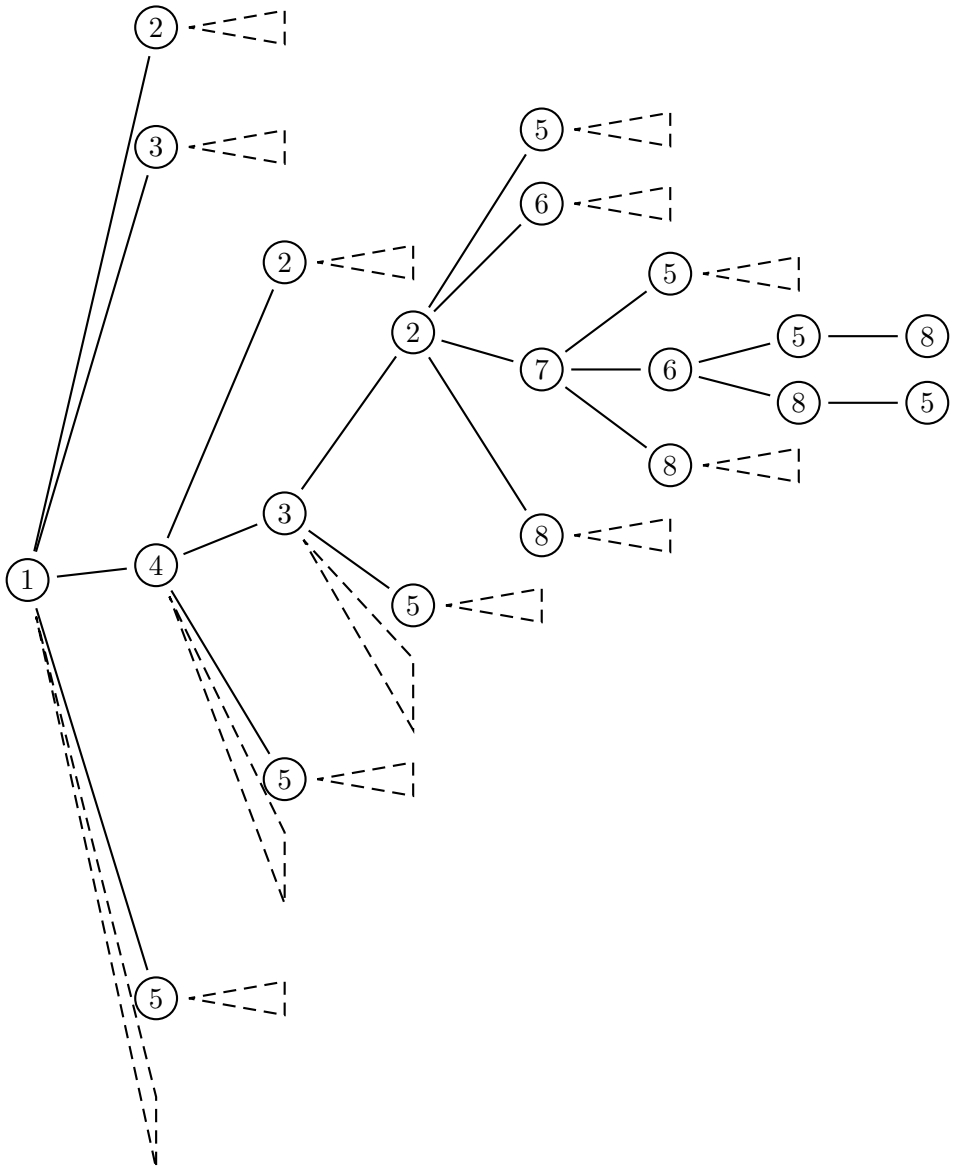

Abbildung 14: Auswahlbaum für einen Hamilton–Zyklus in einem Graphen mit acht Knoten

natürlich nicht beliebig viele Prozesse gleichzeitig ablaufen. Man muss deshalb einen nichtdeterministischen Algorithmus durch einen deterministischen nachahmen, und zwar durch ein Verfahren, das *Rückverfolgung* (engl. *backtracking*) genannt wird: Dabei wird jeder Lösungsversuch, der nicht zum Ziel führt, bis zur letzten Verzweigung im Entscheidungsbaum zurückgezogen und ein anderer Weg von dort aus weiterverfolgt. Dadurch ergibt sich im ungünstigsten Fall jedoch die oben bereits angesprochene Überprüfung aller möglichen Lösungswege, so dass nichtdeterministische Algorithmen zunächst keine praktische Relevanz haben.

Von Bedeutung sind sie aber für theoretische Betrachtungen immer dann, wenn sowohl die Ratephase als auch die Prüfphase polynomialen Zeitaufwand erfordern. Dann ergibt sich eine neue Problemklasse, die sich durch folgende Eigenschaft auszeichnet: Jedes Problem wäre mit polynomialem Zeitaufwand lösbar, wenn es gelänge, die jeweilige Ratephase durch einen polynomialen deterministischen Algorithmus zu ersetzen.

Wir führen zunächst eine Bezeichnung für die praktisch lösbaren Probleme ein (vgl. Definition 10).

Definition 19. Die Klasse P enthält genau diejenigen Probleme, für die ein polynomialer Lösungsalgorithmus existiert.

Unter allen Problemen, die *prinzipiell lösbar* sind, lässt sich nun eine Klasse definieren, die auch viele der bisher als *praktisch unlösbar* bezeichneten Probleme enthält.

Definition 20. Die Klasse NP enthält genau diejenigen Probleme, für die folgende Eigenschaften erfüllt sind:

— Es existiert ein Algorithmus mit exponentiellem Zeitaufwand (z. B. die vollständige Durchsuchung aller Lösungsmöglichkeiten).

— Es ist möglich, durch ein nichtdeterministisches Verfahren (durch Raten) mit polynomialem Zeitaufwand eine Lösung zu bestimmen.

— Es gibt einen Verifikationsalgorithmus mit polynomialem Zeitaufwand zum Nachweis der Richtigkeit einer gefundenen (geratenen) Lösung.

