Die Klassen P und NP

Dr. Eva Richter

29. Juni 2012

Die Klasse P

- $P = DTIME(Pol)$ Klasse der Probleme, die sich von DTM in polynomieller Zeit lösen lassen
- nach Dogma die „praktikablen“ Probleme
- beim Übergang von einer deterministischen Einband- zu einer deterministischen Mehrbandturingmaschine höchstens polynomieller Zeitzuwachs
- beim Übergang von einer nichtdeterministischen zu einer deterministischen Turingmaschine exponentieller Zeitzuwachs
- wegen Zeithierchiesatz ist P echt verschieden von E.

Beschreibung von Algorithmen

- Algorithmen werden mit numerierten Teilen (Schritten oder Stadien) beschrieben
- Ein Stadium ist ähnlich wie ein Schritt einer TM, seine Implementierung auf TM braucht i.A. mehrere Schritte
- Um festzustellen, dass ein Algorithmus in P liegt, müssen wir
  1. Eine polynomielle obere Schranke für die Anzahl der nötigen Stadien angeben
  2. Überprüfen, ob die einzelnen Stadien in Polynomialzeit ausgeführt werden können
- Stadien werden so gewählt, dass die Analyse einfach ist.
- Vernünftige Kodierungen wählen.

Das Pfadproblem

Satz

$$PATH = \{< G, s, t > | G \text{ ist ein gerichteter Graph, in dem es einen gerichteten Pfad von } s \text{ nach } t \text{ gibt.}\}$$

$$PATH \in P.$$

- naiver Algorithmus ist brute-force Methode; für alle potentiellen Pfade in G wird getestet, ob sie Pfade von s nach t sind
- potentieller Pfad hat höchstens Länge m, m ist Anzahl der Knoten
- Zahl der potentiellen Pfade ist unter m^m, was exponentiell in der Anzahl der Knoten ist
- man verwendet Breitensuche: markiere nach und nach alle Knoten, die von s aus in 1, 2 usw. Schritten erreichbar sind

Algorithmus für das Pfadproblem

$M =$

$< G, s, t >$, wobei G ein gerichteter Graph und s und t Knoten sind:

1. Markiere s.
2. Wiederhole folgende Schritte bis kein neuer Knoten mehr markiert wird.
3. Durchsuche alle Kanten von G: wenn eine Kante (a, b) dabei ist, bei der a markiert ist und b unmarkiert, dann markiere b.
4. Falls t markiert ist, akzeptiere, sonst weise ab.

- Stadium 1 und 4 werden einmal, Stadium 3 wird höchstens m Mal ausgeführt, insgesamt mögliche Schritte $1 + 1 + m$
- Stadien 1 und 4 können in Polynomialzeit implementiert werden, 3 benötigt eine Prüfung der Eingabe und Tests, ob bestimmt Knoten markiert sind, was ebenfalls in polynomieller Zeit gemacht werden kann.

Teilerfremde Zahlen

Satz

Sei

$$RELPRIME = \{< x, y > | x \text{ und } y \text{ sind teilerfremd}\}$$

$RELPRIME \in P$.

- in Binärdarstellung wächst die Größe der Zahl exponentiell mit der Länge der Eingabe
- brute-force-Algorithmus ist daher exponentiell
- verwenden Euklidischen Algorithmus; falls g.g.T $(x, y) = 1$ dann ist x relativ prim zu y

Euklidischer Algorithmus

$E =$

$< x, y >$, wobei x und y natürliche Zahlen in Binärdarstellung sind:

1. Wiederhole bis $y = 0$.
2. Berechne $x := x \mod y$.
3. Vertausche x und y.
4. Gebe x aus.

Beispiel: Berechne größten gemeinsamen Teiler von 144 und 60:
$x : 144 \mod 60 = 24$
$x : 60 \mod 24 = 12$
$x : 24 \mod 12 = 0$
$G.g.T(144, 60) = 12$

Polynomialzeitalgorithmus für *RELPRIME*

$R =:$

$< x, y >$, natürliche Zahlen in Binärdarstellung:

1. Starte E auf $< x, y >$.
2. Falls das Ergebnis 1 ist accept, sonst reject.

- Jede Ausführung von Schritt 2 halbiert den aktuellen Wert von x:
  - nach 2 ist $x < y$, wegen Definition von $\mod$, nach 3 gilt dann $x > y$
  - Fall $x/2 \geq y$: es gilt $x \mod y < y \leq x/2$ und x wird mindestens halbiert
  - $x/2 < y$, dann $xmody = x - y < x/2$ und x wird mindestens halbiert
- wegen Vertauschung von x und y werden beide immer wieder halbiert; maximale Anzahl der Schleifen ist $\min\{log_2 x, log_2 y\}$.
- log ist proportional zur Länge der Darstellung, Schrittzahl ist in $O(n)$, alle Schritte von E sind polynomiell, $R \in P$.

