5. Problemkomplexität

Im vorangegangenen Kapitel haben wir uns mit der Frage beschäftigt, ob ein Problem (eine Sprache) entscheidbar (erkennbar) ist. In diesem Kapitel betrachten wir nur entscheidbare Probleme (erkennbare Sprachen) und stellen die Frage nach der Effizienz dieser Entscheidung (Erkennung). Siehe etwa [Weg] für Zusatzinformation.

5.1. Komplexitätsklassen

Def. 5.1.1: Eine rekursive Sprache L hat *Zeitkomplexität* $T(n)$ (*Raumkomplexität* $S(n)$), wenn es einen Algorithmus (d.h. immer terminierende Turing–Maschine) mit Zeitkomplexität $T(n)$ (Raumkomplexität $S(n)$) gibt, welcher L erkennt. Die Sprache L hat *nichtdeterministische Zeitkomplexität* $T(n)$ (*nichtdeterministische Raumkomplexität* $S(n)$), wenn es eine nichtdeterministische Turing–Maschine mit Zeitkomplexität $T(n)$ (Raumkomplexität $S(n)$) gibt, welche L erkennt. ∎

Beispiel 5.1.1: Sei L die Sprache $\left\{wcw^R \mid w \in \{0,1\}^*\right\}$. Dabei ist w^R die Umkehrung des Wortes w. Die Sprache L hat Zeitkomplexität $n+1$. Es gibt nämlich eine Turing–Maschine M_1 mit zwei Bändern, welche die Eingabe links von c auf das zweite Band kopiert, und anschließend synchron den LS–Kopf des ersten Bandes weiter nach rechts und den LS–Kopf des zweiten Bandes nach links bewegt. Dabei werden in jedem Schritt die beiden aktuellen Symbole verglichen. Sind diese jeweils gleich und ist überdies die Anzahl der Symbole links von c gleich der Anzahl der Symbole rechts von c, so akzeptiert M_1 die Eingabe. M_1 braucht nur $n+1$ Schritte, wenn die Eingabe von der Länge n ist.

Die Raumkomplexität von L ist offensichtlich $\mathcal{O}(n)$. ∎

Def. 5.1.2: DTIME($T(n)$) ist die Familie der Sprachen mit Zeitkomplexität $T(n)$, DSPACE($S(n)$) ist die Familie der Sprachen mit Raumkomplexität $S(n)$.
NTIME($T(n)$) ist die Familie der Sprachen mit nichtdeterministischer Zeitkomplexität $T(n)$, NSPACE($S(n)$) ist die Familie der Sprachen mit nichtdeterministischer Raumkomplexität $S(n)$.
Alle diese Familien von Sprachen heißen Komplexitätsklassen. ∎

Def. 5.1.3: Die Sprachen, welche in deterministischer polynomialer Zeit erkannt werden können, bilden eine natürliche und wichtige Klasse, welche wir mit $\mathcal{P}$ bezeichnen,

$$\mathcal{P} = \bigcup_{i \geq 1} \mathrm{DTIME}(n^i).$$

Die Klasse der Sprachen, welche in nichtdeterministischer polynomialer Zeit erkannt werden können, bezeichen wir mit $\mathcal{NP}$,

$$\mathcal{NP} = \bigcup_{i \geq 1} \mathrm{NTIME}(n^i).$$

Weiters definieren wir

$$\mathrm{PSPACE} = \bigcup_{i \geq 1} \mathrm{DSPACE}(n^i)$$

und

$$\mathrm{NSPACE} = \bigcup_{i \geq 1} \mathrm{NSPACE}(n^i). \qquad \blacksquare$$

Das Problem der Hamiltonschen Zyklen P_{hc} lautet folgendermaßen: *"Gibt es im Graphen G einen Zyklus, der jede Ecke genau einmal enthält?"* P_{hc} ist offensichtlich in $\mathcal{NP}$. Ein einfacher nichtdeterministischer Algorithmus ist etwa: errate die Ecken des Zyklus und verifiziere, daß sie tatsächlich einen Hamiltonschen Zyklus bilden. Es scheint jedoch keinen deterministischen Algorithmus mit polynomialer Zeitkomplexität zu geben, der P_{hc} entscheidet.

Der Unterschied zwischen $\mathcal{P}$ und $\mathcal{NP}$ ist vergleichbar mit dem Unterschied zwischen dem effizienten Auffinden eines Beweises für eine Behauptung (etwa "dieser Graph hat einen Hamiltonschen Zyklus") und dem effizienten Nachprüfen eines Beweises (etwa zu prüfen, ob eine gegebene Eckenfolge einen Hamiltonschen Zyklus darstellt). Intuitiv nimmt man wohl an, daß es einfacher ist einen gegebenen Beweis zu überprüfen als einen solchen aufzufinden. Mit Sicherheit wissen wir das aber nicht.

Man kann zeigen, daß PSPACE = NSPACE [1)]. Es gilt offensichtlich $\mathcal{P} \subseteq \mathcal{NP} \subseteq$ PSPACE. Von keiner dieser Inklusionen weiß man aber, ob sie echt ist.

In Kapitel 4 haben wir die Methode der Reduktion benutzt, um Probleme (Sprachen) als unentscheidbar zu erweisen. Dazu betrachteten wir eine Reduktionsfunktion g, sodaß $x \in L' \iff g(x) \in L$. Indem man die Reduktionsfunktion g einschränkt auf einfach zu berechnende Funktionen, ist es oft möglich nachzuweisen, daß

[1)] W.J. Savitch: *"Relations between nondeterministic and deterministic tape complexities"*, *J. Computer and Systems Sciences* 4/2, 177–192 (1970).

eine Sprache L in einer der Klassen $\mathcal{P}$, $\mathcal{NP}$ oder PSPACE liegt. Wir werden uns speziell für zwei Arten von Reduktionen interessieren: Polynom–Zeit Reduktionen und log–Raum Reduktionen.

Def. 5.1.4: Seien L, L' Sprachen. L' ist *Polynom–Zeit reduzierbar auf* L, wenn es eine Turing–Maschine mit polynomialer Zeitkomplexität gibt, welche immer stoppt und für jede Eingabe x eine Ausgabe y erzeugt, sodaß $x \in L' \Longleftrightarrow y \in L$. ■

Satz 5.1.1: Sei L' Polynom–Zeit reduzierbar auf L. Dann
(a) $L \in \mathcal{P} \Longrightarrow L' \in \mathcal{P}$,
(b) $L \in \mathcal{NP} \Longrightarrow L' \in \mathcal{NP}$.

Beweis: (a) Sei $L \in \mathcal{P}$. Sei die Zeitkomplexität der Reduktion beschränkt durch das Polynom $p_1(n)$ und sei L erkennbar in Zeit $p_2(n)$, p_2 ein Polynom. Für ein gegebenes $x \in L'$ der Länge n berechnet man nun mittels der Polynom–Zeit Reduktion y. Da pro Rechenschritt in der Reduktion höchstens 1 Symbol ausgeschrieben werden kann, ist $|\ y\ | \leq p_1(n)$. Der Test "$y \in L$" kann in Zeit $p_2(p_1(n))$ ausgeführt werden. Um also "$x \in L'$" zu entscheiden, braucht man $p_1(n) + p_2(p_1(n))$ Zeit; das ist polynomial in n. Somit ist $L' \in \mathcal{P}$.

Der Beweis von (b) verläuft analog. ■

Def. 5.1.5: Ein *log–Raum Übersetzer* ist eine (mehrbändige) immer haltende Turing–Maschine mit einem read–only Eingabeband, einem write–only Ausgabeband, auf dem der LS–Kopf niemals nach links bewegt wird, und $\log n$ Arbeitsspeicher (Bandzellen) für eine Eingabe der Länge n.
Eine Sprache L' ist *log–Raum reduzierbar auf* L, wenn es einen log–Raum Übersetzer gibt, welcher für jede Eingabe x eine Ausgabe y erzeugt, sodaß $x \in L' \Longleftrightarrow y \in L$. ■

Satz 5.1.2: Sei L' log–Raum reduzierbar auf L. Dann
(a) $L \in \mathcal{P} \Longrightarrow L' \in \mathcal{P}$,
(b) $L \in \text{DSPACE}(\log^k n) \Longrightarrow L' \in \text{DSPSACE}(\log^k n)$.