Die Klasse NP enthält also alle Probleme, die mit einem nichtdeterministischen Verfahren polynomial lösbar sind — daher die Bezeichnung. Für zahlreiche Probleme in NP existiert kein schneller Lösungsalgorithmus, aber alle Probleme in NP haben eine gemeinsame interessante Eigenschaft: Man kann eine Lösung möglicherweise nicht schnell finden, aber eine gefundene Lösung immer schnell verifizieren.

Weil jeder deterministische Algorithmus zu einem nichtdeterministischen erweitert werden kann (der dasselbe Problem löst), liegen alle Probleme aus P auch in NP:

$$P \subseteq NP\ .$$

Von zentraler Bedeutung — aber bisher ungelöst — ist nun die Frage, ob jedes Problem in NP auch durch einen deterministischen Algorithmus mit polynomialem Zeitaufwand gelöst werden kann, ob also die Menge $NP \setminus P$ leer ist.

$$P =^{?} NP\,.$$

Weil für zahlreiche Probleme in NP trotz intensiver Bemühungen noch kein polynomialer Algorithmus gefunden wurde, spricht einiges dafür, dass diese Probleme in $NP \setminus P$ liegen, dass also $P \subsetneq NP$ gilt. Andererseits konnte bisher noch für keines der fraglichen Probleme bewiesen werden, dass es keinen polynomialen Algorithmus geben kann. Das Problem $P =^{?} NP$ ist also weiter offen.

Beispiel 21. Folgende Probleme liegen in der Klasse NP: das Problem *Hamilton–Zyklus*, das *Problem des Handlungsreisenden*, das *Verpackungsproblem*, das *Stundenplanproblem* und das *Primzahlenproblem*.

Weil *jeder* Lösungsalgorithmus für das Problem *Türme von Hanoi* die Zeitkomplexität $O(2^n)$ hat, liegt dieses Problem nicht in NP. Hier erfordert das Umlegen der Scheiben bzw. das Ausgeben einer Lösung bereits $2^n - 1$ Schritte, so dass kein nichtdeterministischer Algorithmus existieren kann, der das Problem mit polynomialem Zeitaufwand löst.

Kandidaten für die Menge $NP \setminus P$ sind alle Probleme aus NP, für die bisher kein Lösungsalgorithmus mit polynomialem Zeitaufwand gefunden wurde. Um die Frage der Gleichheit von P und NP beantworten zu können, muss entweder bewiesen werden, dass für *ein* Problem in NP kein polynomialer Algorithmus existieren kann oder dass für *jedes* Problem in NP (auch jedes noch nicht formulierte) ein solcher Algorithmus gefunden werden kann. Dazu ist es notwendig, die *schwersten* Probleme in NP zu klassifizieren, was Inhalt des folgenden Abschnittes ist.

7 NP–Vollständigkeit

Im Jahr 1971 entdeckte Stephen Cook, dass es in NP Probleme gibt, die man als die schwersten in dieser Klasse beschreiben kann. Bei diesen Problemen hätte die Entdeckung eines deterministischen polynomialen Algorithmus zur Folge, dass jedes Problem in NP deterministisch polynomial lösbar ist. Die Existenz solcher Probleme rechtfertigt die folgende

Definition 22. Ein Problem heißt *NP–vollständig* (engl. *NP–complete*), wenn die folgenden Eigenschaften erfüllt sind:

— Das Problem gehört zur Klasse NP.

— Das Problem gehört genau dann zur Klasse P, wenn $P = NP$ gilt.

Gilt $P = NP$, so ist jedes Problem aus NP auch polynomial lösbar. Interessant ist aber der in der zweiten Bedingung enthaltene Umkehrschluss: Wenn für ein NP–vollständiges Problem gezeigt werden kann, dass ein polynomialer Lösungsalgorithmus existiert, so muss für jedes Problem in NP ein solcher Algorithmus existieren — mit der Konsequenz, dass dann $P = NP$ gilt.

Um verstehen zu können, wie der Nachweis der NP–Vollständigkeit eines Problems geführt werden kann, beweisen wir zunächst eine Eigenschaft NP–vollständiger Probleme.

Definition 23. Ein Problem Q_1 heißt *polynomial reduzierbar* auf ein Problem Q_2, falls es einen polynomialen Algorithmus gibt, der einen Lösungsalgorithmus für Q_2 zu einem Lösungsalgorithmus für Q_1 erweitert. Man schreibt in diesem Fall $Q_1 \leq_p Q_2$.

Beim Nachweis der polynomialen Reduzierbarkeit konstruiert man eine berechenbare und polynomial zeitbeschränkte Funktion f, die jeder Eingabe x für Q_1 eine Eingabe $f(x)$ für Q_2 zuordnet mit der Eigenschaft, dass (die Entscheidungsvariante von) Q_1 für die Eingabe x genau dann mit „ja“ zu beantworten ist, wenn Q_2 für $f(x)$ mit „ja“ zu beantworten ist.

Wenn also für ein Problem $Q \in NP$ gezeigt werden kann, dass ein polynomialer Lösungsalgorithmus existiert, so folgt unmittelbar die Existenz entsprechender polynomialer Algorithmen für alle Probleme, die polynomial auf Q reduzierbar sind. Dies ermöglicht eine weitere Charakterisierung NP–vollständiger Probleme.

Satz 24. Ein Problem $Q \in NP$ ist genau dann NP–vollständig, wenn jedes Problem $Q' \in NP$ polynomial auf Q reduzierbar ist.

$$Q \in NP \text{ ist NP–vollständig} \Leftrightarrow \forall Q' \in NP : \; Q' \leq_p Q \; .$$

Beweis. Sei Q NP–vollständig. Gilt $P = NP$, so existiert für alle Probleme in NP ein polynomialer Lösungsalgorithmus, so dass der in Definition 23 geforderte Erweiterungsalgorithmus bereits durch den Lösungsalgorithmus gegeben ist. Folglich ist in diesem Fall jedes Problem $Q' \in NP$ polynomial auf Q reduzierbar.

Gilt $P \subsetneq NP$, so liegt Q wegen der vorausgesetzten NP–Vollständigkeit in $NP \setminus P$, und es kann kein polynomialer Lösungsalgorithmus für Q existieren. Auch in diesem Fall ist jedes Problem $Q' \in NP$ polynomial auf Q reduzierbar.

