Ist P = NP?
Einführung in die Theorie der NP-Vollständigkeit

Steffen Reith und Heribert Vollmer

Bericht Nr. 269 Februar 2001

Preprint-Reihe
Institut für Informatik
Universität Würzburg

Ist $\mathbf{P} = \mathbf{NP}$?
Einführung in die Theorie der NP-Vollständigkeit

Steffen Reith und Heribert Vollmer
*Theoretische Informatik, Universität Würzburg**

Einführung

Für viele ständig auftretenden Berechnungsprobleme, wie Sortieren, die arithmetischen Operationen (Addition, Multiplikation, Division), Fourier-Transformation etc., sind sehr effiziente Algorithmen konstruiert worden. Für wenige andere praktische Probleme weiß man, dass sie nicht oder nicht effizient algorithmisch lösbar sind. Im Gegensatz dazu ist für einen sehr großen Teil von Fragestellungen aus den verschiedensten Anwendungsbereichen (Operations Research, Netzwerkdesign, Programmoptimierung, Datenbanken, Betriebssystem-Entwicklung, usw.; siehe Kasten „Kleine Sammlung $\mathbf{NP}$-vollständiger Probleme“) jedoch nicht bekannt, ob sie effiziente Algorithmen besitzen. Diese Problematik hängt mit der seit über dreißig Jahren offenen $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage zusammen, wahrscheinlich gegenwärtig das wichtigste ungelöste Problem der theoretischen Informatik. Es wurde sogar kürzlich auf Platz 1 der Liste der sog. *Millenium Prize Problems* des Clay Mathematics Institute gesetzt [3]. Diese Liste umfasst sieben offene Probleme aus der gesamten Mathematik. Das Clay Institute zahlt jedem, der eines der Probleme löst, eine Million US-Dollar.

Aber auch außerhalb der theoretischen Informatik übt das $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problem eine große Faszination aus. Die kürzlich von Anatoly Plotnikov veröffentlichte Arbeit [11], in der $\mathbf{P} = \mathbf{NP}$ behauptet wurde, hat für einige Furore gesorgt, auch in populärwissenschaftlichen Publikationen [8, 7], im Usenet (in comp.theory), auf Slashdot (http://www.slashdot.org), in der Yahoo List „Theory-edge“ (http://groups.yahoo.com/group/theory-edge), etc. Noch nachdem Plotnikov seine Aussage zurückziehen musste, wurden verwandte theoretische Fragen mit ungewohnt breitem Interesse aufgenommen [15, 14].

In diesem Artikel werden diese neuen Entwicklungen diskutiert. Vor allem sollen aber die wesentlichen Begriffe aus dem Kontext des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems und des Begriffes der $\mathbf{NP}$-Vollständigkeit erläutert werden, um so die Grundlagen für das Verständnis derartiger Entwicklungen zu schaffen und deren Beurteilung zu ermöglichen.

Effizient lösbare Probleme: die Klasse P

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

*Lehrstuhl für Theoretische Informatik, Universität Würzburg, Am Hubland, 97074 Würzburg. e-mail: [streit, vollmer]@informatik.uni-wuerzburg.de. WWW: http://www-info4.informatik.uni-wuerzburg.de/∼[streit, vollmer].

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

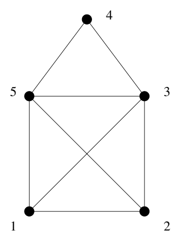
PROBLEM:	PARITY
INSTANZ:	Positive Integerzahl x
FRAGE:	Ist die Anzahl der Ziffern 1 in der Binärdarstellung von x ungerade?

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

PROBLEM:	MULT_D
INSTANZ:	Integerzahlen x, y, positive Integerzahl i
FRAGE:	Ist das i-te Bit in $x \cdot y$ gleich 1?

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

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



Der Nikolausgraph G_N

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

PROBLEM:	k-COL
INSTANZ:	Ein Graph $G = (V, E)$
FRAGE:	Hat G eine Knotenfärbung mit höchstens k Farben?

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

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

Als nächstes ist zu klären, was unter einem „Algorithmus“ zu verstehen ist. Ein Algorithmus ist eine endliche, formale Beschreibung einer Methode, die ein Problem löst (z.B. ein Programm in einer beliebigen Programmiersprache). Diese Methode muss also alle Eingaben mit der gesuchten Eigenschaft von den Eingaben, die diese Eigenschaft nicht haben, unterscheiden können. Man legt fest, dass der Algorithmus für erstere den Wert 1 und für letztere den Wert 0 ausgeben soll. Wie soll die „Laufzeit“ eines Algorithmus gemessen werden? Um dies festlegen zu können, muss man sich zunächst auf ein sogenanntes *Berechnungsmodell* festlegen. Das kann man damit vergleichen, welche Hardware für die Implementation des Algorithmus verwendet werden soll. Für die weiteren Analysen soll das folgendes einfache PASCAL-artige Modell verwendet werden: Es wird die Syntax von PASCAL verwendet und festgelegt, dass jede Anweisung in einem Schritt abgearbeitet werden kann. Gleichzeitig beschränkt man sich auf zwei Datentypen: einen Integer-Typ und zusätzlich Arrays dieses Integer-Typs (wobei Array-Grenzen nicht deklariert werden müssen, sondern sich aus dem Gebrauch ergeben). Dieses primitive Maschinenmodell ist deshalb geeignet, weil man zeigen kann, dass jeder so formulierte Algorithmus auf realen Computern implementiert werden kann, ohne eine substantielle Verlangsamung zu erfahren. (Dies gilt zumindest, wenn die verwendeten Zahlen nicht übermäßig wachsen, d.h., wenn alle verwendeten Variablen nicht zu viel Speicher belegen; formal: nicht mehr Bits benötigen als durch ein festes Polynom angegeben, angewendet auf die Anzahl der Bits, die zur Speicherung Eingabe notwendig sind).[1] Umgekehrt kann man ebenfalls sagen, dass dieses einfaches Modell die Realität genau genug widerspiegelt, da auch reelle Programme ohne allzu großen Zeitverlust auf diesem Berechnungsmodell simuliert werden können. Offensichtlich ist die *Eingabe* der Parameter, von dem die Rechenzeit für einen festen Algorithmus abhängt. In den vergangenen Jahrzehnten, in denen das Gebiet der Analyse von Algorithmen entstand, hat die Erfahrung gezeigt, dass die Länge der Eingabe, also die Anzahl der Bits, die benötigt werden, um die Eingabe zu speichern, ein geeignetes und robustes Maß ist, in der die Rechenzeit gemessen werden kann. Auch der Aufwand, die Eingabe selbst festzulegen (zu konstruieren), hängt schließlich von ihrer Länge ab, nicht davon, ob sich irgendwo in der Eingabe eine 0 oder 1 befindet. Der Kasten „Das Problem der 2-Färbbarkeit“ erläutert die Aufwandsanalyse eines Algorithmus am Beispiel eines Programms für das Problem 2-COL.

[1] Will man beliebig große Zahlen zulassen (diese könnte z.B. bei extensivem Gebrauch der Multiplikation entstehen), so sollte man für die Abarbeitung einer Anweisung nicht mehr nur einen Schritt berechnen, sondern die Größe der verwendeten Zahlen berücksichtigen. Genaueres zu dieser Problematik – man spricht von der Unterscheidung zwischen *uniformem Komplexitätsmaß* und *Bitkomplexität* – findet sich in [13, S. 36f]. Für den Rest dieses Artikels kann jedoch das erläuterte einfache Modell der Einheitskosten verwendet werden.

Das Problem der 2-Färbbarkeit

PROBLEM:	2-COL
INSTANZ:	Ein Graph $G = (V, E)$
FRAGE:	Hat G eine Knotenfärbung mit höchstens 2 Farben?

