6.0 VU Theoretische Informatik (192.017)

Teil 3: Einführung in Komplexitätstheorie

Stefan Woltran

Institut für Logic and Computation

Wintersemester 2023

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

Wiederholung: Entscheidungsproblem

Definition von “Problemen”

Ein Problem ist definiert durch eine (abzählbar) unendliche Menge von möglichen Instanzen (d.h.: möglichen Inputs) zusammen mit einer Frage.

Ein Entscheidungsproblem ist ein Problem, bei dem die Frage eine ja/nein Antwort erwartet.

Beispiel für ein Problem

GRAPH-ERREICHBARKEIT:
INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G einen Pfad von u nach v?

GRAPH-ERREICHBARKEIT ist ein Entscheidungsproblem.

Wiederholung: Algorithmen

Definition eines Algorithmus

Ein Algorithmus für ein Problem $\mathcal{P}$ ist eine Beschreibung von Rechenschritten, die es uns erlauben, jede beliebige Instanz des Problems $\mathcal{P}$ zu lösen.

- Ein Algorithmus muss auf alle Instanzen des Problems anwendbar sein und dabei ...
  1. nach endlich vielen Schritten terminieren und
  2. die korrekte Antwort auf die Frage liefern.

Wiederholung: Unsere Programmiersprache SIMPLE

- Wir verwenden den Kern von prozeduralen Programmiersprachen:
  - Variablen von unterschiedlichem Typ (String, Integer, Boolean, Void)
  - Wertzuweisungen, einfache Rechenoperationen (z.B.: $x := y + z$)
  - "if/then/else" Anweisungen
  - "while" Schleifen
  - "for" und "repeat" Schleifen
    (können durch "while" Schleifen ersetzt werden)
  - "return" Anweisungen
  - Prozeduren, Funktionen
- Der Input für ein Programm kann sein:
  - eine Liste $L = (V_1, V_2, \ldots, V_n)$ von Werten unterschiedlicher Typen;
  - oder einfach ein (großer) String I: Jede Liste L lässt sich als String über einem beliebigen "Alphabet" darstellen (z.B.: ASCII, $\{0, 1\}$, etc.)
- Ein Programm liefert als Ergebnis einen Wert (von beliebigem Typ) mittels "return" Anweisung.

Wiederholung: Unsere Programmiersprache SIMPLE

GRAPH-ERREICHBARKEIT:
INSTANZ: Ein Graph $G = (V, E)$ und Knoten $u, v \in V$.
FRAGE: Gibt es im Graph G einen Pfad von u nach v?

$S := \{u\}$;
repeat
    $S' := S$;
    for all $i \in S'$ **do** $\{$
        for all $j \in V$ **do** $\{$
            if $(i, j) \in E$ **then** $S := S \cup \{j\}$;
        $\}$
    $\}$
until $S = S'$;
if $v \in S$ **then** return *true*;
else return *false*;

Church-Turing These

Church-Turing These - Informelle Variante

Jeder Algorithmus lässt sich mittels SIMPLE realisieren.

Alternativ: Jedes “angemessene” formale Modell eines Algorithmus ist gleichmächtig (äquivalent) zur SIMPLE Programmiersprache.

Erweiterte Church-Turing These

Jedes “angemessene” formale Modell eines Algorithmus ist von seinem Zeit/Speicherbedarf äquivalent (modulo eines polynomiellen Overheads) zur SIMPLE Programmiersprache.

Wiederholung: Reduktionen

Idee 1 (nicht auf die Theoretische Informatik beschränkt)

- Angenommen, wir wollen ein neues Problem A lösen, (d.h.: wir wollen einen Algorithmus für das Problem A entwickeln).
- Und angenommen, wir wissen, wie man ein verwandtes Problem B löst (d.h.: wir haben bereits einen passenden Algorithmus für das Problem B).
- Idee. Wir könnten versuchen, A zu lösen, indem wir es zum Problem B umformen, (d.h.: wir entwickeln einen Algorithmus für Problem A, der den Algorithmus für Problem B verwendet).

Schlussfolgerung: Wenn diese Strategie funktioniert, sagen wir:
"Problem A wird auf Problem B reduziert" und wir schreiben $A \leq B$.
$\Longrightarrow$ Problem A ist mindestens so leicht lösbar ist wie Problem B.

Wiederholung: Reduktionen

Idee 2 (typisch für Berechenbarkeit und Komplexität)

- Angenommen, es gelingt uns nicht, für ein neues Problem B ein geeigtnes Lösungsverfahren zu finden, d.h.: wir finden keinen (effizienten) Algorithmus oder nicht einmal ein Semi-Entscheidungsverfahren für Problem B.
- Und angenommen, wir wissen, dass ein verwandtes Problem A *schwer zu lösen* ist, d.h.: es wurde bereits bewiesen, dass es keinen (effizienten) Algorithmus oder nicht einmal ein Semi-Entscheidungsverfahren für Problem A gibt.
- Idee. Wir entwickeln einen Algorithmus für Problem A, der einen (hypothetischen) Algorithmus für Problem B verwendet.

Schlussfolgerung: Wenn diese Strategie funktioniert, sagen wir wiederum: "Problem A wird auf Problem B reduziert" und wir schreiben $A \leq B$.
$\Longrightarrow$ Problem B ist mindestens so schwer lösbar ist wie Problem A.

Wiederholung: Reduktionen

Beschränkung der Ressourcen

- Eine Problemreduktion von A nach B sollte einfacher sein als die beiden betrachteten Probleme.
- Wenn das nicht der Fall wäre, könnte die Reduktion einen Teil der Komplexität von A beseitigen und ein Vergleich zwischen (den Schwierigkeitsgraden von) A und B wäre nicht mehr möglich.

Typische Anforderung in der Berechenbarkeitstheorie

Reduktionen müssen berechenbar sein.

Typische Anfordering in der Komplexitätstheorie

Reduktionen müssen effizient berechenbar sein, z.B. in polynomieller Zeit, d.h.: $O(n^k)$ für Instanzen der Größe n und Konstante $k \geq 1$.

Wiederholung: Reduktionen

“Many-One” Reduktionen

- Definiere eine Funktion R von der Menge der Instanzen von Problem A auf die Menge der Instanzen von Problem B, d.h.: jede Instanz x von Problem A wird auf eine Instanz $R(x)$ von Problem B abgebildet.
- Wenn man nun die Instanz $R(x)$ mit einem Lösungsverfahren für Problem B löst, dann ist das Ergebnis für die Instanz $R(x)$ bereits das korrekte Ergebnis für die Instanz x des Problems A.
- In diesem Fall sagen wir: x und $R(x)$ sind äquivalent.
  Für Entscheidungsprobleme A und B bedeutet das: x ist eine positive Instanz von $A \Leftrightarrow R(x)$ ist eine positive Instanz von B.
- Ressourcenbeschränkung: Die Funktion R muss (effizient) berechenbar sein. Im Fall einer Beschränkung auf polynomielle Zeit, heißt so eine Reduktion “Karp Reduktion”.

Aussagenlogik

Syntax

Die Syntax der Aussagenlogik (d.h. die Menge der wohl-geformten aussagenlogischen Formeln) basiert auf folgenden Symbolen:

- Boole'sche Variable (oder Atome): $X = \{x_1, x_2, \ldots\}$.
- Boole'sche Konnektive: $\vee$, $\wedge$, und $\neg$.

Die Menge der aussagenlogischen Formeln ist die kleinste Menge, die folgenden Bedingungen genügt:

- alle Atome sind aussagenlogische Formeln;
- wenn ϕ_1 und ϕ_2 aussagenlogische Formeln sind, dann sind dies auch $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$ und $(\phi_1 \vee \phi_2)$.

Im folgenden werden wir einfachheitshalber von Formeln und Atomen sprechen. Weiters werden wir übliche Konventionen (weglassen von Klammern, etc.) nutzen.

Aussagenlogik

- Eine Formel der Form form x_i oder $\neg x_i$ nennen wir Literal.
- Eine Disjunktion von Literalen nennen wir eine Klausel
- Eine Konjunktion von Klauseln nennen wir eine Konjunktive Normalform (KNF).
- Wenn jede Klausel aus n Literalen besteht, nennen wir eine KNF auch n-KNF.
- Eine 3-KNF Formel ist daher von der Form

$$\bigwedge_{i=1}^{m} \left(l_{i,1} \vee l_{i,2} \vee l_{i,3} \right)$$

Aussagenlogik

Wie lassen sich aussagenlogische Formeln auswerten?
Beobachtung: Formeln sind Aussagen die entweder wahr oder falsch sind und setzen sich aus Atomen zusammen, die ebenfalls entweder wahr oder falsch sein können.

Definition

Eine Wahrheitsbelegung I ist eine Funktion $I \colon X \to \{\mathbf{true}, \mathbf{false}\}$.

$I(\cdot)$ wird induktiv von Atomen auf beliebige Formeln erweitert:

- Für $\phi = \neg\phi_1$ gilt, dass $I(\phi) = \mathbf{true}$ genau dann wenn (gdw) $I(\phi_1) = \mathbf{false}$.
- Für $\phi = \phi_1 \wedge \phi_2$ gilt, dass $I(\phi) = \mathbf{true}$ gdw $I(\phi_1) = I(\phi_2) = \mathbf{true}$.
- Für $\phi = \phi_1 \vee \phi_2$ gilt dass $I(\phi) = \mathbf{true}$ gdw $I(\phi_1) = \mathbf{true}$ oder $I(\phi_2) = \mathbf{true}$.

Wenn wir von einer konkreten Formel ϕ sprechen, werden wir üblicherweise die Wahrheitsbelegung auf jene Atome beschränken, die in ϕ vorkommen. Wir sprechen dann von einer Wahrheitsbelegung für ϕ.

Einige klassische Entscheidungsprobleme der Aussagenlogik

MODEL-CHECKING

INSTANZ: Aussagenlogische Formel ϕ und Wahrheitsbelegung I für ϕ.
FRAGE: Gilt $I(\phi) = \textbf{true}$ (d.h. ist ϕ wahr unter I)?

SAT (SATISFIABILITY)

INSTANZ: Aussagenlogische Formel ϕ.
FRAGE: Ist ϕ erfüllbar? (d.h.: gibt es eine Wahrheitsbelegung I für ϕ, sodass $I(\phi) = \textbf{true}$).

3-SAT

INSTANZ: Aussagenlogische Formel ϕ in 3-KNF.

FRAGE: Ist ϕ erfüllbar?