Es gelte nun umgekehrt $Q' \leq_p Q$ für alle Probleme $Q' \in NP$. Gilt $Q \in P$, so existiert für alle Probleme in NP ein polynomialer Lösungsalgorithmus, weil der polynomiale Algorithmus für Q mit polynomialem Aufwand zu einem entsprechenden Algorithmus erweitert werden kann. Aus $Q \in P$ folgt also $P = NP$.

Gilt $Q \notin P$, so folgt aus $Q \in NP$, dass $P \neq NP$ gilt. Insgesamt ist also $P = NP$ genau dann, wenn $Q \in P$ gilt, so dass Q nach Definition 22 NP–vollständig ist. □

Um nun die NP–Vollständigkeit eines Problems $Q \in NP$ nachzuweisen, müssen wir prinzipiell zeigen, dass jedes Problem in NP polynomial auf Q reduzierbar ist. In der Praxis gestaltet sich ein solcher Beweis aber viel einfacher, weil die Relation $\leq_p$ transitiv ist:

$$Q_1 \leq_p Q_2 \wedge Q_2 \leq_p Q_3 \Rightarrow Q_1 \leq_p Q_3 \; .$$

Es ist also ausreichend, wenn wir zeigen können, dass *ein* Problem polynomial reduzierbar auf Q ist, von dem schon bekannt ist, dass es NP–vollständig ist. Dies erleichtert die Beweisführung ganz entscheidend, setzt aber voraus, dass es einen „Urahn" der NP–vollständigen Probleme gibt, für den ein ausführlicher Beweis erforderlich ist, der die Eigenschaft der Transitivität noch nicht ausnutzen kann. Cook leistete diese Vorarbeit, indem er 1971 bewies, dass das Erfüllbarkeitsproblem der Aussagenlogik NP–vollständig ist. Wir klären zunächst einige Begriffe.

Definition 25. Ein *boolescher Ausdruck* besteht aus Variablen, die durch Operatoren verknüpft sind. Die Variablen können nur die Werte *wahr* und *falsch* (engl. *true* und *false*) annehmen. Die erlaubten Operatoren sind $\wedge$ (*Konjunktion*, logisches *und*), $\vee$ (*Disjunktion*, logisches *oder*) und $\neg$ (*Negation, Komplement*, logisches *nicht*). Zur Strukturierung eines Ausdrucks können Klammern gesetzt werden.

Eine Disjunktion von Variablen oder deren Komplementen heißt *Klausel*. Ein boolescher Ausdruck hat *konjunktive Normalform*, wenn er aus Konjunktionen von Klauseln besteht. Ein boolescher Ausdruck heißt *erfüllbar*, wenn es eine Belegung der Variablen mit *wahr* oder *falsch* gibt, so dass der Ausdruck insgesamt wahr ist.

Beispiel 26. Der Ausdruck $(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg d)$ besteht aus den beiden Klauseln $a \vee b \vee \neg c$ und $a \vee \neg b \vee \neg d$, die konjunktiv verknüpft sind. Er hat also konjunktive Normalform. Der Ausdruck ist erfüllbar, weil er durch die Variablenbelegung a = falsch, b = wahr, c = wahr und d = falsch insgesamt wahr wird.

Definition 27. Das *Erfüllbarkeitsproblem der Aussagenlogik* (engl. *satisfiability problem*, kurz SAT) lautet wie folgt:

> Gegeben ist ein boolescher Ausdruck B in konjunktiver Normalform.
> Gibt es eine Belegung der Variablen, so dass B wahr ist?

Satz 28 (Cook)**.** Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist NP–vollständig.

Der Beweis gliedert sich in zwei Schritte (nach Socher 2003, 174): Zunächst wird gezeigt, dass SAT in NP liegt, und anschließend wird nachgewiesen, dass jedes Problem in NP polynomial auf SAT reduzierbar ist.

Der erste Teil ist leicht nachvollziehbar. Ein nichtdeterministischer Algorithmus errät eine Belegung der Variablen und überprüft anschließend, ob der Ausdruck durch diese Belegung wahr ist. Bei n Variablen führt der Algorithmus n Ratephasen durch. Die Überprüfung des Wahrheitswertes erfordert polynomialen Zeitaufwand in Abhängigkeit von der Länge des Ausdrucks, weil ein Ausdruck der Länge n nicht mehr als n Variablen enthalten kann.

Aus dem linearen Aufwand für das Raten und dem polynomialen Aufwand für die Überprüfung ergibt sich insgesamt, dass das Erfüllbarkeitsproblem durch einen nichtdeterministischen Algorithmus mit polynomialem Zeitaufwand lösbar ist. Das Problem gehört also zur Klasse NP.

Der zweite Teil ist wesentlich schwieriger. Um von den unterschiedlichen Formaten der Eingabedaten aller relevanten Probleme (Zahlen, Graphen, Zeichenketten usw.) abstrahieren zu können, identifiziert man jedes Problem $Q \in NP$ mit einer charakteristischen Sprache L. Diese Sprache wird wie folgt konstruiert: Die möglichen Eingabedaten für Q werden eindeutig in Wörter über dem Alphabet von L übersetzt. L enthält dann alle Wörter, die den Eingabedaten entsprechen, für die (die Entscheidungsvariante von) Q mit „ja“ zu beantworten ist.

Cook geht in seinem Beweis von der Tatsache aus, dass es zu jeder solchen Sprache L eine nichtdeterministische Turing–Maschine M_L gibt, die zu jedem Eingabewort w mit polynomialem Zeitaufwand entscheidet, ob $w \in L$ gilt oder nicht. Die eigentliche Beweisidee besteht nun darin, eine Menge von Klauseln zu konstruieren, die die Arbeitsweise von M_L beschreibt und die in polynomialer Zeit aus w konstruiert werden kann. Der durch konjunktive Verknüpfung der Klauseln entstehende boolesche Ausdruck ist genau dann wahr, wenn M_L das Eingabewort w akzeptiert, d. h. wenn $w \in L$ gilt. Damit kann jedes Problem in NP auf das Erfüllbarkeitsproblem der Aussagenlogik zurückgeführt werden.

