Berechenbarkeit, Entscheidbarkeit, Komplexitätstheorie 1

050026 VO Theoretische Informatik

Wolfgang Dvořák

Fakultät für Informatik,
Universität Wien

Sommersemester 2016

Berechenbarkeitstheorie

Ziele der Berechenbarkeitstheorie (auch Rekursionstheorie):

- Den „Algorithmus“ Begriff schärfen.
- Die theoretischen Grenzen des mit einem Computer machbaren erkunden.
- Eine Methodik bereitstellen die es erlaubt Probleme die nicht mit einem Computer gelöst werden können als solche zu erkennen.

Weiterführende Materialien

Berechenbarkeit:

Gottfried Vossen, Kurt-Ulrich Witt
Grundkurs Theoretische Informatik: Eine Anwendungsbezogene Einführung. [Kapitel 9,10]
Vieweg+Teubner Verlag; ISBN: 978-3834815378

Komplexität:

Gottfried Vossen, Kurt-Ulrich Witt
Grundkurs Theoretische Informatik: Eine Anwendungsbezogene Einführung. [Kapitel 11,12]
Vieweg+Teubner Verlag; ISBN: 978-3834815378

Sanjeev Arora and Boaz Barak
Computational Complexity: A Modern Approach
Cambridge University Press

Turing-Berechenbarkeit

Wir brauchen ein formales Modell für Computer / Berechenbarkeit:

Turing-Maschine.

Definition

Eine Funktion $f : \Sigma^* \Rightarrow \Sigma^*$ heißt **(Turing-)berechenbar**, wenn es eine Turingmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ gibt, die f berechnet, d.h. für die $f(x) = f_M(x)$ für alle $x \in \Sigma^*$ gilt.

- Die Maschine muss für **alle** Eingaben einen Endzustand erreichen.
- Wenn die Maschine terminiert steht $f(x)$ auf dem Band.

Zu jeder nichtdet. TM gibt es eine äquivalente det. TM.

$\hookrightarrow$ Wir können uns auf deterministische Maschinen beschränken.

Turing-Berechenbarkeit

Turing-Berechenbarkeit stimmt mit anderen Berechenbarkeitsmodellen (while-Programm, λ-Kalkül, μ-Rekursion, ...) überein

- Eine Funktion ist Turing-berechenbar genau dann wenn sie in den anderen Modellen berechenbar ist.

Church's These

Die Menge der Turing-berechenbaren Funktionen ist genau die Menge der intuitiv berechenbaren Funktionen.

Turing-Vollständigkeit: Eine Programmiersprache ist Turing-vollständig wenn sie alle Turing-berechenbaren Funktionen implementieren kann.

Gängige Programmiersprachen sind Turing-vollständig: C, C++, ...

Entscheidbarkeit

- **Berechenbarkeit von Problemen**: Berechenbarkeit der Lösung
- **Berechenbarkeit von Mengen**: Berechenbarkeit, ob ein Element Teil einer Menge ist – man spricht dann von Entscheidbarkeit.

Die Entscheidbarkeit von Mengen kann auf die Berechenbarkeit von Funktionen zurückgeführt werden.

Definition

Sei Σ ein Alphabet. Eine Menge $L \subseteq \Sigma^*$ heißt **entscheidbar**, falls die charakteristische Funktion von L, $\chi_L : \Sigma^* \to \{0, 1\}$, definiert durch

$$\chi_L(w) = \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{falls } w \notin L \end{cases}$$

berechenbar ist.

Nicht berechenbare Funktionen

Frage: Gibt es (wohldefinierte) Probleme, die nicht berechenbar sind, d.h. Probleme, die mit keinem Computer gelöst werden können?

Antwort: Ja.

Ein Kardinalitätsargument

- Die Menge aller (Turing-)berechenbarer Funktionen über $\mathbb{N}$ ist abzählbar unendlich.
  - Jede Turingmaschine mit $\Sigma = \{1\}$ ist äquivalent zu einer Turingmaschine mit $\Sigma = \{1\}$ und $\Gamma = \{0, 1\}$.
  - Fixiert man Γ und Σ dann ist die Menge aller Turingmaschinen mit Γ und Σ abzählbar unendlich.
- Es gibt überabzählbar viele Funktionen von $\mathbb{N}$ nach $\mathbb{N}$.
- Es gibt überabzählbar viele Teilmengen der natürlichen Zahlen.