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

k-Färbbarkeit

Sei $k \in \mathbb{N}$ mit $k \geq 2$. Dann ist k-Färbbarkeit folgendermaßen definiert:

Instanz: ungerichteter Graph $G = (V, E)$

Frage: Ist der Graph G k-färbbar?

d.h.: gibt es eine Funktion $f: V \to \{0, \ldots, k-1\}$, so dass für alle Kanten $[v_i, v_j] \in E$ gilt: $f(v_i) \neq f(v_j)$.

3-COL (Drei-Färbbarkeit)

Instanz: ungerichteter Graph $G = (V, E)$

Frage: Ist G drei-färbbar? Das heißt, können wir jedem Knoten aus V eine von 3 Farben so zuordnen, dass alle adjazenten Knoten verschieden gefärbt sind?

Anm: Im folgenden verwenden wir als Farben $\{r, g, b\}$ anstatt $\{0, 1, 2\}$.

Wiederholung: Vorgangsweise bei Korrektheitsbeweisen

- Zur Erinnerung: eine Many-One Reduktion R von Problem A auf Problem B ist korrekt, wenn für jede beliebige Instanz x von A die Äquivalenz gilt: x ist eine positive Instanz von $A \Leftrightarrow R(x)$ ist eine positive Instanz von B.
- Üblicherweise zeigt man die 2 Richtungen "$\Rightarrow$" und "$\Leftarrow$" der Äquivalenz getrennt.
- Der Beweis beginnt immer mit der Annahme "Sei x eine positive Instanz von A." bzw. "Sei $R(x)$ eine positive Instanz von B.".
- Mittels *schlüssiger* Beweisschritte versucht man zu zeigen, dass man dann auch beim anderen Problem eine positive Instanz hat.
- Typische Beweisschritte sind, wenn etwas *aufgrund der Annahme*, *aufgrund einer Definition* (z.B. einer "Lösung") oder *aufgrund der Problemreduktion* gilt.
- Es erhöht die Lesbarkeit eines Beweises, wenn man im Beweis klarstellt, was man gerade möchte bzw. was noch zu zeigen ist.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

Problemreduktion

Sei $G = (V, E)$ eine beliebige Instanz von 3-COL, d.h.: $G = (V, E)$ ist ein ungerichteter Graph mit Knoten $V = \{v_1, \ldots, v_n\}$ und Kanten E.

Für die zu konstruierende Formel verwenden wir Atome a_i^c, mit $1 \leq i \leq n$ und $c \in \{r, g, b\}$. Intuition: Wenn a_i^c auf wahr gesetzt ist, dann ist Knoten v_i mit Farbe c gefärbt.

Wir definieren Reduktion $R(\cdot)$, die für jeden Graph G, eine SAT-Instanz $R(G) = \phi_G$ liefert (in polynomieller Zeit), wobei $\phi_G = \phi_1 \wedge \phi_2 \wedge \phi_3$ mit

$$\phi_1 = \bigwedge_{1 \leq i \leq n} (a_i^r \vee a_i^g \vee a_i^b),$$

$$\phi_2 = \bigwedge_{1 \leq i \leq n} \big(\neg(a_i^r \wedge a_i^g) \wedge \neg(a_i^r \wedge a_i^b) \wedge \neg(a_i^g \wedge a_i^b)\big),$$

$$\phi_3 = \bigwedge_{[v_i, v_j] \in E} \big(\neg(a_i^r \wedge a_j^r) \wedge \neg(a_i^g \wedge a_j^g) \wedge \neg(a_i^b \wedge a_j^b)\big).$$

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

Wir müssen folgende Äquivalenz zeigen:
$G = (V, E)$ ist eine positive Instanz von 3-COL $\Leftrightarrow$
ϕ_G ist eine positive Instanz von SAT

$\Rightarrow$: Angenommen $G = (V, E)$ ist eine positive Instanz von 3-COL. Dann gibt es eine Funktion $f: V \to \{r, g, b\}$, die eine gültige 3-Färbung für den Graph $G = (V, E)$ ist, d.h.: für jede Kante $[v_i, v_j]$ in G gilt $f(v_i) \neq f(v_j)$.
Wir konstruieren eine Wahrheitsbelegung I für ϕ_G:

$$I(a_i^c) = \begin{cases} \mathit{true} & \text{if } f(v_i) = c, \\ \mathit{false} & \text{if } f(v_i) \neq c. \end{cases}$$

und zeigen, dass $I(\phi_G) = \textbf{true}$.
Da $\phi_G = \phi_1 \wedge \phi_2 \wedge \phi_3$, müssen wir sowohl $I(\phi_1) = \textbf{true}$, $I(\phi_2) = \textbf{true}$, als auch $I(\phi_3) = \textbf{true}$ zeigen.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$I(\phi_1) =$ **true**: Betrachte beliebige Klausel $(a_i^r \vee a_i^g \vee a_i^b)$ aus ϕ_1.

Laut Annahme ist f eine gültige Dreifärbung von G. Also ist $f(v_i) \in \{r, g, b\}$.

Laut Definition von I gilt dann für mindestens eine der Atome (a_i^r, a_i^g, a_i^b), dass sie in I den Wahrheitswert true hat.

Daher erfüllt I die Klausel $(a_i^r \vee a_i^g \vee a_i^b)$.

Da I jede Klausel von ϕ_1 erfüllt, haben wir gezeigt, dass $I(\phi_1) =$ **true**.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$I(\phi_2) = \textbf{true}$: Betrachte beliebiges Konjunkt

$$c_i = \neg(a_i^r \wedge a_i^g) \wedge \neg(a_i^r \wedge a_i^b) \wedge \neg(a_i^g \wedge a_i^b)$$

aus ϕ_2 $(1 \leq i \leq n)$.

Laut Annahme ist f eine gültige Dreifärbung von G. Also ist $f(v_i) \in \{r, g, b\}$.

Laut Definition von I gilt dann für mindestens zwei der drei Atome (a_i^r, a_i^g, a_i^b), dass sie in I den Wahrheitswert false hat.

Daher sind sowohl $(a_i^r \wedge a_i^g)$, $(a_i^r \wedge a_i^b)$, als auch $(a_i^g \wedge a_i^b)$ falsch unter I. Es folgt, dass $I(c_i) = \textbf{true}$.

Da I jedes solche Konjunkt von ϕ_2 erfüllt, haben wir gezeigt, dass $I(\phi_2) = \textbf{true}$.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$I(\phi_3) = \textbf{true}$: Zur Erinnerung ...

$$\phi_3 = \bigwedge_{[v_i,v_j]\in E} \big(\neg(a_i^r \wedge a_j^r) \wedge \neg(a_i^g \wedge a_j^g) \wedge \neg(a_i^b \wedge a_j^b)\big).$$

Wir beweisen diesen Teil indirekt, also angenommen $I(\phi_3) = \textbf{false}$.
Dann ist zumindest eines der Konjunkte

$$c_{i,j} = \big(\neg(a_i^r \wedge a_j^r) \wedge \neg(a_i^g \wedge a_j^g) \wedge \neg(a_i^b \wedge a_j^b)\big)$$

falsch in I.

Es folgt, dass $(a_i^r \wedge a_j^r)$, $(a_i^g \wedge a_j^g)$, oder $(a_i^b \wedge a_j^b)$ wahr unter I.
Daher $I(a_i^c) = I(a_j^c) = 1$ für eine der drei Farben $c \in \{r, g, b\}$.

Laut Definition von I haben wir auch $f(v_i) = f(v_j) = c$.
Laut Problemreduktion enthält ϕ_3 Konjunkt $c_{i,j}$ nur dann, wenn es eine Kante $[v_i, v_j]$ in G gibt. Laut Annahme ist f eine gültige Dreifärbung von G, also $f(v_i) \neq f(v_j)$. Widerspruch!

Wir haben gezeigt, dass $I(\phi_G) = \textbf{true}$. Daher ist ϕ_G eine positive Instanz von SAT.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

$\Leftarrow$: Angenommen ϕ_G ist eine positive Instanz von SAT, gegeben Graph $G = (V, E)$. Dann gibt es eine Wahrheitsbelegung I, sodass $I(\phi_G) = \textbf{true}$.
Zuerst zeigt man dass aus $I(\phi_1) = \textbf{true}$ und $I(\phi_2) = \textbf{true}$ folgt: für jedes i gilt, dass genau eines der Atome aus (a_i^r, a_i^g, a_i^b) unter I den Wahrheitswert true hat.
Wir definieren nun eine Funktion $f : V \to \{r, g, b\}$ sodass, für alle $1 \leq i \leq n$, $f(v_i) = c$, wobei $c \in \{r, g, b\}$ genau jenes c mit $I(a_i^c) = \textbf{true}$ ist.

Aufwärmbeispiel: Reduktion von 3-COL auf SAT

Fortsetzung der $\Leftarrow$-Richtung:
Wir müssen noch zeigen, dass f eine gültige Drei-Färbung von G ist. Wir zeigen das abermals indirekt und nehmen an, dass es eine Kante $[v_i, v_j]$ in G gibt, sodass $f(v_i) = f(v_j) = c$.
Laut Definition von f, gilt daher dass $I(a_i^c) = I(a_j^c) = \textbf{true}$.
Laut Problemreduktion enthält ϕ_3 dann die Teilformel $\neg(a_i^c \wedge a_j^c)$. Es folgt, dass $I(\neg(a_i^c \wedge a_j^c)) = \textbf{false}$ und daher $I(\phi_G) = \textbf{false}$.
Laut Annahme ist aber $I(\phi_G) = \textbf{true}$. Widerspruch.
Es gilt daher, dass f eine gültige Drei-Färbung von G ist: somit ist G eine positive Instanz von 3-COL.

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung
Teil 3.2: Die Komplexitätsklassen P und NP
Teil 3.3: NP-Vollständigkeit
Teil 3.4: Weitere Komplexitätsklassen

Prolog

Komplexitätsanalyse

- Klassifiziert konkrete Probleme anhand deren Schwierigkeit
- Dies geschieht typischerweise mittels Komplexitätsklassen

Komplexitätstheorie

- Untersucht Eigenschaften von Komplexitätsklassen
- ... und Beziehungen zwischen Komplexitätsklassen

Wir verstehen unter Komplexität hier immer die Worst-Case Komplexität. Wir gehen nicht auf andere Aspekte ein, z.B. average-case Komplexität, best-case Komplexität, Komplexität für “Instanzen, die in der Praxis auftreten” . . .

Laufzeit und Speicherbedarf

Wir messen die die Effizienz von Algorithmen bzw. die Komplexität von Problemen in Bezug auf (Lauf)zeit und Speicher(bedarf):

Laufzeit

Gegeben Programm Π und Input I für Π, die Laufzeit von Π auf I ist die Anzahl von atomaren Operationen die für die Ausführung von Π auf I benötigt wird.