Die von Cook geleistete Konstruktion der Klauselmenge ist sehr kompliziert. Sie im Detail zu beschreiben führt hier zu weit. Wir sehen uns stattdessen an einem Beispiel an, wie sich SAT polynomial auf ein graphentheoretisches Problem reduzieren lässt.

Definition 29. Das *Knotenüberdeckungsproblem* (engl. *vertex covering problem*, kurz *VCP*) ist wie folgt definiert:

> Gegeben ist ein ungerichteter Graph G und eine natürliche Zahl k.
> Gibt es eine Teilmenge von G mit k Knoten, so dass jede Kante mindestens einen Endpunkt aus dieser Teilmenge enthält?

Beispiel 30. Gegeben ist der in Abbildung 15 dargestellte Graph. Gibt es eine Knotenüberdeckung mit vier Knoten? Gibt es eine mit drei Knoten?

Die Knotenmengen $\{1, 3, 5, 7\}$ und $\{2, 4, 6, 8\}$ erfüllen die Bedingung, dass jede Kante des Graphen mindestens einen Endpunkt aus der jeweiligen Menge besitzt. Für $k = 4$ ist das Problem also mit „ja“ zu beantworten, für $k = 3$ mit „nein“.

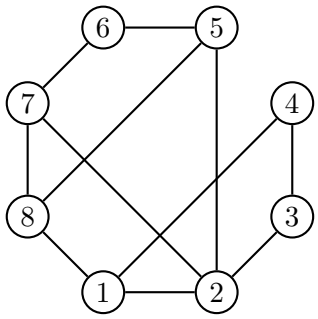

Abbildung 15: Graph zum *Knotenüberdeckungsproblem*

Satz 31. Das *Knotenüberdeckungsproblem* ist NP–vollständig.

Beweis. Wir zeigen, dass sich das Erfüllbarkeitsproblem SAT polynomial auf VCP reduzieren lässt: $SAT \leq_p VCP$. Aus der Transitivität der Relation $\leq_p$ folgt dann, dass sich jedes Problem $Q \in NP$ polynomial auf VCP reduzieren lässt:

$$\text{Nach Cook gilt: } \forall Q \in NP : Q \leq_p SAT$$
$$\text{Wir zeigen: } SAT \leq_p VCP$$
$$\text{Aus } Q \leq_p SAT \leq_p VCP \text{ folgt: } \forall Q \in NP : Q \leq_p VCP$$

Zunächst müssen wir uns aber vergewissern, dass VCP in NP liegt. Dazu geben wir einen nichtdeterministischen Algorithmus an, der das Problem mit polynomialem Zeitaufwand löst.

Gegeben sei also ein Graph G mit n Knoten und eine natürliche Zahl k. Der Algorithmus wählt im ersten Schritt (nichtdeterministisch) k Knoten aus. Dies erfordert linearen Aufwand in Abhängigkeit von n. Im zweiten Schritt muss nun überprüft werden, ob die ausgewählte Teilmenge der Knoten eine Überdeckung darstellt. Dies ist leicht möglich, wenn man sich den Graphen durch eine $n \times n$–Matrix M repräsentiert vorstellt, für die gilt:

$$m_{i,j} = \begin{cases} 1 & \text{falls } G \text{ eine Kante von } i \text{ nach } j \text{ enthält} \\ 0 & \text{sonst .} \end{cases}$$

Für den Graphen in Abbildung 15 sähe diese Matrix als rechteckiges Schema wie

folgt aus:

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

Die Matrix ist symmetrisch zur Hauptdiagonalen, weil G ungerichtet ist. Es muss nun also nur für alle $m_{i,j}$ mit $m_{i,j} = 1$ und $i > j$ überprüft werden, ob sich i oder j unter den k ausgewählten Knoten befindet. Dies ist sicherlich mit polynomialem Aufwand in Abhängigkeit von n möglich, so dass *VCP* in NP liegt.

Wir reduzieren nun *SAT* polynomial auf *VCP*, indem wir ein Verfahren angeben, durch das jedem booleschen Ausdruck B, der auf Erfüllbarkeit getestet wird, eine Eingabe für *VCP* (ein Graph G und ein $k \in \mathbb{N}$) zugeordnet wird, für die gilt: B ist genau dann erfüllbar, wenn G eine Überdeckung mit k Knoten enthält.

Sei also B ein boolescher Ausdruck mit n Variablen. Zur Veranschaulichung verfolgen wir die Konstruktionsschritte am Beispiel des Ausdruckes

$$(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c \vee \neg d) \ .$$

Für jede der Variablen enthält G zwei Knoten, die später das Vorkommen der Variablen in den Klauseln (negiert oder nicht negiert) repräsentieren. Sie werden paarweise durch eine Kante verbunden (vgl. Abbildung 16).

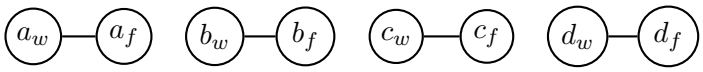

Abbildung 16: Konstruktion eines Graphen zum Beweis von Satz 31, Schritt 1

Für jede Klausel enthält G einen vollständigen Teilgraphen, dessen Knotenanzahl durch die Anzahl der Literale in der jeweiligen Klausel bestimmt ist (vgl. Abbildung 17).

Die Kanten zwischen den oberen und unteren Teilgraphen werden so festgelegt, dass jeder Knoten aus einem unteren Teilgraphen mit genau einem Knoten aus einem oberen Teilgraphen verbunden ist, und zwar so, dass jede Kante ein Literal der entsprechenden Klausel repräsentiert (vgl. Abbildung 18). Im Beispiel repräsentiert dann der linke Teilgraph die Klausel $a \vee b \vee \neg c$ und der rechte Teilgraph die Klausel $a \vee \neg b \vee c \vee \neg d$.