Beweis: (a) Wenn wir zeigen können, daß eine log–Raum Reduktion nicht mehr als polynomiale Zeit braucht, dann folgt die Behauptung aus Satz 5.1.1(a). Da das Ausgabeband die Berechnung nicht beeinflussen kann, ist das Produkt P der Anzahl der Zustände $|\ Q\ |$, Speicherbandinhalte und Positionen der Eingabe und Speicher LS–Köpfe eine obere Schranke für die Anzahl der Rechenschritte, die ausgeführt werden können, bevor der log–Raum Übersetzer in eine Schleife kommt, was per Definition ausgeschlossen ist. Hat nun das Speicherband die Länge $\log n$ und ist A das Bandalphabet, so ist

$$P = \mid Q \mid \cdot \mid A \mid^{\log n} \cdot n \cdot \log n \leq \mid Q \mid \cdot n^{\log \mid A \mid} \cdot n \cdot n = \mid Q \mid \cdot n^{2+\log \mid A \mid} \ .$$

Die Anzahl der Schritte in der Übersetzung ist also polynomial in der Länge n der Eingabe.

(b) Dazu verweisen wir auf [HU] Lemma 13.2(c). ■

5.2. Vollständige Probleme

Wie bereits oben erwähnt, ist nicht bekannt, ob $\mathcal{P} = \mathcal{NP}$. Wollte man etwa versuchen $\mathcal{P} \neq \mathcal{NP}$ zu zeigen, so liegt es nahe nach einem "schwierigsten" Problem in $\mathcal{NP}$ zu suchen. Dieses wäre ein geeigneter Kandidat für $\mathcal{NP} \setminus \mathcal{P}$. Intuitiv könnte man etwa sagen, eine Sprache L_0 ist schwierig, wenn jede andere Sprache in $\mathcal{NP}$ auf L_0 reduzierbar ist bezüglich einer einfach zu berechnenden Reduktion. Ob eine Sprache "schwierig" ist, hängt natürlich von der Art der Reduktion ab.

Def. 5.2.1: Sei $\mathcal{C}$ eine Klasse von Sprachen. Eine Sprache L ist *schwierig für* $\mathcal{C}$ *bezüglich Polynom–Zeit (*bzw. *log–Raum) Reduktion*, wenn jede Sprache in $\mathcal{C}$ Polynom–Zeit (bzw. log–Raum) reduzierbar ist auf L. (L ist nicht notwendigerweise in $\mathcal{C}$.)

L ist *vollständig für* $\mathcal{C}$ *bezüglich Polynom–Zeit (*bzw. *log–Raum) Reduktion*, wenn L schwierig ist für $\mathcal{C}$ bzgl. Polynom–Zeit (bzw. log–Raum) Reduktion und $L \in \mathcal{C}$.

Für zwei häufig verwendete Begriffe führen wir Abkürzungen ein: L ist *NP–vollständig (NP–schwierig)* falls L vollständig (schwierig) ist für $\mathcal{NP}$ bzgl. log–Raum Reduktion [2). L ist *PSPACE–vollständig (PSPACE–schwierig)*, wenn L vollständig (schwierig) ist für PSPACE bzgl. Polynom–Zeit Reduktion. ■

Viele NP–vollständige Probleme sind sehr natürlich und effiziente Lösungen sind seit langer Zeit gesucht worden. Für keines dieser Probleme ist aber eine polynomiale Lösung bekannt. Wäre eines dieser Probleme in $\mathcal{P}$, so wären sie alle in $\mathcal{P}$ (und überdies $\mathcal{P} = \mathcal{NP}$). Daß eines dieser Probleme eine polynomiale Lösung hat, ist also sehr unwahrscheinlich. Im folgenden gehen wir auf ein NP–vollständiges Problem, das Erfüllbarkeitsproblem, näher ein.

Def. 5.2.2: Ein *Boolescher Ausdruck* E ist auf die übliche Weise aufgebaut aus *Variablen* $x_1, x_2, \ldots$, den *Konstanten* `T` (wahr) und `F` (falsch), Klammern, und den *Operatoren* $\neg$ (nicht), $\wedge$ (und) und $\vee$ (oder). Anstatt $\neg x$ schreiben wir oft $\overline{x}$ für eine Variable x.

Eine *Klause* ist ein Boolescher Ausdruck der Form $\alpha_1 \vee \ldots \vee \alpha_k$, wobei jedes α_i, $1 \leq i \leq k$, ein *Literal* ist, also entweder eine Konstante oder x oder $\neg x$ für eine Variable x.

---

2) Wir folgen dabei [HU].

Ein Boolescher Ausdruck E ist in *konjunktiver Normalform (CNF)*, wenn er von der Gestalt $E_1 \wedge \ldots \wedge E_l$ ist, wobei jeder Boolescher Ausdruck E_i, $1 \leq i \leq l$, eine Klause ist. E ist in *3–CNF*, wenn jede Klause genau 3 Literale enthält. ■

Def. 5.2.3: Ein Boolescher Ausdruck E heißt *erfüllbar*, wenn es eine Zuweisung von $\{\mathtt{T}, \mathtt{F}\}$ auf die Variablen in E gibt, sodaß E den Wert $\mathtt{T}$ erhält. Das *Erfüllbarkeitsproblem* besteht darin, für einen Boolescher Ausdruck festzustellen, ob er erfüllbar ist. ■

Beispiel 5.2.1: Der Boolesche Ausdruck $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$ ist erfüllbar; eine erfüllende Zuweisung ist etwa $\{\mathtt{T} \to x_1, \mathtt{F} \to x_2\}$. Dagegen ist der Boolesche Ausdruck $x_1 \wedge \neg x_2 \wedge (\neg x_1 \vee x_2)$ unerfüllbar. ■

Def. 5.2.4: Dem Erfüllbarkeitsproblem kann man auf folgende Weise eine Sprache L_{sat} (satisfiability) zuordnen. Als Alphabet wählt man $\{\mathtt{T}, \mathtt{F}, x, \neg, \wedge, \vee, 0, 1, (,)\}$. Die Funktion *code* bildet ab

$$\begin{array}{rcl} \mathtt{T}, \mathtt{F}, \neg, \wedge, \vee, (,) & \mapsto & \mathtt{T}, \mathtt{F}, \neg, \wedge, \vee, (,), \\ x_i & \mapsto & x b_1 \ldots b_r, \\ & & \text{wobei } b_1 \ldots b_r \text{ die Binärdarstellung von } i \text{ ist.} \end{array}$$

code wird auf natürliche Weise auf Boolesche Ausdrücke ausgedehnt.

$$L_{sat} = \{code(E) \mid E \text{ ist ein erfüllbarer Boolescher Ausdruck}\}.$$ ■

Beispiel 5.2.2: So ist etwa $(x1 \vee x10) \wedge (\neg x10 \vee x1) \in L_{sat}$ und $x1 \wedge \neg x10 \wedge (\neg x1 \vee x10) \notin L_{sat}$. ■

Ist E ein Boolescher Ausdruck der Länge n, so ist die Länge von $code(E) \leq \lceil n \log n \rceil$. Im folgenden werden wir nicht unterscheiden zwischen der Länge n von E und der Länge $n \log n$ von $code(E)$. Dadurch können aber keine Probleme auftreten, da wir log–Raum Reduktionen betrachten werden und $\log(n \log n) \leq 2 \cdot \log n$.

Satz 5.2.1 (Satz von Cook [3]): L_{sat} (also das Erfüllbarkeitsproblem) ist NP–vollständig.

[3] S.A. Cook: *"The complexity of theorem proving procedures", Proc. 3rd Annual ACM Symposium on the Theory of Computing,* 151–158 (1971).

Beweis: Zunächst ist natürlich klar, daß $L_{sat} \in \mathcal{NP}$. Um festzustellen, ob ein Ausdruck der Länge n erfüllbar ist, wählt man nichtdeterministisch Werte für alle Variablen und evaluiert den Ausdruck. Zur Evaluierung braucht man nur polynomiale Zeit.

Um zu zeigen, daß jede Sprache in $\mathcal{NP}$ auf L_{sat} reduzierbar ist, geben wir für jede nichtdeterministische Turing–Maschine M, deren Zeitkomplexität ein Polynom $p(n)$ ist, einen log–Raum Algorithmus $A_{p(n)}$ an, der für ein Eingabewort x einen Booleschen Ausdruck E_x berechnet, der genau dann erfüllbar ist, wenn M das Wort x akzeptiert. Wir geben nun an, wie dieser Ausdruck E_x erzeugt wird.