- Der Term "Atomare Operation" hängt von der Programmiersprache, bzw. der Computer Architektur ab.

Speicherbedarf

Gegeben Programm Π und Input I für Π, der Speicherbedarf von Π auf I ist die Anzahl der Bits die für die Ausführung von Π auf I im Speicher benötigt wird.

Komplexität eines Algorithmus

Messen von Komplexität

Wir basieren unsere Überlegungen auf die asymptotische, worst-case Komplexität eines Algorithmus, d.h. anhand einer Funktion $f(\,)$, sodass für jede Probleminstanz der Größe n, die Antwort in $\mathrm{O}(f(n))$ Operationen (Laufzeit), bzw. mittels $\mathrm{O}(f(n))$ Bits im Speicher (Speicherbedarf) berechnet werden kann.

Recall

Seien f, g: $\mathbf{N} \mapsto \mathbf{N}$.

- $g(n) = \mathrm{O}(f(n))$ (g wächst maximal wie f), wenn es c und n_0 gibt, sodass für alle $n \geq n_0$, $g(n) \leq c \cdot f(n)$.

Anmerkung: Da wir Laufzeit in Bezug auf die Instanzgröße messen, ist eine "vernünftige" Repräsentation der Instanz wichtig.
Vergleiche unäre vs. binäre oder dezimale Zahlendarstellung.

Komplexität eines (Entscheidungs)Problems

Komplexität eines Problems =
Worst-case Komplexität des bestmöglichen Algorithmus für dieses Problem.

Zentrale Unterscheidung:

- Tractable Problems - es gibt einen effizienten Algorithmus
- Intractable Problems - es gibt (vermutlich) keinen effizienten Algorithmus

Positive Resulte sind daher "einfach" zu zeigen.
Für negative Resulte brauchen wir formale Konzepte.

Vergleiche entscheidbare/unentscheidbare Probleme.

Nochmal: Erweiterte Church-Turing These

Frage

Beeinflußt das konkrete Berechnungsmodell (die konkrete Programmiersprache) oder eine spezifische Architektur die Zeit/Speicher-Performanz eines implementierten Algorithmus?

- Wir können dies i.A. vernachlässigen: die Auswahl einer spezifischen Programmiersprache/Architektur bringt nur eine geringfügige Verbesserung in Bezug auf Laufzeit/Speicherbedarf.

  Ein effizientes Programm kann in jede andere Programmiersprache (z.B. SIMPLE) übersetzt werden ohne dass Effizienz verloren geht.

- "effizient" bedeutet hier: polynomiell
- Daher können wir für unsere Analysen in Bezug auf Existenz/Nicht-Existenz von effizienten Algorithmen für ein konkretes Problem auf SIMPLE als Berechnungsmodell basieren.

Die Komplexitätsklasse P

Die Klasse P ist die Menge aller Entscheidungsprobleme die in polynomieller Zeit (in Bezug auf die Instanzgröße) gelöst werden können.

Notation: $|I|$ ist die Größe einer Instanz I.

Definition

Die Klasse P beinhaltet genau jene Entscheidungsprobleme $\mathcal{P}$ die folgende Eigenschaften haben:

1. es gibt ein SIMPLE Programm Π für $\mathcal{P}$, sodass
2. für alle Instanzen I von $\mathcal{P}$ die Laufzeit von Π auf I polynomiell in $|I|$ ist, d.h. die Laufzeit ist durch $\mathrm{O}(|I|^k)$, wobei k eine Konstante, beschränkt.

Die Komplexitätsklasse P

Zahlreiche wichtige Entscheidungsprobleme sind in P:

- Graph-Erreichbarkeit ist in P;
- 2-SAT ist in P (recall Block1: Reduktion von 2-SAT auf Graph-Erreichbarkeit);
- Testen ob ein String aus einer kontextfreien Grammatik (recall Block2) abgeleitet werden kann, ist in P.
- Diskussion: Primzahlentest?

Entscheidungsprobleme aus P

Boolean Circuits

Ein Boolean Circuit der Arität k ist ein gerichteter azyklischer Graph (DAG) bestehend aus: k Input gates $in_1, \ldots, in_k$, einem Output gate, und einer beliebigen Anzahl von *AND*, *OR* und *NOT* gates.
Input gates haben keine Kind-gates.
NOT gates und das Output gate haben genau ein Kind-gate,
AND bzw. OR gates haben zumindest zwei Kind-gates.

CIRCUIT-EVAL

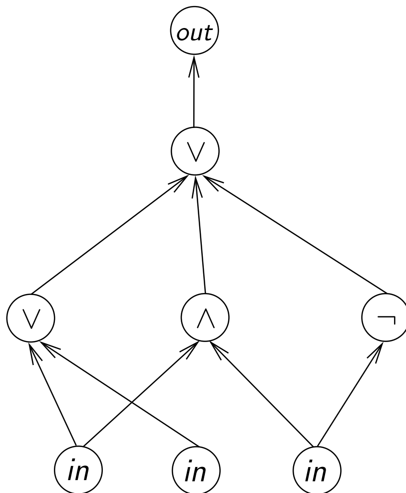
INSTANZ: Ein Boolean Circuit C und eine Funktion I die jedem Input gate von C einen Wert **true** oder **false** zuweist.

FRAGE: Ist der output von C **true** unter Belegung I?

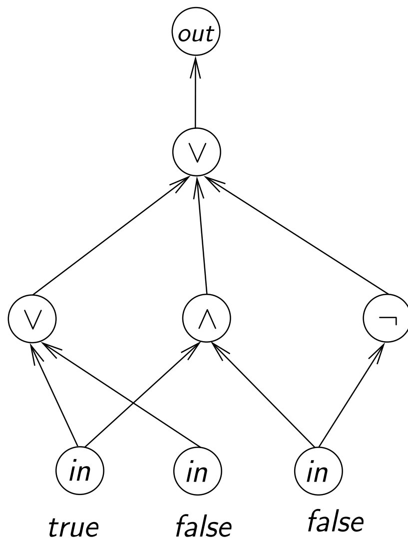
Belegungen werden im Circuit folgendermaßen propagiert:

- *AND* gate: Output **true** gdw alle Kind-gates Output **true**,
- *OR* gate: Output **true** gdw zumindest ein Kind-gate Output **true**,
- *NOT* gate: Output **true** gdw wenn das Kind-gate Output **false**.

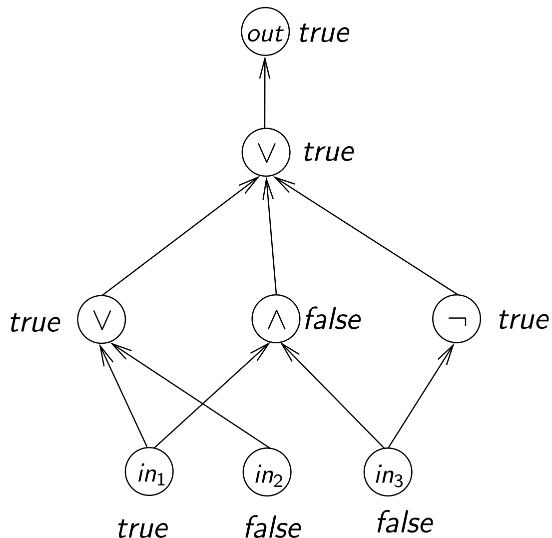
Beispiel für einen Circuit



Beispiel für einen Circuit mit Belegung



Beispiel für einen ausgewerteten Circuit



CIRCUIT-EVAL ist in P

Algorithmus **evaluateCircuit**

Input: Boolean Circuit C der Arität k, Belegung I für C
Output: **true** gdw Output von C ist **true** in I

1. Setze $A := \{(in_1, I(in_1)), \ldots, (in_k, I(in_k))\}$; /* Kopie von I */
2. Sei G die Menge aller Gates in C;
   /* jene $g \in G$ die in A aufscheinen nennen wir value-assigned */
3. **while** existiert $g \in G$ sodass g nicht value-assigned **do**
   - (i) wähle ein $g \in G$ das nicht value-assigned, dessen Kind gates aber allle bereits value-assigned sind (ein solches Gate muss existieren da C ein DAG mit den Input gates als Quellknoten)
   - (ii) gib zu A das Paar (g, v), wobei v der Wert der sich durch den Typ von g und der Werte der Kind gates von g gemäß A ergibt.
4. **if** $(out, true) \in A$ **then return** *true* **else return** *false*

CIRCUIT-EVAL ist in P

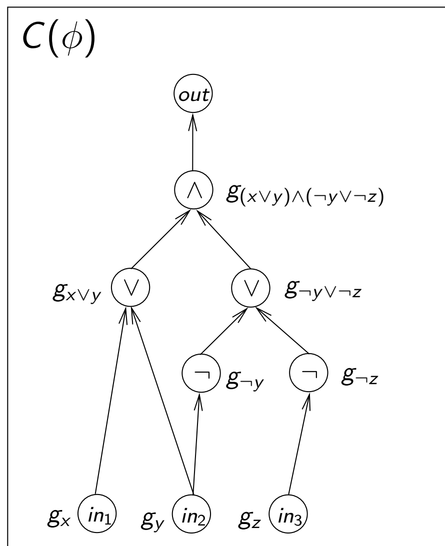
Wir argumentieren dass Algorithmus **evaluateCircuit** in polynomieller Zeit (in Bezug auf Instanzgröße) läuft:

1. Die Relation A ist zu jeder Zeit linear in der Größe von C.
2. Da jede Iteration der Schritte (i-ii) ein neues Gate belegt, haben wir maximal $|G|$ Iterationen des **while** Loops.
3. Gegeben Gate $g \in G$, um zu testen, dass g unbelegt und alle Kind-Gates belegt, kann in quadratischer Zeit (in Bezug auf die Größe von G) bewerkstelligt werden.
4. Aufgrund von (3) hat Schritt (i) kubische Laufzeit (durchsuche G und führe den Test des vorigen Punkts aus).
5. Schritt (ii) ist linear in der Größe von C.
6. Die Laufzeit des Algorithmus ist daher mit $\mathrm{O}(n^4)$ beschränkt.

Tatsächlich gibt es für dieses Problem einen $\mathrm{O}(n)$-Algorithmus.

Aussagenlogische Formulen und Boolean Circuits

Example:
$\phi = (x \vee y) \wedge (\neg y \vee \neg z)$



Model Checking ist in P

MODEL-CHECKING

INSTANZ: Aussagenlogische Formel ϕ und Wahrheitsbelegung I für ϕ.