Kontextfreie Sprachen

Satz

Jede kontextfreie Sprache liegt in P.

Beweisidee:

- naiver Algorithmus, der für ein Wort w der Länge n, mit $2n-1$ Ableitungsschritten alle möglichen Ableitungen durchsucht, ist exponentiell in n
- mit dynamischer Programmierung: alle Variablen, die zur Erzeugung der Teilwörter beitragen, werden $n \times n$ Tabelle eingetragen
- bei (i,j) stehen die Variablen, die das Wort $w_i \dots w_j$ erzeugen
- starte mit den Teilwörtern der Länge 1 und erzeuge nach und nach den Rest
- für Wörter der Länge $k+1$ werden alle möglichen Zerlegungen in zwei Teilwörter gemacht und alle Regeln der Form $A \to BC$ untersucht, ob B im Eintrag für das erste Teilwort und C im Eintrag für das zweite Teilwort steht

Algorithmus für Wortproblem in kontextfreien Sprachen

G sei kontextfreie Grammatik für Sprache L in CNF

$D =$

$w = w_1 \dots w_n$:

1. Falls $w = \epsilon$ und $S \to \epsilon \in G$ accept.
2. Für jedes $i = 1$ bis n
3. Für jede Variable A: teste, ob $A \to b \in G$ mit $b = w_i$, wenn ja, füge A zu $table(i,i)$ hinzu.
4. Für $l = 2$ bis n
5. Für $i = 1$ bis $n - 1 + l$:
6. sei $j = i + l - 1$
7. für $k = i$ bis j–1
8. für jede Regel $A \to BC$: wenn $B \in table(i,k)$ und $C \in table(k+1,j)$, dann füge A zu $table(i,j)$ hinzu.
9. Falls $S \in table(1,n)$, accept; wenn nicht reject.

Analyse des Algorithmus'

- jeder Schritt läuft in Polynomialzeit
- Schritt 4 und 5 werden höchstens $n \cdot v$ –mal ausgeführt, (v ist Zahl der Nichtterminale in G)– also eine Konstante und unabhängig von n, insgesamt in $O(n)$
- Schritt 4 wird höchstens n-mal ausgeführt, dabei läuft 5 höchstens n-mal;
- bei jeder Ausführung von 5 werden 6 und 7 höchstens n-mal ausgeführt,
- für jedes 7 läuft r-mal 8 (r ist Zahl der Regeln in G- eine Konstante)
- Anzahl für innerste Schleife 8 in $O(n^3)$
- Laufzeit von D liegt in $O(n^3)$.

Kann man brute-force-Suche vermeiden?

- für viele Algorithmen lässt sich brute-force-Suche vermeiden und man kann polynomielle Lösungen bekommen
- bei vielen Problemen war die Suche nach Ersatz für brute-force-Suche bisher nicht erfolgreich, unbekannt, ob Polynomialzeitalgorithmen existieren
- vielleicht gibt es für diese Probleme Polynomialzeitalgorithmen, die auf bisher unbekannten Prinzipien beruhen
- möglicherweise sind die Probleme von sich aus zu schwierig
- wichtige Beobachtung: Komplexität vieler Probleme ist miteinander verbunden

Beispiel HAMPATH

Definition

Ein Hamiltonscher Pfad in einem gerichteten Graphen G ist ein gerichteter Pfad, der genau einmal durch jeden Knoten geht.

$$HAMPATH = \{< G, s, t > \mid G \text{ ist gerichteter Graph mit einem Hamiltonschen Pfad zwischen } s \text{ und } t\}$$

- exponentieller Algorithmus für *HAMPATH* aus *PATH*-Algorithmus durch Hinzufügen der Prüfung, ob der gefundene Pfad hamiltonsch ist
- bisher ist nicht bekannt, ob sich *HAMPATH* auch in polynomieller Zeit lösen lässt
- *HAMPATH* ist polynomiell verifizierbar: man kann in polynomieller Zeit bestimmen, ob ein gegebener Pfad eine Lösung ist

Polynomielle Verifizierbarkeit

- ist wichtig, um die Komlexität von Algorithmen zu verstehen
- alles deutet darauf hin, dass das es viel einfacher ist, eine Lösung zu verifizieren, als zu bestimmen, ob es eine Lösung gibt
- weiteres Beispiel für polynomielle Verifizierbarkeit:

$$COMPOSITES = \{x \mid x = pq \text{ für ganze Zahlen } p, q > 1\}$$

(Seit 2004 ist bekannt, dass *PRIMES* in *P* liegt.)

