Formale Grundlagen der Informatik 1
Kapitel 10
Zeit- und Platzkomplexität

Frank Heitmann
heitmann@informatik.uni-hamburg.de

6. Mai 2014

Motivation

Bisher haben wir mit TMs

- Probleme gelöst/entschieden/berechnet.

Dabei war

- entscheidbar = gut
- nicht entscheidbar = schlecht

Aber ...

Motivation

Nur zu wissen, *dass* ein Problem gelöst werden kann, reicht im Allgemeinen nicht!

Die Aufgabe...

Wie viel

- **Zeit** und
- **Platz**

brauchen wir zur Lösung des Problems?

Motivation

Dies wird abhängen

- Von dem **Problem**
- Von der **Größe der Probleminstanz**

Bemerkung

Eine Probleminstanz ist eine mögliche Eingabe. Wollen wir z.B. Zahlen addieren, so ist dies als Sprache $L = \{\langle x, y, z\rangle \mid x + y = z\}$. Eine Probleminstanz ist dann ein Tripel $(2, 3, 5)$ (Ja-Instanz) oder $(2, 3, 6)$ (Nein-Instanz).
Die Größe hängt hier von der Länge der Zahlen ab.

Zeitkomplexität bei DTMs

Definition (Zeit- und Platzkomplexität der DTM)

Sei M eine DTM.

1. Für eine Rechnung benötigt eine DTM so viel Zeit, wie in der Rechnung einzelne Konfigurationen durchlaufen werden.
2. M benötigt in einer Rechnung soviel Platz, wie *maximal* Felder besucht werden.

Beispiel

In

$$z_0ab \vdash Az_1 \vdash ABz_2\# \vdash ABXz3\# \vdash ABz_3X$$

werden 5 Zeiteinheiten benötigt und 4 Felder benutzt (das vierte Feld wird nicht geändert und bleibt leer).

Zeitkomplexität bei DTMs

Definition (Zeit- und Platzkomplexität der DTM (Fortsetzung))

Seien t, s Funktionen von $\mathbb{N}$ nach $\mathbb{R}$. Eine DTM M ist

1. $t(n)$-**zeitbeschränkt** genau dann, wenn

$$t(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert in } k \text{ Schritten}\}$$

2. $s(n)$-**platzbeschränkt** genau dann, wenn

$$s(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert mit Platzbedarf } k\}$$

Anmerkung zur Zeitkomplexität

Anmerkung

1. Man bezeichnet dies als **worst-case**-Komplexität, da alle Eingaben der Länge n betrachtet werden und $t(n)$ mindestens so groß sein muss, wie der Zeitbedarf im schlimmsten Fall (analog für den Platzbedarf und $s(n)$).
2. t und s sind nur deswegen Abbildungen auf die *reelle Zahlen*, damit wir später mit Funktionen wie $n^2 + n/2$ arbeiten können.
3. Oft ignoriert man Anfangswert und erlaubt auch $t(n) = n^2 - 5 \cdot n$, was erst ab $n \geq 4$ einen sinnvollen Wert ergibt.

Anmerkung zur Zeitkomplexität

Beispiel

Ist eine DTM $t(n)$-**zeitbeschränkt** mit

$$t(n) = n^2 - n/2,$$

dann darf eine Rechnung

- auf einem Wort w der Länge 10 nur maximal $100 - 5 = 95$ Schritte benötigen.
- Auf einem Wort der Länge 3 nur maximal $9 - 1,5 = 7,5 \approx 7$ Schritte.

Zeitkomplexität bei NTMs

Definition (Zeit- und Platzkomplexität der NTM)

Sei M eine **N**TM. In einer fest vorgegebenen Erfolgsrechnung

1. benötigt eine NTM so viel Zeit, wie in der Rechnung einzelne Konfigurationen durchlaufen werden und
2. soviel Platz, wie *maximal* Felder besucht werden.

Anmerkung

Das ist erstmal so wie bei DTMs, aber wir halten hier eine (Erfolgs-)Rechnung fest!

Zeitkomplexität bei NTMs

Definition (Zeit- und Platzkomplexität der NTM (Fortsetzung))

Eine NTM M akzeptiert ein $w \in L(M)$ mit

1. der **Zeitbeschränkung** $t \in \mathbb{R}$ genau dann, wenn die kürzestes Erfolgsrechnung $\lceil t \rceil$ Schritte hat
2. der **Platzbeschränkung** $s \in \mathbb{R}$ genau dann, wenn die Erfolgsrechnung mit dem geringsten Platzbedarf $\lceil s \rceil$ Felder besucht.