FRAGE: $I(\phi) = \textbf{true}$?

Wir können nun folgern, dass **MODEL-CHECKING** in P.

- Wir reduzieren **MODEL-CHECKING** auf **CIRCUIT-EVAL**. Die Reduktion selbst ist klarerweise in polynomieller Zeit möglich.
- Weiters haben wir bereits argumentiert, dass **CIRCUIT-EVAL** in P.

Diskussion

- Ein Problem gilt als effizient lösbar (tractable) wenn es in P ist. Zur Erinnerung: das bedeutet, dass es einen Algorithmus für dieses Problem gibt, sodass die Lösungszeit für jede Instanz des Problems polynomiell zu dessen Größe n steht, also die Laufzeit durch $\mathrm{O}(n^d)$ beschränkt ist. (daher: asymptotic, worst-case performance).
- In der Praxis kann es vorkommen, dass dieses Konzept nur bedingt greift. Beispiel: Linear Programming.
  - Es existieren Algorithmen mit polynomieller worst-case Komplexität.
  - In der Praxis hat sich allerdings die Simplex Methode (exponentielle worst-case Komplexität) als brauchbarer herauskristallisiert.
- Wenn es für ein Problem keinen Polynomialzeit-Algorithmus gibt, nennen wir ein solches Entscheidungsproblem intractable.
  - Intractability (kein effizienter Algorithmus bekannt, Existenz eines solchen Algorithmus unwahrscheinlich, aber nicht auszuschliessen) $\longrightarrow$ Komplexitätsklasse NP ... folgt in Kürze.
  - beweisbare Intractability (i.e., Existenz eines effizienten Algorithmus kann formal ausgeschlossen werden) $\longrightarrow$ Komplexitätsklasse EXPTIME.

Die Komplexitätsklasse NP

Die Komplexitätsklasse NP

- Gegeben Formel ϕ und Wahrheitswertwertbelegung I: zu überprüfen, ob $I(\phi) = \textbf{true}$ ist effizient lösbar.
  - Recall: **MODEL-CHECKING** ist in P.
- Um Erfüllbarkeit (SAT) von ϕ zu entscheiden, könnten wir alle möglichen Wahrheitswertbelegungen für ϕ durchgehen und jeweils effizient prüfen ob die Belegung ϕ erfüllt.
- Naives Aufzählen aller Wahrheitswertbelegungen für ϕ braucht allerdings exponentielle Laufzeit:
  - 2^n mögliche Wahrheitswertbelegungen, wenn n die Anzahl der Atome in ϕ.
- Gibt es eine schlauere Strategie? Hard to say...
- Mit Glück finden wir die richtige Belegung möglichst bald und unser Algorithmus für SAT könnte sofort terminieren. (aber das hilft nicht bei negativen Instanzen).
- "Mit Glück" kann durch Nicht-Determinismus erzwungen werden. Tatsächlich steht NP für Non-deterministic Polynomial Time.

Die Komplexitätsklasse NP

Viele Probleme lassen sich durch sogenannte Zertifikate (auch Zeugen, Lösungen, etc) analysieren. Am Beispiel SAT:

- Wenn ϕ erfüllbar gibt es zumindest eine Wahrheitswertbelegung für ϕ, die ϕ wahr macht.
- Für unerfüllbare Formeln existiert keine solche Belegung.

Beobachtung 1: Im Falle von SAT sind alle Zertifikate kompakt ... eine Wahrheitswertbelegung für Formel ϕ kannn durch einen String repräsentiert werden der polynomiell (linear) in der Größe von ϕ ist.

Beobachtung 2: Überprüfen ob es sich bei einer Belegung tatsächlich um ein Zertifikat für eine Instanz handelt, ist effizient lösbar.

Die Komplexitätsklasse NP

SAT ist nur eines von vielen Problemen die der Komplexitätsklasse **NP** zugeschrieben werden.

Informell ist NP die Menge jener Entscheidungsprobleme $\mathcal{P}$ für die gilt:

- positive Instanzen von $\mathcal{P}$ haben kompakte Zertifikate
- gegeben eine Instanz I von $\mathcal{P}$ und Zertifikat-Kandidat C, dann ist es einfach zu entscheiden ob C als Zertifikat von I gilt.

Recall: kompakt und "einfach zu entscheiden" beziehen sich auf die Instanzgröße.

Zertifikat-Relation

Wir definieren zuerst das Konzept der Zertifikat-Relation.

Definition (Zertifikat-Relation)

Sei $\mathcal{P}$ ein Eintscheidungsproblem, $INSTANCES(\mathcal{P})$ die Menge der Instanzen von $\mathcal{P}$.

Eine Zertifikat-Relation für $\mathcal{P}$ ist eine Relation $R \subseteq INSTANCES(\mathcal{P}) \times STRINGS$ (wobei $STRINGS$ die Menge aller Zeichenketten ist), sodass

$I \in INSTANCES(\mathcal{P})$ ist eine positive Instanz von $\mathcal{P}$

gdw

existiert Zertifikat $C \in STRING$ sodass $(I, C) \in R$.

Die Komplexitätsklasse NP

Wir können nun unsere Intuition bzgl. kompakte und einfach zu entscheidene Zertifikate formalisieren.

Definition

Gegeben eine binäre Relation R.

R heißt polynomiell balanziert wenn für alle $(v_1, v_2) \in R$, $|v_2| \leq |v_1|^k$ (für fixes $k \geq 1$) gilt.

R heißt polynomiell entscheidbar wenn es einen Algorithmus gibt, welcher, gegeben v_1 und v_2, $(v_1, v_2) \in R$ in polynomieller Zeit (in der Größe von v_1, v_2) prüft.

Definition (Die Komplexitätsklasse NP)

Ein Entscheidungsproblem $\mathcal{P}$ ist in der Klasse NP wenn es eine polynomiell balanzierte und polynomiell entscheidbare Zertifikats-Relation für $\mathcal{P}$ gibt.

Beispiel: **SAT** ist in **NP**

Theorem

SAT $\in$ **NP**.

Beweis.

Um **SAT** $\in$ **NP** zu beweisen, müssen wir lediglich eine Zertifikats-Relation für **SAT** angeben, die auch polynomiell balanziert und polynomiell entscheidbar ist. Betrachte

$$R = \{(\phi, I) \mid \text{Formel } \phi \text{ ist } \textbf{true} \text{ unter Belegung } I\}.$$

- R ist eine Zertifikats-Relation für **SAT**: ϕ ist eine positive Instanz von **SAT** $\Leftrightarrow$ es gibt eine Wahrheitswertbelegung I die ϕ wahr macht $\Leftrightarrow$ $(\phi, I) \in R$.
- R is polynomiell balanziert: jede Belegung I für ϕ kann einfach als Teilmenge von Atomen in ϕ repräsentiert werden.
- R is polynomiell entscheidbar: **MODEL-CHECKING** ist in P!

Die Komplexitätsklasse NP - Alternative Definition

Wir erweitern unsere SIMPLE programming language um einen Befehl `choice`(*assignment1*,*assignment2*). Ein `choice` Befehl teilt die Berechnung in zwei Zweige auf (die dann parallel und unabhängig voneinander weiterlaufen), wobei in einem Zweig das *assignment1*, im anderen das *assignment2* gesetzt wird.
Ein SIMPLE program mit `choice` Befehlen und Boole'schen Return-Values nennen wir `choice` Programm.

- Die Laufzeit eines `choice` Programms wird als die Laufzeit des längsten Berechungszweigs definiert.
- Ein `choice` Programm liefert true, wenn zumindest ein Berechnungszweig true liefert.

Die Komplexitätsklasse NP - Alternative Definition

SAT with extended SIMPLE

Boolean test(String s, Integer n) /* s a formula over atoms $a_1, \ldots, a_n$ */
    for all $1 \leq i \leq n$ **do** {
        choice($t_i = true$,$t_i = false$);
    }
    return modelcheck (s,[$t_1, \ldots, t_n$]);   /* checks if s is true under
            the assignment mapping a_i to t_i ($1 \leq i \leq n$) */

Die Komplexitätsklasse NP - Alternative Definition

Wir können nun eine alternative und äquivalente Definition der Klasse NP (die der Definition von P ähnelt) angeben:

Definition

Die Klasse NP beinhaltet genau jene Entscheidungsprobleme $\mathcal{P}$ die folgende Eigenschaften haben:

1. es gibt ein choice Programm Π für $\mathcal{P}$, sodass
2. für alle Instanzen I von $\mathcal{P}$ die Laufzeit von Π auf I polynomiell in $|I|$ ist, d.h. die Laufzeit ist durch $\mathrm{O}(|I|^k)$, wobei k eine Konstante, beschränkt.

Beobachtung 1: Jedes Problem in P ist auch in NP.
Beobachtung 2: Mittels choice Programmen können wir in polynomiell vielen Schritten exponentiell viele Berechnungszweige "generieren".

Frage: Ist das ein "angemessenes" Berechnungsmodell?

NP Probleme als Guess & Check Prozeduren

- Jedes Problem in NP kann man als Guess & Check Prozedur (mittels `choice` Programm) effizient implementieren.
- Sei $\mathcal{P} \in \text{NP}$ und R eine polynomiell balanzierte und polynomiell entscheidbare Zertifikats-Relation für $\mathcal{P}$.
- I ist eine positive Instanz von $\mathcal{P}$ wenn wir ein korrektes Zertifikat für I erraten können (guess). Das geschieht mit `choice` statements.
- Da R polynomiell balanziert ist, ist jeder Kandidat für ein Zertifikat für I polynomiell in der Größe von I
  (wir brauchen nur polynomiell viele `choice`-statements)
- Da R polynomiell entscheidbar, kann ein Kandidat als korrektes Zertfikat in polynomieller Zeit identifiziert werden (check).
  .... mittels standard SIMPLE Befehlen.

3 Wege um NP-Membership zu beweisen

3-COL (Drei-Färbbarkeit)

Instanz: ungerichteter Graph $G = (V, E)$

Frage: Ist G drei-färbbar? Das heißt, können wir jedem Knoten aus V eine von 3 Farben so zuordnen, dass alle adjazenten Knoten verschieden gefärbt sind?

1. Reduktion auf Problem in NP (siehe Reduktion 3-COL auf SAT);
2. Zertifikat-Relation:

$$R = \{((V, E), f) \mid f(v_i) \neq f(v_j) \text{ für alle } v_i, v_j \in V \text{ mit } [v_i, v_j] \in E\}$$

3. Extended SIMPLE program (Guess and Check Prozedur)
   - Rate für jeden Knoten eine Farbe (mittel zwei choice statements).
   - Überprüfe ob diese Belegung eine gültige Färbung der Knoten erzeugt.

