Elem. Math. 57 (2002) 1–6
0013-6018/01/010001-05 $1.50 + 0.20/0

©Birkhäuser Verlag, Basel, 2002

Elemente der Mathematik

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

Martin Grötschel

Hinter der für den Uneingeweihten etwas kryptischen Frage „$\mathcal{P} = \mathcal{NP}$?“ verbirgt sich das derzeit wichtigste Problem der Komplexitätstheorie. Dieser Artikel erläutert einige Aspekte der Theorie und erklärt informell, was „$\mathcal{P} = \mathcal{NP}$?“ bedeutet. Es geht nicht nur um komplizierte algorithmische Mathematik und Informatik, sondern um grundsätzliche Fragen unserer Lebensumwelt. Kann man vielleicht beweisen, dass es für viele Probleme unseres Alltags keine effizienten Lösungsmethoden gibt?

Die Witzboldlösung Wir können $\mathcal{N}$ und $\mathcal{P}$ als Variable interpretieren, wie das Witzbolde schon gemacht haben; $\mathcal{P} = \mathcal{NP}$ gilt dann offensichtlich, wenn $\mathcal{P} = 0$ oder $\mathcal{N} = 1$ ist. $\mathcal{P}$ (deswegen meistens in $\mathcal{SCRIPT}$ geschrieben) ist jedoch keine Variable; $\mathcal{P}$ steht für die Klasse aller Entscheidungsprobleme, die auf einer Turing-Maschine mit einem Algorithmus in polynomialer Laufzeit gelöst werden können. Man kommt von der Klasse $\mathcal{P}$ zur Klasse $\mathcal{NP}$, wenn man *polynomial* durch *nichtdeterministisch-polynomial* ersetzt. Wer so etwas Abschreckendes am Anfang eines Artikels liest, hört meistens sofort auf. Es ist aber nicht so schlimm. Und deswegen beginnen wir von neuem.

Was ist ein schneller Algorithmus? Ein zentraler Begriff der Komplexitätstheorie ist der *schnelle Algorithmus*, man sagt auch *guter* oder *effizienter* Algorithmus. Jeder hat eine intuitive Vorstellung von „schnell“. Hier ist mein Beispiel. Wenn ich mit dem Auto fahren will und die beste Wegstrecke nicht kenne, dann rufe ich im Internet einen der vielen Routenplaner auf, gebe Start- und Zielort ein und lasse die kürzeste oder schnellste Fahrtstrecke berechnen. Die Eingabe der Ortsdaten benötigt in der Regel (zumindest, wenn ich tippe) mehr Zeit als die Berechnung der Route. Das empfinde ich als schnell.
Kurz bevor ich diesen Artikel schrieb, musste ich von Berlin-Hohengatow nach Erkner fahren. Die kilometermäßig kürzeste Strecke führt mitten durch Berlin (52 km, 1:46 h laut Routenplaner); im morgendlichen Berufsverkehr ist das kein guter Weg, wenn man einen Termin hat. Die zeitlich schnellste Strecke verläuft über

den nordöstlichen Autobahnring (99 km, 1:28 h). Ein Kollege warnte mich vor Baustellen auf dieser Strecke. Durch Eingabe verschiedener Zwischenpunkte und wiederholten Aufruf des Programms habe ich eine mir „vernünftig“ erscheinende Route gefunden. Dies ist eine typische Situation bei Anwendungen der Mathematik. Ein Problem wird nicht nur einmal gelöst. Man zieht zusätzliche Erwägungen in Betracht, wiederholt den Lösungsvorgang mehrfach und wählt unter vielen Lösungen unter Berücksichtigung weiterer Überlegungen (Verzögerung durch Baustellen) eine „akzeptable“ Lösung aus. Man wendet also einen Algorithmus (hier ein Verfahren zur Berechnung kürzester Wege) mehrfach an, und man wird das nur dann tun, wenn der Rechner in Sekundenschnelle antwortet.