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

```
Eingabe: Graph G = ({v_1, ... v_n}, E);
begin
  for i := 1 to n do Farbe[i] := 0;      (* noch alles ungefärbt *)
  Farbe[1] := 1;      (* Knoten v_1 mit Farbe 1 färben *)
  noch_Knoten_untersuchen := 1;      (* Nachbarn von v_1 sind noch zu untersuchen *)
  while noch_Knoten_untersuchen = 1 do begin
    noch_Knoten_untersuchen := 0;
    for i := 1 to n do
      for j := 1 to n do
        if (v_i, v_j) ∈ E and      (* v_i und v_j durch Kante verbunden und ... *)
          Farbe[i] ≠ 0 and Farbe[j] = 0 then begin      (* v_i gefärbt, v_j noch ungefärbt *)
          Farbe[j] := 3 − Farbe[i];      (* v_j entgegengesetzt zu v_i färben *)
          noch_Knoten_untersuchen := 1;      (* Neuer Knoten hinzugekommen, Nachbarn noch untersuchen *)
          for k := 1 to n do
            if (v_j, v_k) ∈ E and Farbe[j] = Farbe[k] then return 0      (* Kollision! G ist nicht 2-färbbar *)
        end (* if *);
    if noch_Knoten_untersuchen = 0 then begin
      weiter_suchen := 1;
      i := 1;
      while weiter_suchen = 1 do begin      (* Suche nach anderen Komponenten von G *)
        if Farbe[i] = 0 then begin      (* v_i liegt in neuer Komponente von G *)
          Farbe[i] := 1;      (* Färbe v_i mit 1; kein Nachbar von v_i ist bisher gefärbt *)
          noch_Knoten_untersuchen := 1;      (* Komponente von v_i im nächsten while-Durchlauf bearbeiten *)
          weiter_suchen := 0      (* Komponentensuche abbrechen *)
        end (* if *);
        i := i + 1;
        if i > n then weiter_suchen := 0      (* keine weiteren Komponenten vorhanden; Suche abbrechen *)
      end (* while *)
    end (* if *)
  end (* while *);
  return 1      (* Alles gefärbt, keine Kollision dabei aufgetreten *)
end.
```

Zur Laufzeit des Algorithmus: Die erste **for**-Schleife benötigt n Schritte. In der **while**-Schleife wird entweder mindestens ein Knoten gefärbt und die Schleife dann erneut ausgeführt, oder es wird kein Knoten gefärbt und die Schleife dann verlassen; also wird diese Schleife höchstens n-mal ausgeführt. Innerhalb der **while**-Schleife finden sich drei ineinander verschachtelte **for**-Schleifen, die alle jeweils n-mal durchlaufen werden, und eine **while**-Schleife, die maximal n-mal durchlaufen wird.
Damit ergibt sich also eine Gesamtlaufzeit der Größenordnung n^4, wobei n die Anzahl der Knoten des Eingabe-Graphen G ist. Wie groß ist nun die Eingabelänge, also die Anzahl der benötigten Bits zur Speicherung von G? Sicherlich muss jeder Knoten in dieser Speicherung vertreten sein, d.h. also, dass mindestens n Bits zur Speicherung von G benötigt werden. Die Eingabelänge ist also mindestens n. Daraus folgt, dass die Laufzeit des Algorithmus also höchstens von der Größenordnung N^4 ist, wenn N die Eingabelänge bezeichnet.
Tatsächlich sind (bei Verwendung geeigneter Datenstrukturen wie Listen oder Queues) wesentlich effizientere Verfahren für 2-COL möglich. Aber auch schon das obige einfache Verfahren zeigt:
2-COL hat einen Polynomialzeitalgorithmus, also 2-COL $\in \mathbf{P}$.

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

Rechenzeitbedarf von Algorithmen auf einem „1-MIPS“-Rechner

Alle Probleme, für die Algorithmen existieren, die eine Anzahl von Rechenschritten benötigen, die durch ein beliebiges Polynom beschränkt ist, bezeichnet man mit $\mathbf{P}$ („**P**“ steht dabei für „**P**olynomialzeit“). Auch dabei wird die Rechenzeit in der Länge der Eingabe gemessen, d.h. in der Anzahl der Bits, die benötigt werden, um die Eingabe zu speichern (zu kodieren). Die Klasse $\mathbf{P}$ wird auch als Klasse der *effizient lösbaren Probleme* bezeichnet. Dies ist natürlich wieder eine idealisierte Auffassung: Einen Algorithmus mit einer Laufzeit n^{57}, wobei n die Länge der Eingabe bezeichnet, kann man schwer als effizient bezeichnen. Allerdings hat es sich in der Praxis gezeigt, dass für fast alle bekannten Probleme in $\mathbf{P}$ auch Algorithmen existieren, deren Laufzeit durch ein Polynom kleinen Grades beschränkt ist. (Uns ist kein praktisch relevantes Problem aus $\mathbf{P}$ bekannt, für das es keinen Algorithmus mit einer Laufzeit von weniger als n^8 gibt.) In diesem Licht ist die Definition der Klasse $\mathbf{P}$ auch für praktische Belange von Relevanz. Dass eine polynomielle Laufzeit etwas substanziell Besseres darstellt als exponentielle Laufzeit (hier beträgt die benötigte Rechenzeit $2^{c \cdot n}$ für eine Konstante c, wobei n wieder die Länge der Eingabe bezeichnet), zeigt die Tabelle „Rechenzeitbedarf von Algorithmen“. Zu beachten ist, dass bei einem Exponentialzeit-Algorithmus eine Verdoppelung der „Geschwindigkeit“ der verwendeten Maschine (also Taktzahl pro Sekunde) es nur erlaubt, eine um höchstens 1 Bit längere Eingabe in einer bestimmten Zeit zu bearbeiten. Bei einem Linearzeit-Algorithmus hingegen verdoppelt sich auch die mögliche Eingabelänge; bei einer n^k vergrößert sich die mögliche Eingabelänge immerhin noch um den Faktor $\sqrt[k]{2}$. Deswegen sind Probleme, für die nur Exponentialzeit-Algorithmen existieren, praktisch nicht lösbar; daran ändert sich auch nichts Wesentliches durch die Einführung von immer schnelleren Rechnern.

Nun stellt sich natürlich sofort die Frage: Gibt es für jedes Problem einen effizienten Algorithmus? Man kann relativ leicht zeigen, dass die Antwort auf diese Frage „Nein“ ist. Die Schwierigkeit bei dieser Fragestellung liegt aber darin, dass man von vielen Problemen *nicht weiß*, ob sie effizient lösbar sind. Ganz konkret: Ein effizienter Algorithmus für das Problem 2-COL findet sich im Kasten. Ist es möglich, ebenfalls einen Polynomialzeitalgorithmus für 3-COL zu finden? Viele Informatiker beschäftigen sich seit den 60er Jahren des letzten Jahrhunderts intensiv mit dieser Frage. Dabei kristallisierte sich heraus, dass viele praktisch relevante Probleme, für die kein effizienter Algorithmus bekannt ist, eine gemeinsame Eigenschaft besitzen, nämlich die der *effizienten Überprüfbarkeit* von geeigneten Lösungen. Auch 3-COL gehört zu dieser Klasse von Problemen, wie sich in Kürze zeigen wird. Aber wie soll man zeigen, dass für ein Problem kein effizienter Algorithmus existiert? Nur weil kein Algorithmus bekannt ist, bedeutet das noch nicht, dass keiner existiert.