Sei $\#\beta_0\#\beta_1\#\cdots\#\beta_{p(n)}$ eine Berechnung von M. Dabei sei jedes β_i eine Konfiguration von M mit genau $p(n)$ Symbolen (wenn notwendig fügt man rechts zum letzten Bandsymbol noch einige Blanks hinzu). Wird die Eingabe bereits vor dem $p(n)$–ten Schritt akzeptiert, so gestatten wir eine Wiederholung der akzeptierenden Konfiguration, sodaß jede Berechnung aus genau $p(n)+1$ Konfigurationen besteht. In jeder Konfiguration fassen wir den Zustand und das aktuelle Symbol zu einem einzigen Symbol zusammen. Dieses zusammengesetzte Symbol in der i–ten Konfiguration enthalte zusätzlich eine Nummer m, welche die nichtdeterministische Wahl anzeigt, durch welche die $(i+1)$–te aus der i–ten Konfiguration entsteht. Dazu werden die nichtdeterministischen Wahlmöglichkeiten von M bei gegebenem Zustand und Bandsymbol durchnummeriert.

Für jedes Symbol X, das in der Berechnung vorkommen kann, und für jedes i, $0 \leq i < (p(n)+1)^2$, definieren wir eine Boolesche Variable c_{iX}, welche angibt, ob X das i–te Symbol in der Berechnung ist. (Das 0–te Symbol ist #.)

Wir konstruieren nun den Ausdruck E_x so, daß er genau dann für eine gegebene Zuweisung zu den Variablen c_{iX} wahr ist, wenn die c_{iX}'s mit Wert `T` einer akzeptierenden Berechnung entsprechen. Der Ausdruck E_x sagt folgendes aus:

(1) Die c_{iX}'s mit Wert `T` entsprechen einer Folge von Symbolen, für jedes i ist also genau ein c_{iX} wahr.
(2) Die Konfiguration β_0 ist eine initiale Konfiguration von M mit Eingabe x.
(3) Die letzte Konfiguration enthält einen Endzustand.
(4) Jede Konfiguration folgt aus der vorangegangenen durch den angegebenen Schritt von M.

Der Ausdruck E_x ist die Konjunktion von vier Ausdrücken, deren jeder eine dieser Bedingungen erzwingt. Der Ausdruck, der (1) erzwingt, ist

$$\bigwedge_i \Big[\bigvee_X c_{iX} \wedge \neg\Big(\bigvee_{X \neq Y} (c_{iX} \wedge c_{iY}) \Big) \Big].$$

Sei $x = a_1 a_2 \cdots a_n$. Der Ausdruck, der (2) erzwingt, ist seinerseits die Konjunktion der folgenden Unterausdrücke (i) — (iv).

(i) $c_{0\#} \wedge c_{p(n)+1,\#}$. Die Symbole an der 0–ten und $(p(n)+1)$–ten Stelle sind #.

(ii) $c_{1Y_1} \vee c_{1Y_2} \vee \cdots \vee c_{1Y_k}$, wobei $Y_1, Y_2, \ldots, Y_k$ alle die zusammengesetzten Symbole sind, welche das Bandsymbol a_1, den Anfangszustand q_0, und die Nummer eines erlaubten Schrittes von M beim Zustand q_0 und aktuellen Symbol a_1 darstellen. Dieser Teilausdruck behauptet, daß das erste Symbol von β_0 korrekt ist.

(iii) $\bigwedge_{2 \le i \le n} c_{ia_i}$. Die Symbole von der zweiten bis zur n–ten Stelle von β_0 sind korrekt.

(iv) $\bigwedge_{n < i \le p(n)} c_{i\sqcup}$. Die restlichen Symbole von β_0 sind Blank.

Die Formel, welche (3) erzwingt, ist

$$\bigvee_{p(n)(p(n)+1) < i < (p(n)+1)^2} \Big(\bigvee_{X \in F} c_{iX} \Big),$$

wobei F die Menge der zusammengesetzten Symbole ist, welche einen Endzustand enthalten.

Wir kommen nun zum Ausdruck, welcher (4) erzwingen soll. Jedes Symbol in β_i kann hergeleitet werden vom entsprechenden Symbol in β_{i-1} und den Symbolen links und rechts davon (wovon eines # sein kann). Enthält die vorangegangene Konfiguration einen Endzustand, so sind die gegenwärtige und die vorangegangene Konfiguration gleich. Es ist also nicht schwierig ein Prädikat $f(W,X,Y,Z)$ zu definieren, das genau dann erfüllt ist, wenn das Symbol Z in Position j einer Konfiguration vorkommen kann, vorausgesetzt daß W, X und Y die Symbole in den Positionen $j-1$, j und $j+1$ der vorangegangenen Konfiguration sind (W ist # falls $j = 1$ und Y ist # falls $j = p(n)$). Der Einfachheit halber sei auch $f(W, \#, X, \#)$ erfüllt; auf diese Weise können die Markierungen zwischen den Konfigurationen genau so behandelt werden wie die Symbole in den Konfigurationen. Wir erhalten somit für (4) den Ausdruck

$$\bigwedge_{p(n) < j < (p(n)+1)^2} \Big(\bigvee_{\substack{W,X,Y,Z \text{ sodass} \\ f(W,X,Y,Z)}} (c_{j-p(n)-2,W} \wedge c_{j-p(n)-1,X} \wedge c_{j-p(n),Y} \wedge c_{jZ}) \Big).$$

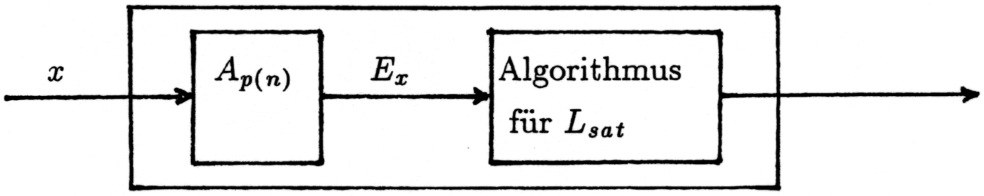
Ausgehend von einer akzeptierenden Berechnung von M auf x ist es leicht, Wahrheitswerte für die Variablen c_{iX} zu finden, welche den Ausdruck E_x wahr machen. Ist umgekehrt eine Zuordnung von Wahrheitswerten gegeben, welche E_x wahr macht, so garantieren die oben definierten vier Ausdrücke, daß eine akzeptierende Berechnung von M auf x existiert.

Der Ausdruck E_x hat die Länge $\mathcal{O}(p^2(n))$ und ist einfach genug, daß ihn eine log–Raum Turing–Maschine mit Eingabe x erzeugen kann. Diese Turing–Maschine braucht nur genügend Speicherplatz, um bis $(p(n)+1)^2$ zählen zu können. Wegen $\log(p(n)) = c \cdot \log n$ für eine Konstante c, genügen dazu $\mathcal{O}(\log n)$ Speicherzellen.

Somit haben wir gezeigt, daß jede Sprache in $\mathcal{NP}$ log–Raum reduzierbar ist auf L_{sat}, also L_{sat} NP–vollständig ist. ■

Man kann zeigen, daß L_{csat}, das Erfüllbarkeitsproblem für CNFs, NP–vollständig ist, ja sogar L_{3-csat}, das Erfüllbarkeitsproblem für 3–CNFs.

Hätten wir also einen deterministischen Algorithmus mit polynomialer Zeitkomplexität zur Entscheidung von L_{sat}, so könnten wir ihn dazu benutzen, jede Sprache in $\mathcal{NP}$ in polynomialer Zeit zu erkennen. Sei L eine Sprache, die von einer nichtdeterministischen Turing–Maschine M mit Zeitkomplexität $p(n)$ erkannt wird und sei $A_{p(n)}$ der im Beweis zu Satz 5.2.1 definierte log–Raum (also Polynom–Zeit) Übersetzer. Ein deterministischer Polynom–Zeit Algorithmus zur Erkennung von L wäre dann



Aus der Existenz eines deterministischen Polynom–Zeit Algorithmus für L_{sat} würde also $\mathcal{P} = \mathcal{NP}$ folgen.