Das Problem des kürzesten Weges: Eine Variante Bei der Bestimmung eines akzeptablen Weges von Hohengatow nach Erkner rief ich das Kürzeste-Wege-Programm unter Angabe von Zwischenstationen Z_1, Z_2, ... Z_k auf. Das Programm berechnet den kürzesten Weg von Hohengatow nach Z_1, von Z_1 nach Z_2, usw. Die Reihenfolge der Zwischenstationen wird wie gewünscht berücksichtigt. Hätte ich einen Ausflug machen und Sehenswürdigkeiten in Z_1, ... Z_k anschauen wollen, wäre mir die Reihenfolge gleichgültig gewesen. Jetzt kommt die Überraschung. Niemand kennt einen Algorithmus, der diese „kleine Variante“ garantiert in Sekundenschnelle optimal lösen kann. Hat man Pech, dauert die Routenberechnung länger als der Ausflug. Wie kommt das? Genau dies ist der Kern der Frage „$\mathcal{P} = \mathcal{NP}$?“.

Die Laufzeit eines Algorithmus Zur Erklärung der „$\mathcal{P} = \mathcal{NP}$?“–Frage müssen wir etwas formaler werden und einige Grundbegriffe der Komplexitätstheorie erläutern. Um über Algorithmen sprechen zu können, braucht man ein *Rechnermodell*. In der Theorie betrachtet man so genannte Turing-Maschinen. Für die Zwecke dieses Artikels reicht es, sich einen PC vorzustellen. Ein *Algorithmus* ist ein Programm, das auf einem PC abläuft. Die Schnelligkeit eines Programms bestimmen wir in der Praxis durch Messung der Ausführungszeit. In der Theorie müssen wir vorsichtiger sein, denn uns interessiert die Qualität des Programms, nicht die des PC. Deswegen wird die Laufzeit rechnerunabhängig definiert. Man zerlegt dazu die Ausführung eines Programms in einzelne, so genannte *elementare Rechenschritte*. Elementare Rechenschritte sind zum Beispiel die Addition zweier Zahlen oder das Schreiben auf einen Speicherplatz. Die *Laufzeit* eines Algorithmus ist bei dieser Sichtweise die Anzahl der ausgeführten elementaren Rechenschritte. Dies ist ein theoretisch brauchbares Maß, das auch die Praxis (wenn man z.B. die Zykluszeiten eines PC kennt) gut wiedergibt.
Es ist klar, dass die Laufzeit eines Algorithmus von den Input-Daten abhängt. Zur Berechnung eines kürzesten Weges von Hohengatow nach Erkner braucht man nur die Straßendaten von Berlin und Brandenburg. Die Bestimmung der besten Route von Moskau nach Madrid benötigt größere Datenmengen und mehr Rechenzeit. Analoges gilt auch für das Rechnen mit Zahlen. Einstellige Zahlen können wir im Kopf multiplizieren. Der Rechner macht das in einem elementaren Schritt. Für die Multiplikation zweier hundertstelliger Zahlen muss der Rechner (genauso wie wir) rund zehntausend elementare Multiplikationen mit einstelligen Zahlen und etwa ebenso viele Additionen durchführen. Formal ausgedrückt, die Multiplikation

zweier k-stelliger Zahlen benötigt rund k^2 elementare Rechenschritte.
Zur präzisen Definition der Laufzeit eines Algorithmus müssen wir noch genauer werden. Wir legen fest, welche Inputs erlaubt sind und wie die Input-Daten kodiert werden. Das Übliche in Theorie und Praxis ist die Binärkodierung, also die Darstellung von Zahlen als Folge von Nullen und Einsen. Die Zahl -100 wird dann binär als -1100100 dargestellt, benötigt also inklusive Vorzeichen acht Bits Kodierungslänge. Analog wird festgelegt, wie man mit den Buchstaben eines Alphabets oder den Knoten und Kanten eines Graphen verfährt.
Man definiert dann: Die *Laufzeit* $l_A(n)$ eines Algorithmus A auf Inputs der Kodierungslänge n ist die maximale Anzahl elementarer Rechenoperationen, die der Algorithmus A ausführt, wenn er mit Inputs der Kodierungslänge höchstens n aufgerufen wird. Es ist klar, dass man $l_A(n)$ nicht wirklich berechnen kann. Man spricht von einer *polynomialen Laufzeit*, wenn $l_A(n)$ durch ein Polynom in n abgeschätzt werden kann. Gilt zum Beispiel

$$l_A(n) \leq an^2 + bn + c$$

für alle möglichen Inputlängen n (a, b, c sind Konstante), so sagt man, dass der Algorithmus A eine quadratische Laufzeit hat.
Der naive Primzahltest, man ziehe die Wurzel aus z und prüfe dann, ob eine der natürlichen Zahlen kleiner oder gleich $\sqrt{z}$ ein Teiler von z ist, ist kein polynomialer Algorithmus, da die Anzahl der Divisionen nicht durch ein Polynom in der Kodierungslänge $\lceil \log(z+1) \rceil + 1$ von z abgeschätzt werden kann.