Es ist bekannt, dass die oberen Schranken (also die Laufzeit von bekannten Algorithmen) und die unteren Schranken (mindestens benötigte Laufzeit) für das Problem PARITY sehr nahe zusammen liegen. Das gilt ebenfalls für einige wenige weitere, ebenfalls sehr einfach-geartete Probleme; z.B. die arithmetischen Operationen[2]. Das bedeutet also, dass nur noch unwesentliche Verbesserungen der Algorithmen für des PARITY-Problem erwartet werden können. Beim Problem 3-COL ist das ganz anderes: Die bekannten unteren und oberen Schranken liegen extrem weit auseinander. Deshalb ist nicht klar, ob nicht doch (extrem) bessere Algorithmen als heute bekannt im Bereich des Möglichen liegen. Aber wie untersucht man solch eine Problematik? Man müsste ja über unendlich viele Algorithmen für das Problem 3-COL Untersuchungen anstellen. Dies ist äußerst schwer zu handhaben und deshalb ist der einzige bekannte Ausweg, das Problem mit einer Reihe von weiteren (aus bestimmten Gründen) interessierenden Problemen zu vergleichen und zu zeigen, dass unser zu untersuchendes Problem nicht leichter zu lösen ist als diese anderen. Hat man das geschafft, ist eine untere Schranke einer speziellen Art gefunden: Unser Problem ist nicht leichter lösbar, als alle Probleme der Klasse von Problemen, die für den Vergleich herangezogen wurden. Nun ist aus der Beschreibung der Aufgabe aber schon klar, dass auch diese Aufgabe schwierig zu lösen ist, weil ein Problem nun mit unendlich vielen anderen Problemen zu vergleichen ist. Es zeigt sich aber, dass diese Aufgabe nicht aussichtslos ist. Bevor diese Idee weiter ausgeführt wird, soll zunächst die Klasse von Problemen untersucht werden, die für diesen Vergleich herangezogen werden sollen, nämlich die Klasse $\mathbf{NP}$.

Effizient überprüfbare Probleme: die Klasse NP

Wie schon erwähnt, gibt es eine große Anzahl verschiedener Probleme, für die kein effizienter Algorithmus bekannt ist, die aber eine gemeinsame Eigenschaft haben: die *effiziente Überprüfbarkeit von Lösungen* für dieses Problem.

Diese Eigenschaft soll am schon bekannten Problem 3-COL veranschaulicht werden: Angenommen, man hat einen beliebigen Graphen G gegeben; wie bereits erwähnt ist kein effizienter Algorithmus bekannt, der entscheiden kann, ob der Graph G eine 3-Färbung hat (d.h., ob der fiktive Mobilfunkprovider mit 3 Funkfrequenzen auskommt). Hat man aber aus irgendwelchen Gründen eine *potenzielle* Knotenfärbung vorliegen, dann ist es leicht, diese potenzielle Knotenfärbung zu überprüfen und festzustellen, ob sie eine *korrekte* Färbung des Graphen ist, wie folgender Algorithmus zeigt:

Eingabe: Graph $G = (\{v_1, \dots v_n\}, E)$ und eine potenzielle Knotenfärbung;
begin
  for i := 1 **to** n **do**
    for j := 1 **to** n **do**
      if $(v_i, v_j) \in E$ **and** i und j gleich gefärbt **then return** 0;
  return 1
end.

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

Sei wieder ein Graph $G = (\{v_1, \dots v_n\}, E)$ gegeben. Diesmal ist eine Rundreise entlang der Kanten von G gesucht, die bei einem Knoten v_{i_1} aus G startet, wieder bei v_{i_1} endet und jeden Knoten genau einmal besucht. Genauer wird diese Rundreise im Graphen G durch eine Folge von n Knoten $(v_{i_1}, v_{i_2}, v_{i_3}, \dots, v_{i_{n-1}}, v_{i_n},)$ beschrieben, wobei gelten soll, dass alle Knoten $v_{i_1}, \dots, v_{i_n}$ verschieden sind und die Kanten $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n})$ und (v_{i_n}, v_{i_1}) in G vorkommen. Eine solche

[2] Auch für Division ist seit Neuestem, nachdem das Problem 15 Jahre lang offen war, ein Zusammenfallen der oberen und unteren Schranken bekannt.

Folge von Kanten wird als *Hamiltonscher Kreis* bezeichnet. Ein Hamiltonscher Kreis in einem Graphen G ist also ein Kreis, der jeden Knoten des Graphen genau einmal besucht. Das Problem, einen Hamiltonschen Kreis in einem Graphen zu finden, bezeichnet man mit HAMILTON:

PROBLEM:	HAMILTON
INSTANZ:	Ein Graph G
FRAGE:	Hat G einen Hamiltonschen Kreis?

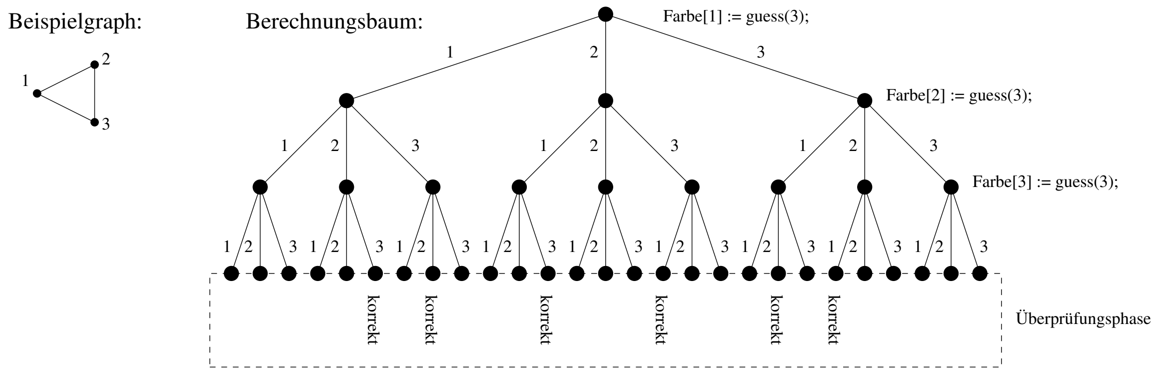
Auch für dieses Problem ist kein effizienter Algorithmus bekannt. Aber auch hier ist offensichtlich: Bekommt man einen Graphen gegeben und eine Folge von Knoten, dann kann man sehr leicht überprüfen, ob sie ein Hamiltonscher Kreis ist – dazu ist lediglich zu testen, ob alle Knoten genau einmal besucht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

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

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

Hätte man einen Parallelrechner zur Verfügung mit so vielen Prozessoren wie es potenzielle Lösungen gibt, dann könnte man das Problem schnell lösen, denn jeder Prozessor kann unabhängig von allen anderen Prozessoren eine potenzielle Färbung überprüfen. Es zeigt sich aber, dass auch dieses Berechnungsmodell zu mächtig wäre. Es gibt Probleme, die wahrscheinlich nicht im obigen Sinne effizient überprüfbar sind, aber mit solch einem Parallelrechner trotzdem (effizient) gelöst werden könnten. In der Praxis würde uns ein derartiger paralleler Algorithmus auch nichts nützen, da man einen Rechner mit exponentiell vielen Prozessoren, also enormem Hardwareaufwand, zu konstruieren hätten. Also muss auch dieses Berechnungsmodell wieder etwas schwächer gemacht werden.