$\Rightarrow$ Es gibt nicht-berechenbare Funktionen und unentscheidbare Mengen.

Turing's Halteproblem

Wir kennen aber auch konkrete unentscheidbare Probleme:

Turing's Halteproblem

Gegeben: Eine beliebige Turingmaschine mit einer beliebiger Eingabe.
Frage: Hält die Maschine mit dieser Eingabe?

Turing hat gezeigt: Es gibt keinen Algorithmus der das Halteproblem löst, d.h. das Halteproblem ist unentscheidbar.

Praktische Konsequenz: Es ist nicht entscheidbar, ob ein beliebiges Programm mit einer beliebigen Eingabe terminiert oder nicht!
(Für Spezialfälle ist eine Entscheidbarkeit durchaus möglich.)

Beispiel: Terminierung bei Lösung von $x^n + y^n = z^n$

Beispiel

```
read n
do forever
  (x,y,z) = next-tuple-in-some-order();
  if exp(x,n)+exp(y,n)=exp(z,n) then
    print x,y,z;
    exit;
  end if
end for
```

- Terminiert wenn eine Lösung für $x^n + y^n = z^n$ gefunden wird.
- Fermatsche Vermutung: Es gibt keine positive ganzzahlige Lösung für $x^n + y^n = z^n$ für $n > 2$.
- Es hat über 300 Jahre gebraucht, um den Satz zu beweisen (1995).
- Zur Entscheidbarkeit der Terminierung müsste der Algorithmus die Fermatsche Vermutung einer Lösung zuführen.

Anmerkung zur Entscheidbarkeit

- **Eindruck:** Die meisten Probleme erscheinen entscheidbar.
- **Gründe:**
  - Selektive Sicht.
  - Man betrachtet zumeist einfache, gut strukturierte Probleme, und diese sind tatsächlich häufig entscheidbar.
- In "Wirklichkeit" sind die meisten Probleme unentscheidbar (überabzählbar viele).
- Auch viele interessante Probleme sind unentscheidbar.

Post's Korrespondenzproblem PKP

Benannt nach Emil L. Post (1897-1954).

Gegeben:

- Alphabet Σ
- zwei Listen von Zeichenketten über Σ:

$$A = (x_1, \ldots, x_k)$$
$$B = (y_1, \ldots, y_k)$$

Es gilt also $x_i, y_i \in \Sigma^+$.

Frage: Gibt es eine nicht-leere Folge von Indizes $i_1, \ldots, i_m$ sodass

$$x_{i_1} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}?$$

Beispiel

$A = (1, 10111, 10)$ und $B = (111, 10, 0)$.
Für $m = 4$ gibt es eine Lösung mit Indexfolge $(2, 1, 1, 3)$ so dass
$x_2 x_1 x_1 x_3 = y_2 y_1 y_1 y_3 = 101111110$.

Post's Korrespondenzproblem PKP

Beispiel

Für $A = (10, 011, 101)$ und $B = (101, 11, 011)$ gibt es keine Lösung.

Beispiel

Die kürzeste Lösung für $A = (001, 01, 01, 10)$ und $B = (0, 011, 101, 001)$ hat $m = 66$.
(Das beweist man indem man alle Folgen für $m \leq 66$ testet)

Satz

Post's Korrespondenzproblem ist unentscheidbar.

Das Post's Korrespondenzproblem ist sehr hilfreich um die Unentscheidbarkeit anderer Probleme zu beweisen.