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

Komplexität: Härte und Vollständigkeit

Konvention

- Im folgenden werden wir, wenn nicht anders angegeben, polynomielle many-one Reduktionen nutzen. Wir schreiben $\mathcal{P} \leq_{\mathrm{R}} \mathcal{P}'$ wenn Problem $\mathcal{P}$ auf $\mathcal{P}'$ auf diese Art reduziert werden kann.
- Beachte dass die Relation $\leq_{\mathrm{R}}$ Probleme nach deren Schwierigkeit ordnet. Klarerweise ist $\leq_{\mathrm{R}}$ reflexiv und transitiv.

Komplexität: Härte und Vollständigkeit

Definition

Sei C eine Komplexitätsklasse und sei $\mathcal{P}$ ein Entscheidungsproblem.

$\mathcal{P}$ heißt C-hart (C-schwer) wenn jedes Problem $\mathcal{P}' \in \mathrm{C}$ auf $\mathcal{P}$ reduziert werden kann.

$\mathcal{P}$ heißt C-vollständig wenn $\mathcal{P}$ in C liegt und $\mathcal{P}$ C-hart ist.

In anderen Worten:

Vollständigkeit = Mitgliedschaft und Härte

Beobachtung

Für alle Probleme $\mathcal{P}$ und $\mathcal{P}^*$ gilt: wenn $\mathcal{P}$ C-hart und $\mathcal{P} \leq_{\mathrm{R}} \mathcal{P}^*$ dann ist auch $\mathcal{P}^*$ C-hart.

Die Rolle von Vollständigkeit in der Komplexitätstheorie

- Vollständige Probleme stellen die maximalen Elemente ihrer Klasse in Bezug auf die Reduktions-Relation $\leq_{\mathrm{R}}$ dar.
- Vollständige Probleme sind ein zentrales Konzept und Werkzeug in der Komplexitätstheorie:
  - Die Komplexität eines Problems wird durch das Zeigen der Vollständigkeit für eine Komplexitätsklasse kategorisiert.
  - Andererseits ist die Essenz einer Komplexitätsklasse durch ihre vollständigen Probleme charakterisiert.
- Vollständigkeit dient als Basis für "negative" Komplexitätsresultate: Vollständige Probleme einer Klasse C sind jene Probleme dieser Klasse für die es am wenigsten wahrscheinlich ist, dass sie in einer "schwächeren Klasse" $\mathrm{C}' \subseteq \mathrm{C}$ liegen.

Die Rolle von Vollständigkeit in der Komplexitätstheorie

Theorem

Gilt für ein NP*-vollständiges Problem* $\mathcal{P}$ *dass es in* P *liegt, dann ist* NP $=$ P*.*

Beweis

Wir wissen P $\subseteq$ NP (siehe Definition). Wir zeigen NP $\subseteq$ P:
Sei $\mathcal{P}'$ ein beliebiges Problem in NP.
Da $\mathcal{P}$ NP-vollständig ist, existiert eine Reduktion R von $\mathcal{P}'$ auf $\mathcal{P}$. Dann können wir jedoch sofort einen (deterministischen) Polynomialzeit-Algorithmus für $\mathcal{P}'$ konstruieren: Wandle Instanz x des Problems $\mathcal{P}'$ in $R(x)$ um; entscheide $R(x)$ mit der Entscheidungsprozedur für $\mathcal{P}$. Da wir eine polynomielle Reduktion $\mathcal{P}' \leq_{\mathrm{R}} \mathcal{P}$ und $\mathcal{P} \in$ P haben, folgt dass $\mathcal{P}' \in$ P.
Wir habem somit NP $\subseteq$ P und folgerichtig auch NP $=$ P.

Remark: Es gibt einen relativ großen Konsens dass NP $\neq$ P. In diesem Fall gibt es keine effizienten Algorithmen für NP-vollständige Probleme!

Wiederholung: klassische Entscheidungsprobleme der Aussagenlogik

SAT (Satisfiability)

Instanz: Aussagenlogische Formel ϕ.
Frage: Ist ϕ erfüllbar? (d.h.: gibt es eine Wahrheitsbelegung I für ϕ, sodass $I(\phi) = \textbf{true}$).

3-SAT

Instanz: Aussagenlogische Formel ϕ in 3-KNF.

Frage: Ist ϕ erfüllbar?

2-SAT

Instanz: Aussagenlogische Formel ϕ in 2-KNF.

Frage: Ist ϕ erfüllbar?

Enscheidungsprobleme der Aussagenlogik

- Wir haben bereits argumentiert dass 2-SAT tractable ist.
- SAT und 3-SAT sind aber intractable (Wir geben hier nur einen Sketch der Beweisidee der NP-Vollständigkeit für diese zwei Probleme).

Wichtige Beobachtung

Das SAT Problem gilt als *das* klassische NP-vollständige Problem. Es bleibt NP-vollständig wenn wir die Instanzen auf 3-KNF einschränken (3-SAT). 3-SAT ist i.A. das Problem der Wahl wenn wir für weitere Probleme NP-Vollständigkeit zeigen wollen (da dies das Finden bzw. die Definition der Reduktionen einfacher macht).

Komplexität von SAT und $3\text{-}\mathrm{SAT}$

Cook-Levin Theorem

SAT ist NP-vollständig.

Theorem

$3\text{-}\mathrm{SAT}$ ist NP-vollständig.

NP-membership

Wir haben bereits besprochen, dass sowohl SAT als auch $3\text{-}\mathrm{SAT}$ mittels eines einfachen NP-Algorithmus entschieden werden können:

Gegeben Formel ϕ,

1. rate Wahrheitswertbelegung *I* für die Atome in ϕ.
2. prüfe ob ϕ auf **true** unter *I* evaluiert.

Beweisidee NP-Härte von SAT

Um zu zeigen dass SAT NP-hart ist, müssen wir für alle Probleme $\mathcal{P}$ in NP eine polynomielle many-one Reduktion von $\mathcal{P}$ nach SAT finden.

Da NP unendlich viele Probleme beinhaltet, können wir nicht jedes Problem in NP individuell behandeln. Wir brauchen eine Methode die alle diese Probleme umfasst $\Longrightarrow$ Berechnungsmodell nutzen.

Idee. Gegeben ein beliebiges Problem $\mathcal{P} \in$ NP und eine beliebige Instanz I von $\mathcal{P}$. Wir müssen zeigen, dass wir in polynomieller Zeit eine Formel ϕ bilden können, sodass I ist eine positive Instanz von $\mathcal{P} \Leftrightarrow \phi$ ist erfüllbar.

Da $\mathcal{P} \in$ NP gibt es eine polynomiell balanzierte und polynomiell entscheidbare Zertifikats-Relation R für $\mathcal{P}$.
Recall: I ist eine positive Instanz von $\mathcal{P} \Leftrightarrow (I, C) \in R$ mit (i) Größe von C polynomiell in $|I|$, und (ii) es gibt einen Polynomialzeit-Algorithmus A um zu testen ob $(I, C) \in R$.

Beweisidee NP-Härte von SAT

Um die Reduktion zu definieren brauchen wir daher für jedes $\mathcal{P}$ und jede Instanz I eine Formel ϕ sodass Folgendes gilt:
ϕ ist erfüllbar $\Leftrightarrow$ es gibt ein polynomial-size Objekt C sodass Algorithmus A Returnwert *true* für (I, C) liefert.

Informell muss die gesuchte Formel ϕ zwei Aufgaben kodieren:

1. Zertifikat-Kandidat generieren
2. die Berechnungen von A auf dem Paar (I, C) simulieren.

Der zweite Teil ist schwierig: wie können wir eine Formel angeben die SIMPLE Programme simuliert? Wie sollen wir **while** loops, **for** loops, **if**/**then**/**else** statements kodieren?

Antwort: Im Prinzip geht das, aber die bessere Wahl ist natürlich ein einfacheres Berechnungsmodell zu verwenden. Turing Maschinen sind ein wesentlich einfacheres Modell, aber äquivalent - erweiterte Church-Turing These - und lassen sich einfacher in Aussagenlogik kodieren.

Komplexität von SAT und $3\text{-}\mathrm{SAT}$

Beweisidee NP-Härte von $3\text{-}\mathrm{SAT}$

Wir brauchen eine Polynomialzeit Reduktion von SAT nach $3\text{-}\mathrm{SAT}$, d.h. für eine beliebige Formel ϕ existiert eine 3-KNF Formel $R(\phi) = \psi$, sodass ϕ erfüllbar $\Leftrightarrow$ ψ erfüllbar.

Anmerkungen:

- Das ist nicht komplett trivial. Beachte: die Standardumwandlung in KNF mittels de Morgan'sche Regeln und Distributiv-Regeln von $\wedge$ und $\vee$ führt im allgemeinen zu einem Exponential Blow-up. (Betrachte KNFs die logisch äquivalent zu $(x_1 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$ sind.)
- Jede Formel ϕ kann allerdings in in Polynomialzeit in eine sat-äquivalente Formel ψ in 3-KNF überführt werden
- Grundidee: Dies geschieht durch zusätzliche Atome die für Teilformeln von ϕ stehen (Tseitin-Transformation).
- ϕ und ψ sind daher i.A. nicht logisch äquivalent.

NP-Härte von INDEPENDENT SET

Da wir nun ein NP-hartes Problem kennen, können wir aufgrund unserer vorgehenden Beobachtung NP-Härte für weitere Probleme mittels Reduktion zeigen.

INDEPENDENT SET

INSTANZ: Ungerichteter Graph $G = (V, E)$ und Integer K.

FRAGE: Existiert ein *Independent-Set* S der Größe $\geq K$ für G d.h., gibt es eine Menge $S \subseteq V$, sodass für alle $i, j \in S$ gilt: $[i, j] \notin E$?

Theorem

3-SAT *lässt sich in polynomieller Zeit auf* INDEPENDENT SET *reduzieren.*

Folgerung:

Das INDEPENDENT SET Problem ist NP-hart. Tatsächlich ist es NP-vollständig.

Reduktion 3-SAT $\rightarrow$ INDEPENDENT SET

Beweis

Sei ϕ eine beliebige Instanz von 3-SAT.
Nehmen wir an dass ϕ m Klauseln mit jeweils exakt 3 Literalen hat.
Wir konstruieren die folgende Instanz von INDEPENDENT SET:

Ungerichteter Graph G mit einem Knoten für jedes Literal in ϕ:
$V = \{\ell_{11}, \ell_{12}, \ell_{13}, \ldots, \ell_{m1}, \ell_{m2}, \ell_{m3}\}$, also ist $|V| = 3m$.