Eine Abschwächung der gerade untersuchten Idee des Parallelrechners führt zu folgendem Vorgehen: Man „rät" für den ersten Knoten eine beliebige Farbe, dann für den zweiten Knoten auch wieder eine beliebige Farbe, solange bis für den letzten Knoten eine Farbe gewählt wurde. Danach überprüft man die geratene Färbung und akzeptiert die Eingabe, wenn die geratene Färbung eine korrekte Knotenfärbung ist. Die Eingabe ist eine positive Eingabeinstanz des $\mathbf{NP}$-Problems 3-COL, falls es eine potenzielle Lösung (Färbung) gibt, die sich bei der Überprüfung als korrekt herausstellt, d.h. im beschriebenen Rechnermodell: falls es eine Möglichkeit zu raten gibt, sodass am Ende akzeptiert (Wert 1 ausgegeben) wird. Man kann also die Berechnung durch einen Baum mit 3-fachen Verzweigungen



Ein Berechnungsbaum für das 3-COL-Problem

darstellen, vgl. Abbildung „Ein Berechnungsbaum für das 3-COL-Problem“. An den Kanten des Baumes findet sich das Resultat der Rateanweisung der darüberliegenden Verzweigung. Jeder Pfad in diesem sogenannten *Berechnungsbaum* entspricht daher einer Folge von Farbzuordnungen an die Knoten, d.h. einer potenziellen Färbung. Der Graph ist 3-färbbar, falls sich auf mindestens einem Pfad eine korrekte Färbung ergibt, falls also auf mindestens einem Pfad die Überprüfungsphase erfolgreich ist; der Beispielgraph besitzt 6 korrekte 3-Färbungen, ist also eine positive Instanz des 3-COL-Problems.

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

Es hat sich also gezeigt, dass eine Art „Rateanweisung“ benötigt wird. In der Programmiersprache für $\mathbf{NP}$ verwendet man dazu das neue Schlüsselwort $\mathbf{guess}(m)$, wobei m die Anzahl von Möglichkeiten ist, aus denen eine geraten wird, und legt fest, dass auch die Anweisung $\mathbf{guess}(m)$ nur einen Takt Zeit für ihre Abarbeitung benötigt. Berechnungen, die, wie soeben beschrieben, verschiedene Möglichkeiten raten können, heißen *nichtdeterministisch*. Es sei wiederholt, dass *festgelegt* (definiert) wird, dass ein nichtdeterministischer Algorithmus bei einer Eingabe den Wert 1 berechnet, falls es *eine Möglichkeit* zu raten gibt, sodass der Algorithmus auf die Anweisung „**return** 1“ stösst. Die Klasse $\mathbf{NP}$ umfasst nun genau die Probleme, die von nichtdeterministischen Algorithmen mit polynomieller Laufzeit gelöst werden können. „**NP**“ steht dabei für „**n**ichtdeterministische **P**olynomialzeit“, nicht etwa, wie mitunter zu lesen, für „Nicht-Polynomialzeit“. (Eine formale Präsentation der Äquivalenz zwischen effizienter Überprüfbarkeit und Polynomialzeit in der $\mathbf{NP}$-Programmiersprache findet sich z.B. in [4, Kapitel 2.3].)

Folgender nichtdeterministischer Algorithmus löst das 3-COL-Problem. Die zweite Phase des Algorithmus, die *Überprüfungsphase*, entspricht übrigens genau dem oben angegebenen Algorithmus zum effizienten Überprüfen von möglichen Lösungen des 3-COL-Problems.

Eingabe: Graph $G = (\{v_1, \dots v_n\}, E)$;
begin
  $(*$ Ratephase: $*)$
  for i := 1 **to** n **do**
    Farbe[i] := **guess**(3);
  $(*$ Überprüfungsphase $*)$
  for i := 1 **to** n **do**
    for j := 1 **to** n **do**

    if $(v_i, v_j) \in E$ **and** Farbe[i] = Farbe[j] **then return** 0;
  return 1
end.

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

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

Schwierigste Probleme in NP: der Begriff der NP-Vollständigkeit

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

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

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

Eine zweite, erfolgversprechendere Idee ist die folgende: Ein Problem A ist nicht (wesentlich) schwieriger als ein Problem B, wenn man A mit der Hilfe von B (als Unterprogramm) effizient lösen kann. Ein einfaches Beispiel ist die Multiplikation von n Zahlen. Angenommen, man hat schon ein Programm, dass 2 Zahlen multiplizieren kann; dann ist es nicht wesentlich schwieriger, auch n Zahlen zu multiplizieren, wenn die Routine für die Multiplikation von 2 Zahlen verwendet wird. Dieser Ansatz ist unter dem Namen *relative Berechenbarkeit* bekannt, der genau den oben beschriebenen Sachverhalt widerspiegelt: Multiplikation von n Zahlen (sog. *iterierte Multiplikation*) ist relativ zur Multiplikation zweier Zahlen (leicht) berechenbar.

Da das Prinzip der relativen Berechenbarkeit so allgemein gehalten ist, gibt es innerhalb der theoretischen Informatik sehr viele verschiedene Ausprägungen dieses Konzepts. Für die $\mathbf{P}$-$\mathbf{NP}$-Problematik ist folgende Version der relativen Berechenbarkeit (d.h. folgende erlaubte Art von „Unterprogrammaufrufen“) geeignet:

Seien zwei Probleme A und B gegeben. Das Problem A ist nicht schwerer als B, falls es eine effizient zu berechnende Transformation T gibt, die Folgendes leistet: Wenn x eine Eingabeinstanz von Problem A ist, dann ist $T(x)$ eine Eingabeinstanz für B. Weiterhin gilt: x ist *genau dann* eine positive Instanz von A (d.h. ein Entscheidungsalgorithmus für A muss den Wert 1 für Eingabe x liefern), wenn $T(x)$ eine positive Instanz von Problem B ist. Erneut soll „effizient berechenbar“ hier bedeuten: in Polynomialzeit berechenbar. Es muss also einen Polynomialzeitalgorithmus geben, der die Transformation T ausführt. Das Entscheidungsproblem A ist damit effizient transformierbar in das Problem B. Man sagt auch: A ist reduzierbar auf B; oder intuitiver: A ist nicht schwieriger als B, oder B ist mindestens so schwierig wie A. Formal schreibt man: $A \leq B$.

Um für dieses Konzept ein wenig mehr Intuition zu gewinnen, sei erwähnt, dass man sich eine solche Transformation auch wie folgt vorstellen kann: A lässt sich auf B reduzieren, wenn ein Algorithmus für A angegeben werden kann, der ein Unterprogramm U_B für B verwendet und genau die folgende Form hat:

Eingabe: Instanz x von Problem A;
begin
  berechne $y := T(x)$; $(* \, y$ ist Instanz von Problem $B \, *)$
  $z := U_B(y)$;
  return z
end.

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

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

(1) C ist ein Problem aus $\mathbf{NP}$.

(2) C ist mindestens so schwierig wie jedes andere $\mathbf{NP}$-Problem A; d.h.: für alle Probleme A aus $\mathbf{NP}$ gilt: $A \leq C$.

Solche schwierigsten Probleme in $\mathbf{NP}$ sind unter der Bezeichnung $\mathbf{NP}$-*vollständige Probleme* bekannt. Nun sieht die Aufgabe, von einem Problem zu zeigen, dass es $\mathbf{NP}$-vollständig ist, ziemlich hoffnungslos aus. Immerhin ist zu zeigen, dass für alle Probleme aus $\mathbf{NP}$ – und damit unendlich viele

– gilt, dass sie höchstens so schwer sind wie das zu untersuchende Problem, und damit scheint man der Schwierigkeit beim Nachweis unterer Schranken nicht entgangen zu sein. Dennoch konnten der russische Mathematiker Leonid Levin und der amerikanische Mathematiker Stephan Cook Anfang der siebziger Jahre des letzten Jahrhunderts unabhängig voneinander die Existenz von solchen $\mathbf{NP}$-vollständigen Probleme zeigen. Hat man nun erst einmal *ein* solches Problem identifiziert, ist die Aufgabe, *weitere* $\mathbf{NP}$-vollständige Probleme zu finden, wesentlich leichter. Dies ist sehr leicht einzusehen: Ein $\mathbf{NP}$-Problem C ist ein schwierigstes Problem in $\mathbf{NP}$, wenn es ein anderes schwierigstes Problem B gibt, sodass C nicht leichter als B ist. Das führt zu folgendem „Kochrezept":

Nachweis der NP-Vollständigkeit eines Problems C:

(1) Zeige, dass C in $\mathbf{NP}$ enthalten ist, indem dafür ein geeigneter nichtdeterministischer Polynomialzeitalgorithmus konstruiert wird.

