BuK Panikzettel

Luca Oeljeklaus, der Dude, Caspar Zecha,
Tobias Polock, Philipp Schröer

24. Oktober 2025

Dieser Panikzettel ist über die Vorlesung Berechenbarkeit und Komplexität. Er basiert auf dem Foliensatz von Prof. Dr. Pascal Schweitzer aus dem Wintersemester 16/17.
Dieser Panikzettel ist Open Source. Wir freuen uns über Anmerkungen und Verbesserungsvorschläge auf `https://git.rwth-aachen.de/philipp.schroer/panikzettel`.

Inhaltsverzeichnis

*Pseudonyme gehören anonymen Autoren, die anonym bleiben wollen.

1. Einleitung

Wir haben uns entschieden, nur die wichtigsten Ergebnisse der Vorlesung hier im Panikzettel zusammenzufassen. Wir werden auf Grundlagen wie TMs und RAMs eingehen. Wir haben die großen Erkenntnisse aus dem Kapitel der Berechenbarkeit zusammengefasst und versuchen, auf Intuition und wichtige Werkzeuge bei Beweisen einzugehen. Im Kapitel zur Komplexität stellen wir die aus der Vorlesung bekannten Komplexitätsklassen vor und erklären P, NP und EXP.

Wie in anderen Panikzetteln haben wir die Anzahl der Beweise minimal gehalten. Wir wollen aber darauf hinweisen, dass die Beweise aus der Vorlesung besonders wichtig insbesondere für das Verständnis der vielen Reduktionen sind.

Im Anhang A haben wir eine Übersicht der Entscheidbarkeit für die aus der Vorlesung bekannten Probleme zusammengestellt. Im Anhang B fassen wir die Komplexitäten verschiedener Probleme zusammen.

2. Grundlagen

2.1. Die Turingmaschine (TM)

Damit wir uns mit Themen wie Berechenbarkeit und Komplexität von Problemen auseinandersetzen können, müssen wir uns erstmal klar machen was „berechnen“ überhaupt bedeutet. Wir führen hierzu das Modell der Turingmaschine ein, welche nach der Church-Turing-These (3.1) eine universelle Rechenmaschine ist.

2.1.1. Definition

Eine Turingmaschine M wird definiert durch das 7-Tupel

$$M = (Q, \Sigma, \Gamma, B, q_0, \overline{q}, \delta),$$

wobei Q eine endliche Zustandsmenge, Σ das Eingabealphabet, Γ das Bandalphabet, B das Leerzeichen, q_0 der Anfangszustand, $\overline{q}$ der Endzustand und

$$\delta : (Q \setminus \overline{q}) \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$$

die Zustandsübergangsfunktion ist.

Eine **Konfiguration** einer TM ist ein String $\alpha q \beta$ mit $q \in Q, \alpha, \beta \in \Gamma^*$. $\alpha_1 \dots \alpha_n q \underbrace{\beta_1}_{\text{Kopf}} \dots \beta_n$ bedeutet, dass die TM sich im Zustand q befindet, während der Lesekopf auf dem ersten Symbol von β und eine Position hinter dem letzten Symbol von α steht.

Eine Konfiguration k' ist die direkte Nachfolgekonfiguration einer Konfiguration k, falls diese durch einen Rechenschritt der betrachteten TM aus k entsteht. Wir schreiben $k \vdash k'$. Falls k' in einer beliebigen Anzahl an Rechenschritten aus k entsteht so ist k' eine Nachfolgekonfiguration von k: $k \vdash^* k'$.

2.1.2. Ausgabe

Die Ausgabe der TM bezeichnet nach dem Halten das Wort aus Elementen des Eingabealphabets ab der Position des Lesekopfes bis zum ersten Zeichen aus $\Gamma \backslash \Sigma$.

Für Entscheidungsprobleme, also Probleme, wo die Antwort entweder Ja oder Nein ist, gilt: Die TM akzeptiert die Eingabe, wenn sie terminiert und auf der Position des Lesekopfes eine 1 steht. Sie verwirft, wenn sie terminiert und eine 0 an der Position des Lesekopfes steht.