Zeitkomplexität bei NTMs

Definition (Zeit- und Platzkomplexität der NTM (Fortsetzung))

Seien t, s Funktionen von $\mathbb{N}$ nach $\mathbb{R}$. Eine NTM M ist

1. $t(n)$-**zeitbeschränkt** genau dann, wenn

$$t(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert mit Zeitbeschränkung } k\}$$

2. $s(n)$-**platzbeschränkt** genau dann, wenn

$$s(n) \geq \max\{k \mid \exists w \in L(M) : n = |w| \text{ und } M \text{ akzeptiert mit Platzbeschränkung } k\}$$

Bandkompression und lineares speed-up

Satz (Bandkompression)

Zu jeder $s(n)$-platzbeschränkten TM und jedem $c \in \mathbb{R}$, $c > 0$ gibt es eine äquivalente $c \cdot s(n)$-platzbeschränkte TM.

Satz (lineares speed-up)

Zu jeder $t(n)$-zeitbeschränkten TM mit $\inf_{n\to\infty} \frac{t(n)}{n}$ und jedem $c \in \mathbb{R}$, $c > 0$ gibt es eine äquivalente $c \cdot t(n)$-zeitbeschränkte TM.

Hinweis

Beide Beweise gehen indem mehrere Bandzelle zu einer zusammengefasst werden. Man beachte, dass es in beiden Fällen nur um einen linearen Faktor geht! Beweise findet man im Skript.

Komplexitätsklassen

Definition (Komplexitätsklasse)

Für $s, t : \mathbb{N} \to \mathbb{R}$ mit $t(n) \geq n+1$ und $s(n) \geq 1$:

$$\begin{aligned} DTIME(t(n)) &:= \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte DTM } A\} \\ NTIME(t(n)) &:= \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte NTM } A\} \\ DSPACE(s(n)) &:= \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte DTM } A\} \\ NSPACE(s(n)) &:= \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte NTM } A\} \end{aligned}$$

Komplexitätsklassen

Definition (Komplexitätsklasse)

$$\begin{aligned} P &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L(A) = L\} \\ NP &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-zeitbeschränkte NTM } A \text{ mit } L(A) = L\} \\ PSPACE &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-platzbeschränkte DTM } A \text{ mit } L(A) = L\} \\ NPSPACE &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine } p(n)\text{-platzbeschränkte NTM } A \text{ mit } L(A) = L\} \end{aligned}$$

P und NP

Wegen der Speedup- und Kompressionssätze gilt dann:

$$\begin{aligned} P &= \cup_{i \geq 1} DTIME(n^i) \\ NP &= \cup_{i \geq 1} NTIME(n^i) \\ PSPACE &= \cup_{i \geq 1} DSPACE(n^i) \\ NPSPACE &= \cup_{i \geq 1} NSPACE(n^i) \end{aligned}$$

Teilmengenbeziehungen

Satz

Es gilt

$$\begin{aligned} DTIME(f) &\subseteq NTIME(f) \\ DSPACE(f) &\subseteq NSPACE(f) \\ P &\subseteq NP \\ PSPACE &\subseteq NPSPACE \end{aligned}$$

Der Beweis folgt direkt aus den Definitionen...

Von TMs zu realen Computern

Bis hierhin haben wir uns mit der Komplexität bei TMs beschäftigt.

Man kann aber zeigen, dass

- ein Computer eine TM simulieren kann
- eine TM einen Computer simulieren kann

Anmerkung

Man zeigt hierzu, wie eine TM eine gängige imperative Programmiersprache simuliert (bzw. eine Von-Neumann-Architektur) - und andersherum. Siehe [HMU].

Von TMs zu realen Computern

Anmerkung

Bei (Un-)Entscheidbarkeit hatten wir das schon kurz erwähnt. Daher ließen sich die Resultate dort auf unsere realen Computer übertragen (mit der Konsequenz, dass wichtige Probleme nicht von uns berechnet werden können).

Dass dies nicht nur für unsere Computer gilt, sondern für jedes erdenkbare Rechnermodell ist die *Church-Turing-These*.

Die *erweiterte Church-Turing-These* sagt nun, dass die Modelle alle in Polynomialzeit ineinander umwandelbar sind! (Insb. also auch obige Umwandlung TM $\leftrightarrow$ Computer, für die man die Gültigkeit der Aussage der These zeigen kann.)

