Ist P gleich NP? Vermutlich nicht!

Komplexitätsklassen von algorithmischen Problemen

Achtung: Das Wort "Klasse" bedeutet hier etwas ähnliches wie "Menge" oder "Zusammenfassung" und hat praktisch nichts mit dem Begriff "Klasse" der objektorientierten Programmierung zu tun.

InformatikerInnen haben viele Eigenschaften von *algorithmischen Problemen* untersucht und solche Probleme zu *Klassen* zusammengefasst, die *ähnliche Eigenschaften* haben.

Beispiel-01: Zur Klasse der nicht-mit-einem-Computerprogramm-lösbaren Probleme gehören (unter vielen anderen) das *Halte-Problem* und das Problem der *Lösbarkeit von diophantischen Gleichungen*.

Beispiel-02: Zur Klasse der mit einer Zeitkomplexität von O(N * log(N)) lösbaren Probleme gehört (unter vielen anderen) das *vergleichsbasierte Sortieren von Reihungen*.

Unter diesen zahlreichen Klassen gibt es insbesondere zwei, die besonders intensiv untersucht wurden und werden. Sie heißen ganz kurz **P** und **NP**.

Def. P: Zur Klasse **P** gehören alle algorithmischen Probleme, von denen gezeigt wurde, dass man sie mit einem *deterministischen Programm* (DP) in ***p**olynomialer Zeit* lösen kann.

"In polynomialer Zeit" bedeutet: Der Lösungs-Algorithmus hat eine Zeitkomplexität mit einem Polynom hinter dem großen O, z.B. $O(N^0)$ (gleich O(1)) oder $O(N^2)$ oder $O(N^{17})$ etc.
(oder eine kleinere Zeitkomplexität, z.B. O(log N) oder O(N * log N) etc.). Wenn die besten bekannten Lösungs-Algorithmen eine Zeit-Komplexität von z.B. $O(2^N)$ oder von O(N!) haben, dann gehört das algorithmische Problem *nicht* zur Klasse **P**.

Def. NP: Zur Klasse **NP** gehören alle algorithmischen Probleme, von denen gezeigt wurde, dass man sie mit einem ***n**icht-deterministischen Programm* (NDP) in ***p**olynomialer Zeit* lösen kann.

Etwas vereinfacht kann man sagen: Die meisten auf heutigen Computern ausgeführten Programme sind *deterministische Programme* (DPs). Die Reihenfolge, in der die Befehle eines solchen Programms ausgeführt werden, wird durch die Eingabedaten genau festgelegt ("determiniert"). Der Ausführer folgt "stur" bestimmten Regeln und trifft keine "kreativen Entscheidungen".

Heutige Computer können zu einem bestimmten Grad auch nicht-deterministische Programme (NDPs) ausführen, aber diese Fähigkeit ist noch so schwach entwickelt, dass wir sie hier ignorieren. Wenn die ersten Computer mit mehr als einer Million Prozessoren weniger kosten als 500 Arbeitsstunden zum Mindestlohn (Anfang 2018 sind/waren das 4.420,- Euro), sollten wir diese Haltung kritisch überdenken.

Nicht-deterministische Programme, zwei Arten

Es gibt zwei Möglichkeiten, sich nicht-deterministische Programme (NDPs) vorzustellen. Präzise Definitionen dieser Möglichkeiten findet man im Internet unter den Stichworten "nicht-deterministische Turingmaschine" und "Orakel Turingmaschine". Hier sollen diese Möglichkeiten nicht präzise definiert, sondern nur anhand von Beispielen erläutert werden.

Algorithmisches Problem 1 (AP-01):

Gegeben eine natürlich Zahl `n`. Ist `n` eine Primzahl?

Dieses Problem soll jetzt mit zwei Arten von nicht-deterministischen Programmen (NDPs) gelöst werden: Mit einem *NDP-mit-Orakel* und mit einem *NDP-mit-unbegrenzter-Nebenläufigkeit*.