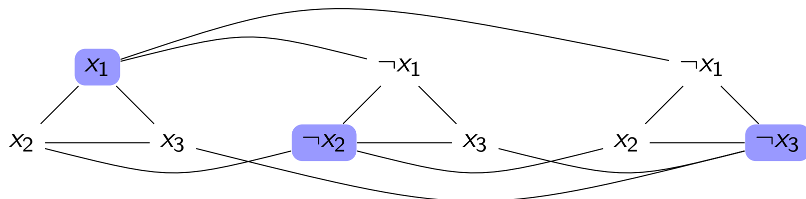
Die Knoten die Literale einer Klausel c_i von ϕ repräsentieren, sind verbunden: $E \supseteq \{[\ell_{i1}, \ell_{i2}], [\ell_{i1}, \ell_{i3}], [\ell_{i2}, \ell_{i3}]\}$ für alle i;
in anderen Worten: G hat für jede Klausel ein Dreieck.

Weiters verbinden wir alle Paare $\ell_{i\alpha}$, $\ell_{j\beta}$, wenn diese komplementäre Literale repräsentieren,
sprich: E umfasst auch alle solche Kanten $[\ell_{i\alpha}, \ell_{j\beta}]$.

Abschließend setzen wir $K = m$.

Beispiel

Die 3-KNF Formel $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$ wird auf folgenden Graph reduziert:



Der Independent-Set $S = \{\ell_{11}, \ell_{22}, \ell_{33}\}$ repräsentiert die Wahrheitswertbelegung $I(x_1) = I(\neg x_2) = I(\neg x_3) = \textbf{true}$.

Reduktion 3-SAT $\rightarrow$ INDEPENDENT SET

Beweis - Fortsetzung

Um die Korrektheit der Reduktion zu zeigen, müssen wir folgende Äquivalenz beweisen: ϕ ist erfüllbar $\Leftrightarrow$ die durch die Reduktion von ϕ gegebene Instanz (G, K) ist eine positive Instanz von $\textsc{Independent Set}$.

Wir beweisen die beiden Richtungen getrennt.

"$\Leftarrow$" Sei (G, K) eine positive Instanz von $\textsc{Independent Set}$, d.h. es gibt ein Set $S \subseteq V$, sodass für alle $i, j \in S$ $[i, j] \notin E$ gilt und $|S| \geq K = m$. Wir müssen zeigen dass ϕ eine positive Instanz von $3\text{-}\textsc{Sat}$ ist.

Beobachtung: Die Knoten eines Dreiecks in G sind alle adjazent, daher kann S maximal einen Knoten aus jedem der m Dreiecke enthalten. Daher gilt für alle $\ell_{i\alpha}, \ell_{i\beta} \in S$, dass $\alpha = \beta$. Da $|S| \geq K$ und $K = m$, ist in S genau ein Knoten pro Dreieck. Wir identifizieren S als $\{\ell_{1\alpha_1}, \dots, \ell_{m\alpha_m}\}$ für eine passende Kombination $\alpha_1, \dots, \alpha_m$ für Werte aus $\{1, 2, 3\}$.

Beweis - Fortsetzung $\Leftarrow$-Richtung

Wir definieren nun eine Wahrheitswertbelegung I für ϕ:

1. Ein Atom x von ϕ ist **true** unter I wenn wir ein $\ell_{i\alpha} \in S$ haben und x ist ein positives Literal an Position α in der $i$ten Klausel von ϕ.
2. Die verbleibenden Atome werden in I auf **false** gesetzt.

Wir zeigen nun dass alle Klauseln aus ϕ zumindest ein Literal haben, welches **true** unter I ist; klarerweise ist ϕ dann erfüllbar.

Sei C eine beliebige Klausel in ϕ und nehmen wir an es ist die i-te Klausel in ϕ. Betrachte Literal $\ell_{i\alpha_i}$ in C. Wir wissen dass $\ell_{i\alpha_i} \in S$. Wir zeigen nun dass $\ell_{i\alpha_i}$ **true** in I.

Fall 1: Wenn $\ell_{i\alpha_i}$ ein positives Literal ist, d.h. ein Atom x, dann ist $\ell_{i\alpha_i}$ **true** in I da laut Definition x in I auf **true** gesetzt ist (1).

Fall 2: Sei $\ell_{i\alpha_i}$ ein negatives Literal der Form $\neg x$. Wir müssen verifizieren dass x **false** in I ist. Angenommen das ist nicht der Fall, d.h. x ist **true** in I. Dann gibt es laut Definition von I, ein $\ell_{j\beta} \in S$ wobei x ein positives Literal auf Position β in der $j$ten Klausel von ϕ ist. Daher sind $\ell_{i\alpha_i}$ und $\ell_{j\beta}$ komplementäre Literale und daher laut Reduktion $[\ell_{i\alpha_i}, \ell_{j\beta}] \in E$. Widerspruch zur Annahme dass S ein Independent-Set von G ist.

Beweis - Fortsetzung $\Rightarrow$-Richtung

"$\Rightarrow$" Angenommen ϕ ist erfüllbar. Wir zeigen dass die reduzierte Instanz (G, K) eine positive Instanz von INDEPENDENT SET ist.

Da ϕ erfüllbar, gibt es Wahrheitswertbelegung I für ϕ, sodass $I(\phi) =$ **true**. Klarerweise macht I zumindest ein Literal in jeder Klausel von ϕ **true**. Daher können wir eine Knotenmenge $S = \{\ell_{1i_1}, \ldots, \ell_{mi_m}\}$ definieren sodass jeder Index i_j der Position eines Literals in der $j$ten Klausel von ϕ welches **true** in I ist, entspricht. (Falls mehrere Literale in einer Klausel **true** in I sind, wählen wir ein beliebiges aus). Wir müssen noch zeigen dass (i) S ein Independent-Set in G ist und (ii) $|S| \geq K$.

(ii) ist trivial: Laut Konstruktion $|S| = m$ und laut Reduktion $K = m$.

Wir zeigen (i): Angenommen dies gilt nicht, d.h. es gibt ein Paar $\ell_{i\alpha}, \ell_{j\beta} \in S$ sodass $[\ell_{i\alpha}, \ell_{j\beta}] \in E$. Laut Definition von G, hat G keine self-loops. Daher gilt $i \neq j$. Laut Reduktion stellen $\ell_{i\alpha}$ und $\ell_{j\beta}$ daher komplementäre Literale dar. Da I eine Wahrheitswertbelegung, ist eines der beiden Literale falsch unter I. Widerspruch: Laut Konstruktion haben wir für S nur Literale verwendet die **true** unter I.

Weitere NP-vollständige Graph Probleme

Die folgenden beiden Entscheidungsprobleme sind dem INDEPENDENT SET-Problem ähnlich:

CLIQUE

INSTANZ: Ungerichteter Graph $G = (V, E)$ und Integer K.

FRAGE: Enthält G eine *Clique* C der Größe $\geq K$?
D.h., gibt es ein $C \subseteq V$, sodass für alle $i, j \in C$ mit $i \neq j$, $[i, j] \in E$?

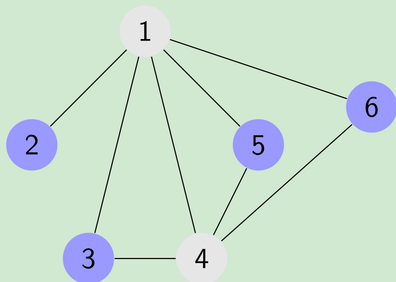
VERTEX COVER

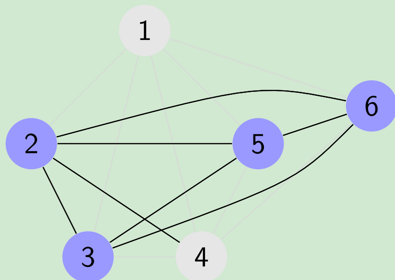
INSTANZ: Ungerichteter Graph $G = (V, E)$ und Integer K.

FRAGE: Gibt es ein *Vertex Cover* N der Größe $\leq K$ in G?
D.h., gibt es ein $N \subseteq V$, sodass für alle Kanten $[i, j] \in E$, $i \in N$ oder $j \in N$ (oder beides)?

INDEPENDENT SET vs. CLIQUE

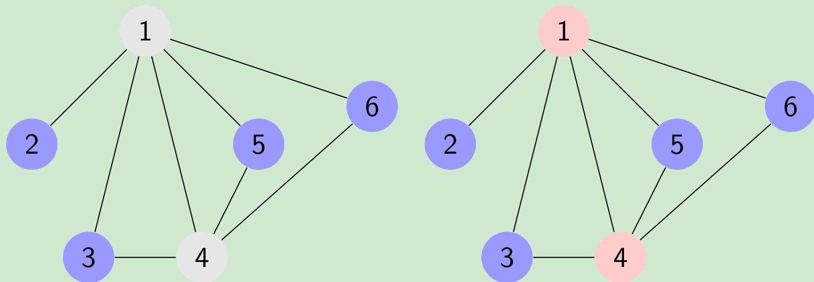
Example





INDEPENDENT SET vs. VERTEX COVER

Example



Komplexität

Theorem

INDEPENDENT SET, CLIQUE, *und* VERTEX COVER *sind* NP-*vollständig.*

Proof

NP-Membership. NP-Algorithmen für diese Probleme raten zuerst eine Teilmenge S der Knoten V und checken in polynomieller Zeit dass S den jeweilegen Eigenschaften genügt. (z.B. dass S ein Independent-Set der Größe $\geq K$ ist).

NP-Härte. Die zuvor skizzierten Äquivalenzen entsprechen einfachen Reduktionen mit denen man NP-Härte für die zwei verbleibenden Probleme zeigt. (Recall: Wir haben NP-Härte für das Problem INDEPENDENT SET bereits mittels Reduktion von 3-SAT gezeigt.)

Weitere Graph Probleme

HAMILTON-PATH

INSTANZ: Graph (gerichtet oder ungerichtet) $G = (V, E)$

FRAGE: Gibt es für G einen *Hamilton'schen Pfad*, also einen Pfad der alle Knoten exakt einmal besucht?

HAMILTON-CYCLE

INSTANZ: Graph (gerichtet oder ungerichtet) $G = (V, E)$

FRAGE: Gibt es für G einen *Hamilton'schen Kreis*, also einen zylischen Pfad der alle Knoten exakt einmal besucht?