- Antibeispiel: $\overline{HAMPATH}$ ist (bisher) nicht polynomiell verifizierbar

Polynomielle Verifizierer

Definition

1. *Ein Verifizierer für A ist ein Algorithmus V, sd.* $A = \{w \mid V \text{ akzeptiert } < w, c > \text{ für eine Zeichenkette } c\}$.
2. *Ein Polynomialzeit-Verifizierer läuft in Polynomialzeit in Abhängigkeit von der Länge von w.*
3. *Eine Sprache heißt polynomiell verifizierbar, wenn es einen Polynomialzeit-Verifizierer für sie gibt.*
4. *Die zusätzliche Information c heißt Zertifikat oder Beweis (Zeuge) für Enthaltensein in A.*

- Für polynomielle Verifizierer hat das Zertifikat polynomielle Länge (in der Länge von w).
- Für *HAMPATH* ist ein Hamiltonscher Pfad von s nach t ein Zertifikat. Für *COMPOSITES* ist einer der Teiler das Zertifikat.

Die Klasse NP

Definition

NP ist die Klasse der Sprachen, die polynomielle Verifizierer besitzen.

Zeigen später, dass *NP* mit $NTIME(Pol)$ übereinstimmt.

NTM für HAMPATH

N_1 =

„Bei Eingabe $< G, s, t >$, G ist gerichteter Graph mit Knoten s und t:

1. Schreibe eine Liste von m Zahlen $p_1 \dots p_m$, wobei m die Zahl der Knoten in G ist (jede Zahl wird nichtdeterministisch zwischen 1 und m gewählt).
2. Überprüfe, ob es Wiederholungen gibt, wenn ja reject.
3. Prüfe, ob $s = p_1$ und $t = p_m$, falls eine der Bedingungen nicht gilt, reject.
4. Für jedes i zwischen 1 und $m - 1$ prüfe, ob (p_i, p_{i+1}) eine Kante von G ist, falls ein Paar keine Kante von G ist, reject; sonst accept.“

- $NTime_M(w)$ ist gleich der Anzahl der Schritte des kürzesten akzeptierenden Pfades
- Schritt 1 ist polynomiell, einfacher Test in 2 und 3, zusammen polynomielle Zeit, Schritt 4 ist ebenfalls polynomiell.

$NP = NTIME(Pol)$

Satz

Eine Sprache liegt in genau dann NP , wenn sie durch eine nichtdeterministische Turingmaschine in Polynomialzeit entschieden werden kann.

Beweisidee:

- zeigen, wie man einen Polynomialzeit-Verifizierer in eine nichtdeterministische Turingmaschine umwandelt und umgekehrt
- NTM simuliert den Verifizierer durch Raten des Zeugen
- Verifier simuliert die NTM, indem es den akzeptierenden Pfad als Zeugen verwendet

$NP = NTIME(Pol)$-*Beweis I*

Sei $A \in NP$ und V der Verifier für A mit $NTime_V \in O(n^k)$

N =

„Bei Eingabe w der Länge n:

1. Wähle nichtdeterministisch ein Wort c der Länge n^k.
2. Starte V mit der Eingabe $< w, c >$.
3. Falls V akzeptiert, accept; sonst reject.“

$NP = NTIME(Pol)$-*Beweis II*

Sei N NTM, die A in polynomieller Zeit entscheidet

V =

„Bei Eingabe $< w, c >$, wobei w und c Wörter sind:

1. Simuliere N auf der Eingabe w und behandle dabei jedes Symbol von c als Beschreibung einer nichtdeterministischen Entscheidung in jedem Schritt
2. Wird dieser Zweig von N akzeptiert, accept; sonst reject.“

Die Klasse NP hängt nicht von der Wahl des Berechenbarkeitsmodells ab.

Beispiele für NP-Probleme: CLIQUE

Definition

Sei G ein ungerichteter Graph, dann ist eine Clique ein Teilgraph, bei dem je zwei Knoten durch eine Kante verbunden sind. Eine k-Clique ist ein Teilgraph mit k Knoten.

$$CLIQUE = \{< G, k > | G \text{ ist ein ungerichteter Graph, der eine } k\text{-Clique enthält } \}$$

Satz

CLIQUE liegt in NP.