Von TMs zu realen Computern - Zusammengefasst

Church-Turing-These

Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein.

Folgerung

Alle hinreichend mächtigen Berechnungsmodelle sind äquivalent.

Erweiterte Church-Turing-These

Sind B_1 und B_2 zwei Rechnermodelle, so gibt es ein Polynom p, so dass t Rechenschritte von B_1 bei einer Eingabe der Länge n durch $p(t, n)$ Rechenschritte von B_2 simuliert werden können.

Folgerung

Zwei (Turing-)äquivalente Modelle lassen sich mit polynomiellen Mehraufwand ineinander umwandeln.

Zur Bedeutung von P

Wichtige Anmerkung

Damit erhält die Komplexitätsklasse P eine besondere Bedeutung! Alles was in P ist, kann mit jedem Berechnungsmodell in P berechnet werden (auch wenn die Polynome unterschiedlich sind).
Außerdem kann man statt mit TMs bei Algorithmen auch mit gängigen Programmiersprachen argumentieren, so fern einen nur interessiert, ob ein Problem in P ist (und nicht der exakte Zeitbedarf auf einem festen Rechnermodell).

Wichtige Anmerkung

Die herausragende Bedeutung von P liegt auch darin, dass man i.A. davon ausgeht, dass **in P die Probleme liegen, die wir effizient lösen können**. Wenn wir also für ein Problem einen langsameren Algorithmus haben, versuchen wir immer einen zu finden, der das Problem in Polynomialzeit löst!

Zur Bedeutung von P (und NP)

Weiterhin zur Bedeutung von P:

- Man geht allgemein davon aus, dass ein Problem, dass in Polynomialzeit gelöst werden kann, *effizient* lösbar ist.
  - In Polynomialzeit bedeutet, es gibt ein Polynom p, so dass eine Instanz der Länge n in $p(n)$ Schritten gelöst werden kann.
- Ein Polynom wächst mit n in eine vertretbaren Aufwand.
  - Für z.B. n^2 ist $10^2 = 100$ und $50^2 = 2500$.
  - Für aber 2^n ist $2^{10} \approx 1000$ und $2^{50} \approx 1$ Billarde.
- Für die meisten Probleme, für die man einen Algorithmus in P hat, findet man einen, mit einem kleinen Polynom.

Zur Bedeutung von NP später mehr...

Landau-Notation

Wir merken uns

- Polynome sind wichtig!

Meist interessieren uns aber Faktoren vor einem Polynom nicht und auch die "kleineren Teile" in einem Polynom interessieren uns nicht. Die Polynome

- $10 \cdot n^3 + 2 \cdot n^2 - 50 \cdot n + 25$ und
- n^3

verhalten sich "im Grunde genommen gleich". (Wenn man sehr große Werte einsetzt, kommt da nämlich fast das gleiche raus...)

Landau-Notation

Um das zu formalisieren, führen wir die O- bzw. Landau-Notation ein. Warum man das braucht sieht man gut bei Algorithmen....

Algorithmenanalyse...

Algorithmus 1 Lineare Suche

```
1: for i = 0 to n do
2:     if a[i] == max mustermann then
3:         return true
4:     end if
5: end for
6: return false
```

Wir behandeln nachfolgend *Zeit-* und *Platzkomplexität* (im uniformen Maß). Eine (elementare) Anweisung zählt dabei als eine Zeiteinheit. Eine benutzte (elemantare) Variable als eine Platzeinheit.

Zeit- und Platzbedarf wird *abhängig von der Länge n der Eingabe* gezählt. Wir arbeiten hierfür mit Funktionen $f : \mathbb{N} \to \mathbb{R}$.

O-Notation - Motivation

Wir werden nachfolgend *konstante Faktoren* ignorieren. Diese 'verschwinden' in der O-Notation (auch *Landau*-Notation).

- $\Rightarrow$ Konzentration auf das wesentliche.
- $\Rightarrow$ Abstrahiert von verschiedenen Rechnerarchitekturen.
- $\Rightarrow$ Guter Vergleich von Algorithmen möglich (praxisbewährt).
- $\Rightarrow$ Ferner werden wir 'anfängliche Schwankungen' ignorieren können.
  - ABER: $5 \cdot n$ und $5000 \cdot n$ wird als 'im Prinzip' gleich angesehen werden!