Mathematisch schnelle Algorithmen Es hat sich eingebürgert, Algorithmen mit polynomialer Laufzeit schnell zu nennen, auch wenn jedem bewusst ist, dass eine Laufzeit von n^{1000} hoffnungslos ist. Hier geht es nur um eine grobe Rasterung des Laufzeitverhaltens. Ein n^{10}-Algorithmus ist praktisch nicht verwendbar, aber er ist immerhin für alle $n \geq 10$ erheblich schneller als ein n^n-Algorithmus.
Es mag überraschend erscheinen, aber der gegenwärtige Wissensstand ist, dass für viele interessante Probleme der Theorie und der industriellen Praxis keine polynomialen Algorithmen bekannt sind, nicht einmal Algorithmen mit Laufzeit $n^{1000^{1000}}$.

Die Klasse $\mathcal{P}$ Wir machen noch eine Vereinfachung. Statt mathematische Probleme allgemeiner Art zuzulassen, beschränken wir uns ab jetzt auf *Entscheidungsprobleme*. Das sind solche Probleme, bei denen nur eine *Ja*- oder *Nein*-Antwort gegeben werden muss. Beispiele hierfür sind:

- Ist eine gegebene natürliche Zahl Summe von zwei Quadratzahlen?
- Besitzt ein gegebener Graph einen Hamiltonschen Weg?

Ein *Hamiltonscher Weg* ist ein Weg, der in einem beliebigen Knoten beginnt, in einem beliebigen anderen Knoten endet, der über alle übrigen Knoten führt und dabei nur Kanten aus dem Graphen benutzt. Versuchen Sie einmal, einen Hamiltonschen Weg im Graphen in Abbildung 1 zu finden!
Optimierungsprobleme kann man in Entscheidungsprobleme verwandeln. Statt einen kürzesten Weg von Hohengatow nach Erkner zu suchen, fragt man beispielsweise, ob es einen Weg mit höchstens 53 km Länge gibt.

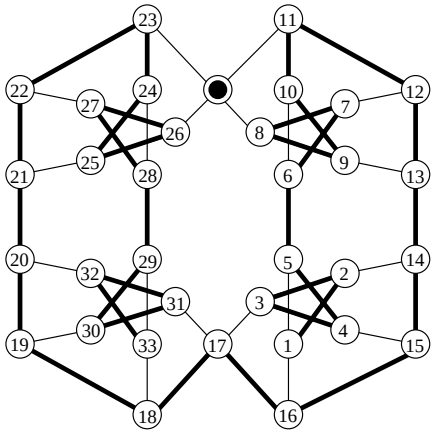

Abbildung 1

Die Klasse $\mathcal{P}$, so die formale Definition, besteht aus allen Entscheidungsproblemen, für die es einen Algorithmus gibt, der in polynomialer Laufzeit eine *Ja*- oder *Nein*-Antwort liefert. Entscheidungsprobleme aus der Klasse $\mathcal{P}$ sind zum Beispiel:

- Gibt es in einem Graphen einen Weg von A nach B mit Länge höchstens c?
- Ist eine gegebene $n \times n$-Matrix invertierbar?
- Ist ein gegebener Graph zusammenhängend?

Die Klasse $\mathcal{NP}$ Vom Hamiltonschen Wegeproblem und den folgenden beiden Entscheidungsproblemen weiß man nicht, ob sie zur Klasse $\mathcal{P}$ gehören

- Gegeben sei ein Graph G, kann man die Knoten von G mit höchstens k Farben so färben, dass je zwei benachbarte Knoten verschieden gefärbt sind?
- Gegeben eine positive ganze Zahl z, ist z das Produkt von zwei von 1 verschiedenen ganzen Zahlen, das heißt, ist z zusammengesetzt?