Methodik zum Beweisen der Unentscheidbarkeit

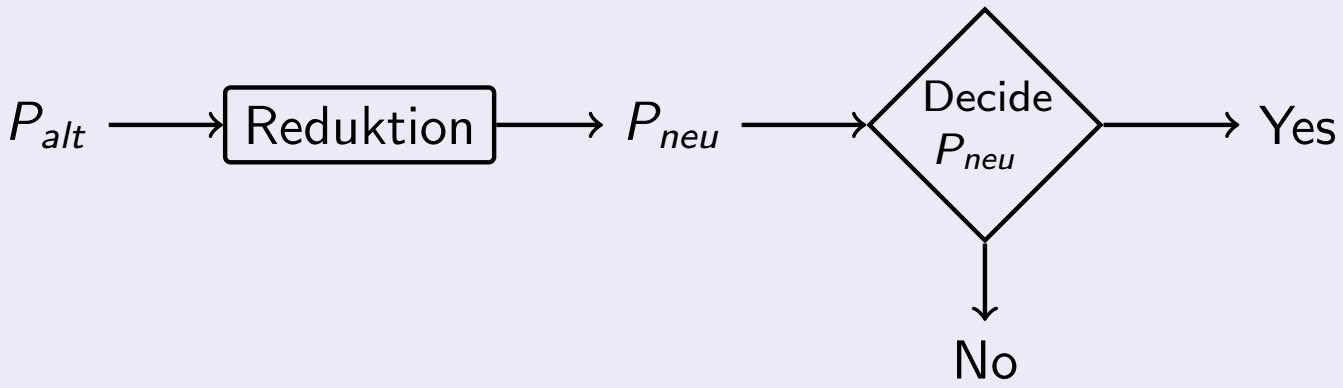
Frage: Ist eine neues Problem P_{neu} entscheidbar?

Bekannt: Es gibt ein Problem P_{alt} das nicht entscheidbar ist und ähnlich dem Problem P_{neu} ist.

Beweisführung (Idee)

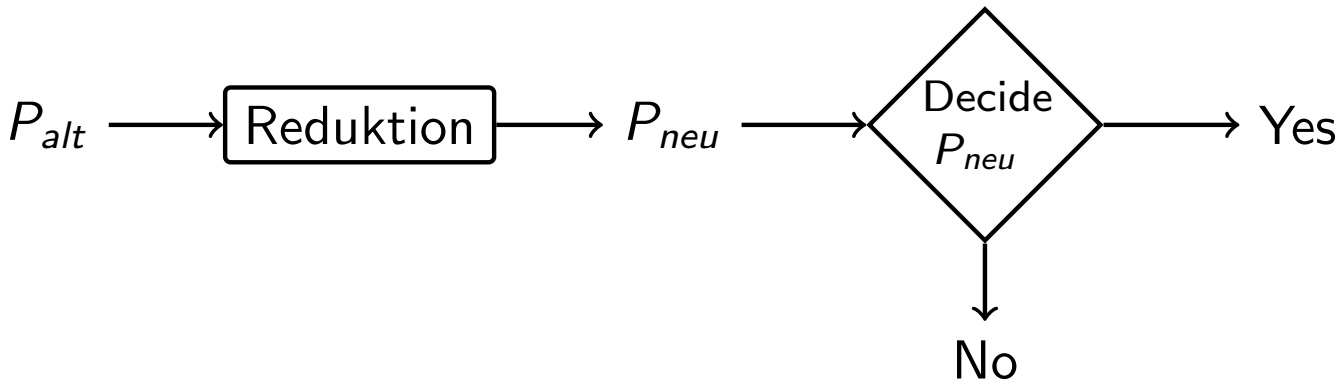
Reduktion und Widerspruch:

- Hat man eine Reduktion des Problems P_{alt} in das Problem P_{neu} und
- nimmt an dass ein Algorithmus Decide P_{neu} entscheidet,
- dann kann man einen Algorithmus für P_{alt} wie folgt konstruieren:



Widerspruch zu P_{alt} unentscheidbar.

Methodik zum Beweisen der Unentscheidbarkeit



Eine **Reduktion** ist eine Funktion $R(.)$.

1. Jede Instanz I von P_{alt} wird auf eine Instanz $R(I)$ von P_{neu} abbildet.
2. I yes/no Instanz von $P_{alt} \Leftrightarrow R(I)$ yes/no Instanz von P_{neu}

   oder

   I yes/no Instanz von $P_{alt} \Leftrightarrow R(I)$ no/yes Instanz von P_{neu}
3. $R(.)$ ist berechenbar

01-Post's Korrespondenzproblem

Satz

Post's Korrespondenzproblem ist unentscheidbar für $|\Sigma| = 2$.

Reduktion: Wir müssen eine Methode angeben um eine beliebige Instanz (Σ, A, B) von PKP in eine äquivalente mit $|\Sigma'| = \{0, 1\}$ umzuwandeln.

Sei $\Sigma = \{a_1, \ldots a_n\}$ dann können wir a_j als 01^j kodieren.

Wir bilden A', B' indem wir in A, B jedes a_j durch 01^j ersetzen.

Beispiel

$\Sigma = \{a, b, c\}$ mit $A = (ab, ccb, a)$ und $B = (a, bcc, ba)$.