O-Notation - Definition

Definition (O-Notation (und Verwandte) - Die wichtigen)

- $O(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\Omega(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|\}$

Definition (O-Notation (und Verwandte) - Die nicht ganz so wichtigen)

- $o(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\omega(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$

Bemerkung

f ist dabei stets eine Funktion von $\mathbb{N}$ nach $\mathbb{R}$, also $f : \mathbb{N} \to \mathbb{R}$.

Wichtige Funktionen

Bemerkung

Wichtige Funktionen in der Algorithmik/Algorithmenanalyse:

- Konstante Funktionen
- Logarithmen ($\log, \ln$)
- Wurzelfunktionen ($\sqrt{n}$)
- Polynome (beachte: $\sqrt{n} = n^{1/2}$)
- Exponentialfunktionen (2^n)
- ... Kombinationen davon

In welchem Zusammenhang stehen diese bzgl. der O-Notation?
($\Rightarrow$ Modul A&D!)

Zum Logarithmus

Nebenbemerkung

Mit log werden wir stets eine Variante des Logarithmus zur Basis 2 meinen, nämlich:

$$\log(n) := \begin{cases} 1 & \text{, falls } n \leq 1 \\ \lfloor \log_2(n) \rfloor + 1 & \text{, sonst.} \end{cases}$$

Damit ist $log(n)$ die Länge der Binärdarstellung einer natürlichen Zahl n. (Die Anzahl von Speicherstellen/Schritten ist stets eine natürliche Zahl!) Andere Logarithmen werden nur selten benötigt und sind bis auf einen konstanten Faktor ohnehin gleich. Die Rechengesetze wie z.B. $\log(x \cdot y) = \log(x) + \log(y)$ und $\log(x^r) = r \cdot \log(x)$ sind oft hilfreich.

O-Notation - Beispiele I

$$\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in R^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Beispiel

Wir wollen $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$ zeigen. Wir suchen also Konstanten c_1, c_2, n_0, so dass

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

für alle $n \geq n_0$ erfüllt ist. Teilen durch n^2 führt zu

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

. Dies kann man z.B. mit $c_1 = \frac{2}{8}, c_2 = 1, n_0 = 24$ erfüllen.

O-Notation - Variante

Satz

1. $f(n) \in O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
2. $f(n) \in \Omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
3. $f(n) \in \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^+$
4. $f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
5. $f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

O-Notation - Beispiele II

Definition

$f(n) \in O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

Beispiel

Wir wollen $\frac{1}{2}n^2 - 3n \in O(n^2)$ zeigen.

$$\lim_{n\to\infty} \frac{1/2n^2 - 3n}{n^2} = \lim_{n\to\infty} \frac{1}{2} - \frac{3}{n} = \frac{1}{2}$$

Ähnlich zeigt man $\frac{1}{2}n^2 - 3n \in \Omega(n^2)$ woraus das gleiche Ergebnis wie oben folgt.

Merkhilfe

Bemerkung

Eine kleine *Merkhilfe* (nicht mehr!)

- $f \in O(g) \approx f \leq g$
- $f \in \Omega(g) \approx f \geq g$
- $f \in \Theta(g) \approx f = g$
- $f \in o(g) \approx f < g$
- $f \in \omega(g) \approx f > g$

Wir sagen, dass f asymptotisch kleiner gleich, größer gleich, gleich, kleiner bzw. größer ist als g, wenn $f \in O(g), f \in \Omega(g), f \in \Theta(g), f \in o(g)$ bzw. $f \in \omega(g)$ gilt.

Algorithmus 1: Lineare Suche

Algorithmus 2 Algorithmus 1

```
1: for i = 0 to n do
2:     if a[i] == max mustermann then
3:         return true
4:     end if
5: end for
6: return false
```

Analyse

Laufzeit ist linear in der Länge n des Arrays, d.h. in $O(n)$ oder genauer sogar in $\Theta(n)$. (*Korrektheit* ist noch zu zeigen. Dazu in A&D mehr...)

Algorithmus 2: Fakultät

Algorithmus 3 `fac`(n)

```
1: res = 1
2: for i = 1 to n do
3:     res = res · i
4: end for
5: return res
```

Bemerkung

Die Laufzeit ist in $O(n)$. Aber Achtung! Dies ist **exponentiell in der Größe der Eingabe**! Diese ist nämlich nur in $O(\log n)$.