Einige weitere NP–vollständige Probleme sind

— das Problem der Hamiltonschen Zyklen;

— das Problem der *ganzzahligen linearen Programmierung*: für $A \in \mathbf{Z}^{m \times n}, b \in \mathbf{Z}^m$, entscheide ob es einen Vektor $x \in \mathbf{Z}^n$ gibt, sodaß $Ax \geq b$;

— das *chromatische Zahlenproblem*: für einen gegebenen Graphen G und eine ganze Zahl k, entscheide ob G mit k Farben gefärbt werden kann, sodaß keine zwei benachbarten Ecken dieselbe Farbe haben;

— das *Problem des Handlungsreisenden*: zu einem gegebenen Graphen mit gewichteten Kanten bestimme den Hamiltonschen Zyklus mit kleinstem Gewicht. Um das Problem als Sprache ausdrücken zu können, verlangt man, daß die Gewichte ganze Zahlen seien und fragt dann, ob es einen Hamiltonschen Zyklus mit Gewicht $\leq k$ gibt.

— Das *Partitionsproblem*: existiert zu einer gegebenen Liste von ganzen Zahlen $i_1, i_2, \ldots, i_k$ eine Unterliste, deren Summe $\frac{1}{2}(i_1 + i_2 + \cdots + i_k)$ ist. Man beachte, daß die Länge einer Instanz nicht $i_1 + i_2 + \cdots + i_k$ ist, sondern die Summe der Längen der i_j's.

Für eine intensive Untersuchung dieser und vieler anderer NP–vollständiger Probleme verweisen wir auf das Buch von Garey und Johnson [5].

Man kann sich dem $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Problem auch auf folgende Weise nähern.

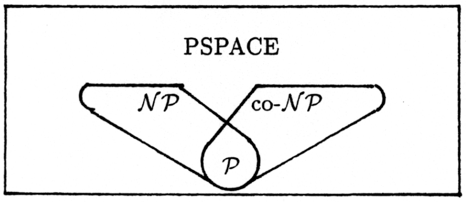
Def. 5.2.5 Die Komplexitätsklasse co-$\mathcal{NP}$ enthält alle diejenigen Sprachen, deren Komplement in $\mathcal{NP}$ ist, also co–$\mathcal{NP} = \{L | \overline{L} \in \mathcal{NP}\}$. ∎

Es ist nicht bekannt, ob $\mathcal{NP}$ abgeschlossen ist bezüglich der Bildung des Komplements, also ob co–$\mathcal{NP} = \mathcal{NP}$. Sollte das nicht der Fall sein, so wäre $\mathcal{P} \neq \mathcal{NP}$, denn $\mathcal{P}$ ist abgeschlossen bezüglich der Bildung des Komplements. Von keinem NP–vollständigen Problem weiß man, daß sein Komplement ebenfalls in $\mathcal{NP}$ ist. Um etwa die Nichterfüllbarkeit eines booleschen Ausdrucks mit n Variablen zu bestimmen, scheint es notwendig zu sein alle 2^n Zuweisungen zu testen, auch wenn der Algorithmus nicht–deterministisch ist. Sollte von einem NP–vollständigen Problem entdeckt werden, daß auch sein Komplement in $\mathcal{NP}$ gibt, so wäre $\mathcal{NP}$ abgeschlossen bezüglich der Bildung des Komplements.

Satz 5.2.2 $\mathcal{NP}$ ist abgeschlossen bezüglich Bildung des Komplements genau dann, wenn es ein NP–vollständiges Problem gibt, dessen Komplement in $\mathcal{NP}$ ist.

Beweis: Die Implikation "$\Longrightarrow$" ist offensichtlich. Für die Implikation "$\Longleftarrow$" sei S ein NP–vollständiges Problem. Wir nehmen an, $\overline{S}$ wäre in $\mathcal{NP}$. Da jedes L in $\mathcal{NP}$ log–Raum reduzierbar ist auf S, ist jedes $\overline{L}$ log–Raum reduzierbar auf $\overline{S}$. Also ist $\overline{L}$ in $\mathcal{NP}$. ∎

Die Komplexitätsklassen $\mathcal{P}$, $\mathcal{NP}$, co-$\mathcal{NP}$, PSPACE stehen also in folgender Relation



[5] M.R. Garey und D.S. Johnson: *"Computers and Intractability: A Guide to the Theory of NP–Completeness"*, H. Freeman, San Francisco (1978).

5.3. The P versus NP Problem (by S. Cook)

Der folgende Artikel von S. Cook ist eine in [Coo] veröffentlichte Beschreibung eines der wichtigsten offenen Probleme in Mathematik und Informatik.

The P versus NP Problem

Stephen Cook
University of Toronto

1. Statement of the Problem

The **P** versus **NP** problem is to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time. To define the problem precisely it is necessary to give a formal model of a computer. The standard computer model in computability theory is the Turing machine, introduced by Alan Turing in 1936 [Tur36]. Although the model was introduced before physical computers were built, it nevertheless continues to be accepted as the proper computer model for the purpose of defining the notion of *computable function.*

Informally the class **P** is the class of decision problems solvable by some algorithm within a number of steps bounded by some fixed polynomial in the length of the input. Turing was not concerned with the efficiency of his machines, but rather his concern was whether they can simulate arbitrary algorithms given sufficient time. However it turns out Turing machines can generally simulate more efficient computer models (for example machines equipped with many tapes or an unbounded random access memory) by at most squaring or cubing the computation time. Thus **P** is a robust class, and has equivalent definitions over a large class of computer models. Here we follow standard practice and define the class **P** in terms of Turing machines.

Formally the elements of the class **P** are languages. Let Σ be a finite alphabet (that is, a finite nonempty set) with at least two elements, and let Σ^* be the set of finite strings over Σ. Then a *language over* Σ is a subset L of Σ^*. Each Turing machine M has an associated *input alphabet* Σ. For each string w in Σ^* there is a computation associated with M with input w. (The notions of Turing machine and computation are defined formally in the appendix.) We say that M *accepts* w if this computation terminates in the accepting state. Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate. The *language accepted by* M, denoted $L(M)$, has associated alphabet Σ and is defined by

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

We denote by $t_M(w)$ the number of steps in the computation of M on input w (see the Appendix). If this computation never halts, then $t_M(w) = \infty$. For $n \in \mathbb{N}$ we denote by $T_M(n)$ the *worst case run time* of M; that is

$$T_M(n) = \max\{t_M(w) \mid w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n. We say that M *runs in polynomial time* if there exists k such that for all n, $T_M(n) \leq n^k + k$. Now we define the class $\mathbf{P}$ of languages by

$$\mathbf{P} = \{L \mid L = L(M) \text{ for some Turing machine } M \text{ which runs in polynomial time}\}$$

The notation **NP** stands for "nondeterministic polynomial time", since originally **NP** was defined in terms of nondeterministic machines (that is, machines that have more than one possible move from a given configuration). However now it is customary to give an equivalent definition using the notion of a *checking relation*, which is simply a binary relation $R \subseteq \Sigma^* \times \Sigma_1^*$ for some finite alphabets Σ and Σ_1. We associate with each such relation R a language L_R over $\Sigma \cup \Sigma_1 \cup \{\#\}$ defined by

$$L_R = \{w\#y \mid R(w, y)\}$$

where the symbol $\#$ is not in Σ. We say that R is *polynomial-time* iff $L_R \in \mathbf{P}$.

Now we define the class **NP** of languages by the condition that a language L over Σ is in **NP** iff there is $k \in \mathbb{N}$ and a polynomial-time checking relation R such that for all $w \in \Sigma^*$,

$$w \in L \Longleftrightarrow \exists y(|y| \leq |w|^k \text{ and } R(w, y))$$

where $|w|$ and $|y|$ denote the lengths of w and y, respectively.

Problem Statement: Does $\mathbf{P} = \mathbf{NP}$?

It is easy to see that the answer is independent of the size of the alphabet Σ (we assume $|\Sigma| \geq 2$), since strings over an alphabet of any fixed size can be efficiently coded by strings over a binary alphabet. (For $|\Sigma| = 1$ the problem is still open, although it is possible that $\mathbf{P} = \mathbf{NP}$ in this case but not in the general case.)