Wir codieren a als 01, b als 011 und c als 0111.

$$A' = (01011, 01110111011, 01)$$
$$B' = (01, 01101110111, 01101)$$

01-Post's Korrespondenzproblem

1. Jede PKP Instanz wird auf eine 01-PKP Instanz ($\Sigma = \{0, 1\}$) abbildet.
2. Ein Folge ist Lösung von (Σ, A, B) genau dann wenn sie Lösung von $(\{0, 1\}, A', B')$ ist.
3. Die Kodierung ist berechenbar.

Wir haben also eine Reduktion für die Unentscheidbarkeit von 01-PKP gefunden.

Satz

Post's Korrespondenzproblem ist unentscheidbar für $|\Sigma| \geq 2$ *(für* $|\Sigma| = 1$ *ist PKP entscheidbar).*

Einige Unentscheidbare Probleme

Grammatiken

- **Äquivalenzproblem**: Seien G_1 und G_2 kontextfreie Grammatiken. Es ist nicht entscheidbar, ob $L(G1) = L(G2)$.
- **Mehrdeutigkeit**: Es ist nicht entscheidbar ob eine eine kontextfreie Grammatik G mehrdeutig ist?

Turingmaschinen

- **Programmäquivalenz**: M, M' Turingmaschinen. Berechnen M und M' die gleiche Funktion?
- **Turing's Halteproblem**: Hält eine Turingmaschine M für eine gegebene Eingabe I?

Prädikatenlogik

- **Erfüllbarkeitsproblem** der Prädikatenlogik: Ist diese Prädikatenlogische Formel erfüllbar?
- **Hilbert's Entscheidungsproblem**: Folgt diese Prädikatenlogische Formel aus den gegebenen Axiomen?

Komplexität von berechenbaren Problemen

Bis jetzt haben wir

- nicht berechenbare Probleme und
- (theoretisch) berechenbare Probleme unterschieden.

Nicht jedes theoretisch berechenbare Problem ist auch tatsächlich praktisch lösbar (aufgrund des hohen Berechnungsaufwands).

Verschiedene berechenbare Probleme haben ganz unterschiedliche Anforderungen an

- Rechenzeit
- Speicher
- ...

Wir wollen die berechenbaren Probleme anhand dieser Anforderungen weiter unterscheiden.

Komplexitätstheorie

Zeitkomplexität

Prinzip: Aufwand wird als Funktion der Inputgröße angegeben (viel aussagekräftiger als Zeitmessung in sec).

Algorithmus mit polynomialem Aufwand. Für Inputs der Länge n beträgt die max. Laufzeit $O(n^k)$.

Beispiel

- **Minimum / Maximum – Suche**: n Elemente, daher $(n-1)$ Vergleiche, daher Aufwand $O(n)$.
- **Sortieren durch Minimum / Maximum – Suche**: n Elemente, daher $(n-1)+(n-2)+\ldots+1$ Vergleiche, i.e. $\sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} = O(n^2)$.

Zeitkomplexität

Definition

Eine Turingmaschine M besitzt eine Zeitkomplexität $T(n)$, wenn M bei jeder Eingabe der Länge n nach maximal $T(n)$ Schritten hält.

Wir betrachten also asymptotische worst case Zeitkomplexität.

Wichtige Klasse von Funktionen: $T(n)$ ist polynomial

$T(n)$ ist polynomial wenn $T(n) = \sum_{i=0}^{k} a_i n^i$ für ein $k < \infty$

Intuition: Probleme die in polynomialer Zeit von deterministischen Turingmaschinen berechnet werden können, können auch auf einem typischen Computer in polynomialer Zeit berechnet werden.

Komplexitätsklassen

2 Komplexitätsklassen sind von besonderer Bedeutung[1]:

- **Klasse P (polynomial)**: Probleme die mit einer deterministischen Turingmaschine in polynomialer Zeit gelöst werden können.
- **Klasse NP (nichtdeterministisch polynomial)**: Probleme die mit einer nichtdeterministischen Turingmaschine in polynomialer Zeit gelöst werden können.

Beachte: Zu jeder nichtdet. TM gibt es auch eine äquivalente det. TM. Die Zeitkomplexität der det. TM kann jedoch exponentiell werden.

[1] Es gibt aber bei weitem mehr: `https://complexityzoo.uwaterloo.ca`.