Auf manchen Eingaben halten manche Turingmaschinen nicht, die Ausgabe wird dann als $\perp$ (undefiniert) notiert. Eine TM berechnet daher eine Funktion

$$f_M : \Sigma^* \to \Sigma^* \cup \{\perp\}.$$

2.1.3. Erkannte und entschiedene Sprachen

Die Sprache $L(M) = \{ w \in \Sigma^* \mid M \text{ akzeptiert } w \}$ ist die von der TM M erkannte Sprache.

Falls M zusätzlich auf allen Eingaben $v \notin L(M)$ mit 0 hält (d.h. immer terminiert), so ist $L(M)$ gerade die von M *entschiedene* Sprache.

2.1.4. TM-Berechen- und Entscheidbarkeit

Eine Funktion $f : \Sigma^* \to \Sigma^* \cup \{\perp\}$ ist (TM-)berechenbar, wenn eine TM M existiert, sodass $f = f_M$ gilt.

Eine Sprache $L \in \Sigma^*$ ist (TM-)entscheidbar, falls eine TM existiert, die L entscheidet.

2.1.5. Laufzeit

Die Laufzeit $T_M(w)$ einer TM M auf einem Wort w ist die Anzahl der Schritte, bis die TM hält.

Die *worst-case Laufzeit* $t_M(n)$ ist das Maximum der Laufzeiten auf Wörtern der Länge n.

2.1.6. Gödelnummer

Um einfacher formal über Turingmaschinen argumentieren zu können, kodieren wir diese durch natürliche Zahlen.

Die Gödelnummer bestimmen wir als eine Kodierung aller s Übergänge, wobei der t-te Übergang $\delta(q_i, X_j) = (q_k, X_l, D_m)$ wie folgt kodiert wird

$$code(t) = 0^i 10^j 10^k 10^l 10^m.$$

Außerdem trennen wir die Kodierung der einzelnen Übergänge mit dem Trennsymbol 11 und signalisieren den Anfang und das Ende der Gödelnumnmer mit 111.

Die Gödelnummer einer TM M schreiben wir als $\langle M \rangle$.

2.1.7. Universelle Turingmaschine

Die Universelle Turingmaschine U führt beliebige Turingmaschinen aus, die durch Gödelnummern kodiert werden. Ihre Eingabe ist $\langle M \rangle w$, die zu simulierende TM M und w die Eingabe für M.

2.1.8. Mehrband-TMen

Mehrband-TMen sind eine Verallgemeinerung von TMen, da sie einfach mehrere Speicherbänder und mehrere Lese-/Schreibköpfe haben. Für k Bänder gilt:

$$\delta : (Q \setminus \overline{q}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k.$$

Mehrere Bänder erhöhen aber nicht die Berechnungskraft, da jede Mehrband-TM, die mit Rechenzeit $t(n)$ und Platz $s(n)$ auskommt, mit Zeitbedarf $\mathcal{O}(t^2(n))$ und Platzbedarf $\mathcal{O}(s(n))$ auf einer herkömmlichen TM simuliert werden kann.

2.2. Die Registermaschine (RAM)

Die Registermaschine besteht aus einem Programm, einem unendlichen Speicher und einem Befehlszähler. Der Speicher besteht aus einem Akkumulator ($c(0)$) und unendlich vielen Registern ($c(1), c(2), \ldots$), welche natürliche Zahlen enthalten.

Die Eingabe wird dann zu Beginn in die ersten Register geschrieben. Alle anderen Register und der Akkumulator, haben initial den Wert 0.

Der Befehlszähler hat Anfangs den Wert 1. In einem Arbeitsschritt führt die RAM stets denjenigen Befehl aus, dessen Nummer im Befehlszähler gespeichert ist. Der Befehl END beendet die Ausführung des Programmes.

Nach Ende der Berechnung ist das Ergebnis der Inhalt der ersten Register.

Man unterscheidet bei der Betrachtung der Laufzeit zwischen zwei verschiedenen *Kostenmodellen*:

- *Uniformes Kostenmaß*: Jeder Schritt kostet eine Zeiteinheit
- *Logarithmisches Kostenmaß*: Jeder Schritt verursacht Kosten proportional zur (binären) Länge der in den angesprochenen Registern gespeicherten Zahlen.