Nun ergänzen wir die Eingabedaten für *VCP* noch durch die Angabe einer natürlichen Zahl k, die sich wie folgt berechnet: Man addiert die Anzahl der

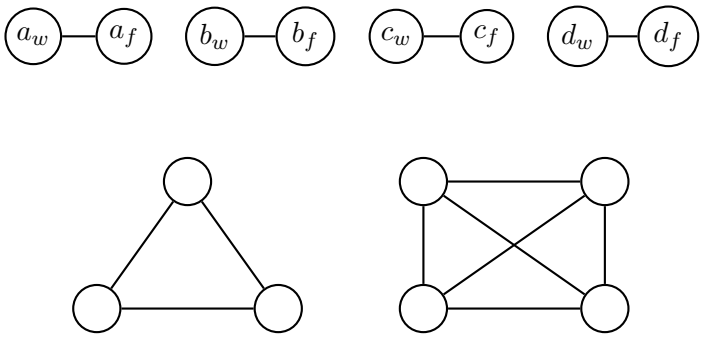

Abbildung 17: Konstruktion eines Graphen zum Beweis von Satz 31, Schritt 2

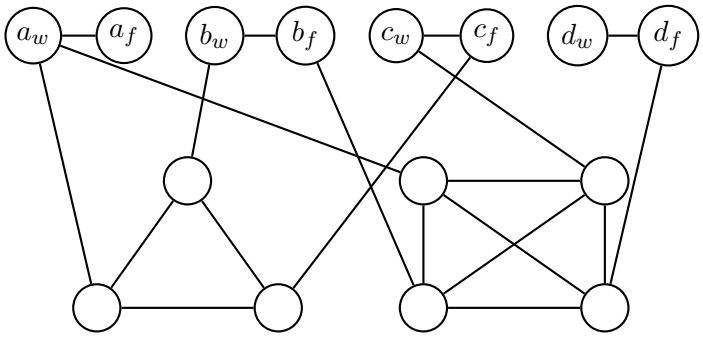

Abbildung 18: Konstruktion eines Graphen zum Beweis von Satz 31, Schritt 3

Variablen zur Anzahl der Literale in B und subtrahiert die Anzahl der Klauseln. In unserem Beispiel ergibt sich

$$\begin{aligned} k &= 4 + (3 + 4) - 2 \\ &= 9 \, . \end{aligned}$$

Wir müssen jetzt zeigen, dass jeder Ausdruck B genau dann erfüllbar ist, wenn G eine Überdeckung mit k Knoten besitzt. Sei also zunächst B erfüllbar. Dann gibt es eine Belegung der Variablen mit *wahr* oder *falsch*, so dass jede Klausel wahr ist. Man wählt nun aus dem oberen Teilgraphen diejenigen Knoten in die Überdeckung, die einer solchen Belegung entsprechen. Im Beispiel erfüllt etwa $a = falsch$, $b = wahr$, $c = wahr$ und $d = falsch$ den gegebenen Ausdruck, so dass a_f, b_w, c_w und d_f gewählt werden. Aus dem unteren Teilgraphen gehören alle Knoten zur Überdeckung mit Ausnahme je eines Knotens, der mit einem bereits ausgewählten Knoten des oberen Teilgraphen verbunden ist (vgl. Abbildung 19). Ein solcher Knoten existiert nach Konstruktion der Kanten und aufgrund der Knotenauswahl im oberen Teilgraphen.

Die Anzahl der im oberen Teilgraphen ausgewählten Knoten entspricht der Anzahl der Variablen. Die Gesamtzahl der in den unteren Teilgraphen ausgewähl-

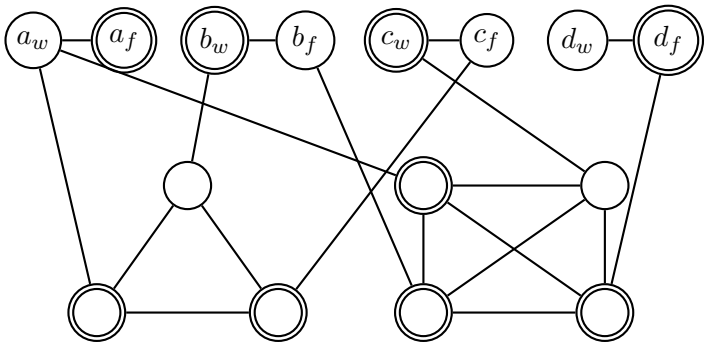

Abbildung 19: Auswahl von Knoten zum Beweis von Satz 31

ten Knoten entspricht der Anzahl der Literale, abzüglich der Anzahl der Klauseln. Also sind genau k Knoten ausgewählt, die nach Konstruktion eine vollständige Überdeckung des Graphen darstellen.

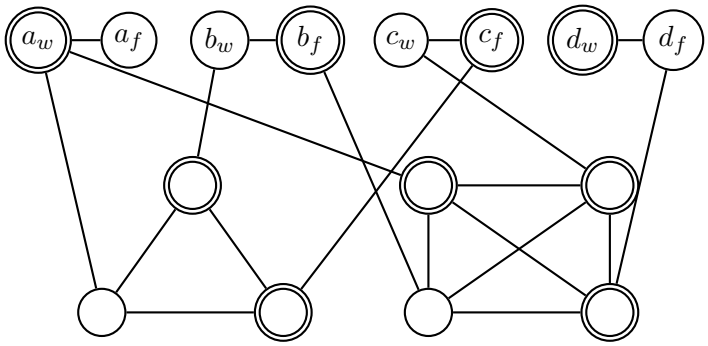

Abbildung 20: Vollständige Überdeckung mit 9 Knoten

Sei nun umgekehrt eine vollständige Überdeckung des Graphen mit k Knoten gegeben, z. B. wie in Abbildung 20. Weil die unteren Teilgraphen nach Konstruktion vollständig sind, kann jeweils höchstens ein Knoten nicht zur Überdeckung gehören, weil diese sonst nicht vollständig wäre. Aus dem gleichen Grund muss im oberen Teilgraphen für jede Kante mindestens ein Knoten zur Überdeckung gehören. Nach Konstruktion von k gehört daher bei den unteren Teilgraphen jeweils *genau* ein Knoten nicht zur Überdeckung.

Man wählt nun eine Belegung der Variablen im Ausdruck B, die der Auswahl der Knoten im oberen Teilgraphen entspricht, im Beispiel $a = wahr$, $b = falsch$, $c = falsch$ und $d = wahr$. Es bleibt nur noch zu zeigen, dass diese Belegung den Ausdruck erfüllt.