Alternative Charakterisierung von **NP**

Zwei Phasen:

1. **Generate/Guess:** Es werden mögliche Lösungsvorschläge für ein Problem generiert.
2. **Check:** Ein Problem ist in der Klasse **NP**, wenn es in polynomialer Zeit möglich ist, einen Lösungsvorschlag zu verifizieren.

Verwendbar für Beweis, dass ein Problem aus **NP** ist

- SAT-Problem: Eine Variablen-Belegung kann in polynomialer Zeit verifiziert werde

Komplexitätsklassen und Praktische Lösbarkeit

- Probleme mit Komplexität **P** können i.A. praktisch mit vertretbarem Aufwand gelöst werden, und werden auch als “traktabel“ (engl. tractable) bezeichnet.
- Probleme mit Komplexität **NP** können i.A. praktisch nicht mehr gelöst werden (oder nur für kleine Instanzen).

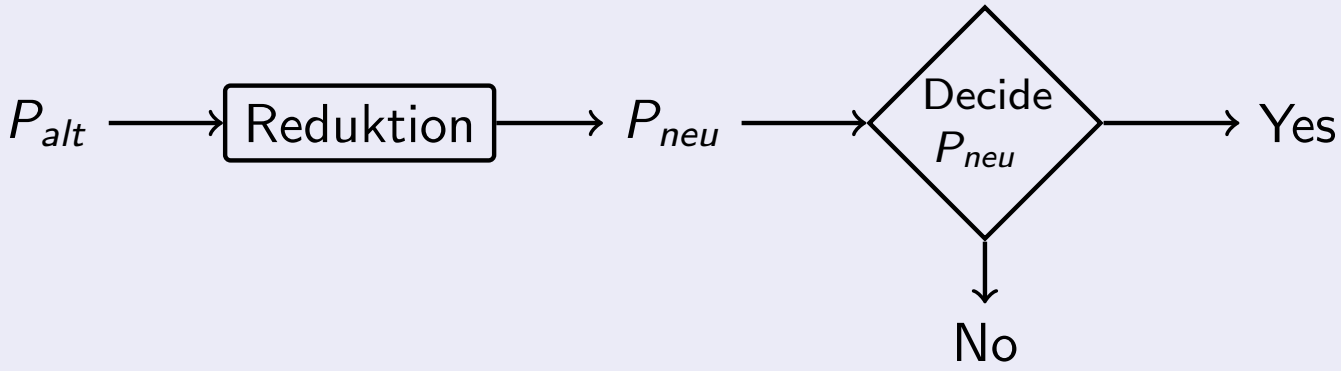
Es ist daher in der Praxis von großer Wichtigkeit zeigen zu können, ob ein Problem in der Klasse **P** liegt oder nicht.

- Falls in **P**: polynomialen Algorithmus angeben
- Falls nicht in **P**: Beweis mit Hilfe einer polynomialen Reduktion.

Polynomiale Reduktion

Kann man ein Problem P_{alt} das nicht in **P** ist auf ein P_{neu} polynomial reduzieren dann ist auch P_{neu} nicht in **P**.

Polynomiale Reduktion und Widerspruch



Eine **polynomiale Reduktion** ist eine Funktion $R(.)$.

1. Jede Instanz I von P_{alt} wird auf eine Instanz $R(I)$ von P_{neu} abbildet.
2. I yes/no Instanz von $P_{alt} \Leftrightarrow R(I)$ yes/no Instanz von P_{neu}
3. $R(.)$ ist in polynomial Zeit berechenbar

Gebe es einen polynomialen Algorithmus für P_{neu} dann auch für P_{alt}.

NP-vollständige Probleme **NPC** (NP-complete)

- Wir benötigen ein "erstes" Problem nicht in P.
- Kann man ein Problem A polynomial auf ein Problem B reduzieren, schreiben wir $A \preceq B$ und sagen B ist schwerer als A.
- Für gleich schwere Probleme gilt dann $A \preceq B$ und $B \preceq A$.

Definition

Ein Problem L heißt **NP**-vollständig, falls

1. $L \in \mathbf{NP}$, und
2. Für jedes Problem $L' \in \mathbf{NP}$ gilt $L' \preceq L$, d.h. es existiert eine polynomiale Reduktion von L' nach L

NP-vollständige Probleme **NPC** (NP-complete)

Für **NP**-vollständige Probleme gilt:

- Einerseits: bis dato wurden keine Algorithmen mit polynomialem Aufwand gefunden.
- Andererseits: bis dato konnte nicht ausgeschlossen werden, dass Algorithmen mit polynomialem Aufwand existieren könnten.