Konvention: Im Folgenden ist mit einem *E-Teiler einer Ganzzahl n* immer ein *echter* Teiler gemeint (d.h. einer, der ungleich 1 und ungleich n ist).

NDP-01: Ein NDP-mit-Orakel zur Lösung von AP-01 könnte etwa so funktionieren:

1. Das NDP-01 verlangt von seinem Orakel: "Gib mir einen E-Teiler von n".

2. Daraufhin liefert das Orakel (in *einem* Schritt) eine Zahl `t`.

3. Dann prüft das NDP-01, ob `n` sich (mit Rest 0) durch `t` teilen lässt. Wenn ja, dann ist `n` offenbar *keine* Primzahl. Wenn nein, dann ist `n` eine Primzahl, denn das Orakel muss das Verlangen in 1. so gut erfüllen wie möglich und darf nur dann einen nicht-E-Teiler liefern, wenn es keinen E-Teiler gibt.

Das Befragen des Orakels zählt als 1 Schritt. Für die Prüfung, ob `n` durch `t` teilbar ist, braucht man höchstens eine Zeitkomplexität von $O(N^2)$, wobei N die Anzahl der Ziffern von `n` ist (Sie wissen ja, wie man große Zahlen mit Papier und Bleistift dividieren kann). Also kann NDP-01 das AP-01 mit einer Zeitkomplexität von $O(N^2)$ lösen oder, etwas ungenauer gesagt:

NDP-01 kann das AP-01 in *polynomialer Zeit* lösen.

Ein Orakel ist keineswegs so märchenhaft und weltfremd wie es auf den ersten Blick scheinen mag. Wenn man seine Lösungsvorschläge (in polynomialer Zeit) überprüfen kann, ehe man sie akzeptiert, ist zumindest ein "kleines Orakel" durchaus realisierbar.

NDP-02: Ein NDP-mit-unbegrenzter-Nebenläufigkeit könnte etwa so funktionieren

1. Der Ausführer A bekommt die Zahl `n` und will alle Zahlen im Intervall `[2, n-1]` daraufhin prüfen, ob sie ein E-Teiler von `n` sind. Dazu erzeugt er zwei neue Ausführer A1 und A2.
Dem einen übergibt er die Zahl `n` und das Intervall `[2, n/2]` ("die untere Hälfte") und
dem anderen übergibt er die Zahl `n` und das Intervall `[n/2+1, n-1]` ("die obere Hälfte").

Die Ausführer A1 und A2 verhalten sich ähnlich: Wenn das Intervall von A1 mehr als eine Zahl enthält, erzeugt A1 zwei Ausführer A11 und A12 und übergibt jedem `n` und eine Hälfte seines Intervalls, u.s.w.

Wenn ein Ausführer (z.B. der namens A2112211121222) feststellt, dass sein Intervall nur noch *eine* Zahl `t` enthält, prüft er, ob `n` durch `t` teilbar ist. Wenn ja, schreibt er `t` in eine globale Variable `ERG` (die mit 0 initialisiert wurde), und sonst macht er einfach nichts.

Wenn alle Ausführer (A, A1, A2, A11, A12, A21, A22, ...) fertig sind und in `ERG` immer noch 0 steht, ist `n` ein Primzahl. Sonst steht in `ERG` ein E-Teiler von `n` und `n` ist keine Primzahl.

Aufgabe-01: Wie lange dauert es (wie viele Schritte müssen *nacheinander* ausgeführt werden), bis alle benötigten Ausführer erzeugt wurden und ihre Aufgabe fertig erledigt haben?

Man sieht: Auch das NDP-02 kann das AP-01 in polynomialer Zeit lösen.

Verhältnis von NDPs zu DPs

Man kann ein NDP ("ein Programm für einen nicht-deterministischen Rechner") auch von einem "Rechner für deterministische Programme" ausführen lassen, allerdings wird dann die Zeitt-Komplexität deutlich schlechter: Wenn das NDP auf einem NDP-Rechner *polynomiale Zeit* braucht, dann braucht es auf einem DP-Rechner *exponentielle Zeit* (z.B. 2^N oder 5^N etc.). In diesem Sinne sind NDPs (oder: Rechner für NDPs) viel mächtiger als DPs (oder: Rechner für DPs, z.B. heute übliche Computer).