Jeder der unteren Teilgraphen enthält einen Knoten, der nicht zur Überdeckung gehört. Ihn verbindet eine Kante mit einem Knoten im oberen Teilgraphen, der zur Überdeckung gehören muss (weil diese ansonsten nicht vollständig wäre).

Nach Konstruktion der Belegung ist das durch ihn repräsentierte Literal wahr, so dass auch die Klausel, die dem zugehörigen unteren Teilgraphen entspricht, wahr ist. Die Belegung erfüllt also den Ausdruck B.

Insgesamt ergibt sich, dass B genau dann erfüllbar ist, wenn G eine Überdeckung mit k Knoten enthält. Es gilt also $SAT \leq_p VCP$. Wegen der NP–Vollständigkeit des Erfüllbarkeitsproblems und der Transitivität der Relation $\leq_p$ lässt sich jedes Problem $Q \in NP$ polynomial auf VCP reduzieren — VCP ist NP–vollständig. □

Nach diesem ausführlichen Beweis beschränken wir uns nun darauf, weitere Probleme ohne Nachweis als NP–vollständig einzustufen.

Definition 32. Das Problem *3KNF–SAT* ist wie folgt definiert:

> Gegeben ist ein boolescher Ausdruck B in konjunktiver Normalform mit höchstens drei Literalen. Ist B erfüllbar?

Dieses Problem scheint leichter lösbar zu sein als das allgemeine Erfüllbarkeitsproblem, ist aber auch NP–vollständig. Seine Bedeutung für die Informatik liegt darin, dass es leichter auf andere Probleme polynomial reduzierbar ist.

Definition 33. Das *Mengenüberdeckungsproblem* ist wie folgt definiert:

> Gegeben sind Teilmengen einer endlichen Menge M, sowie eine natürliche Zahl k. Gibt es eine Auswahl von k Teilmengen, bei der bereits alle Elemente von M vorkommen?

Das Mengenüberdeckungsproblem ist NP–vollständig.

Definition 34. Sei G ein ungerichteter Graph. Unter einer *Clique* versteht man einen vollständigen Teilgraphen, also eine Menge von Knoten aus G, die paarweise durch eine Kante mit einander verbunden sind. Das Problem *Clique* ist wie folgt definiert:

> Gegeben ist ein ungerichteter Graph G und eine natürliche Zahl k. Besitzt G eine Clique mit mindestens k Knoten?

Das Problem Clique ist NP–vollständig.

Definition 35. Das *Rucksackproblem* (engl. *subset sum problem*) ist wie folgt definiert:

> Gegeben sind natürliche Zahlen $a_1, a_2, \ldots, a_k$ und b. Gibt es eine Teilmenge I von $\{1, 2, \ldots, k\}$, so dass $\sum_{i \in I} a_i = b$ gilt?

Die Bezeichnung dieses Problems entstammt der Praxis: Gegeben sind k Gegenstände von unterschiedlichem Gewicht. Gefragt ist nach der Möglichkeit, einen Rucksack so mit einer Auswahl der Gegenstände zu füllen, dass der Inhalt ein vorgegebenes Gesamtgewicht hat. Auch das Rucksackproblem ist NP–vollständig.

Definition 36. Das *Erbteilungsproblem* (engl. *partition problem*) ist wie folgt definiert:

> Gegeben sind natürliche Zahlen $a_1, a_2, \ldots, a_k$. Gibt es eine Teilmenge I von $\{1, 2, \ldots, k\}$ mit $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

Auch hier lässt die formale Definition den Ursprung der Problembezeichnung nur erahnen. Er wird deutlich, wenn man sich vorstellt, die Zahlen $a_1, a_2, \ldots, a_k$ seien Werte von Münzen einer gerecht unter zwei Personen aufzuteilenden Erbschaft. Das Erbteilungsproblem ist NP–vollständig.

Definition 37. Das *k–Färbbarkeitsproblem* ist wie folgt definiert:

> Gegeben ist ein ungerichteter Graph G und eine natürliche Zahl k. Gibt es eine Färbung der Knoten von G mit k verschiedenen Farben, so dass keine benachbarten Knoten dieselbe Farbe haben?

Hier ist entscheidend, dass die Zahl k zu den Eingabedaten des Problems gehört und nicht vorab festgelegt ist. In dieser Formulierung ist das Färbbarkeitsproblem NP–vollständig, für eine fest vorgegebene Zahl k aber in polynomialer Zeit lösbar.

Von den in den vorangegangenen Abschnitten angesprochenen Problemen sind die folgenden NP–vollständig: das Problem Hamilton–Zyklus, das Problem des Handlungsreisenden, das Verpackungsproblem, das Stundenplanproblem und das Primzahlenproblem.

8 Näherungslösungen

Zahlreiche Probleme, die derzeit nicht der Klasse P zugerechnet werden können, besitzen große Relevanz für praktische Aufgaben. Da aber die bekannten Lösungsalgorithmen exponentielle Laufzeiten haben, ist es im Allgemeinen nicht möglich, korrekte bzw. optimale Lösungen zu bestimmen. Die Aufgaben müssen aber gelöst werden, so dass man gezwungen ist in Kauf zu nehmen, dass eine „Lösung“ nur teilweise korrekt oder nicht optimal ist.

Beispiel 38. An jeder Schule gibt es in jedem Schuljahr einen neuen Stundenplan, obwohl wir das Stundenplanproblem als praktisch unlösbar eingestuft haben.

Beispiel 39. Die Planung der Kurskopplung in der Oberstufe ist eine Anwendung des k–Färbbarkeitsproblems. Der zugehörige Graph enthält für jeden eingerichteten Kurs einen Knoten. Alle Knoten, die den verschiedenen Kursen eines Schülers entsprechen, werden jeweils durch Kanten verbunden. Kurse können zeitlich gekoppelt werden, wenn die jeweiligen Knoten nicht benachbart sind und somit gleich gefärbt werden können.