(2) Suche ein geeignetes „ähnliches" schwierigstes Problem B in $\mathbf{NP}$ und zeige, dass C nicht leichter als B ist; formal: Finde ein $\mathbf{NP}$-vollständiges Problem B und zeige $B \leq C$ mit Hilfe einer geeigneten Transformation T.

Den zweiten Schritt kann man oft relativ leicht mit Hilfe von bekannten Sammlungen $\mathbf{NP}$-vollständiger Problemen erledigen. Das Buch von Garey und Johnson [4] ist eine solche Sammlung, die mehr als 300 $\mathbf{NP}$-vollständige Probleme enthält. Ein Beispiel für das oben beschriebene Vorgehen findet sich im Kasten „Traveling Salesperson ist $\mathbf{NP}$-vollständig".

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

Eingabe: Instanz x für Problem A;
begin
  $y := T(x)$;
  $z := \mathcal{A}_C(y)$;
  return z
end.

Es gilt also: Ist irgendein $\mathbf{NP}$-vollständiges Problem effizient lösbar, dann ist $\mathbf{P} = \mathbf{NP}$.

Auch die Arbeit des ukrainischen Mathematikers Anatoly D. Plotnikov, in der zunächst $\mathbf{P} = \mathbf{NP}$ behauptet wurde [11], ist nach diesem Prinzip aufgebaut: Er definierte das sog. *Minimum Clique Partition Problem*, ein $\mathbf{NP}$-vollständiges Problem, und versuchte zu zeigen, dass es einen Polynomialzeit-Algorithmus besitzt. Dazu bediente er sich fortgeschrittener graphentheoretischer Konzepte und entwickelte eine Reihe bemerkenswerter algorithmischer Tricks, aber es zeigte sich, dass sein Algorithmus das Problem nicht in seiner ganzen Allgemeinheit löst, sondern bei einigen Eingabeinstanzen (die eben die Schwierigkeit des Minimum Clique Partition Problem ausmachen) versagt (vgl. auch[2]).

Sei nun angenommen, dass jemand $\mathbf{P} \neq \mathbf{NP}$ gezeigt hat. In diesem Fall ist aber auch klar, dass dann für kein $\mathbf{NP}$-vollständiges Problem ein Polynomialzeitalgorithmus existieren kann, denn sonst würde sich ja der Widerspruch $\mathbf{P} = \mathbf{NP}$ ergeben. Ist das Problem C also $\mathbf{NP}$-vollständig, so gilt: C hat genau dann einen effizienten Algorithmus, wenn $\mathbf{P} = \mathbf{NP}$, also wenn jedes Problem in $\mathbf{NP}$ einen effizienten Algorithmus besitzt. Diese Eigenschaft macht die $\mathbf{NP}$-vollständigen Probleme für die Theoretiker so interessant, denn eine Klasse von unendlich vielen Problemen kann untersucht werden, indem man

Traveling Salesperson ist NP-vollständig

Wie kann man zeigen, dass Traveling Salesperson $\mathbf{NP}$-vollständig ist? Dazu zuerst die Definition dieses Problems:

PROBLEM:	TRAVELING SALESPERSON (TSP)
INSTANZ:	Eine Menge von Städten $C = \{c_1, \ldots, c_n\}$ und eine $n \times n$ Entfernungsmatrix D, wobei das Element $D[i,j]$ der Matrix D die Entfernung zwischen Stadt c_i und c_j angibt. Weiterhin eine Obergrenze $k \geq 0$ für die maximal erlaubte Länge der Tour
FRAGE:	Gibt es eine Rundreise, die einerseits alle Städte besucht, aber andererseits eine Gesamtlänge von höchstens k hat?

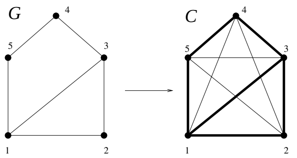
Nun zum ersten Schritt des Nachweises der $\mathbf{NP}$-Vollständigkeit von TSP: Offensichtlich gehört auch das Traveling Salesperson Problem zur Klasse $\mathbf{NP}$, denn man kann nichtdeterministisch eine Folge von n Städten raten (eine potenzielle Rundreise) und dann leicht überprüfen, ob diese potenzielle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

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

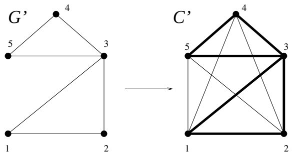
Folgende Idee führt zum Ziel: Gegeben ist eine Instanz $G = (V, E)$ von HAMILTON. Transformiere G in folgende Instanz von TSP: Als Städtemenge C wählen wir die Knoten V des Graphen G. Die Entfernungen zwischen den Städten sind definiert wie folgt: $D[i,j] = 1$, falls es in E eine Kante von Knoten i zu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

Zwei Beispiele für die Wirkungsweise der Transformation:

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



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



Folgender Polynomialzeit-Algorithmus führt die Transformation aus:

Eingabe: Graph $G = (V, E)$, wobei $V = \{1, \ldots n\}$;
begin
  $C := V$; (∗ Die Knoten entsprechen den Städten ∗)
  for $i := 1$ **to** n **do** (∗ Baue die Entfernungsmatrix D ∗)
    for $j := 1$ **to** n **do**
      if $(i,j) \in E$ **then** $D[i,j] := 1$ (∗ Kante in $G \Rightarrow$ Entfernung = 1 ∗)
      else $D[i,j] := n+1$; (∗ keine Kante in $G \Rightarrow$ Entfernung = Anzahl Städte $+ 1$ ∗)
  $k := n$; (∗ Gesamtlänge k der Rundreise ist Anzahl der Städte ∗)
  return (C, D, k)
end.

nur ein einziges Problem betrachtet. Man kann sich das auch so vorstellen: Alle relevanten Eigenschaften aller Probleme aus $\mathbf{NP}$ wurden in ein einziges Problem „destilliert". Die $\mathbf{NP}$-vollständigen Probleme sind also in diesem Sinn *prototypische* $\mathbf{NP}$-Probleme.

Trotz intensiven Bemühungen in den letzten 30 Jahren konnte bisher niemand einen Polynomialzeitalgorithmus für ein $\mathbf{NP}$-vollständiges Problem finden; deshalb geht man heute davon aus, dass $\mathbf{P} \neq \mathbf{NP}$ gilt. Leider konnte auch dies bisher nicht gezeigt werden, aber in der theoretischen Informatik gibt es starke Indizien für die Richtigkeit dieser Annahme, sodass heute die große Mehrheit der Forscher von $\mathbf{P} \neq \mathbf{NP}$ ausgeht.

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

Nun ist auch eine Antwort für das 3-COL-Problem gefunden. Es wurde gezeigt [4], dass k-COL für $k \geq 3$ $\mathbf{NP}$-vollständig ist. Der fiktive Mobilfunkplaner hat also Pech gehabt: Es ist unwahrscheinlich, dass er jemals ein korrektes effizientes Planungsverfahren finden wird.