Diese Probleme haben eine besondere Eigenschaft. Falls die Antwort *Ja* lautet, kann man einen schnell überprüfbaren Beweis der Korrektheit der *Ja*-Antwort liefern.
Betrachten wir den Graphen G aus Abbildung 1. Entfernen wir den fetten Knoten aus G und alle Kanten, die mit diesem Knoten inzidieren, enthält der so entstehende Graph G' einen Hamiltonschen Weg. Dieser ist fett eingezeichnet und läuft von 1 über 2, 3, ... bis zum Knoten 33. Dass dies ein Hamiltonscher Weg ist, ist einfach zu überprüfen. Ist ein Graph mit höchstens k Farben färbbar, so kann man eine Färbung liefern und mühelos feststellen, ob alle Kanten verschieden gefärbte Endknoten haben. Ist eine Zahl z das Produkt von zwei anderen Zahlen z_1 und z_2, so kann man durch Multiplikation prüfen, ob $z = z_1 \cdot z_2$ gilt.
Wir führen nun die Klasse $\mathcal{NP}$ ein. Ein Entscheidungsproblem gehört zur Klasse $\mathcal{NP}$, wenn es folgende Eigenschaften besitzt:

(a) Lautet für einen gegebenen Input die Antwort auf die Frage *Ja*, gibt es ein Zertifikat, mit dessen Hilfe die Korrektheit der *Ja*-Antwort überprüfbar ist.
(b) Es gibt einen Algorithmus (genannt *Prüfalgorithmus*), der die normale Inputsequenz und das Zertifikat als Input akzeptiert und der in einer Laufzeit, die polynomial in der Kodierungslänge des normalen Inputs ist, überprüft, ob das Zertifikat ein Beweis für die Korrektheit der *Ja*-Antwort ist.

Das hört sich kompliziert an, ist aber verständlicher als es scheint. Für das Hamiltonsche Wegeproblem besteht der normale Input aus dem Bitstring, der den gegebenen Graphen repräsentiert. Das Zertifikat ist im obigen Beispiel die binär kodierte Knotenfolge 1, 2, 3, ..., 33. Unser Prüfalgorithmus liest erst den Graphen (dies definiert die Kodierungslänge) und dann die Knotenfolge. Danach prüft er, ob die Knotenfolge einen Hamiltonschen Weg repräsentiert oder nicht. Diese Überprüfung muss in einer Laufzeit erfolgen, die polynomial in der Kodierungslänge des Graphen ist.
Ein anderes Beispiel. Um zu zeigen, dass 1090621093 zusammengesetzt ist, muss der Prüfalgorithmus keine Primfaktorzerlegung durchführen. Werden ihm die Zahlen 4585 und 237969 als Zertifikat geliefert, stellt er durch Multiplikation fest, dass das Zertifikat gar keins ist. Besteht der Zusatzinput aus 4583 und 237971, so ist das Produkt 1090621093. Damit ist bewiesen, dass diese Zahl keine Primzahl ist.

Die Klasse co-$\mathcal{NP}$ Eine Besonderheit der Definition von $\mathcal{NP}$ ist die Unsymmetrie in *Ja* und *Nein*. Das Problem „Enthält ein Graph G einen Hamiltonschen Weg?“ ist in $\mathcal{NP}$. Müsste dann nicht auch das Problem „Enthält G keinen Hamiltonschen Weg?“ in $\mathcal{NP}$ sein? Niemand weiß derzeit, wie man hierfür ein Zertifikat angeben kann, das in polynomialer Zeit überprüfbar ist.
In der Tat erscheinen *Probleme der Nichtexistenz* „irgendwie“ noch schwieriger. Schauen Sie sich noch einmal den Graphen G in Abbildung 1 an und versuchen Sie nachzuweisen, dass er keinen Hamiltonschen Weg enthält. Dies ist mühselig!
Man bezeichnet die Klasse der Entscheidungsprobleme, die komplementär (Vertauschung von *Ja* und *Nein*) zu Problemen in $\mathcal{NP}$ sind, mit co-$\mathcal{NP}$. „Enthält G keinen Hamiltonschen Weg?“ ist also ein Problem der Klasse co-$\mathcal{NP}$. Auch das Problem „Ist die ganze Zahl z prim?“ ist damit in co-$\mathcal{NP}$. Mit dem Einsatz von Zahlentheorie kann man beweisen, dass dieses Problem auch in $\mathcal{NP}$ ist. Das Primzahlproblem ist also in $\mathcal{NP} \cap$ co-$\mathcal{NP}$.