Beispiel 40. Das Problem des Handlungsreisenden findet eine regelmäßige Anwendung in der Routenplanung der Müllabfuhr oder bei der Postzustellung. Hier geht es neben einer Maximierung der Anzahl der zu besuchenden Orte zunächst um die Frage, welche „Rundreisen“ unter dem Kostenaspekt „Zeit“ überhaupt möglich sind.

Beispiel 41. Eine Reederei, deren Containerschiffe verschiedene Häfen anlaufen, hat es mit einer Kombination aus zwei NP–vollständigen Problemen zu tun: Das Problem des Handlungsreisenden ist zu lösen, wenn es darum geht, die Fahrtrouten der Schiffe so festzulegen, dass die Gesamtstrecke minimal ist. Dabei ist aber auch das Verpackungsproblem relevant, weil die zu transportierenden Waren so auf die Schiffe verteilt werden müssen, dass deren maximale Nutzlast möglichst erreicht, aber nicht überschritten wird.

Es gibt verschiedene Möglichkeiten, wie man mit praktisch unlösbaren Problemen umgehen kann, um doch zu einem akzeptablen Ergebnis zu gelangen. Zunächst ist es manchmal möglich, einen bekannten exponentiellen Algorithmus zu verwenden, wenn dieser bei konkret auftretenden Eingabedaten meistens schnell genug arbeitet. Es ist dann möglich, eine zeitliche Schranke festzulegen, nach der die Berechnung abgebrochen wird, wenn die aktuellen Eingabedaten einen für die Problemlösung ungünstigen Fall darstellen.

Eine Alternative besteht darin, bei Optimierungsproblemen die Forderung aufzugeben, dass das Ergebnis tatsächlich optimal ist. Beim Stundenplanproblem ist es beispielsweise sinnvoller, einen Plan mit Lücken zu akzeptieren, als gar keinen Stundenplan zu haben. Beim Problem des Handlungsreisenden ist es besser, schnell einen vertretbar kurzen Weg zu finden, als mit extrem hohem Rechenaufwand den absolut kürzesten Weg zu bestimmen.

Beispiel 42 (Algorithmus der billigsten Erweiterung)**.** Eine Näherungslösung für das Problem des Handlungsreisenden (vgl. Beispiel 14) kann auf folgende Art gefunden werden (nach Gasper et al. 1992, 245):

> Man beginnt mit einer beliebigen Stadt. Nun vergrößert man schrittweise die Rundreise um eine weitere Stadt, indem man von den noch nicht zur Rundreise gehörenden Städten diejenige hinzunimmt, durch die der bisherige Weg am wenigsten verlängert wird.

Der Algorithmus findet mit der Komplexität $O(n^2 \cdot \log_2 n)$ eine Rundreise, die höchstens doppelt so lang ist wie die optimale.

Genetische Algorithmen

Zur Konstruktion solcher Näherungslösungen gibt es ein Verfahren, das sich an den Mechanismen der Evolution orientiert. *Genetische Algorithmen* erzeugen

zunächst zufällige Lösungskandidaten und konstruieren daraus dann in mehreren Generationen neue Lösungen mit Hilfe der genetischen Prinzipien *Selektion, Mutation* und *Kreuzung*. Die Menge der Lösungen einer Generation wird in diesem Zusammenhang auch als *Population* bezeichnet, die Lösungen selbst als *Individuen*. Die Selektion wird dadurch simuliert, dass Lösungen gemäß der Problemstellung bewertet, mit einander verglichen und bei niedriger „Qualität“ aus der Population entfernt werden. Mutationen werden durch zufällige Änderungen einzelner Lösungen simuliert und Kreuzungen dadurch, dass je zwei Lösungen mit einander kombiniert werden.

Anstelle einer formalen Definition des Verfahrens betrachten wir das folgende

Beispiel 43. Gesucht ist eine Näherungslösung für das Rucksackproblem (vgl. Definition 35), und zwar in der Optimierungsvariante: Gegeben sind k Zahlen $a_1, a_2, \ldots, a_k$, die das Gewicht der zur Verfügung stehenden Gegenstände angeben, sowie eine Zahl b, die das maximale Gesamtgewicht darstellt. Gesucht ist eine Auswahl von Gegenständen, deren Gesamtgewicht möglichst groß ist, jedoch ohne b zu überschreiten.

Eine Population von n Individuen (also eine Menge mit n Lösungskandidaten) lässt sich als $n \times k$–Matrix L darstellen mit

$$l_{ij} = \begin{cases} 1 & \text{falls der Gegenstand mit dem Gewicht } a_j \text{ Bestandteil der Lösung } i \text{ ist} \\ 0 & \text{sonst .} \end{cases}$$

Die Einträge der Matrix werden zunächst rein zufällig festgelegt und bilden die Ausgangspopulation. Die Konstruktion der nächsten Population erfolgt nun in drei Schritten:

1. Selektion: Für jede Zeile i der Matrix wird der jeweilige Nutzenwert w_i bestimmt. Dieser ergibt sich aus dem Gewicht g_i der ausgewählten Gegenstände. Überschreitet das Gesamtgewicht den Wert b, wird der Nutzenwert -1 vergeben:

$$g_i = \sum_{j=1}^{k} l_{ij} \cdot a_j$$

$$w_i = \begin{cases} g_i & \text{falls } g_i \leq b \\ -1 & \text{sonst .} \end{cases}$$

   Nun wird eine Hilfspopulation gleicher Größe erzeugt (eine $n \times k$–Matrix L^*), indem n–mal zufällig zwei Lösungen mit einander verglichen werden. Diejenige mit dem größeren Nutzenwert wird in die Hilfspopulation aufgenommen. Ergibt sich die Zeile i von L^* durch den Vergleich der Zeilen p und q von L, so gilt also

$$l^*_{ij} = \begin{cases} l_{pj} & \text{falls } w_p > w_q \\ l_{qj} & \text{sonst .} \end{cases} \quad (j = 1, \ldots, k)$$

Anschließend wird L durch L^* ersetzt. Diesen Algorithmus bezeichnet man als *binäre Wettkampfselektion*.