Die RAM als Konzept ist gleichmächtig zur Turingmaschine:

Für eine RAM mit Laufzeit $\mathcal{O}(t(n))$ im logarithmischen Kostenmaß gibt es ein Polynom p und eine TM mit Laufzeit $\mathcal{O}(p(n + t(n)))$, die diese RAM simuliert.

Eine TM mit Laufzeit $\mathcal{O}(t(n))$ kann durch eine RAM mit Laufzeit $\mathcal{O}(n + t(n))$ im uniformen oder $\mathcal{O}((n + t(n)) \log(n + t(n)))$ im logarithmischen Kostenmodell simuliert werden.

$*$	Befehl	Effekt	Zähler
✓	LOAD i	$c(0) := c(i)$	$b := b + 1$
	INDLOAD i	$c(0) := c(c(i))$	$b := b + 1$
✓	CLOAD i	$c(0) := i$	$b := b + 1$
✓	STORE i	$c(i) := c(0)$	$b := b + 1$
	INDSTORE i	$c(c(i)) := c(0)$	$b := b + 1$
	ADD i	$c(0) := c(0) + c(i)$	$b := b + 1$
	INDADD i	$c(0) := c(0) + c(c(i))$	$b := b + 1$
✓	CADD i	$c(0) := c(0) + i$	$b := b + 1$
	SUB i	$c(0) := \max\{0, c(0) - c(i)\}$	$b := b + 1$
	INDSUB i	$c(0) := \max\{0, c(0) - c(c(i))\}$	$b := b + 1$
✓	CSUB i	$c(0) := \max\{0, c(0) - i\}$	$b := b + 1$
	MULT i	$c(0) := c(0) \cdot c(i)$	$b := b + 1$
	INDMULT i	$c(0) := c(0) \cdot c(c(i))$	$b := b + 1$
	CMULT i	$c(0) := c(0) \cdot i$	$b := b + 1$
	DIV i	$c(0) := \lfloor \frac{c(0)}{c(i)} \rfloor$ falls $c(i) \neq 0$, sonst 0	$b := b + 1$
	INDDIV i	$c(0) := \lfloor \frac{c(0)}{c(c(i))} \rfloor$ falls $c(i) \neq 0$, sonst 0	$b := b + 1$
	CDIV i	$c(0) := \lfloor \frac{c(0)}{i} \rfloor$ falls $i \neq 0$, sonst 0	$b := b + 1$
✓	GOTO j		$b := j$
✓	IF $c(0) = x$ THEN GOTO j		$b := \begin{cases} j & c(0) = x, \\ b + 1 & \text{sonst} \end{cases}$
✓	END	Ende der Programmausführung.	

$*$: Die Registermaschine lässt sich äquivalent auch als *eingeschränkte RAM* formulieren, mit nur den mit ✓ markierten Befehlen. Diese eingeschränkte RAM kann dann natürlich nur endlich viele Register verwenden, weil `IND*`-Befehle fehlen.

2.3. Die Nichtdeterministische Turingmaschine (NTM)

Bei einer nichtdeterministischen Turingmaschine haben wir statt einer Übergangsfunktion eine Relation, d.h. es kann in einer Konfiguration mehrere direkt nachfolgende Konfigurationen geben. Diese Relation nennen wir im Allgemeinen δ, und es ist

$$\delta \subseteq ((Q \setminus \overline{q}) \times \Gamma) \times (Q \times \Gamma \times \{L, R, N\})$$

Eine NTM akzeptiert eine Eingabe, falls es mindestens einen Rechenweg in einen akzeptierenden Zustand gibt.

Die Laufzeit $T_M(w)$ einer NTM M auf einem Wort w ist die Länge des kürzesten akzeptierenden Pfades. Für $w \notin L(M)$ definieren wir $T_M(w) = 0$. Die worst-case Laufzeit $t_M(n)$ ist dann analog zur normalen TM die Laufzeit des kürzesten akzeptierenden Pfades für das Wort der Länge n, bei dem diese Laufzeit maximal ist.

Die NTM besitzt die gleiche Mächtigkeit wie die TM.