Algorithmisches Problem 2 (AP-02):

Gegeben ein einfacher Graph (d.h. ungerichtet, keine Schleifen, keine Mehrfachkanten) mit nicht-negativen Kantengewichten. Frage: Gibt es einen Weg, der an jedem Knoten genau einmal vorbeikommt und nicht länger als L ist (wobei L eine nicht-negative Zahl ist)?

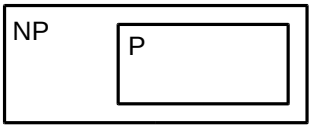
Einen Weg, der an jedem Knoten genau einmal vorkommt bezeichnen wir kurz
als einen *vollständigen Weg*.

Aufgabe-02: Beschreiben Sie ein NDP-mit-Orakel namens **NDP-03**, welches
das Problem **AP-02** löst.

Aufgabe-03: Beschreiben Sie ein NDP-mit-unbegrenzter-Nebenläufigkeit namens **NDP-04**, welches
das Problem **AP-02** löst.

Eine einfache Erkenntnis: NP enthält P

Keine Zahl kann gleichzeitig *positiv* und *nicht-positiv* sein. Diese *Regel der deutschen Sprache* gilt für viele, aber nicht für alle Eigenschaftsworte. Zu den Ausnahmen gehört das Wort *deterministisch* im Zusammenhang mit Programmen und Maschinen: Jedes *deterministische* Programm gilt auch als ein *nicht-deterministisches* Programm (welches sein Orakel nie befragt bzw. 0 weitere Ausführer erzeugt). Daraus folgt, dass **P** eine Teilmenge von **NP** ist (alle Probleme, die man mit einem DP in polynomialer Zeit lösen kann, kann man erst recht mit einem NDP in polynomialer Zeit lösen).



Beispiele für algorithmische Probleme aus NP

Das SAT-Problem

Von diesem Problem wurde zuerst bewiesen, dass es
1. in NP liegt und
2. *NP-vollständig* ist (siehe unten)
und zwar 1971 von Steve A. Cook.

Eine *aussagenlogische Formel* besteht aus booleschen Variablen `A, A1, A2, ..., B, B1, B2, ...`, den booleschen Operatoren `und`, `oder` und `nicht` und runden Klammern. Die `nicht`-Operation wird häufig platzsparend durch *Überstreichung* dargestellt (z.B. bedeutet $\overline{\mathbf{A}}$ soviel wie **`nicht A`**).

Beispiel-01: Eine aussagenlogische Formel

`A und (A oder` $\overline{\mathbf{B}}$ **`oder B)`

Beispiel-02: Eine aussagenlogisch Formel in `oder`-Normalform (in disjunktiver Normalform):

`(A und` $\overline{\mathbf{B}}$ **`und C und` $\overline{\mathbf{A}}$`) oder (B und` $\overline{\mathbf{C}}$ **`und D) oder (A und D)`**

Beispiel-03: Eine aussagenlogisch Formel in `und`-Normalform (in konjunktiver Normalform):

`(A oder` $\overline{\mathbf{B}}$ **`oder C oder` $\overline{\mathbf{A}}$`) und (B oder` $\overline{\mathbf{C}}$ **`oder D) und (A oder D)`**

Def.: Eine *Variablen-Belegung* ordnet jeder Variablen einer Formel einen der Werte **wahr** bzw. **falsch** zu.

Def.: Eine aussagenlogische Formel F ist *erfüllbar* (engl. satisfiable), wenn es eine Variablen-Belegung (für die Variablen von F) gibt, so dass F den Wert **wahr** hat.