2. Kreuzung: Vor der Kombination der Lösungen werden diese paarweise gruppiert. Dann wird für jedes Paar zufällig eine Zahl $t \in \{1, \ldots, k-1\}$, die Trennstelle, festgelegt. In den Spalten 1 bis t der Matrix werden jeweils die Einträge der „gepaarten“ Lösungen getauscht. Auch hier kann wieder eine Hilfspopulation L^* verwendet werden. Für die Einträge der Matrix gilt beispielsweise bei der Kreuzung der ersten beiden Lösungen

$$l^*_{1j} = \begin{cases} l_{2j} & \text{für } j = 1, \ldots, t \\ l_{1j} & \text{für } j = t+1, \ldots, k \end{cases}$$

$$l^*_{2j} = \begin{cases} l_{1j} & \text{für } j = 1, \ldots, t \\ l_{2j} & \text{für } j = t+1, \ldots, k\,. \end{cases}$$

   Dieses Kreuzungsverfahren wird als *Ein–Punkt–Crossover* bezeichnet.

3. Mutation: Um auch Lösungen zu ermöglichen, die allein durch Kreuzungen nicht entstehen können, werden zufällig einige Individuen ausgewählt, die an einer Stelle verändert werden. In der Matrix wird dann eine Null durch eine Eins bzw. eine Eins durch eine Null ersetzt. Die Wahrscheinlichkeit, mit der ein Individuum mutiert wird, bezeichnet man als *Mutationsrate*, das hier dargestellte Verfahren als *Bit–Flip–Mutation*.

Diese Schritte werden mehrmals (in der Praxis für mehrere hundert Generationen) wiederholt. Abschließend wird noch ein Mal der Nutzenwert aller Lösungen bestimmt, um die beste als Ergebnis auszugeben.

Während die Selektion durch rechnerische Bestimmung der Zielgröße erfolgt, müssen Mutation und Kreuzung mittels rein formaler Operationen auf Zeichenketten bzw. Bitfolgen durchgeführt werden. Dabei ergibt sich das Problem, dass das so konstruierte Individuum wieder ein gültiger Lösungskandidat sein muss. Beim Rucksackproblem ist dies unproblematisch, weil schlimmstenfalls eine Lösung mit dem Nutzenwert -1 entsteht, die im Laufe der folgenden Selektionen wieder aus der Population verschwindet. Beim Problem des Handlungsreisenden führt aber die Kreuzung in den meisten Fällen auf eine Knotenfolge, die gar keine Rundreise mehr darstellt. Bei der Konstruktion genetischer Algorithmen besteht allgemein häufig die größte Schwierigkeit darin, eine geeignete Kodierung der Lösungskandidaten zu finden. Für das Problem des Handlungsreisenden gibt A. K. Dewdney eine elegante Lösung für diese Problematik an (Dewdney 2001, 106).

Probabilistische Algorithmen

Eine weitere Alternative beim Umgang mit praktisch unlösbaren Problemen besteht in der Aufgabe der Forderung, dass der Lösungsalgorithmus korrekt arbei-

ten muss. Algorithmen, die mit einer gewissen (kleinen) Wahrscheinlichkeit ein falsches Ergebnis liefern, bezeichnet man als *probabilistisch*.

Beispiel 44. Das NP–vollständige Problem *Independent Set* ist wie folgt definiert:

> In einem ungerichteten Graphen G heißt eine Teilmenge I der Knotenmenge *unabhängig* (bzw. *Independent Set*), wenn G keine Kante enthält, die zwei Knoten aus I verbindet. Gesucht ist eine unabhängige Menge mit maximaler Kardinalität.

Hat ein ungerichteter Graph G n Knoten und bezeichnet d den durchschnittlichen Grad der Knoten ($d \geq 1$), dann berechnet der folgende Algorithmus eine unabhängige Teilmenge I der Knotenmenge von G, die im Erwartungswert $\frac{n}{2d}$ Knoten enthält (nach Hofmeister 2006, 4):

— Bilde eine (zufällige) Teilmenge I der Knotenmenge von G, indem jeder Knoten mit Wahrscheinlichkeit $\frac{1}{d}$ in die Menge I aufgenommen wird.

— Überprüfe, ob es in I Knoten gibt, die im Graph G durch eine Kante verbunden sind. Falls ja, entferne einen der betroffenen Knoten aus I.

— Wiederhole den vorherigen Schritt, bis I eine unabhängige Menge ist.

Zum Vergleich: Es gibt einen exponentiellen deterministischen Algorithmus, der eine unabhängige Menge mit mindestens $\frac{n}{d+1}$ Knoten berechnet.

Dieser letzte Abschnitt hat gezeigt, dass es oft Wege gibt, mit NP–vollständigen Problemen in der Praxis fertig zu werden. Andererseits kann für viele NP–vollständige Probleme bewiesen werden, dass es unmöglich ist, eine akzeptable Näherungslösung zu finden — es sei denn, es gilt $P = NP$.

Literatur

— Dewdney A. K.: *The (new) Turing Omnibus*. New York: Henry Holt and Company 2001

— Gasper F., Leiß I., Spengler M., Stimm H.: *Technische und theoretische Informatik*. München: Bayerischer Schulbuch-Verlag 1992

— Goldschlager L., Lister A.: *Informatik: eine moderne Einführung*. München, Wien: Carl Hanser Verlag 1990

— Gumm H.-P., Sommer M.: *Einführung in die Informatik*. München: Oldenbourg Wissenschaftsverlag 2002

— Hartmann W., Näf M., Reichert R.: *Informatikunterricht planen und durchführen*. Berlin, Heidelberg: Springer–Verlag 2006

— Hofmeister Th.: *Randomisierte Algorithmen*. Vorlesungsskript, Universität Dortmund 2006

— Knuth D. E.: *The Art of Computer Programming, Vol. 3 / Sorting And Searching*. Reading (Mass.): Addison-Wesley Publishing Company 1973

— Mayr U.: *Komplexität / NP–vollständige Probleme*. Skript zum SIL–Kurs Informatik VI, ohne Datum

— Rechenberg P.: *Was ist Informatik?* München, Wien: Carl Hanser Verlag 2000

— Socher R.: *Theoretische Grundlagen der Informatik*. München, Wien: Fachbuchverlag Leipzig im Carl Hanser Verlag 2003