Vorgehen

Vorgehen, wenn wir einen Algorithmus analysieren:

1. Überlegen bzgl. welcher *Kenngröße* der Eingabe wir messen wollen. Diese kann sich von der Größe der Eingabe unterscheiden! (Beispiel: Anzahl Knoten eines Graphen).
2. Laufzeit/Speicherbedarf bzgl. dieser Kenngröße ausdrücken.
3. Nochmal überlegen, ob dies die Aussage des Ergebnisses verfälscht (so wie bei der Fakultätsberechnung oben).

⇒ I.A. sollte sich die eigentliche Eingabegröße leicht durch die Kenngröße ausdrücken lassen und der Unterschied sollte nicht zu groß sein.

- \+ Beim Graphen mit n Knoten ist die Adjazenmatrix in $O(n^2)$.
- – Ist eine Zahl n die Eingabe, so ist die Eingabegröße in $O(\log n)$. Eine Laufzeit von $O(n)$ wäre also exponentiell in der Eingabe!

Fragen

Algorithmus 4 Find Maximum

```
1: max = 1
2: for i = 2 to n do
3:     if a[i] > a[max] then
4:         max = i
5:     end if
6: end for
7: return max;
```

Laufzeit?

1. n	2. $O(n)$	3. $O(n-1)$
4. $O(3 \cdot n)$	5. $O(n^2)$	6. nichts davon

Fragen

Algorithmus 5 Sortieren mit Max

```
1: for i = n downto 1 do
2:     idx = max(A)
3:     B[i] = A[idx]
4:     A[idx] = 0
5: end for
6: return B
```

Laufzeit?

1. $O(n)$
2. $O(n^2)$
3. $O(n \cdot \log n)$
4. $O(\log n)$
5. $O(2^n)$
6. nichts davon

Fragen

Das P in NP steht für Polynomialzeit. Wofür steht das N?

1. nicht(-polynomiell)
2. nicht-deterministisch

Fragen

Was gilt?

1. $P \subseteq NP$
2. $NP \subseteq P$
3. $P \cap NP = \emptyset$
4. unvergleichbar!

Fragen

Algorithmus 6 Suchraum durchsuchen (deterministisch!)

```
1: for all A ⊆ S do
2:     if ∑_{x∈A} x = ∑_{x∈Ā} x then
3:         return true
4:     end if
5: end for
6: return false
```

Laufzeit?

1. $O(n)$
2. $O(n^2)$
3. $O(n \cdot \log n)$
4. $O(n^3)$
5. $O(2^n)$
6. nichts davon

Fragen

Algorithmus 7 Suchraum durchsuchen (nichtdeterministisch!?)

```
1: for all A ⊆ S do
2:    if ∑_{x∈A} x = ∑_{x∈Ā} x then
3:       return true
4:    end if
5: end for
6: return false
```

Wie schnell könnt ihr das gleiche Problem *nichtdeterministisch* lösen?
1. $O(\log n)$ 2. $O(n)$ 3. $O(n \cdot \log n)$
4. $O(n^2)$ 5. $O(2^n)$ 6. nichts davon

Zur Nachbereitung

Zur Nachbereitung

1. $O(n)$ (n ist die Länge des Arrays a)
2. $O(n^2)$ (ebenso)
3. nichtdeterministisch
4. $P \subseteq NP$ gilt. (Ob auch $NP \subseteq P$ gilt, ist offen...)
5. $O(2^n)$ (n ist die Größe der Menge S)
6. $O(n)$ (wie eben - im Algorithmus die Menge A raten!)

Die Klasse P

$$\begin{aligned} P &:= \{L \mid \text{es gibt ein Polynom } p \text{ und eine} \\ &\qquad p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L(A) = L\} \\ &= \cup_{i \geq 1} DTIME(n^i) \end{aligned}$$

Anmerkung

- Ein Algorithmus (eine TM) A akzeptiert eine Sprache L in *polynomialer Zeit*, wenn sie von A akzeptiert wird und zusätzlich ein $k \in \mathbb{N}$ existiert, so dass jedes $x \in L$ mit $|x| = n$ in Zeit $O(n^k)$ akzeptiert wird.
- Soll A die Sprache L entscheiden, wird für jeden String x der Länge n in Zeit $O(n^k)$ entschieden, ob $x \in L$ gilt (oder nicht).

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit entschieden}\}$

Beweis.

Die Richtung von rechts nach links ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Akzeptieren und Entscheiden

Satz

$P = \{L \mid L \text{ wird von einem Algo. in Polynomialzeit entscheidet}\}$

Anmerkung

Man beachte, dass der obige Beweis nicht konstruktiv war! Wenn man weiss, dass ein Algorithmus mit polynomialer Laufzeit existiert, weiss man nicht zwingend wie dieser aussieht geschweige denn, wie die Schranke aussieht.