Beispiel-04: Die folgende Formel F1

`(A oder B) und (` $\overline{\mathbf{A}}$ **`oder` $\overline{\mathbf{B}}$**`) und (A oder` $\overline{\mathbf{B}}$**`)`**

ist *erfüllbar*, denn die Variablen-Belegung `{A:wahr, B:falsch}` macht F1 `wahr`.

Beispiel-05: Die folgende Formel F2 ist *nicht erfüllbar*

`(A oder B) und (` $\overline{\mathbf{A}}$ **`oder` $\overline{\mathbf{B}}$**`) und (A oder` $\overline{\mathbf{B}}$**`) und (`** $\overline{\mathbf{A}}$ **`oder B)`**

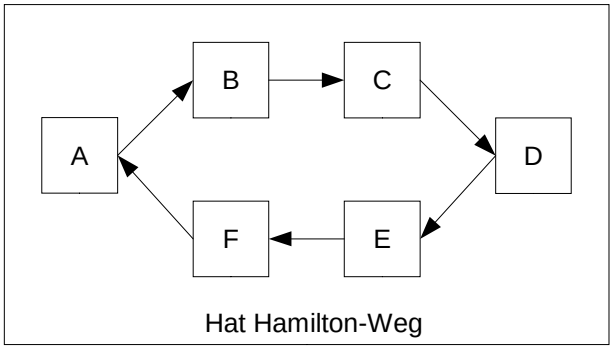
denn für alle Variablen-Belegungen (für die zwei Variablen A und B gibt es insgesamt 4 Belegungen) hat F2 den Wert `falsch`.

SAT-Problem: Gegeben sei eine aussagenlogische Formel F in `und`-Normalform. Ist F erfüllbar?

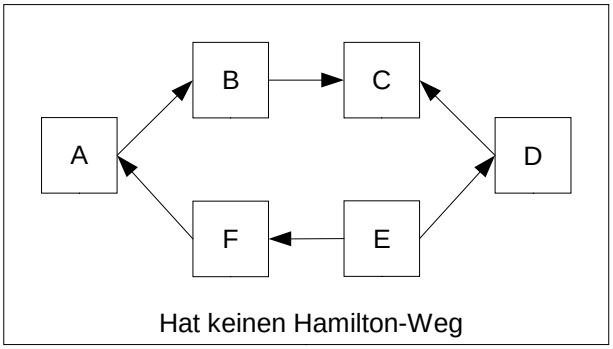
Der Name SAT-Problem kommt (nicht vom gleichnamigen Fernsehsender sondern) vom englischen Wort *satisfiable* (deutsch: erfüllbar).

Hamilton-Problem: Gegeben sei ein gerichteter Graph G.
Gibt es einen Weg, der an jedem Knoten von G genau *einmal* vorbeikommt?

Beispiel-01: Zwei gerichtete Graphen, einer mit und einer ohne Hamilton-Weg:



Hat Hamilton-Weg



Hat keinen Hamilton-Weg

Dieses Problem wurde zum ersten Mal von dem englischen Mathematiker und Priester Thomas P. **Kirkman** (1806-1895) erwähnt aber trotzdem nach dem irischen Mathematiker und Physiker Sir William Rowan **Hamilton** (1805-1865) benannt. Ganz ähnlich werden die *Quaternionen*, die heute in Computeranimationen verwendet werden, auch als **Hamilton**-Zahlen bezeichnet, obwohl sie vor diesem von dem Franzosen Olinde **Rodrigues** (1795-1851) beschrieben wurden.

Rucksack-Problem: Gegeben sei eine Menge von Gegenständen. Jeder Gegenstand hat ein *Gewicht* und einen *Wert* ("für den Wanderer"). Ein Rucksack soll so mit einigen dieser Gegenstände gefüllt werden, dass ein bestimmtes Gesamtgewicht (z.B. 5 kg) nicht überschritten wird. Gleichzeitig sollen die Gegenstände im Rucksack ("für den Wanderer") *so wertvoll wie möglich* sein.