Weitere Varianten des Erfüllbarkeitsproblems

Not-all-equal SAT (NAESAT)

INSTANZ: Aussagenlogische Formel ϕ in 3-KNF

FRAGE: Gibt es eine Wahrheitswertbelegung I für ϕ, die ϕ wahr macht und in jeder Klausel max. zwei der drei Literale auf wahr setzt?

1-IN-3-SAT

INSTANZ: Aussagenlogische Formel ϕ in 3-KNF

FRAGE: Gibt es eine Wahrheitswertbelegung I für ϕ, die in jeder Klausel genau eines der 3 Literale auf wahr setzt?

NP-Vollständige Probleme

Theorem

Die folgenden Probleme sind alle NP*-vollständig.*

- k-COL *für jedes* $k \geq 3$ *(also insbesonders auch* 3-COL*)*
- HAMILTON-PATH, HAMILTON-CYCLE
- k-SAT *für jedes* $k \geq 3$, NAESAT, 1-IN-3-SAT

Noch einmal: Varianten des Erfüllbarkeitsproblems

Anmerkungen

- Für positve Instanzen gilt $\mathrm{1\text{-}IN\text{-}3\text{-}SAT} \subset \mathrm{NAESAT} \subset \mathrm{3\text{-}SAT}$. Die Menge aller Instanzen dieser 3 Probleme ist aber gleich. In anderen Worten, es ist die Frage die sich hier ändert.
- Daher folgt aus der NP-Vollständigkeit eines dieser 3 Probleme nicht die NP-Vollständigkeit eines der anderen Probleme (a priori ist es nicht klar ob das Beschränken der positiven Instanzen das Problem leichter oder schwieriger macht!)
- Im Gegensatz dazu ist $\mathrm{3\text{-}SAT}$ ein Spezialfall von SAT, d.h., die Instanzen des einen Problems sind sind eine Teilmenge der Instanzen des anderen Problems — bei gleicher Fragestellung.
  1. NP-Härte des Spezialfalls impliziert NP-Härte des allgemeineren Problems
  2. NP-Membership des allgemeineren Problems impliziert NP-Membership des Spezialfalls

NP-Vollständigkeit – Spezialfall

Wir nennen einen ungerichteten Graphen G non-terminal wenn jeder Knoten in G zumindest zwei Kanten zu anderen Knoten hat. Beispiele: $(\{a, b, c\}, \{[a, b], [b, c], [a, c]\})$ ist non-terminal, während $(\{a, b, c\}, \{[a, b], [b, c]\})$ oder $(\{a, b, c, d\}, \{[a, b], [b, c], [a, c]\})$ diese Eigenschaft nicht haben.

Betrachten wir das folgende Entscheidungsproblem:

3-COL-NT

INSTANZ: Ein non-terminal Graph $G = (V, E)$.

FRAGE: Ist der Graph G 3-färbbar?
d.h.: gibt es eine Funktion $f : V \to \{0, 1, 2\}$, so dass für alle Kanten $[v_i, v_j] \in E$ gilt: $f(v_i) \neq f(v_j)$.

Wir nutzen die Tatsache dass die Standard Version des 3-Färbbarkeitsproblems NP-vollständig ist.

NP-Vollständigkeit – Spezialfall

Für die NP-Vollständigkeit von 3-COL-NT müssen wir NP-membership und NP-Härte zeigen.

NP-membership: 3-COL-NT ist ein Spezialfall von 3-COL, d.h. jede Instanz von 3-COL-NT ist auch Instanz von 3-COL. Da $3\text{-COL} \in$ NP, folgt $3\text{-COL-NT} \in$ NP direkt.

NP-Vollständigkeit – Spezialfall

NP-Härte: Wir geben eine polynomielle many-one Reduktion von 3-COL auf 3-COL-NT.
Sei $G = (V, E)$ eine beliebige Instanz von 3-COL, also ein beliebiger ungerichteter Graph. Wir konstruieren eine Instanz $G' = (V', E')$ von 3-COL-NT mit

- $V' = \{v, v^1, v^2 \mid v \in V\}$ und
- $E' = E \cup \{[v, v^1], [v^1, v^2], [v, v^2]\ midv \in V\}$.

Laut Definition ist jeder solcher Graph G' non-terminal und wir haben somit eine Reduktion wie gefordert. Außerdem ist klar, dass die Reduktion in polynomieller Zeit berechnet werden kann.

Wir müssen noch die Korrektheit dieser Reduktion zeigen, also: G ist positive Instanz von $3\text{-COL} \Leftrightarrow G'$ ist positive Instanz von 3-COL-NT.

NP-Vollständigkeit – Spezialfall

$\Rightarrow$: Sei $G = (V, E)$ eine positive Instanz von 3-COL. Dann gibt es eine Funktion f die den Knoten aus V einen Wert aus $\{0, 1, 2\}$ so zuweist, dass $f(u) \neq f(v)$ für alle Kanten $[u, v] \in E$ gilt.

Wir konstruieren nun eine Färbung $f^* : V' \to \{0, 1, 2\}$ für G' wie folgt:

$$\text{für alle } v \in V, \text{ sei } f^*(v) = f(v) \text{ und } f^*(v^i) = (f(v) + i) \textit{ mod } 3.$$

Wir müssen zeigen dass für allen Kanten $[x, y] \in E'$, $f^*(x) \neq f^*(y)$. Wir unterscheiden dabei die folgenden Fälle:

(1) Betrachte zuerst $x, y \in V$: Laut Annahme ist f eine 3-Färbung für G; laut Definition von f^* gilt daher $f^*(x) \neq f^*(y)$.

(2) Laut Konstruktion gilt andererseits $x, y \in \{v, v^1, v^2\}$ für ein $v \in V$; laut Definition von f^* haben wir unmittelbar $f^*(x) \neq f^*(y)$. Es folgt dass G' 3-färbbar ist und daher eine positive Instanz von 3-COL-NT.

NP-Vollständigkeit – Spezialfall

$\Leftarrow$: Sei G' eine positive Instanz von 3-COL-NT. Da G' 3-färbbar ist auch jeder Subgraph von G' 3-färbbar. Da G ein solcher Subgraph ist, gilt unmittelbar, dass G' eine positive Instanz von 3-COL.

Wir haben somit die Korrektheit unserer Reduktion gezeigt.

Teil 3: Einführung in Komplexitätstheorie

Teil 3.1: Grundlegende Konzepte und Wiederholung

Teil 3.2: Die Komplexitätsklassen P und NP

Teil 3.3: NP-Vollständigkeit

Teil 3.4: Weitere Komplexitätsklassen

The Klasse L (1)

Definition von L

L ist die Klasse jener Probleme die mittels eines SIMPLE Programms gelöst werden können welches lediglich logarithmischen Speicherbedarf hat, d.h. $\mathcal{P} \in$ L wenn es ein Programm gibt sodass für alle Instanzen I von $\mathcal{P}$ dieses Programm $\mathrm{O}(log_2|I|)$ Bits im Arbeitsspeicher braucht.

- Gemäß Definition von $\mathrm{O}(log_2|I|)$ heisst das, dass das Programm max. $c \cdot log_2|I| + d$ Bits des Arbeitspeicher braucht, wobei c, d Konstante.
- Logarithmic space = really little space, z.B. $log_2(65536) = 16$.
- Wir dürfen also im Programm nur konstant viele Pointer (Addressen im Speicher), Zähler, und logarithmisch viele Boole'sche Variable nutzen.
- Die Laufzeit ist apriori nicht eingeschränkt!
  (aber ist maximal polynomiell, wie wir sehen werden).

Die Klasse L (2)

- Für ein logarithmic-space Programm nehmen wir an dass der Input in einem separaten Read-only Memory abgelegt wird.
- Wir messen lediglich die Bits im Arbeitsspeicher
  - Klarerweise gilt $c \cdot log_2 n + d < n$ für ein ausreichend großes n.
  - Daher brauchen wir diesen separaten Speicher für den Input, da wir unter den gegebenen Speicherbeschränkungen sonst nicht einmal den Input parsen könnten.

- Ein Beispielproblem für L:

FIND-NODE

INSTANZ: Natürliche Zahl n und Baum T, wobei jeder Baumknoten mit einer natürlichen Zahl gelabelt ist.

FRAGE: Enthält T einen Knoten mit Label n?

- **FIND-NODE** $\in$ L da wir T depth-first mit nur drei Pointer durchlaufen könnnen: *current*, *next*, und *aux*.

FIND-NODE in Logarithmic Space

Wir können die folgenden Subroutinen voraussetzen (laufen in log. space):

- $root(T)$ zeigt auf Wurzel von T
  - $root(T) = nil$ wenn T der leere Baum
- $firstChild(T, x)$ liefert erstes Kind des Knoten x in T
  - $firstChild(T, x) = nil$ wenn ein solches Kind nicht existiert
- $rightSibling(T, x)$ liefert den rechten Nachbarn von x in T
  - $rightSibling(T, x) = nil$ wenn dieser nicht existiert
- $parent(T, x)$ liefert den Elternknoten von x in T
  - $parent(T, x) = nil$ wenn x Wurzel von T
- $isChildOf(T, x_1, x_2)$ liefert **true** wenn x_1 Kind von x_2 in T
- $isLeaf(T, x)$ liefert **true** wenn $firstChild(T, x) = nil$
- $hasRightSibling(T, x)$ liefert **true** wenn $rightSibling(T, x) \neq nil$
- $labelling(T, x)$ liefert das Label von x in T

// Speichereffiziente Prozedur eines depth-first Durchlaufs

```
Boolean find(Tree T, Integer val)
  current = root(T);
  if labelling(T, current) = val then return true;
  next = firstChild(T, current);

  while (next != nil) {

    if isChildOf(T, next, current) and !isLeaf(T, next) then {
      current := next; next := firstChild(T, next) }

    else if isChildOf(T, next, current) and isLeaf(T, next) then {
      aux := current; current := next; next := aux }

    else if isChildOf(T, current, next) and hasRightSibling(T, current) then
      { aux := current; current := next; next := rightSibling(T, aux) }

    else if isChildOf(T, current, next) and !hasRightSibling(T, current) {
      current := next; next := parent(T, next) }

    if labelling(T, current) = val then return true
  }
  return false
```

L und P

Theorem

L $\subseteq$ P, *also jedes Problem mit logarithmischem Specherbedarf ist in Polynomialzeit lösbar.*

Beweisidee

Zur Erinnerung: ein Programm Π dass logarithmischen Speicherbedarf in Bezug auf Input I hat, verbraucht maximal $c \cdot log_2|I| + d$ Bits im Read/Write Memory, mit c, d Konstante. Jeder Berechnungszustand (Konfiguration) des Programms im Zuge der Ausführung von Π auf I muss eindeutig durch die Belegung des Speichers beschreibbar sein. Wir haben daher dass es max. $2^{c \cdot log_2|I|+d}$ mögliche Zustände gibt, in denen sich Π befinden kann. Da $2^{c \cdot log_2|I|+d} = (2^{\log_2|I|})^c \cdot 2^d = |I|^c \cdot 2^d$, folgt dass es maximale polynomiell viele Zustände (in Bezug auf $|I|$) für Π geben kann. Folgerichtig kann die Laufzeit von Π auch nur polynomiell sein.