Ein $\mathbf{NP}$-Vollständigkeitsnachweis eines Problems ist also ein starkes Indiz für seine praktische Nicht-Handhabbarkeit. Auch die $\mathbf{NP}$-Vollständigkeit eines Problems, das mit dem Spiel *Minesweeper* zu tun hat [6, 14, 15], bedeutet demnach lediglich, dass dieses Problem höchstwahrscheinlich nicht effizient lösbar sein wird. Ein solcher Vollständigkeitsbeweis hat nichts mit einem Schritt in Richtung auf eine Lösung des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems zu tun, wie irreführenderweise gelegentlich zu lesen war. Übrigens ist auch für eine Reihe weiterer Spiele ihre $\mathbf{NP}$-Vollständigkeit bekannt. Dazu gehören u.a. bestimmte Puzzle- und Kreuzwortspiele. Typische Brettspiele, wie Dame, Schach oder GO, sind hingegen (verallgemeinert auf Spielbretter der Größe $n \times n$) **PSPACE**-vollständig. Die Klasse **PSPACE** ist eine noch deutlich mächtigere Klasse als $\mathbf{NP}$. Damit sind also diese Spiele noch viel komplexer als Minesweeper und andere $\mathbf{NP}$-vollständige Probleme.

Wie geht man in der Praxis mit NP-vollständigen Problemen um?

Viele in der Praxis bedeutsame Probleme sind $\mathbf{NP}$-vollständig (siehe Kasten „Kleine Sammlung $\mathbf{NP}$-vollständiger Probleme"). Ein Anwendungsentwickler wird es aber sicher schwer haben, seinem Management mitteilen zu müssen, dass ein aktuelles Projekt nicht durchgeführt werden kann, weil keine geeigneten Algorithmen zur Verfügung stehen. (Wahrscheinlich würden in diesem Fall einfach „geeignetere" Entwickler eingestellt werden.) Es stellt sich daher also die Frage, wie man mit solchen $\mathbf{NP}$-vollständigen Problemen in der Praxis umgeht. Zu dieser Fragestellung hat die theoretische Informatik ein ausgefeiltes Instrumentarium entwickelt (siehe hierzu auch [16]).

Eine erste Idee wäre es, sich mit Algorithmen zufrieden zu geben, die mit Zufallszahlen arbeiten und nur mit sehr großer Wahrscheinlichkeit die richtige Lösung berechnen, aber sich auch mit kleiner (vernachlässigbarer) Wahrscheinlichkeit irren dürfen. Solche Algorithmen sind als *probabilistische* oder *randomisierte Algorithmen* bekannt [9] und werden beispielsweise in der Kryptographie mit sehr großem Erfolg angewendet. Das prominenteste Beispiel hierfür sind Algorithmen, die testen, ob eine gegebene Zahl eine Primzahl ist und sich dabei fast nie irren. Primzahlen spielen bekanntermaßen im RSA-Verfahren und damit bei PGP und ähnlichen Verschlüsselungen eine zentrale Rolle. Es konnte aber gezeigt werden, dass probabilistische Algorithmen uns bei den $\mathbf{NP}$-vollständigen Problemen wohl nicht weiterhelfen. So weiß man heute, dass die Klasse der Probleme, die sich mit probabilistischen Algorithmen lösen lässt, höchstwahrscheinlich nicht die Klasse $\mathbf{NP}$ umfasst. Deshalb liegen (höchstwahrscheinlich) insbesondere alle $\mathbf{NP}$-vollständigen Probleme außerhalb der Möglichkeiten von effizienten probabilistischen Algorithmen.

Kleine Sammlung NP-vollständiger Probleme

(Referenzen und Problemnummern in „[...]“ beziehen sich auf die Sammlung von Garey und Johnson [4].)

PROBLEM: CLUSTER
INSTANZ: Netzwerk $G = (V, E)$, positive Integerzahl K
FRAGE: Gibt es eine Menge von mindestens K Knoten, die paarweise miteinander verbunden sind?
REFERENZ: Clique [GT19]

PROBLEM: NETZWERK-AUFTEILUNG
INSTANZ: Netzwerk $G = (V, E)$, Kapazität für jede Kante in E, positive Integerzahl K
FRAGE: Kann man das Netzwerk so in zwei Teile zerlegen, dass die Gesamtkapazität aller Verbindungen zwischen den beiden Teilen mindestens K beträgt?
REFERENZ: Max Cut [ND16]

PROBLEM: NETZWERK-REDUNDANZ
INSTANZ: Netzwerk $G = (V, E)$, Kosten für Verbindungen zwischen je zwei Knoten aus V, Budget B
FRAGE: Kann G so um Verbindungen erweitert werden, dass zwischen je zwei Knoten mindestens zwei Pfade existieren und die Gesamtkosten für die Erweiterung höchstens B betragen?
REFERENZ: Biconnectivity Augmentation [ND18]

PROBLEM: OBJEKTE IN KACHELN SPEICHERN
INSTANZ: Menge U von Objekten mit Speicherbedarf $s(u)$ für jedes $u \in U$; Kachelgröße S, positive Integerzahl K
FRAGE: Können die Objekte in U auf K Kacheln verteilt werden?
REFERENZ: Bin Packing [SR1]

PROBLEM: DATENKOMPRESSION
INSTANZ: endliche Menge R von Strings über festgelegtem Alphabet, positive Integerzahl K
FRAGE: Gibt es einen String S der Länge höchstens K, sodass jeder String aus R als Teilfolge von S vorkommt?
REFERENZ: Shortest Common Supersequence [SR8]

PROBLEM: KLEINSTER SCHLÜSSEL IN DATENBANKEN
INSTANZ: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, positive Integerzahl K
FRAGE: Gibt es einen Schlüssel mit höchstens K Attributen?
REFERENZ: Minimum Cardinality Key [SR26]

PROBLEM: VERLETZUNG DER BCNF
INSTANZ: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, Teilmenge $A' \subseteq A$
FRAGE: Verletzt A' die Boyce-Codd-Normalform?
REFERENZ: Boyce-Codd Normal Form Violation [SR29]

PROBLEM: MEHRPROZESSOR-SCHEDULING
INSTANZ: Menge T von Tasks, Länge für jede Task, Anzahl m von Prozessoren, Positive Integerzahl D („Deadline“)
FRAGE: Gibt es ein m-Prozessor-Schedule für T mit Ausführungszeit höchstens D?
REFERENZ: Multiprocessor Scheduling [SS8]

PROBLEM: PREEMPTIVES SCHEDULING
INSTANZ: Menge T von Tasks, Länge für jede Task, Präzedenzrelation auf den Tasks, Anzahl m von Prozessoren, Positive Integerzahl D („Deadline“)
FRAGE: Gibt es ein m-Prozessor-Schedule für T, das die Präzedenzrelationen berücksichtigt und Ausführungszeit höchstens D hat?
REFERENZ: Preemptive Scheduling [SS12]

PROBLEM: DEADLOCK-VERMEIDUNG
INSTANZ: Menge von Prozessen, Menge von Ressourcen, aktuelle Zustände der Prozesse und aktuell allokierte Ressourcen
FRAGE: Gibt es einen Kontrollfluss, der zum Deadlock führt?
REFERENZ: Deadlock Avoidance [SS22]

PROBLEM: SCHLEIFENVARIABLEN
INSTANZ: Menge V von Variablen, die in einer Schleife benutzt werden, für jede Variable einen Gültigkeitsbereich, positive Integerzahl K
FRAGE: Können die Schleifenvariablen mit höchstens K Registern gespeichert werden?
REFERENZ: Register Sufficiency for Loops [PO3]

PROBLEM: FORMALE REKURSION
INSTANZ: Menge A von Prozedur-Identifiern, Pascal-Programmfragment mit Deklarationen und Aufrufen der Prozeduren aus A
FRAGE: Ist eine der Prozeduren aus A formal rekursiv?
REFERENZ: Programs with Formally Recursive Procedures [PO20]

Kleine Sammlung NP-vollständiger Probleme (II)

PROBLEM:	LR(K)-GRAMMATIK
INSTANZ:	Kontextfreie Grammatik G, positive Integerzahl K (unär)
FRAGE:	Ist G keine LR(K)-Grammatik?
REFERENZ:	Non-LR(K) Context-free Grammar [AL15]