Problem des Handlungsreisenden: Gegeben ein gerichteter Graph mit nicht-negativen Kantengewichten. Gesucht ist ein *kürzester* Rundweg (beginnt und endet beim selben Knoten), der an jedem Knoten genau einmal vorbeikommt.

Teilsummen-Problem: Gegeben ist eine Menge von (positiven und negativen) Zahlen. Gesucht ist eine Teilmenge, deren Summe gleich 0 ist.

Umwandlung-von-Normalformen-Problem: Gegeben eine aussagenlogische Formel in `und`-Normalform (bzw. in `oder`-Normalform). Gesucht ist eine äquivalente Formel in `oder`-Normalform (bzw. in `und`-Normalform).

Primzahl-Zerlegungs-Problem: Gegeben eine natürliche Zahl n. Gesucht sind die Primfaktoren von n. Beispiel: Die Zahl 20 hat die Primfaktoren 2, 2, 5, den 2*2*5 ist gleich 20.

Graph-Isomorphie-Problem: Gegeben zwei Graphen g1 und g2. Sind g1 und g2 isomorph oder nicht? "isomorph" bedeutet hier etwa Folgendes: Man kann die beiden Graphen (so zeichnen und) aufeinander legen, dass sie sich genau bedecken, und nirgends eine Ecke oder eine Kante "hervorschaut".

Die Klasse NP

Von mehreren tausend algorithmischen Problemen weiß man heute (2016), dass sie von einem nicht-deterministischen Programm in polynomialer Zeit gelöst werden können und somit zur Klasse **NP** gehören.

Diese Zahl von Problemen klingt möglicherweise beeindruckend groß. Aber noch wichtiger ist die Erkenntnis, das (nicht alle aber) viele dieser Probleme "miteinander zusammenhängen" und in einem gewissen (stark vereinfachten) Sinn "im Grunde alle das gleiche Problem" sind.

Für viele Probleme P1 und P2 aus **NP** hat man nämlich beweisen: Man kann
jede Instanz I1 des Problems P1 (in polynomialer Zeit) so in
eine Instanz I2 des Problems P2 übersetzen dass jede Lösung für I2 einer Lösung für I1 entspricht.

Beispiel-01: Aus einer Instanz I1 des Hamilton-Wege-Problems
(d.h. aus einem gerichteten Graphen G1)
kann man eine Instanz des SAT-Problems
(d.h. eine Menge M1 von aussagenlogischen Formeln in und-Normalform)
erzeugen, so dass gilt:
Im Graphen G1 gibt es genau dann einen Hamilton-Weg wenn die Formelmenge M1 erfüllbar ist.

Und von vielen Problemen hat man bewiesen, dass diese "Übersetzbarkeit in polynomialer Zeit" in beiden Richtungen gilt, d.h. von P1 nach P2 und von P2 nach P1.

Man beachte, dass dieser Zusammenhang zwischen Problemen in NP *transitiv* ist, d.h. es gilt:

Wenn die folgenden beiden Übersetzungen in polynomialer Zeit möglich sind
1. P1-Probleme in äquivalente P2-Probleme und
2. P2-Probleme in äquivalente P3-Probleme
dann ist auch folgende Übersetzung in polynomialer Zeit möglich:
3. P1-Probleme in äquivalente P3-Probleme.

Das folgt aus der Tatsache, dass die Summe zweier Polynome wieder ein Polynom ist.

Für etwa 3000 Probleme P1 aus **NP** hat man bewiesen: Man kann jedes Problem P2 aus **NP**
in polynomialer Zeit nach P1 übersetzen (wie im **Beispiel-01** skizziert).

Def: Ein Problem P1 aus **NP** heißt *NP-vollständig*, wenn man jedes Problem P2 aus **NP**
in polynomialer Zeit nach P1 übersetzen kann.