It is trivial to show that $\mathbf{P} \subseteq \mathbf{NP}$, since for each language L over Σ, if $L \in \mathbf{P}$ then we can define the polynomial-time checking relation $R \subseteq \Sigma^* \cup \Sigma^*$ by

$$R(w, y) \iff w \in L$$

for all $w, y \in \Sigma^*$.

Here are two simple examples, using decimal notation to code natural numbers: The set of perfect squares is in $\mathbf{P}$ and the set of composite numbers is in $\mathbf{NP}$ (and not known to be in $\mathbf{P}$). For the latter, the associated polynomial time checking relation R is given by

$$R(\overline{a}, \overline{b}) \iff 1 < b < a \text{ and } b|a \tag{1}$$

In general the decimal notation for a natural number c is denoted by $\overline{c}$.

2. History and Importance

The importance of the $\mathbf{P}$ vs $\mathbf{NP}$ questions stems from the successful theories of $\mathbf{NP}$-completeness and complexity-based cryptography, as well as the potentially stunning practical consequences of a constructive proof of $\mathbf{P} = \mathbf{NP}$.

The theory of $\mathbf{NP}$-completeness has its roots in computability theory, which originated in the work of Turing, Church, Gödel, and others in the 1930's. The computability precursors of the classes $\mathbf{P}$ and $\mathbf{NP}$ are the classes of decidable and c.e. (computably enumerable) languages, respectively. We say that a language L is c.e. (or *semi-decidable*) iff $L = L(M)$ for some Turing machine M. We say that L is *decidable* iff $L = L(M)$ for some Turing machine M which satisfies the condition that M halts on all input strings w. There is an equivalent definition of c.e. which brings out its analogy with $\mathbf{NP}$, namely L is c.e. iff there is a computable "checking relation" $R(x, y)$ such that $L = \{x \mid \exists y R(x, y)\}$.

Using the notation $\langle M \rangle$ to denote a string describing a Turing machine M, we define the Halting Problem HP as follows:

$$HP = \{\langle M \rangle \mid M \text{ is a Turing machine which halts on input } \langle M \rangle\}$$

Turing used a simple diagonal argument to show that HP is not decidable. On the other hand, it is not hard to show that HP is c.e.

Of central importance in computability theory is the notion of reducibility, which Turing defined roughly as follows: A language L_1 is *Turing reducible* to a language L_2 iff there is an oracle Turing machine M which accepts L_1, where M is allowed to make membership queries of the form $x \in L_2$? which are correctly answered by an "oracle" for L_2. Later the more restricted notion of many-one reducibility ($\leq_m$) was introduced and defined as follows.

Definition 1: Suppose that L_i is a language over Σ_i, $i = 1, 2$. Then $L_1 \leq_m L_2$ iff there is a (total) computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1 \iff f(x) \in L_2$, for all $x \in \Sigma_1^*$.

It is easy to see that if $L_1 \leq_m L_2$ and L_2 is decidable, then L_1 is decidable. This fact provides an important tool for showing undecidability; for example if $HP \leq_m L$ then L is undecidable.

The notion of **NP**-complete is based on the following notion from computability theory:

Definition 2: A language L is c.e.-*complete* iff L is c.e., and $L' \leq_m L$ for every c.e. language L'.

It is easy to show that HP is c.e.-complete. It turns out that most "natural" undecidable c.e. languages are in fact c.e.-complete. Since $\leq_m$ is transitive, to show that a c.e. language L is c.e. complete is suffices to show that $HP \leq_m L$.

The notion of polynomial-time computation was introduced in the 1960's by Cobham [Cob64] and Edmonds [Edm65] as part of the early development of computational complexity theory (although earlier von Neumann [vN53] in 1953 distinguished between polynomial time and exponential-time algorithms). Edmonds called polynomial-time algorithms "good algorithms", and linked them to tractable algorithms.

It has now become standard in complexity theory to identify polynomial-time with feasible, and here we digress to discuss this point. It is of course not literally true that every polynomial-time algorithm can be feasibly executed on small inputs; for example a computer program requiring n^{100} steps could never be executed on an input even as small as $n = 10$. Here is a more defensible statement (see [Coo91]):

Feasibility Thesis: A natural problem has a feasible algorithm iff it has a polynomial-time algorithm.

Examples of natural problems that have both feasible and polynomial-time algorithms abound: Integer arithmetic, linear algebra, network flow, linear programming, many graph problems (connectivity, shortest path, minimum spanning tree, bipartite matching) etc. On the other hand the deep results of Robertson and Seymour [RS95] provide a potential source of counterexamples to the thesis: They prove that every minor-closed family of graphs can be recognized in polynomial time (in fact in time $O(n^3)$), but the algorithms supplied by their method have such huge constants that they are not feasible. However each potential counter example can be removed by finding a feasible algorithm for it. For example a feasible recognition algorithm is known for the class of planar graphs, but none is currently known for the class of graphs embeddable in $\mathbb{R}^3$ with no two cycles linked. (Both examples are minor-closed families.) Of course the words "natural" and "feasible" in the thesis above should be explained; generally we do not consider a class with a parameter as natural, such as the set of graphs embeddable on a surface of genus k, $k > 1$.

We mention two concerns related to the "only if" direction of the thesis. The first comes from randomized algorithms. We discuss at the end of section 3 the possibility that a source of random bits might be used to greatly reduce the recognition time required for some language. Note however that it is not clear whether a truly random source exists in nature. The second concern comes from Quantum computers. This computer model incorporates the idea of superposition of states from quantum mechanics, and allows a potential exponential speed-up of some computations over Turing machines. For example, Shor [Sho97] has shown that some quantum computer algorithm is able to factor integers in polynomial time, but no polynomial-time integer factoring algorithm is known for Turing machines. However physicists have so far been unable to build a quantum computer that can handle more than a half-dozen bits, so this threat to the feasibility thesis is hypothetical at present.

Returning to the historical treatment of complexity theory, in 1971 the present author [Coo71] introduced a notion of **NP**-completeness as a polynomial-time analog of c.e. completeness, except that the reduction used was a polynomial- time analog of Turing reducibility rather than of many-one reducibility. The main results in [Coo71] are that several natural problems, including Satisfiability and 3-SAT (defined below) and subgraph isomorphism are **NP**-complete. A year later Karp [Kar72] used these completeness results to show that 20 other natural problems are **NP**-complete, thus forcefully demonstrating the importance of the subject. Karp also introduced the now standard notation **P** and **NP** and redefined **NP**-completeness using the polynomial-time analog of many-one reducibility, a definition which has become standard. Meanwhile Levin [Lev73] independently of Cook and Karp defined the notion of "universal search problem", similar to **NP**-complete problem, and gave six examples, including Satisfiability.

The standard definitions concerning **NP**-completeness are close analogs of Definitions 1 and 2 above.

Definition 3: Suppose that L_i is a language over Σ_i, $i = 1, 2$. Then $L_1 \le_p L_2$ (L_1 is p-reducible to L_2) iff there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1 \iff f(x) \in L_2$, for all $x \in \Sigma_1^*$.

Definition 4: A language L is **NP**-*complete* iff L is in **NP**, and $L' \le_p L$ for every language L' in **NP**.

The following proposition is easy to prove: Part (b) uses the transitivity of $\le_p$, and part (c) follows from part (a).

Proposition 1: (a) If $L_1 \le_p L_2$ and $L_2 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$.

(b) If L_1 is **NP**-complete, $L_2 \in \mathbf{NP}$, and $L_1 \le_p L_2$ then L_2 is **NP**-complete.

(c) If L is **NP**-complete and $L \in \mathbf{P}$, then **P**=**NP**.

Notice that parts (a) and (b) have close analogs in computability theory. The analog of part (c) is simply that if L is c.e.-complete then L is undecidable. Part (b) is the basic method for showing new problems are **NP**-complete, and part (c) explains why it is probably a waste of time looking for a polynomial-time algorithm for an **NP**-complete problem.

In practice a member of **NP** is expressed as a decision problem, and the corresponding language is understood to mean the set of strings coding YES instances to the decision problem using standard coding methods. Thus the problem Satisfiability is: Given a propositional formula F, determine whether F is satisfiable. To show that this is in **NP** we define the polynomial-time checking relation $R(x, y)$, which holds iff x codes a propositional formula F and y codes a truth assignment to the variables of F which makes F true. This problem was shown to be **NP**-complete in [Coo71] essentially by showing that for each polynomial-time Turing machine M which recognizes a checking relation $R(x, y)$ for an **NP** language L, there is a polynomial-time algorithm which takes as input a string x and produces a propositional formula F_x (describing the computation of M on input (x, y), with variables representing the unknown string y) such that F_x is satisfiable iff M accepts the input (x, y) for some y with $|y| \leq |x|^{O(1)}$.