PROBLEM:	CONSTRAINT SATISFACTION
INSTANZ:	Menge von Booleschen Constraints, positive Integerzahl K
FRAGE:	Können mindestens K der Constraints gleichzeitig erfüllt werden?
REFERENZ:	Maximal 2-Satisfiability [LO5]

PROBLEM:	INTEGER PROGRAMMING
INSTANZ:	Lineares Programm
FRAGE:	Hat das Programm eine Lösung, die nur ganzzahlige Werte enthält?
REFERENZ:	Integer Programming [MP1]

PROBLEM:	KREUZWORTRÄTSEL-ENTWURF
INSTANZ:	Menge W von Wörtern, Gitter mit schwarzen und weißen Feldern
FRAGE:	Können die weißen Felder des Gitters gefüllt werden mit Wörtern aus W?
REFERENZ:	Crossword Puzzle Construction [GP14]

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

Es konnte gezeigt werden, dass DNA-Computer [10] jedes $\mathbf{NP}$-vollständige Problem in Polynomialzeit lösen können. Für diese Berechnungsstärke hat man aber einen Preis zu zahlen: Die Anzahl und damit die Masse der DNA-Moleküle, die für die Berechnung benötigt werden, wächst exponentiell in der Eingabelänge. Das bedeutet, dass schon bei recht kleinen Eingaben mehr Masse für eine Berechnung gebraucht würde, als im ganzen Universum vorhanden ist. Bisher ist kein Verfahren bekannt, wie dieses Masseproblem gelöst werden kann, und es sieht auch nicht so aus, als ob es gelöst werden kann, wenn $\mathbf{P} \neq \mathbf{NP}$ gilt. Dieses Problem erinnert an das oben im Kontext von Parallelrechnern schon erwähnte Phänomen: Mit exponentiell vielen Prozessoren lassen sich $\mathbf{NP}$-vollständige Probleme lösen, aber solche Parallelrechner haben natürlich explodierende Hardware-Kosten.

Der anderer Ausweg könnten Quantencomputer sein [5]. Hier scheint die Situation zunächst günstiger zu sein: Die Fortschritte bei der Quantencomputer-Forschung verlaufen immens schnell, und es besteht die berechtigte Hoffnung, dass Quantencomputer mittelfristig verfügbar sein werden [12]. Aber auch hier sagen theoretische Ergebnisse voraus, dass Quantencomputer (höchstwahrscheinlich) keine $\mathbf{NP}$-vollständigen Probleme lösen können. Trotzdem sind Quantencomputer interessant, denn es ist bekannt, dass wichtige Probleme existieren, für die kein Polynomialzeitalgorithmus bekannt ist und die wahrscheinlich nicht $\mathbf{NP}$-vollständig sind, die aber auf Quantencomputern effizient gelöst werden können. Das prominenteste Beispiel hierfür ist die Aufgabe, eine ganze Zahl in ihre Primfaktoren zu zerlegen.

Die bisher angesprochenen Ideen lassen also die Frage, wie man mit $\mathbf{NP}$-vollständigen Problemen umgeht, unbeantwortet. In der Praxis gibt es im Moment zwei Hauptansatzpunkte: Die erste Möglichkeit ist die, die Allgemeinheit des untersuchten Problems zu beschränken und eine spezielle Version zu betrachten, die immer noch für die geplante Anwendung ausreicht. Zum Beispiel sind Graphenprobleme oft einfacher, wenn man zusätzlich fordert, dass die Knoten des Graphen in der (Euklidischen) Ebene lokalisiert sind. Deshalb sollte die erste Idee bei der Behandlung von $\mathbf{NP}$-vollständigen Problemen immer sein, zu untersuchen, welche Einschränkungen man an das Problem machen kann, ohne die praktische Aufgabenstellung zu verfälschen. Gerade diese Einschränkungen können dann effiziente Algorithmen ermöglichen.

Die zweite bekannte Möglichkeit sind sogenannte *Approximationsalgorithmen* (siehe [1]). Die Idee hier ist es, nicht die optimalen Lösungen zu suchen, sondern sich mit einem kleinen garantierten Fehler zufrieden zu geben. Dazu folgendes Beispiel. Es ist bekannt, dass das TSP-Problem auch dann noch $\mathbf{NP}$-vollständig ist, wenn man annimmt, dass die Städte in der Euklidischen Ebene lokalisiert sind, d.h. man kann die Städte in einer fiktiven Landkarte einzeichnen, sodass die Entfernungen zwischen den Städten proportional zu den Abständen auf der Landkarte sind. Das ist sicherlich in der Praxis keine einschränkende Abschwächung des Problems und zeigt, dass die oben erwähnte Methode nicht immer zum Erfolg führen muss: Hier bleibt auch das eingeschränkte Problem **NP**-vollständig. Aber für diese eingeschränkte TSP-Variante ist ein Polynomialzeitalgorithmus bekannt, der immer eine Rundreise berechnet, die höchstens um einen beliebig wählbaren Faktor schlechter ist, als die optimale Lösung. Ein Chip-Hersteller, der bei der Bestückung seiner Platinen die Wege der Roboterköpfe minimieren möchte, kann also beschließen, sich mit einer Tour zufrieden zu geben, die um 5 % schlechter ist als die optimale. Für dieses Problem existiert ein effizienter Algorithmus! Dieser ist für die Praxis völlig ausreichend.

Danksagung

Für hilfreiche Diskussionen während des Entstehens dieses Artikels danken wir Sven Kosub, Würzburg.

Literatur

[1] AUSIELLO, G., P. CRESCENZI, G. GAMBOSI, V. KANN, A. MARCHETTI-SPACCAMELA und M. PROTASI: *Complexity and Approximation – Combinatorial Optimization Problems and Their Approximability.* Springer Verlag, Berlin Heidelberg, 1999.

[2] BUSYGIN, S.: *NP-Completeness Page.* www.geocities.com/st_busygin/.

[3] CLAY MATHEMATICS INSTITUTE: *Millenium Prize Problems.* www.claymath.org/prize_problems /index.htm.

[4] GAREY, M. R. und D. S. JOHNSON: *Computers and Intractability, A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[5] GRUSKA, J.: *Quantum Computing.* McGraw-Hill, 1999.

[6] KAYE, R.: *Minesweeper Pages.* www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm.

[7] LAMPERT, C.: *Informatik-Problem P=NP möglicherweise geknackt.* www.heise.de/newsticker /data/ts-16.10.00-001, Oktober 2000.

[8] LAMPERT, C.: *Ukrainische Überraschung – Theoretische Informatik: Gilt doch P=NP?* c't, 22:62, 2000.

[9] MOTWANI, R. und P. RAGHAVAN: *Randomized Algorithms.* Cambridge University Press, 1995.

[10] PĂUN, G.: *Computing with Bio-Molecules.* Springer Series in Discrete Mathematics and Theoretical Computer Science. Springer Verlag, Sinpapore, 1998.

[11] PLOTNIKOV, A. D.: *An Efficient Algorithm for the Minimum Clique Partition Problem.* Erhältlich bei www.vinnica.ua/∼aplot, 2000.

[12] RINK, J.: *Quäntchen für Quäntchen – Fortschritte in der Quanteninformationsverarbeitung.* c't, 16:150, 1998.

[13] SCHÖNING, U.: *Algorithmen – kurz gefasst.* Hochschultaschenbuch. Spektrum Akademischer Verlag, Heidelberg Berlin, 1997.

[14] STEWART, I. A.: *Million-Dollar Minesweeper.* www.claymath.org/prize_problems/million-dollar-minesweeper.htm, 2000.

[15] STÖBE, M.: *P=NP=Minesweeper.* www.heise.de/newsticker/data/mst-03.11.00-000, November 2000.

[16] VOLLMER, H.: *Was leistet die Komplexitätstheorie für die Praxis?* Informatik-Spektrum, 22(5):317–327, 1999.