Ein Problem in P

Typisches Beispiel eines Problems in P:

$$\texttt{PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus höchstens } k \text{ Kanten besteht.} \end{array} \}$$

Verifikation in polynomialer Zeit

$$\texttt{L-PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus } \textit{mindestens } k \text{ Kanten besteht.} \end{array} \}$$

Anmerkung

Schwierig zu lösen (zumindest effizient), aber gegeben ein Pfad, kann schnell *überprüft* werden, ob er die Kriterien erfüllt.

Verifikation in polynomialer Zeit

Definition (Verifikationsalgorithmus)

Ein *Verifikationsalgorithmus* A ist ein Algorithmus mit zwei Argumenten $x, y \in \Sigma^*$, wobei x die gewöhnliche Eingabe und y ein *Zertifikat* ist. A *verifiziert* x, wenn es ein Zertifikat y gibt mit $A(x, y) = 1$. Die von A verifizierte Sprache ist

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : A(x, y) = 1\}.$$

Anmerkung

Es geht also insb. um die Eingabe x. Diese bilden die Sprache. Das Zertifikat y kann vom Algorithmus genutzt werden, um zu entscheiden, ob $x \in L$ gilt, oder nicht.

Die Klasse *NP*

In *NP* sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (*NP*)

$L \in NP$ gdw. ein Algorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c

$$L = \{x \in \{0,1\}^* \mid \begin{array}{l}\text{es existiert ein Zertifikat } y \text{ mit } |y| \in O(|x|^c), \\ \text{so dass } A(x,y) = 1 \text{ gilt}\end{array} \}$$

gilt.

Nichtdeterminismus vs. Verifikation

Nichtdeterminismus

Ein nichtdeterministischer Algorithmus A kann 'raten' und so in einem Zustand z.B. eine Variable auf 0 und auf 1 setzen.

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent. Die Verifikationsidee wird oft in der Algorithmik benutzt.

NP-Probleme lösen

Satz

Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.

Beweis.

Beweisskizze/Idee: Ist $L \in NP$, so gibt es einen Verifikationsalgorithmus in $O(n^k)$ (n ist die Eingabelänge). Das Zertifikat y hat eine Länge in $O(|x|^c)$. Man geht alle $2^{O(|x|^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgortihmus aus. Dieses Verfahren ist in $2^{O(n^k)}$. □

Mengenpartitionsproblem - Deterministisch

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Algorithmus 8 Suchraum durchsuchen (deterministisch!)

```
1: for all A ⊆ S do
2:     if Σ_{x∈A} x = Σ_{x∈Ā} x then
3:         return true
4:     end if
5: end for
6: return false
```

Laufzeit ist *deterministisch* in $O(2^{|S|})$

Mengenpartitionsproblem - Nichtdeterministisch

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Algorithmus 9 Suchraum durchsuchen (nichtdeterministisch!)

```
1: Rate ein A ⊆ S
2: if ∑_{x∈A} x = ∑_{x∈Ā} x then
3:     return true
4: end if
5: return false
```

Laufzeit ist *nichtdeterministisch* in $O(|S|)$ (also in NP).

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?

P vs. NP

Alle oben genannten Probleme sind

- in NP - und damit schnell *nichtdeterministisch* lösbar

Die besten bekannten deterministischen Algorithmen

- benötigen aber exponentielle Laufzeit!

Geht es wirklich nicht schneller?!?

P vs. NP

Zusammenhänge zwischen P und NP:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der NP-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine NP-vollständige Sprache in P liegt.
- Alle oben genannten Probleme sind NP-vollständig...

Ausblick

Was wir getan haben und noch tun werden:

1. Wir wollten über Rechnung/Algorithmen reden können
   - Dafür haben wir die TM eingeführt
   - und gezeigt, dass sie das gleiche kann wie Computer
   - dass das vermutlich für alle Berechnungsmodelle gilt sagt die Church-Turing-These.
2. Dann wollten wir über das Berechenbare reden
   - was mit TMs gut ging
   - bis wir gemerkt haben, dass es unentscheidbare Probleme gibt.
3. Dann wollten wir zumindest die entscheidbaren effizient lösen
   - und dort wird uns die *NP*-vollständigkeit unsere Grenzen aufzeigen...