An important special case of Satisfiability is 3-SAT, which was also shown to be **NP**-complete in [Coo71]. Instances of 3-SAT are restricted to formulas in conjunctive normal form with three literals per clause. For example, the formula

$$(P \vee Q \vee R) \wedge (\bar{P} \vee Q \vee \bar{R}) \wedge (P \vee \bar{Q} \vee S) \wedge (\bar{P} \vee \bar{R} \vee \bar{S}) \tag{2}$$

is a YES instance to 3-SAT since the truth assignment τ satisfies the formula, where $\tau(P) = \tau(Q) = True$ and $\tau(R) = \tau(S) = False$.

Many hundreds of **NP**-complete problems have been identified, including SubsetSum (given a set of positive integers presented in decimal notation, and a target T, is there a subset summing to T?), many graph problems (given a graph G, does G have a Hamiltonian cycle? Does G have a clique consisting of half of the vertices? Can the vertices of G be colored with three colors with distinct colors for adjacent vertices?). These problems give rise to many scheduling and routing problems with industrial importance. The book [GJ79] provides an excellent reference to the subject, with 300 **NP**-complete problems listed in the appendix.

Associated with each decision problem in **NP** there is a search problem, which is, given a string x, find a string y satisfying the checking relation $R(x, y)$ for the problem (or determine that x is a NO instance to the problem). Such a y is said to be a *certificate* for x. In the case of an **NP**-complete problem it is easy to see that the search problem can be efficiently reduced to the corresponding decision problem. In fact if $\mathbf{P} = \mathbf{NP}$ then the associated search problem for every **NP** problem has a polynomial-time algorithm. For example, an algorithm for the decision problem Satisfiability can be used to find a truth assignment τ satisfying a given satisfiable formula F by, for each variable P in F in turn, setting P to True in F or False in F, which ever case keeps F satisfiable.

The set of complements of **NP** languages is denoted co**NP**. The complement of an **NP**-complete language is thought not to be in **NP**; otherwise $\mathbf{NP} = \text{co}\mathbf{NP}$. The set TAUT of tautologies (propositional formulas true under all assignments) is the standard example of a co**NP**-complete language. The conjecture $\mathbf{NP} \neq \text{co}\mathbf{NP}$ is equivalent to the assertion that no formal proof system (suitably defined) for tautologies has short (polynomial-bounded) proofs for all tautologies [CR79]. This fact has motivated the development of a rich theory of proportional proof complexity [Kra95], one of whose goals is to prove superpolynomial lower bounds on the lengths of proofs for standard propositional proof systems.

There are interesting examples of **NP** problems not known to be either in **P** or **NP**-complete. One example is the set of composite numbers, mentioned in the first section, with checking relation (1). Here it is conjectured that there is no polynomial-time Turing reduction from the search problem (integer factoring) to the decision problem. Specifically, Miller [Mil76] showed how to determine in polynomial time whether a given number is composite, assuming the Extended Riemann Hypothesis, but a polynomial-time integer factoring algorithm is thought unlikely to exist. In fact an efficient factoring algorithm would break the RSA public key encryption scheme [ARS78] commonly used to allow (presumably) secure financial transactions over the internet.

The complement of the set of composite numbers (essentially the set of primes) was proved to be in **NP** by an interesting argument due to Pratt [Pra75], and hence is unlikely to be **NP**-complete.

There is an **NP** decision problem with complexity equivalent to that of integer factoring, namely

$$L_{fact} = \{\langle a, b\rangle \mid \exists d(1 < d < a \text{ and } d|b)\}$$

Given an integer $b > 1$, the smallest prime divisor of b can be found with about $\log_2 b$ queries to L_{fact}, using binary search. Using Pratt's theorem, it is easy to see that the complement of L_{fact} is also in **NP**: a certificate showing $\langle a, b\rangle$ is a non-instance of L_{fact} could be the complete prime decomposition of b, together with Pratt certificates proving that each of the prime factors is indeed prime. Thus it is considered unlikely that L_{fact} is **NP**-complete.

Computational complexity theory plays an important role in modern cryptography [Gol00]. The security of the internet, including most financial transactions, depend on complexity-theoretic assumptions such as the difficulty of integer factoring or of breaking DES (the Data Encryption Standard). If **P**= **NP** these assumptions are all false. Specifically, an algorithm solving 3-SAT in n^2 steps could be used to factor 200-digit numbers in a few minutes.

Although a practical algorithm for solving an **NP**-complete problem (showing **P** = **NP**) would have devastating consequences for cryptography, it would also have stunning practical consequences of a more positive nature, and not just because of the efficient solutions to the many **NP**-hard problems important to industry. For example, it would transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of reasonable length, since formal proofs can easily be recognized in polynomial time. Example theorems may well include all of the CMI prize problems. Although the formal proofs may not be initially intelligible to humans, the problem of finding intelligible proofs would be reduced to that of finding a recognition algorithm for intelligible proofs. Similar remarks apply to diverse creative human endeavors, such as designing airplane wings, creating physical theories, or even composing music. The question in each case is to what extent an efficient algorithm for recognizing a good result can be found. This is a fundamental problem in artificial intelligence, and one whose solution itself would be aided by the **NP**-solver by allowing easy testing of recognition theories.

Even if **P** $\neq$ **NP** it may still happen that every **NP** problem is susceptible to a polynomial-time algorithm which works on "most" inputs. This could render cryptography impossible and bring about most of the benefits of a world in which **P** = **NP**. This also motivates Levin's theory [Lev86, Imp95] of average case completeness, in which the **P** = **NP** question is replaced by the question of whether every **NP** problem with any reasonable probability distribution on its inputs can be solved in time polynomial on average.

In [Sma98] Smale lists the **P** vs **NP** question as problem 3 of mathematical problems for the next century. However Smale is interested not just in the classical version of the question, but also a version expressed in terms of the field of complex numbers. Here Turing machines must be replaced by a machine model that is capable of doing exact arithmetic and zero tests on arbitrary complex numbers. The **P** vs **NP** question is replaced by a question related to Hilbert's Nullstellensatz: Is there a polynomial-time algorithm which, given a set of k multivariate polynomials over $\mathbb{C}$, determines whether they have a common zero? See [BCSS98] for a development of complexity theory in this setting, including the intriguing result that the Mandelbrot set is undecidable.

The books by Papadimitriou [Pap94] and Sipser [Sip97] provide good introductions to mainstream complexity theory.

3. The Conjecture and Attempts to Prove it

Most complexity theorists believe that $\mathbf{P} \neq \mathbf{NP}$. Perhaps this can be partly explained by the potentially stunning consequences of $\mathbf{P} = \mathbf{NP}$ mentioned above, but there are better reasons. We explain these by considering the two possibilities in turn: $\mathbf{P} = \mathbf{NP}$ and $\mathbf{P} \neq \mathbf{NP}$.

Suppose first that $\mathbf{P} = \mathbf{NP}$ and consider how one might prove it. The obvious way is to exhibit a polynomial-time algorithm for 3-SAT or one of the other 1000 or so known **NP**-complete problems, and indeed many false proofs have been presented in this form. There is a standard toolkit available [CLR90] for devising polynomial-time algorithms, including the greedy method, dynamic programming, reduction to linear programming, etc. These are the subjects of a course on algorithms, typical in undergraduate computer science curriculums. Because of their importance in industry, a vast number of programmers and engineers have attempted to find efficient algorithms for **NP**-complete problems over the past 30 years, without success. There is similar strong motivation for breaking the cryptographic schemes which assume $\mathbf{P} \neq \mathbf{NP}$ for their security.

Of course it is possible that some nonconstructive argument, such as the Roberton/Seymour proofs mentioned earlier in conjunction with the Feasibility Thesis, could show that $\mathbf{P} = \mathbf{NP}$ without yielding any feasible algorithm for the standard **NP**-complete problems. However at present the best proven upper bound on an algorithm for solving 3-SAT is approximately 1.5^n, where

n is the number of variables in the input formula.