$\mathcal{P}$ und $\mathcal{NP}$ Alle Probleme in $\mathcal{P}$ sind natürlich in $\mathcal{NP}$ und in co-$\mathcal{NP}$. Für Probleme in $\mathcal{P}$ gibt es ja einen Algorithmus, der in polynomialer Laufzeit (nur mit den normalen Input-Daten und ohne Zertifikat) eine *Ja*- oder *Nein*-Antwort liefert. Damit haben wir folgende Erkenntnisse gewonnen.
Die Klasse $\mathcal{P}$ ist sowohl in $\mathcal{NP}$ als auch in co-$\mathcal{NP}$ enthalten. Niemand weiß jedoch, ob $\mathcal{P} = \mathcal{NP}$, ob $\mathcal{P} = \mathcal{NP} \cap$ co-$\mathcal{NP}$ oder ob $\mathcal{NP} =$ co-$\mathcal{P}$ ist. Als wichtigste Frage (unter vielen anderen offenen Problemen der Komplexitätstheorie) gilt das Problem

$$\text{„}\mathcal{P} = \mathcal{NP}?\text{“}$$

da sehr viele Aufgaben des täglichen Lebens (in ihrer Version als Entscheidungsproblem) zur Klasse $\mathcal{NP}$ gehören.

$\mathcal{NP}$-Vollständigkeit Es gibt eine faszinierende Unterklasse der Probleme in $\mathcal{NP}$. Ein Entscheidungsproblem Π wird *$\mathcal{NP}$-vollständig* genannt, wenn es in $\mathcal{NP}$ ist und folgende Eigenschaft besitzt:

- Falls es einen polynomialen Algorithmus für Π gibt, dann ist $\mathcal{P} = \mathcal{NP}$.

Es ist kaum zu glauben, dass es $\mathcal{NP}$-vollständige Probleme gibt; aber in der Tat sind sehr viele Probleme $\mathcal{NP}$-vollständig, so zum Beispiel das Problem des Hamiltonschen Weges und das Knotenfärbungsproblem.
Die Aussicht, durch den Entwurf eines polynomialen Algorithmus für ein einziges $\mathcal{NP}$-vollständiges Problem nachweisen zu können, dass $\mathcal{P} = \mathcal{NP}$ ist, hat zu intensiver Beschäftigung mit diesem Thema eingeladen. Jede Menge falscher Beweise (nicht nur von Laien) pflastern diesen Weg: viel Schweiß und bisher kein Erfolg.

Diagonalisierung Fast alle, die sich mit der Komplexitätstheorie beschäftigen, sind der Überzeugung, dass $\mathcal{P} \neq \mathcal{NP}$ gilt. Zum Beweis müßte man z.B. ein Entscheidungsproblem finden, das nachweisbar nicht in polynomialer Zeit gelöst werden kann. Hierfür scheinen uns jedoch wirksame Beweistechniken zu fehlen. Man hat es u.a. mit Diagonalisierung (sie geht auf G. Cantor zurück) versucht. Dies ist die Methode, mit der man beweist, dass es mehr reelle als natürliche Zahlen gibt. Es konnte jedoch nachgewiesen werden, dass $\mathcal{P} \neq \mathcal{NP}$ damit nicht bewiesen werden kann, siehe [1]. Anderen Techniken ist es ähnlich ergangen.

Folgerungen aus der Problemlösung Der Nachweis von $\mathcal{P} \neq \mathcal{NP}$ würde nach meiner Einschätzung dauerhafte Beschäftigung für Mathematiker und Informatiker garantieren. $\mathcal{NP}$-vollständige Probleme treten überall auf, sie müssen täglich gelöst werden. Ohne allgemeine Lösungsansätze muss man anwendungsspezifisch vorgehen und spezielle Problemtypen aus der Praxis untersuchen. So wird das heute bereits gemacht, und so kann man häufig schwierige industrielle Fragestellungen in akzeptabler Laufzeit und Qualität lösen.
$\mathcal{P} = \mathcal{NP}$ könnte durch nicht-konstruktive Argumente bewiesen werden. Das könnte z.B. heißen, dass man die Existenz eines polynomialen Algorithmus für ein $\mathcal{NP}$-vollständiges Problem nachweist, ohne einen polynomialen Algorithmus explizit anzugeben. Ein solches Ergebnis würde große Ratlosigkeit hinterlassen.
Über die Konsequenzen eines konstruktiven Beweises von $\mathcal{P} = \mathcal{NP}$ sind sich die Auguren nicht einig. Für die gegenwärtige Kryptographie wäre dies verheerend, da damit alle vorhandenen Verschlüsselungssysteme potentiell unsicher würden. Die Industrie würde profitieren. Wichtige Probleme der Praxis (Produktionsplanung, Chipdesign, Transport und Verkehr, Telekommunikation, ...) wären dann in kurzer Zeit optimal lösbar. Ich persönlich glaube, dass in diesem Falle die Komplexitätstheorie revidiert werden muss. Ich „weiß aus Erfahrung“, dass Kürzeste-Wege-Probleme viel einfacher zu lösen sind als Hamiltonsche-Wege-Probleme. Wenn $\mathcal{P} = \mathcal{NP}$ gilt, dann ist die Theorie zu grob und muss so verfeinert werden, dass man die in Rechenexperimenten beobachteten Unterschiede auch theoretisch sauber auseinander halten kann. Das ist keine wissenschaftliche Aussage, sondern einzig ein „Glaubensbekenntnis“. Und dann könnte sich die Frage „$\mathcal{P} = \mathcal{NP}$?“ als unabhängig