Preprint-Reihe
Institut für Informatik
Universität Würzburg

Verantwortlich: Die Vorstände des Institutes für Informatik.

[213] F. Duckstein, R. Kolla. *Ray Tracing Of Parametric Surfaces Based On Adaptive Simplicial Complexes.* Oktober 1998.

[214] N. Vicari. *Effects of Variations in the Available Bandwidth on the Performance of the GFR Service.* Oktober 1998.

[215] M. Dümmler, A. Schömig. *Using Discrete-time Analysis in the Performance Evaluation of Manufacturing.* November 1998.

[216] C. Glaßer. *A Normalform for Classes of Concatenation Hierarchies.* Dezember 1998.

[217] S. Kosub. *Persistent Computations.* Dezember 1998.

[218] S. Reith, K. W. Wagner. *The Complexity of Problems Defined by Subclasses of Boolean Functions.* Januar 1999.

[219] C. Glaßer, G. Wechsung. *Relativizing Function Classes.* Januar 1999.

[220] H. Schmitz. *Some Forbidden Patterns in Automata for Dot-Depth One Languages.* Januar 1999.

[221] T. Peichl, H. Vollmer. *Finite Automata with Generalized Acceptance Criteria.* Januar 1999.

[222] H. Vollmer. *Uniform Characterizations of Complexity Classes.* Januar 1999.

[223] U. Ehrenberg, K. Leibnitz. *Impact of Clustered Traffic Distributions in CDMA Radio Network Planning.* März 1999.

[224] H. Vollmer. *Was leistet die Komplexitätstheorie für die Praxis?* März 1999.

[225] R. Kolla, A. Vodopivec, J. Wolff von Gudenberg. *The IAX Architecture: Interval Arithmetic Extension.* April 1999.

[226] H. Schmitz. *Generalized Deterministic Languages and their Automata: A Characterization of Restricted Temporal Logic.* April 1999.

[227] M. Dümmler. *Using Simulation and Genetic Algorithms to Improve Cluster Tool Performance.* Mai 1999.

[228] R. Harris, S. Köhler. *Possibilities for QoS in Existing Internet Routing Protocols.* Mai 1999.

[229] F. Heister, R. Müller. *An approach for the identification of nonlinear, dynamic processes with Kalman Filter-trained recurrent neural structures.* Mai 1999.

[230] C. Glaßer. *The Boolean Hierarchy over Dot-Depth* $1/2$. June 1999.

[231] M. Dümmler. *Simulation und Leistungsbewertung von Cluster Tools in der Halbleiterfertigung.* Juli 1999.

[232] C. Glaßer, S. Reith, H. Vollmer. *Approximation Algorithms for Cellular Network Optimization.* Juli 1999.

[233] S. Kosub, K. W. Wagner. *The Boolean Hierarchy of Partitions.* Juli 1999.

[234] B. Kluge, D. Schäfer. *Featurebasierte Lokalisation eines autonomen mobilen Roboters.* Juli 1999.

[235] M. Dümmler, N. Vicari. *A Numerical Analysis of the* $M/D^b/N$ *Queueing System.* Juli 1999.

[236] M. Buck, H. Noltemeier, D. Schäfer. *Practical Strategies for Hypotheses Elimination on the Self-Localization Problem.* August 1999.

[237] S. Köhler, U. Schäfer. *Performance Comparison of different Class-and-Drop treatment of Data and Acknowledgements in DiffServ IP Networks.* August 1999.

[238] N. Vicari, S. Köhler. *Measuring Internet User Traffic Behavior Dependent on Access Speed.* Oktober 1999.

[239] T. Ebert, H. Vollmer. *On the Autoreducibility of Random Sequences.* Oktober 1999.

[240] H. Schmitz. *Boolean Hierarchies inside Dot-Depth One.* Oktober 1999.

[241] U. Hertrampf, S. Reith, H. Vollmer. *A note on closure properties of logspace MOD classes.* Oktober 1999.

[242] P. McKenzie, H. Vollmer, K. W. Wagner. *Arithmetic Circuits and Polynomial Replacement Systems.* November 1999.

[243] C. Glaßer, H. Schmitz. *Languages of Dot-Depth 3/2.* November 1999.

[244] B. Kluge, D. Schäfer. *Featurebasierte Lokalisation eines autonomen mobilen Roboters - Experimentelle Fallstudie.* Dezember 1999.

[245] C. Glaßer. *Consequences of the Existence of Sparse Sets Hard for NP under a Subclass of Truth-Table Reductions.* Januar 2000.

[246] N. Vicari, S. Köhler, J. Charzinski. *The Dependence of Internet User Characteristics on Access Speed.* Januar 2000.

[247] M. Menth. *Carrying Wireless Traffic over IP Using RTP Multiplexing.* Januar 2000.

[248] P. McKenzie, T. Schwentick, D. Therien, H. Vollmer. *The Many Faces of a Translation.* Februar 2000.

[249] F. Wolz, R. Kolla. *Discrete Floorplanning by Multidimensional Pattern Matching.* Februar 2000.

[250] O. Rose, M. Dümmler, A. Schömig. *On the Validity of Approximation Formulae for Machine Downtimes.* Februar 2000.

[251] S. Reith, H. Vollmer. *Optimal Satisfiability for Propositional Calculi and Constraint Satisfaction Problems.* März 2000.

[252] C. Glaßer, S. Reith, H. Vollmer. *The Complexity of Base Station Positioning in Cellular Networks.* März 2000.

[253] F. Klügl, F. Puppe, P. Schwarz und H. Szczerbicka. *Multiagentsystems and Individual-Based Simulation.* März 2000.

[254] M. Dümmler, O. Rose. *Analysis of the Short Term Impact of Changes in Product Mix.* März 2000.

[255] S. Reith, K. W. Wagner. *The Complexity of Problems Defined by Boolean Circuits.* März 2000.

[256] C. Glaßer, H. Schmitz. *Concatenation Hierarchies and Forbidden Patterns.* März 2000.

[257] S. Kosub. *On NP-Partitions over Posets with an Application to Reducing the Set of Solutions of NP Problems* April 2000.

[258] D. Staehle, S. Köhler, U. Kohlhaas. *Optimization of the routing parameters for IP networks.* Mai 2000.

[259] S. Köhler, M. Menth, N. Vicari. *Analytic Performance Evaluation of the RED Algorithm for QoS in TCP/IP Networks.* Mai 2000.

[260] F. Klügl. *Visuelles Programmieren als Basis einer Modellierumgebung: Eigenschaften, Konzepte, Anforderungen.* Mai 2000.

[261] D. Staehle, K. Leibnitz, P. Tran-Gia. *Source Traffic Modeling of Wireless Applications.* Juni 2000.

[262] L. Berry, S. Köhler, D. Staehle, P. Tran-Gia. *Fast heuristics for optimal routing in large IP networks.* Juli 2000.

[263] M. Menth. *The Performance of Multiplexing Voice and Circuit Switched Data in UMTS over IP-Networks.* Juli 2000.

[264] M. Galota, C. Glaßer, S. Reith, H. Vollmer. *A Polynomial-Time Approximation Scheme for Base Station Positioning in UMTS Networks.* August 2000.

[265] J. Wolff von Gudenberg. *Multimedia Architectures and Interval Arithmetic.* Oktober 2000.

[266] S. Kosub. *Boolean Partitions and Projective Closure.* November 2000.

[267] S. Kosub. *Types of Separability.* November 2000.

[268] K. Leibnitz, A. Krauss. *Performance Evaluation of Interference and Cell Loading in UMTS Networks.* Februar 2001.

[269] S. Reith, H. Vollmer. *Ist P=NP? Einführung in die Theorie der NP-Vollständigkeit.* Februar 2001.