Suppose on the other hand that $\mathbf{P} \neq \mathbf{NP}$, and consider how one might prove it. There are two general methods that have been tried: diagonalization/reduction and Boolean circuit lower bounds.

The method of diaonalization with reduction has been used very successfully in computability theory to prove a host of problems undecidable, beginning with the Halting Problem. It has also been used successfully in complexity theory to prove super-exponential lower bounds for very hard decidable problems. For example, Presburger arithmetic, the first-order theory of integers under addition, is a decidable theory for which Fischer and Rabin [FR74] proved that any Turing machine deciding the theory must use at least $2^{2^{cn}}$ steps in the worst case, for some $c > 0$. In general, lower bounds using diagonalization and reduction relativize; that is they continue to apply in a setting in which both the problem instance and the solving Turing machine can make membership queries to an arbitrary oracle set A. However in [BGS75] it was shown that there is an oracle set A relative to which $\mathbf{P} = \mathbf{NP}$, suggesting that diagonalization/reduction cannot be used to separate these two classes. (There are nonrelativizing results in complexity theory, as will be mentioned below.) It is interesting to note that relative to a *generic* oracle, $\mathbf{P} \neq \mathbf{NP}$ [BI87, SCY97].

A *Boolean circuit* is a finite acyclic graph in which each non-input node, or *gate*, is labelled with a Boolean connective; typically from $\{AND, OR, NOT\}$. The input nodes are labeled with variables $x_1, ..., x_n$, and for each assignment of 0 or 1 to each variable the circuit computes a bit value at each gate, including the output gate, in the obvious way. It is not hard to see that if L is a language over $\{0, 1\}$ that is in $\mathbf{P}$, then there is a polynomial-size family of Boolean circuits $\langle B_n \rangle$ such that B_n has n inputs, and for each bit string w of length n, when w is applied to the n input nodes of B_n, then the output bit of B_n is 1 iff $w \in L$. In this case we say that $\langle B_n \rangle$ *computes* L.

Thus to prove $\mathbf{P} \neq \mathbf{NP}$ it suffices to prove a super-polynomial lower bound on the size of any family of Boolean circuits solving some specific $\mathbf{NP}$-complete problem, such as 3-SAT. Back in 1949 Shannon [Sha49] proved that for almost all Boolean functions $f : \{0, 1\}^n \to \{0, 1\}$, any Boolean circuit computing f requires at least $2^n/n$ gates. Unfortunately his counting argument gives no clue as to how to prove lower bounds for problems in $\mathbf{NP}$. Exponential lower bounds for **NP** problems have been proved for restricted circuit models, including monotone circuits [Raz85, AB87] and bounded depth circuits with unbounded fan-in gates [Has89, Smo87] (see [BS90]). However all attempts to find even super-linear lower bounds for unrestricted Boolean circuits for "explicitly given" Boolean functions have met with total failure; the best such lower bound proved so far is about $4n$. Razborov and Rudich [RR97] explain this failure by pointing out that all methods used so far can be classified as "Natural Proofs", and Natural Proofs for general circuit lower bounds are doomed to failure, assuming a certain complexity-theoretic conjecture asserting that strong pseudo-random number generators exist. Since such generators have been constructed assuming the hardness of integer factorization, we can infer the surprising result that a Natural Proof for a general circuit lower bound would give rise to a more efficient factoring algorithm than is currently known.

The failure of complexity theory to prove interesting lower bounds on a general model of computation is much more pervasive than the failure to prove $\mathbf{P} \neq \mathbf{NP}$. It is consistent with present knowledge that not only could Satisfiability have a polynomial-time algorithm, it could have a linear time algorithm on a multitape Turing machine. The same applies for all 21 problems mentioned in Karp's original paper [Kar72]. There are complexity class separations which we know exist but cannnot prove. For example, consider the sequence of complexity class inclusions

$$\mathbf{LOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$$

A simple diagonal arugment shows that the first is a proper subset of the last, but we cannot prove any particular adjacent inclusion is proper.

As another example, let **LINEAR-SIZE** be the class of languages over $\{0, 1\}$ which can be computed by a family $\langle B_n \rangle$ of Boolean circuits of size $O(n)$. It is not known whether either **P** or **NP** is a subset of **LINEAR-SIZE**, although Kannan [Kan82] proved that there are languages in the polynomial hierarchy (a generalization of **NP**) not in **LINEAR-SIZE**. Since if $\mathbf{P} = \mathbf{NP}$ then the polynomial hierarchy collapses to **P**, we have

Proposition 2: If $\mathbf{P} \subseteq$ **LINEAR-SIZE**, then $\mathbf{P} \neq \mathbf{NP}$.

This proposition could be interpreted as a method of proving $\mathbf{P} \neq \mathbf{NP}$, but a more usual belief is that the hypothesis is false.

A fundamental question in complexity theory is whether a source of random bits can be used to substantially speed up the recognition of some languages, provided one is willing to accept a small probability of error. The class **BPP** consists of all languages L which can be recognized by a randomized polynomial-time algorithm, with at most an exponentially small error probability on every input. Of course $\mathbf{P} \subseteq \mathbf{BPP}$, but it is not known whether the inclusion is proper. The set of prime numbers is in **BPP** [SS77], but it is not known to be in **P**. A reason for thinking that $\mathbf{BPP} = \mathbf{P}$ is that randomized algorithms are often successfully executed using a deterministic pseudo-random number generator as a source of "random" bits.

There is an indirect but intriguing connection between the two questions $\mathbf{P} = \mathbf{BPP}$ and $\mathbf{P} = \mathbf{NP}$. Let **E** be the class of languages recognizable in exponential time; that is the class of languages L such that $L = L(M)$ for some Turing machine M with $T_M(n) = O(2^{cn})$ for some $c > 0$. Let **A** be the assertion that some language in **E** requires exponential circuit complexity. That is

Assertion A: There is $L \in \mathbf{E}$ and $\epsilon > 0$ such that for every circuit family $\langle B_n \rangle$ computing L and for all sufficiently large n, B_n has at least $2^{\epsilon n}$ gates.

Proposition 3: If **A** then $\mathbf{BPP} = \mathbf{P}$. If **not A** then $\mathbf{P} \neq \mathbf{NP}$.

The first implication is a lovely theorem by Impagliazzo and Wigderson [IW97] and the second is an intriguing observation by V. Kabanets which strengthens Proposition 2. In fact Kabanets concludes $\mathbf{P} \neq \mathbf{NP}$ from a weaker assumption than **not A**; namely that every language in **E** can be computed by a Boolean circuit family $\langle B_n \rangle$ such that for at least one n, B_n has fewer gates than the maximum needed to compute any Boolean function $f : \{0,1\}^n \to \{0,1\}$. But there is no consensus on whether this hypothesis is true.

We should point out that Proposition 3 relativizes, and in particular relative to any **PSPACE**-complete oracle Assertion A holds and $\mathbf{BPP} = \mathbf{P} = \mathbf{NP}$. Thus a nonrelativizing construction will be needed if one is to prove $\mathbf{P} \neq \mathbf{NP}$ by giving small circuits for languages in **E**. However nonrelativizing constructions have been used successfully before, for example in showing **IP** (Interactive Polynomial time) contains all of **PSPSACE** [Sha92]. In this and other such constructions a key technique is to represent Boolean functions by multivariate polynomials over finite fields.

Appendix: Definition of Turing machine

A Turing machine M consists of a finite state control (i.e. a finite program) attached to read/write head moving on an infinite tape. The tape is divided into squares, each capable of storing one symbol from a finite alphabet Γ which includes the blank symbol b. Each machine M has a specified input alphabet Σ, which is a subset of Γ, not including the blank symbol b. At each step in a computation M is in some state q in a specified finite set Q of possible states. Initially a finite input string over Σ is written on adjacent squares of the tape, all other squares are blank (contain b), the head scans the left-most symbol of the input string, and M is in the initial state q_0. At each step M is in some state q and the head is scanning a tape square containing some tape symbol s, and the action performed depends on the pair (q, s) and is specified by the machine's transition function (or program) δ. The action consists of printing a symbol on the scanned square, moving the head left or right one square, and assuming a new state.