von den Axiomen der Mengenlehre erweisen; sie könnte also eine Rolle wie die Kontinuumshypothese spielen. Aber darüber wollen wir hier nicht spekulieren.

Nichtdeterministisch? Was hat es nun mit dem Wort *nichtdeterministisch*, von dem das $\mathcal{N}$ in $\mathcal{NP}$ kommt, auf sich? Für die Klasse $\mathcal{NP}$ gibt es verschiedene äquivalente Definitionen. Bei einigen wird das „Nichtdeterministische“ sichtbarer als bei der von mir aus Gründen der einfachen Darstellbarkeit gewählten Definition. Hier ist ein Erklärungsversuch. Wir stellen uns ein Entscheidungsproblem vor. Wir lesen die normale Inputsequenz ein. Ein deterministischer Algorithmus würde nun „loslegen“. Ein nichtdeterministischer Algorithmus darf zuerst raten, und zwar alle möglichen Zertifikate, die zu einem Beweis der Korrektheit der *Ja*-Antwort führen könnten. Nach jedem Rateschritt läuft mit dem normalen Input und dem geratenen Zertifikat ein deterministischer Algorithmus ab, der überprüft, ob das Zertifikat die *Ja*-Antwort bestätigt. Einen solchen Algorithmus nennt man *nichtdeterministisch*. Ist die Antwort auf eine gegebene Inputsequenz *Ja* und führt nur ein einziges der möglichen Zertifikate in polynomialer Laufzeit zum Korrektheitsbeweis der *Ja*-Antwort, dann sagt man, dass der nichtdeterministische Algorithmus eine polynomiale Laufzeit hat. Die Klasse $\mathcal{NP}$ besteht aus allen Entscheidungsproblemen, die mit einem nichtdeterministischen Algorithmus in polynomialer Zeit gelöst werden können. Diese Interpretation der Klasse $\mathcal{NP}$ macht deutlich, warum kaum jemand an $\mathcal{P} = \mathcal{NP}$ glaubt. Es ist schwer vorstellbar, dass ein deterministischer Algorithmus genauso viel in polynomialer Zeit konstruieren kann (Klasse $\mathcal{P}$) wie ein durch (ganz schön mächtig erscheinende) Raterei „aufgepeppter“ nichtdeterministischer Algorithmus. Oder doch?

Schlussbemerkungen Gute Bücher zum Thema sind [3], [4] und [5]. Sie erläutern präzise und ausführlicher, was in diesem Artikel nur angedeutet wurde. Eine ausgezeichnete Übersicht gibt Stephen Cook [2]. Cook wurde dadurch berühmt, dass er als Erster die Existenz von $\mathcal{NP}$-vollständigen Problemen nachwies.

Literatur

[1] T. Baker, J. Gill, and R. Solovay *Relativizations of the P =? NP question.* SIAM Journal on Computing, 4(1975);431-442.

[2] S. Cook *The P versus NP Problem.*
URL: `http://www.claymath.org/prizeproblems/pvsnp.htm`.

[3] M.R. Garey and D.S. Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[4] Ch. Papadimitriou. *Computational Complexity.* Addison-Wesley, Amsterdam, 1994.

[5] M. Sipser *Introduction to the Theory of Computation.* PWS, Boston, 1997.