Beweise für CLIQUE

Die Clique ist das Zertifikat bzw. der Zeuge.

$V =$

„Bei Eingabe $<< G, k >, c >$:

1. Prüfe, ob c eine Menge von k Knoten in G ist.
2. Prüfe, ob alle Verbindungskanten von c in G enthalten sind.
3. accept, wenn 1 und 2 erfolgreich sind; sonst reject.“

$N =$

„Bei Eingabe $< G, k >$, wobei G ein Graph und $k \in \mathbb{N}$:

1. Wähle nichtdeterministisch eine Teilmenge c mit k Knoten aus der Knotenmenge von G.
2. Überprüfe, ob alle Kanten, die Knoten von c verbinden, in G enthalten sind.
3. Wenn ja accept, wenn nein reject.“

Beispiele für NP-Probleme: SUBSET − SUM

$$SUBSET - SUM = \{< S, t > | S = \{x_1 \dots x_k\} \text{ und für eine Menge } \{y_1, \dots, y_c\} \subseteq \{x_1 \dots x_k\} \text{ gilt } \Sigma_i y_i = t\}$$

Die Mengen sind Multimengen!

Satz

SUBSET − SUM liegt in NP.

$< \{4, 11, 16, 21, 27\}, 25 >$ liegt in *SUBSET − SUM* wegen $4 + 21 = 25$.

Beweise für SUBSET − SUM

Verifizierer

$V =$

„Bei Eingabe $<< S, t >, c >$:

1. Prüfe, ob c eine Menge von Zahlen ist, die sich zu t summieren, wenn nicht
2. Prüfe, ob S alle in c vorkommenden Zahlen enthält, wenn nicht reject; sonst accept.“

nichtdeterministische Turingmaschine

$N =$

„Bei Eingabe $< S, t >$:

1. Wähle nichtdeterministisch eine Teilmenge c aus S aus.
2. Prüfe, ob die Zahlen von c sich zu t aufaddieren, wenn nicht reject, sonst accept.“

Die Klasse $co - NP$

- Man sieht nicht ohne weiteres, ob die Komplemente dieser Mengen, d.h. $\overline{CLIQUE}$ und $\overline{SUBSET - SUM}$ in NP liegen.
- Festzustellen, dass irgendetwas nicht existiert, scheint schwieriger zu sein, als festzustellen, dass etwas vorhanden ist.

Definition

$co - NP$ ist die folgende Klasse von Problemen:

$$co - NP = \{L | \overline{L} \in NP\}$$

Die P-ungleich- NP-Frage

P = Klasse der Sprachen, bei denen Zugehörigkeit schnell entschieden werden kann.
NP = Klasse der Sprachen, bei der Zugehörigkeit schnell überprüft werden kann.

- Beispiele *HAMPATH* und *CLIQUE* liegen die in NP, unbekannt ob auch in P.
- Anscheinend ist NP größer als P, aber könnten auch gleich sein. Bisher ist kein Problem bekannt, das in NP liegt, aber mit Sicherheit nicht in P.
- Wäre $P = NP$, wären polynomiell verifizierbare Probleme auch polynomiell entscheidbar, d.h. Beweissuche genauso schwer wie Beweisverifikation.
- Allgemein lässt sich zeigen $NP \subseteq EXPTIME = \bigcup_k DTIME(2^{n^k})$ aber es ist nicht bekannt, ob es nicht eine kleinere Klasse gibt, in der NP enthalten ist.

NP-Vollständigkeit

- Cook und Levin entdeckten bestimmte Probleme, deren individuelle Komplexität mit der der ganzen Klasse in Beziehung steht.
- Die Probleme dieser Klasse heißen NP-vollständig, NP-Vollständigkeit ist sowohl vom praktischen als auch vom theoretischen Aspekt her wichtig.
- Wenn man von einem NP-Problem sagen kann, dass es mehr als polynomielle Zeit braucht, dann von einem NP-vollständigen.
- Umgekehrt muss man einen P-Algorithmus finden, wenn man zeigen will, dass $P = NP$.
- Für NP-vollständiges Probleme kann man sich die Mühe, einen effizienten Algorithmus zu finden sparen.