Per Definition können alle **NP**-vollständige Probleme gegenseitig aufeinander polynomial reduziert werden.

$\hookrightarrow$ Entweder können alle in **P** gelöst werden oder keines.

Methode

Für $P_{neu} \notin \mathbf{P}$, zeigt man das P_{neu} mind. so schwer ist wie **NPC**.
$\hookrightarrow$ Dazu reduziert man ein $P_{alt} \in \mathbf{NPC}$ polynomial auf P_{neu}.

Polynomiale Reduktion - Beispiel

SAT: Testen ob eine aussagenlogische Formel erfüllbar ist.

CNFSAT: Testen ob eine aussagenlogische Formel in KNF erfüllbar ist.

Wir wollen aus $SAT \in \mathbf{NPC}$ auf $CNFSAT \notin \mathbf{P}$ schließen.

1.Ansatz: Wir kennen einen Algorithmus der jede aussagenlogische Formel in eine semantisch äquivalente in KNF umwandelt.

- Bedingung 1 & 2 sind also erfüllt ✓
- aber der Algorithmus ist im worst case exponentiell. ↯

2.Ansatz: Es gibt einen polynomialen Algorithmus der jede Formel in eine erfüllbarkeitsäquivalente in KNF umwandelt, d.h. die Formeln sind entweder beide erfüllbar oder beide unerfüllbar.

- Bedingung 1 - 3 sind erfüllt ✓

Beziehung **P**, **NP**, **NPC**

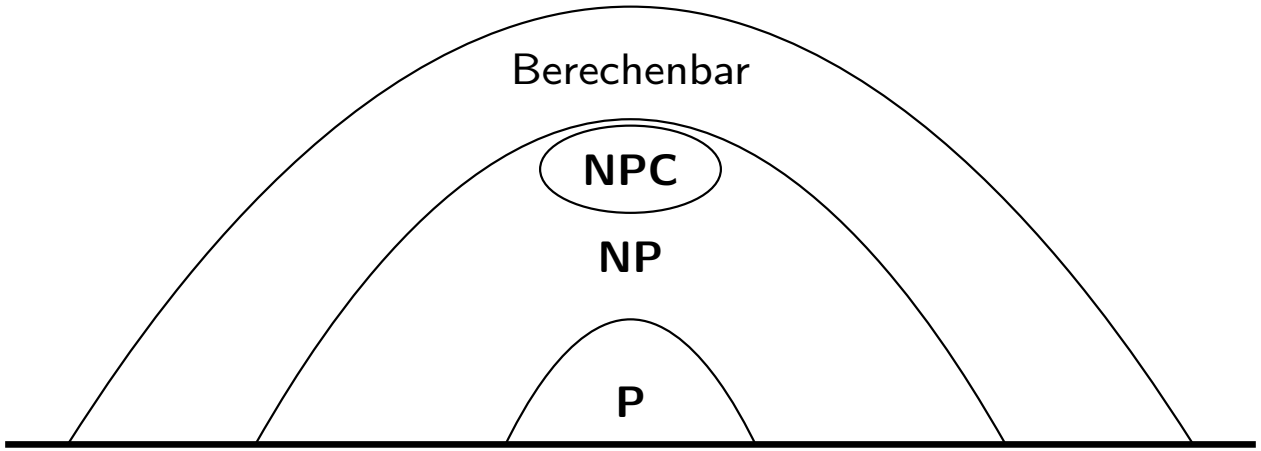
Wir wissen, dass

- **P** $\subseteq$ **NP**: jedes polynomiale Problem ist auch nichtdeterministisch polynomial lösbar.
- **NPC** $\subseteq$ **NP**: jedes **NP**-vollständige Problem ist in **NP**