Anmerkung-01: Dass man (zumindest bisher) von keinem Problem aus **P** zeigen konnte, dass es NP-vollständig ist, sollte nicht überraschen. Von vielen der übrigen Probleme in **NP** (d.h. für viele Probleme in **NP minus P**) hat man gezeigt, dass sie NP-vollständig sind, aber nicht für alle. Von den Problemen "Eine Ganzzahl in ihre Primfaktoren zerlegen" und "Von zwei Graphen feststellen, ob sie isomorph sind oder nicht" vermutet man, dass sie *nicht* NP-vollständig sind (aber bewiesen wurde das noch nicht).

Anmerkung-02: Als die Informatiker Aho, Hopcroft und Ullman nach einem Namen für die Eigenschaft suchten, die dann **NP-complete** (deutsch: NP-vollständig) genannt wurde, war auch **hard boiled** (deutsch: *hart gekocht*, wie bei Eiern) ein Kandidat. Damit sollte Steve Cook (siehe oben Abschnitt 3.1) geehrt werden, dessen Name ja auch mit Kochen zu tun hat.

Das Problem

Wenn man ein algorithmisches Problem mit einem *nicht-deterministischen* Programm in polynomialer Zeit lösen kann, dann folgt daraus *nicht* die Unmöglichkeit, das Problem auch mit einem *deterministischen Programm* in polynomialer Zeit zu lösen (wenn einer mit einem Porsche in einer Stunde von Kreuzberg nach Charlottenburg gefahren ist, dann folgt daraus nicht automatisch, dass man das nicht auch mit einem Fahrrad schaffen kann).

Bisher konnte noch niemand von einem NP-vollständigen Problem beweisen, dass es garantiert *nicht* in der Klasse **P** liegt (d.h. dass es garantiert *nicht* mit einem deterministischen Programm in polynomialer Zeit gelöst werden kann).

Um eine *obere Schranke* für die Komplexität eines Problems zu beweisen, muss man "nur" *einen* Algorithmus mit dieser Komplexität angeben. Um eine *untere Schranke* zu beweisen, muss man von *unendlich vielen* Algorithmen zeigen, dass sie das Problem *nicht* lösen können oder ihre Zeitkomplexität "schlechter als polynomial" ist (z.B. $\mathbf{O(2^N)}$ oder $\mathbf{O(5^N)}$ oder **O(N!)** etc.). Deshalb ist das Beweisen von *unteren* Schranken in aller Regel sehr viel schwerer als das Beweisen von *oberen* Schranken.

Falls es gelingen sollte, von *einem* einzigen NP-vollständigen Problem aus **NP** zu zeigen, dass es auch zur Klasse **P** gehört, dann würde das automatisch auch für alle anderen Probleme aus **NP** gelten und die beiden Klassen **P** und **NP** wären gleich, es würde also gelten: **P = NP**.

Vermutlich nehmen alle (oder zumindest fast alle) Informatiker und Mathematiker an, dass **P ≠ NP** gilt, aber beweisen konnte das bisher noch niemand (obwohl viele es sehr intensiv versucht haben).

Wenn es Ihnen gelingt (als erste/erster) zu beweisen, dass **P ≠ NP** gilt, gewinnen Sie damit erst mal **eine Million Dollar** (siehe http://www.claymath.org/millennium-problems/p-vs-np-problem). Außerdem haben Sie dann gute Chancen, den **Turing-Award** (für Informatiker), die **Fields-Medaille** (für Mathematiker) und eine Prof-Stelle in Kalifornien (Büro mit Blick auf den Pazifik und ein **gutes Surfbrett** inklusive) zu bekommen. Und wenn Sie (als erste/erster) beweisen können, dass **P = NP** gilt, wird zusätzlich auch noch ein **Hollywood-Film** über Sie gedreht, wahrscheinlich mit Ihnen (oder mit Scarlett Johannsson bzw. mit Leonardo DiCaprio als Ersatz für Sie) in der Hauptrolle.

Eine Beschreibung des **P=NP?**-Problems anhand des populären Computerspiels Minesweeper (in "lockerem Englisch") findet man hier: http://www.claymath.org/sites/default/files/minesweeper.pdf.