Die Klasse PSPACE

PSPACE

PSPACE ist die Klasse jener Probleme die mittels eines SIMPLE Programms gelöst werden können welches polynomielen Speicherbedarf hat, d.h. $\mathcal{P} \in$ PSPACE wenn es ein Programm gibt sodass für alle Instanzen I von $\mathcal{P}$ dieses Programm $\mathrm{O}(|I|^k)$ Bits im Arbeitsspeicher braucht (k eine Konstante).

- Diese Klasse ist sehr umfangreich und enthält zahlreiche schwierige Probleme.
- "Polynomieller Speicherbedarf" bedeutet dass das Programm in exponentiell vielen unterschiedlichen Zuständen sein kann (das Arugment ist analog zu jenem bzgl. den polynomiell vielen Zuständen im Fall von L)
- PSPACE-vollständige Probleme gelten als intractable (wir werden noch sehen dass NP $\subseteq$ PSPACE).

Beispiel eines Problems in PSPACE: Tic-Tac-Toe

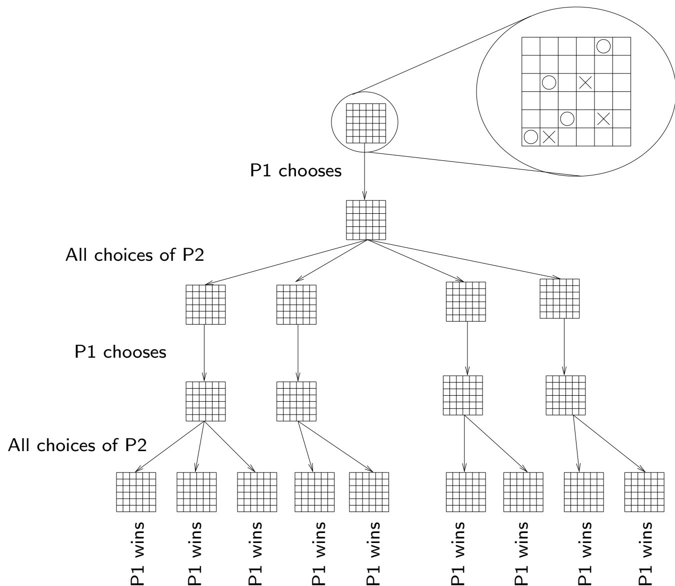
- Wir verallgemeinern das klassische (3×3 Spielbrett) Tic-Tac-Toe Spiel
- Ein (n, k)-Tic-Tac-Toe bezeichnet Tic-Tac-Toe
  - auf einem $n \times n$ Spielbrett;
  - man gewinnt wenn man eine Reihe der Länge k legt.
- Beobachtung: ein Spiel dauert max. n^2 steps.

TTT-WINNING-STRATEGY

INSTANZ: Zahlen n, k, ein Spielstand C auf dem $n \times n$ Brett.

FRAGE: Hat Spieler 1 eine Gewinnstrategie für das (n, k)-Tic-Tac-Toe ausgehend von Spielstand C?

Gewinnstrategie für Verallgemeinertes Tic-Tac-Toe



TTT-WINNING-STRATEGY ist PSPACE-vollständig

TTT-WINNING-STRATEGY

Die Problem ist in PSPACE (tatsächlich ist es PSPACE-vollständig):

- er reicht im Speicher einen Stack von max. n^2 Konfigurationen zu halten;
- jede Konfiguration braucht nur polynomiellen Speicher.

Anmerkungen

- Alternation ist typisch für PSPACE-vollständige Probleme, also:
  es gibt einen Zug für Spieler P1, sodass
  für alle möglichen Züge von P2,
  es gibt einen Zug für P1, ...
- Probleme in PSPACE können mittels polynomiell tief geschachtelten Schleifen gelöst werden: Im Fall von TTT-WINNING-STRATEGY ist die Tiefe mit n^2 beschränkt.

Weitere Probleme in PSPACE

SQL Query

Auswerten einer SQL Query über einer Datenbank.

Kontextsensitive Grammatik

Entscheiden ob ein gegebenes Wort von einer gegebenen kontextsensitiven Grammatik erzeugt werden kann.

Universalität eines regulären Ausdrucks

Testen ob ein regulärer Ausdruck alle möglichen Strings erzeugt.

PSPACE vs. P

Theorem

$P \subseteq PSPACE$: *jedes Problem dass sich in polynomieller Zeit lösen lässt, braucht max. polynomiellen Speicher.*

Beweis

Offensichtlich: In polynomieller Zeit ist es nicht möglich exponentiell viel Speicher zu nutzen/allokieren.

PSPACE vs. NP

Theorem

NP $\subseteq$ PSPACE.

Beweis Idee

Sei $\mathcal{P} \in$ NP. Dann gibt es eine Zertifikatsrelation R für $\mathcal{P}$, die polynomiell balanziert und polynomiel entscheidbar ist.

Wir können relativ einfach ein Programm spezifieren, welches Instanzen I von $\mathcal{P}$ in polynomiellen Speicherbedarf löst.

- Gehe alle Zertifikat-Kanditaten einzeln durch (klarerweise ist dann nur jeweils ein Kandidat im Speicher zu halten). Da R polynomiell balanziert, wissen wir dass ein solcher Kandidat nur polynomiellen Speicher braucht.
- Für jeden Kandidat C, teste $(I, C) \in R$. Da R polynomiell entscheidbar, braucht ein solcher Test nur polynomielle Zeit und daher, wie wir bereits gesehen haben, geht auch das mit polynomiellen Speicherbedarf.

Die Klasse EXPTIME

EXPTIME

Die Klasse EXPTIME beinhaltet genau jene Entscheidungsprobleme $\mathcal{P}$ die folgende Eigenschaften haben:

1. es gibt ein SIMPLE Programm Π für $\mathcal{P}$, sodass
2. für alle Instanzen I von $\mathcal{P}$ die Laufzeit von Π auf I exponentiell in $|I|$ ist, d.h. die Laufzeit ist durch $\mathrm{O}(2^{|I|^k})$, wobei k eine Konstante, beschränkt.

Die folgenden Beziehungen sind bekannt:

- PSPACE $\subseteq$ EXPTIME (analog zu L $\subseteq$ P)
- P $\subsetneq$ EXPTIME (nicht-triviales Diagonalisierungs-Argument)

Beispiele für Probleme in EXPTIME:

- Existenz einer Gewinnstrategie in GO auf einem $n \times n$ Spielbrett (Begründung: es existiert keine polynomielle Schranke für die Dauer des Spiels),
- Auswertung von DATALOG queries (SQL mit Rekursion).

Diskussion

Es gibt noch zahlreiche weitere Klassen:

- 2-EXPTIME $=$ Probleme die in Zeit $\mathrm{O}(2^{2^{|I|^k}})$ lösbar
- 3-EXPTIME $=$ Probleme die in Zeit $\mathrm{O}(2^{2^{2^{|I|^k}}})$ lösbar
- ...
- EXPSPACE $=$ Probleme die mit Speicher $\mathrm{O}(2^{|I|^k})$ lösbar
- 2-EXPSPACE $=$ Probleme die mit Speicher $\mathrm{O}(2^{2^{|I|^k}})$ lösbar
- 3-EXPSPACE $=$ Probleme die mit Speicher $\mathrm{O}(2^{2^{2^{|I|^k}}})$ lösbar
- ...

Nichtdeterminismus und Platzbedarf

Wir können analog zu NP auch Klassen wie z.B. NL, NPSPACE oder NEXPSPACE definieren.

Klarerweise gilt:

$$\mathsf{L} \subseteq \mathsf{NL} \quad \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \quad \mathsf{EXPSPACE} \subseteq \mathsf{NEXPSPACE}$$

Es ist offen ob $\mathsf{L} = \mathsf{NL}$ gilt (vgl. P vs. NP)

Man kann aber zeigen:

$$\mathsf{PSPACE} = \mathsf{NPSPACE} \quad \mathsf{EXPSPACE} = \mathsf{NEXPSPACE}$$

Beziehung zwischen Komplexitätsklassen

Theorem

L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ NEXPTIME $\subseteq$ EXPSPACE

Bzgl. echter Teilmengeninklusion ist nicht mehr bekannt als:
L $\subset$ PSPACE, P $\subset$ EXPTIME, PSPACE $\subset$ EXPSPACE.

Komplementäre Probleme und Komplexitätsklassen

Gegeben ein Entscheidungsproblem $\mathcal{P}$ mit Instanzen I und positive Instanzen $Y \subseteq I$. Das Komplementärproblem $\overline{\mathcal{P}}$ ist definiert über Instanzen I und positive Instanzen $I \setminus Y$ (informell: die Frage des Entscheidungsproblem $\mathcal{P}$ wird verneint).

Definition

Jede Komplexitätsklasse $\mathcal{C}$ hat eine Komplementärklasse, co-$\mathcal{C}$ die wie folgt definiert ist:

- co-$\mathcal{C} = \{\overline{\mathcal{P}} \mid P \in \mathcal{C}\}$.

Anmerkungen

- Beispiele: co-P, co-NP, co-NEXPTIME, etc.
- Deterministische Komplexitätsklassen $\mathcal{C}$ sind abgeschlossen unter Komplement, d.h. sie stimmen mit der co-Klasse co-$\mathcal{C}$ überein, z.B: P = co-P.
- Für nichtdeterministische Komplexitätsklassen $\mathcal{C}$ gilt das i.A. nicht. So ist z.B. offen ob NP = co-NP oder NP $\neq$ co-NP.

Zusammenfassung von Teil 3 der LVA

- Wiederholung grundlegender Konzepte, insbes. Reduktionen
- Korrektheitsbeweis von Reduktionen
- zentrale Komplexitätsklassen: P und NP
- Verschiedene Charakterisierungen von NP
- NP-Vollständigkeit
- SAT als prototypisches NP-vollständiges Problem
- Beispiele fuer NP-vollständige Probleme
- weitere Komplexitätsklassen

Zusammenfassung von Teil 3 der LVA

... als Video:

https://www.youtube.com/watch?v=YX40hbAHx3s