Formally a Turing machine M is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$ where Σ, Γ, Q are finite nonempty sets with $\Sigma \subseteq \Gamma$ and $b \in \Gamma - \Sigma$. The state set Q contains three special states $q_0, q_{accept}, q_{reject}$. The *transition function* δ satisfies

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$$

If $\delta(q, s) = (q', s', h)$ the interpretation is that if M is in state q scanning the symbol s then q' is the new state, s' is the symbol printed, and the tape head moves left or right one square depending on whether h is -1 or 1.

We assume that the sets Q and Γ are disjoint.

A *configuration* of M is a string xqy with $x, y \in \Gamma^*$, y not the empty string, and $q \in Q$.

The interpretation of the configuration xqy is that M is in state q with xy on its tape, with its head scanning the left-most symbol of y.

If C and C' are configurations, then $C \xrightarrow{M} C'$ if $C = xqsy$ and $\delta(q, s) = (q', s', h)$ and one of the following holds:

$C' = xs'q'y$ and $h = 1$ and y is nonempty.

$C' = xs'q'b$ and $h = 1$ and y is empty.
$C' = x'q'as'y$ and $h = -1$ and $x = x'a$ for some $a \in \Gamma$.
$C' = q'bs'y$ and $h = -1$ and x is empty.

A configuration xqy is *halting* if $q \in \{q_{accept}, q_{reject}\}$. Note that for each non-halting configuration C there is a unique configuration C' such that $C \overset{M}{\to} C'$.

The *computation* of M on input $w \in \Sigma^*$ is the unique sequence $C_0, C_1, ...$ of configurations such that $C_0 = q_0 w$ (or $C_0 = q_0 b$ if w is empty) and $C_i \overset{M}{\to} C_{i+1}$ for each i with C_{i+1} in the computation, and either the sequence is infinite or it ends in a halting configuration. If the computation is finite, then the number of steps is one less than the number of configurations; otherwise the number of steps is infinite. We say that M *accepts* w iff the computation is finite and the final configuration contains the state q_{accept}.

Acknowledgments

My thanks to Avi Wigderson and Hugh Woodin for many helpful suggestions for improving an earlier version of this paper.

References

[AB87] N. Alon and R. B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987.

[ARS78] L. Adleman, R. L. Rivest, and A. Shamir. A method for obtaining digital signature and public-key cryptosystems. *Communication of the ACM*, 21(2), 1978.

[BCSS98] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP question. *SICOMP: SIAM Journal on Computing*, 1975.

[BI87] M. Blum and R. Impagliazzo. Generic oracles and oracle classes. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 118–126, Los Angeles, CA, October 1987. IEEE Computer Society Press.

[BS90] R. B. Boppana and M. Sipser. The complexity of finite functions. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 759–804. Elsevier and The MIT Press, 1990.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill, New York, 1990.

[Cob64] A. Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pages 24–30. Elsevier/North-Holland, 1964.

[Coo71] Stephen Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Coo91] Stephen Cook. Computational complexity of higher type functions. In Ichiro Satake, editor, *Proceedings of the International Congress of Mathematicians, Kyoto, Japan*, pages 55–69. Springer–Verlag, 1991.

[CR79] S. Cook and R. Reckhow. The relative efficiency of propositional proof systems. *J. Symbolic Logic*, 44(1), 1979.

[Edm65] Jack Edmonds. Minimum partition of a matroid into independent subsets. *J. Res. Nat. Bur. Standards Sect. B*, 69:67–72, 1965.

[FR74] M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In R. M. Karp, editor, *Complexity of Computation*, volume 7, pages 27–41. American Mathematical Society, Providence, RI, 1974.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, 1979.

[Gol00] Oded Goldreich. *The Foundations of Cryptography–Volume 1*. Cambridge University Press, 2000.

[Has89] J. Hastad. Almost optimal lower bounds for small depth circuits. In S. Micali, editor, *Randomness and Computation, Advances in Computing Research, Vol. 5*, pages 143–170. JAI Press Inc., 1989.

[Imp95] R. Impagliazzo. A personal view of average-case complexity. In *10th IEEE Annual Conference on Structure in Complexity Theory*, pages 134–147, 1995.

[IW97] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1997.

[Kan82] Ravi Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and Control*, 55:40–56, 1982.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[Kra95] J. Krajicek. *Bounded Arithmetic, Propositional Logic, and Complexity Theory.* Cambridge, 1995.

[Lev73] L. Levin. Universal search problems (in russian)". *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. English translation in Trakhtenbrot, B. A.: A survey of Russian approaches to Perebor (brute-force search) algorithms. Annals of the History of Computing, 6 (1984), pages. 384-400.

[Lev86] L. Levin. Average case complete problems. *SIAM J. Computing*, 15:285–286, 1986.

[Mil76] G. L. Miller. Reimann's hypothesis and tests for primality. *J. Comput. System Sci*, 13:300–317, 1976.

[Pap94] Christos Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.

[Pra75] V. R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[Raz85] Alexander A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Soviet Math. Dokl.*, 31:354–357, 1985.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, August 1997.

[RS95] N. Robertson and P. D. Seymour. Graph minors i–xiii. *Journal of Combinatorial Theory B*, 1983–1995.

[SCY97] R. Impagliazzo S. Cook and T. Yamakami. A tight relationship between generic oracles and type-2 complexity theory. *Information and Computation*, 137(2):159–170, 15 September 1997.

[Sha49] C. Shannon. The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

[Sha92] Adi Shamir. Ip = pspace. *J.A.C.M.*, 39(4):869–877, 1992.

[Sho97] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Computing*, 26:1484–1509, 1997.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS, 1997.

[Sma98] Steve Smale. Mathematical problems for the next century. *MATH-INT: The Mathematical Intelligencer*, 20, 1998.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *ACM Symposium on Theory of Computing (STOC)*, volume 19, pages 77–82, 1987.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, March 1977.

[Tur36] Alan Turing. On computable numbers with an application to the entscheidnungsproblem. *Proc. London Math. Soc. ser. 2*, 42:230–265, 1936-7.

[vN53] J. von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem. In H. W. Kahn and A. W. Tucker, editors, *Contributions to the Theory of Games II*. Princeton Univ. Press, 1953.

Literatur

[AHU] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, Massachusetts (1974)

[Cap] C.H. Cap: *Theoretische Grundlagen der Informatik*, Springer–Verlag Wien New York (1993)

[Coo] S. Cook: *The P versus NP Problem*, `http://www.claymath.org/prize_problems/p_vs_np.pdf` (2000)

[Dav] M.D. Davis: *Computability & Unsolvability*, McGraw–Hill, New York–Toronto–London (1958)

[DW] M.D. Davis, E.J. Weyuker: *Computability, Complexity, and Languages*, Academic Press, Orlando, Florida (1983)

[GJ] M.R. Garey, D.S. Johnson: *Computers and Intractability — A Guide to the Theory of NP–Completeness*, Freeman and Company, New York (1979)

[Har] J. Hartmanis (ed.): *Computational Complexity Theory*, Amer.Math.Soc. (1989)

[Hen] M. Hennessy: *The Semantics of Programming Languages*, J. Wiley (1990)

[HU] J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, Massachusetts (1979)

[Koz] D.C. Kozen: *Automata and Computability*, Springer-Verlag New York (1997)

[LP] H.R. Lewis, C.H. Papadimitriou: *Elements of the Theory of Computation*, Prentice–Hall, Englewood Cliffs, New Jersey (1981)

[Lip] J.D. Lipson: *Elements of Algebra and Algebraic Computing*, Addison–Wesley, Reading, Massachusetts (1981)

[Man] Z. Manna: *Mathematical Theory of Computation*, McGraw–Hill (1974)

[McN] R. McNaughton: *Elementary Computability, Formal Languages, and Automata*, Prentice–Hall, Englewood Cliffs, New Jersey (1982)

[Pep] P. Pepper: *Grundlagen der Informatik*, Oldenburg, München (1992)

[Rog] H. Rogers: *Theory of Recursive Functions and Effective Computability*, McGraw–Hill, New York, New York (1967)

[Sch] U. Schöning: *Theoretische Informatik — kurzgefaßt*, 2. Aufl., Spektrum Akad. Verlag, Heidelberg Berlin Oxford (1995)

[Sed] R. Sedgewick: *Algorithms*, Addison–Wesley, Reading, Massachusetts (1983)

[Weg] I. Wegener: *Theoretische Informatik*, Teubner, Stuttgart (1993)