Das Erfüllbarkeitsproblem

- Boolesche Variablen können Werte *TRUE* and *FALSE* annehmen (1und 0)
- Boolesche Operatoren AND, OR und NOT werden durch $\wedge, \vee$ und $\neg$ bzw. $\overline{x}$ dargestellt.
- Eine Boolesche Formel ist ein Ausdruck der aus Boolschen Variablen und Operatoren besteht, wie z.B.

$$\varphi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

- Eine Formel heißt erfüllbar, wenn es eine Belegung der Variablen mit 0 und 1 gibt, so dass die Formel den Wert 1 bekommt.

$$SAT = \{\varphi \mid \varphi \text{ ist erfüllbar}\}$$

Satz

(Cook-Levin) $SAT \in P$ genau dann, wenn $P = NP$.

Polynomialzeit – Reduzierbarkeit

Definition

Seien $A, B \subseteq \Sigma^*$. Sei FP die Menge der polynomialzeit-berechenbaren totalen Funktionen $f : \Sigma^* \to \Sigma^*$.
Dann heißt A in Polynomialzeit funktional reduzierbar (bzw. polynomiell reduzierbar) auf B, geschrieben als $A \leq_P B$, wenn es eine Funktion $f \in FP$ gibt, so dass für alle $w \in \Sigma^*$ gilt, dass $w \in A$ genau dann, wenn $f(w) \in B$.
Die Funkton f heißt polynomielle Reduktion von A auf B

Wenn f effizient berechenbar ist, kann die Lösung von B auf effiziente Weise in eine Lösung von A verwandelt werden.

Polynomialzeit – Reduzierbarkeit

Satz

Sei $A \leq_P B$ und $B \in P$. Dann gilt $A \in P$.

Sei M ein polynomieller Entscheider für B und sei f eine polynomielle Reduktion von A auf B.
Beweis:

$N :=$

„Auf Eingabe von w:
1. Berechne $f(w)$.
2. Starte M auf $f(w)$, gebe aus, was M ausgibt.“

N läuft in polynomieller Zeit, da jeder der beiden Schritte polynomiell ist. □

Das 3SAT-Problem

- *3SAT* ist eine spezielle Variante von *SAT*
- Formeln, deren Erfüllbarkeit festgestellt werden soll haben eine bestimmte Form.
- Ein Literal ist eine negierte oder nichtnegierte Boolesche Variable, wie x oder $\overline{x}$.
- Eine Klausel ist die Oder-Verknüpfung von Literalen wie $x_1 \vee x_2 \vee x_3$.
- Formel ist in konjunktiver Normalform, wenn sie aus der konjunktiven Verknüpfung von Klauseln besteht.
- Falls jede Klausel genau drei Literale enthält, sagen wir die Formel ist in 3-CNF.

$3SAT = \{\varphi \mid \varphi$ ist eine erfüllbare Boolesche Formel in 3-CNF$\}$

Reduktion von 3SAT auf CLIQUE

Satz

3SAT ist polynomiell reduzierbar auf CLIQUE.

Beweisidee:

- Reduktionsfunktion ordnet den Formeln geeignete Graphen zu.
- In den Graphen entsprechen Cliquen (mit fester bestimmter Größe) erfüllenden Belegungen.
- Strukturen innerhalb der Graphen korrespondieren mit Abhängigkeiten von Variablen und Klauseln.

Beweis durch Reduktion

- $\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$ wobei a_i, b_j, c_l (möglicherweiese gleiche) Literale
- f erzeugt einen String $< G, k >$, wobei G ein ungerichteter Graph ist, der wie folgt definiert wird:
  1. pro Klausel entsteht Dreiergruppe von Knoten, mit den Namen der Literale
  2. alle Knoten werden miteinander verbunden, außer:
     1. den Knoten einer Klausel,
     2. Literal und seiner Negation

Beweis durch Reduktion

Wenn es erfüllende Belegung für die Formel gibt, existiert Clique.

- wähle aus jedem Tripel ein Element , das mit wahr belegt ist für Clique, ergibt k Knoten
- jedes Paar der gewählten Knoten ist verbunden
- wegen der beiden Kantenverbote gibt es keine Widersprüche.

Falls Cliquen existiert, dann gibt es erfüllende Belegung.

- Knoten müssen alle in verschieden Gruppen liegen
- Belege die entsprechenden Variablen mit wahr, macht jede Klausel wahr.
- Ist eine gültige Belegung, weil keine Kanten zwischen x und $\neg x$.

Definition von NP-Vollständigkeit

Definition

Eine Sprache B heißt NP-vollständig, wenn sei zwei Bedingungen erfüllt:

(i) B ist in NP

(ii) Jedes A in NP ist polynomiell reduzierbar auf B.

Satz

Falls B NP-vollständig ist und $B \in P$, dann gilt $P = NP$.