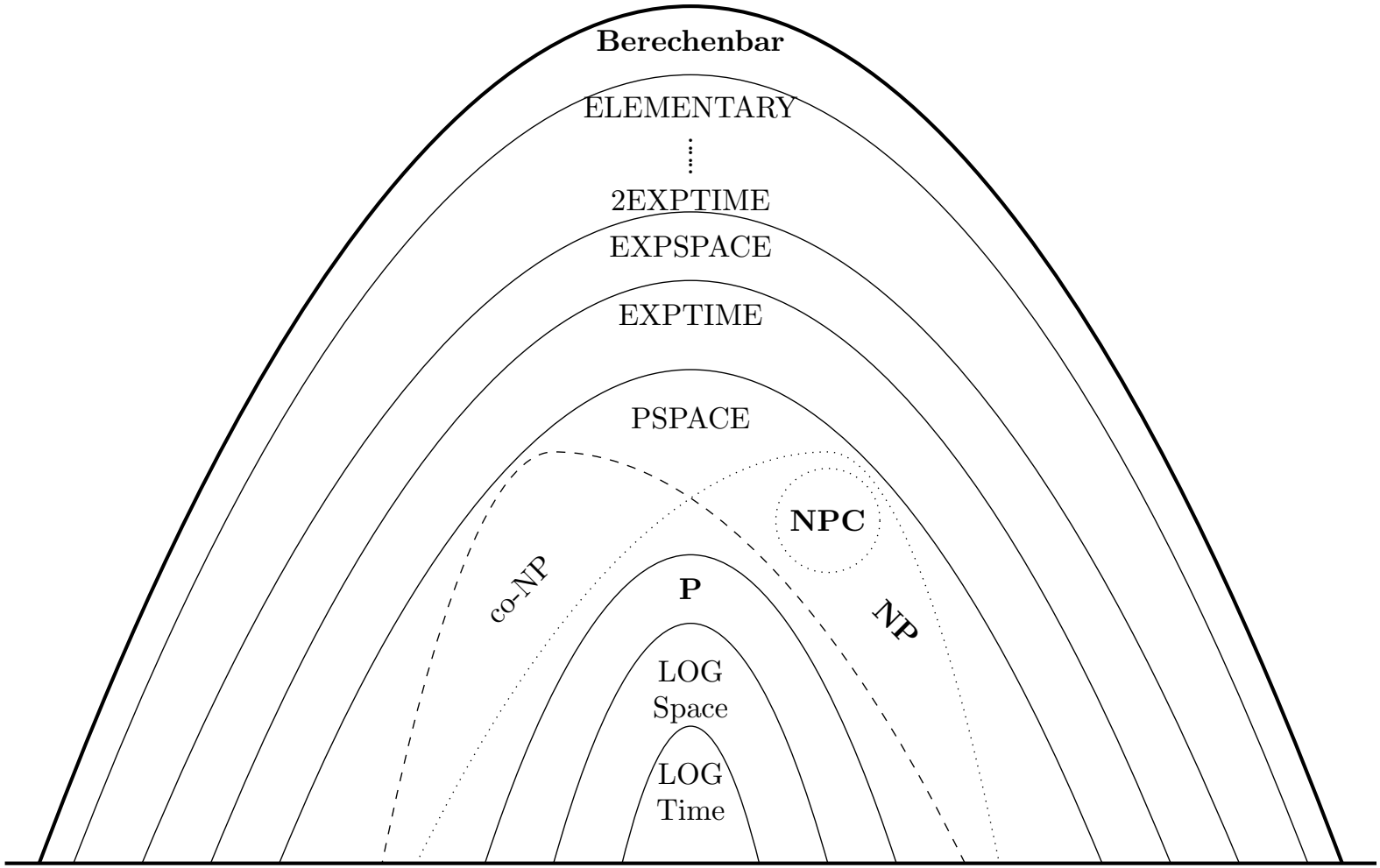
P-NP Problem

Ob **P** $\neq$ **NP** (oder **P** $=$ **NP**) ist eines der wichtigsten offenen Probleme der theoretischen Informatik.

Die starke Vermutung ist, dass **P** $\neq$ **NP** und daher **P** $\cap$ **NPC** $= \emptyset$.



Mehr Komplexitätsklassen [2]



[2] Bild basiert auf Sebastian Sardina: `http://www.texample.net/tikz/examples/complexity-classes/`

Zusammenfassung & Ausblick

Heute haben wir Folgendes behandelt:

- Berechenbarkeit
  - Unentscheidbare Problem
  - Reduktionen
- Komplexität
  - Komplexitätsklassen **P**, **NP**, **NPC**
  - Polynomiale Reduktionen

Nächste Einheit (Prof. Mehofer):

- Beispiele für NP-vollständige Probleme
- Praktische Implikationen von Komplexität (und Auswege)