3. Berechenbarkeit

3.1. Die Church-Turing-These

*Die Klasse der **algorithmisch berechenbaren Probleme** stimmt mit der Klasse der von **TMen berechenbaren Funktionen** überein.*

Diese These kann nicht bewiesen werden, man könnte sie aber anhand eines mächtigeren Rechnermodells widerlegen. Im Allgemeinen wird die Church-Turing-These aber als wahr angenommen.

3.2. Nicht rekursive Probleme

3.2.1. Existenz unentscheidbarer Probleme

Die Menge aller Gödelnummern (und damit die aller TMs) ist als Teilmenge von $\{0,1\}^*$ abzählbar unendlich, während jedoch die Menge aller (binären) Sprachen $\mathcal{L}$ als die Menge aller Teilmengen von $\{0,1\}^*$ ($\mathcal{P}(\{0,1\}^*)$) überabzählbar unendlich ist. Daher gibt es mehr Entscheidungsprobleme als TMs und damit Entscheidungsprobleme, welche man nicht entscheiden kann.

3.2.2. Unentscheidbarkeit der Diagonalsprache

Da Σ^* abzählbar ist, können wir Wörter nummerieren. Wir bezeichen also das i-te Wort als w_i. Das gleiche gilt für Turingmaschinen.

Dann ist die Diagonalsprache D die Menge der Wörter w_i, die von M_i nicht akzeptiert werden. Diese ist unentscheidbar.

Beweis. Angenommen D sei entscheidbar. Dann existiert eine TM M_k, die D entscheidet.

Wir betrachten w_k:

$$w_k \in D \overset{M_k \text{ entscheidet } D}{\Longrightarrow} M_k \text{ akzeptiert } w_k \overset{\text{Def. von } D}{\Longrightarrow} w_k \notin D$$
$$w_k \notin D \overset{M_k \text{ entscheidet } D}{\Longrightarrow} M_k \text{ verwirft } w_k \overset{\text{Def. von } D}{\Longrightarrow} w_k \in D$$

Also führen beide Fälle zu einem Widerspruch. □

3.2.3. Unentscheidbarkeit des Halteproblems

Das Halteproblem ist ein Beispiel für ein sehr wichtiges, aber dennoch leider *unentscheidbares* Problem. Es ist definiert als:

$$H = \{\langle M \rangle w \mid M \text{ hält auf } w\}$$

Die Unentscheidbarkeit lässt sich mithilfe der Diagonalsprache beweisen, aber hier soll folgender Beweisansatz ausreichen:

Angenommen, das Halteproblem sei entscheidbar.

Dann können wir eine TM M_H bauen, die H entscheidet. Dann können wir auch eine TM M bauen, die als Eingabe eine Gödelnummer $\langle M_{in} \rangle$ einer TM bekommt. Diese TM

M führt dann M_H auf der Eingabe $\langle M_{in}\rangle\langle M_{in}\rangle$ aus. Wenn M_H akzeptiert, geht M in eine Endlosschleife, ansonsten akzeptiert M.

Dann hält M genau dann auf $\langle M\rangle$, wenn M nicht auf $\langle M\rangle$ hält.

Also kann H nicht entschieden werden.

3.2.4. Unentscheidbarkeit des speziellen Halteproblems

Das spezielle Halteproblem ist eine Variante des Halteproblems. Die Definition lautet wie folgt:

$$H_\varepsilon = \{\langle M\rangle \mid M \text{ hält auf } \varepsilon\}$$

Das spezielle Halteproblem ist unentscheidbar, da man aus einer TM M und einem Wort w eine TM M' konstruieren kann, wobei sich M' auf der Eingabe ε genauso verhält wie M auf der Eingabe w. Wäre also das spezielle Halteproblem H_ε entscheidbar, wäre auch das Halteproblem H entscheidbar.

3.2.5. Unterprogrammtechnik (Turingreduktion)

Mithilfe der Unterprogrammtechnik kann auch die Unentscheidbarkeit eines Entscheidungsproblems aufgezeigt werden:

Um zu zeigen, dass ein Problem N unentscheidbar ist, konstruiert man mithilfe einer TM, die N entscheidet, eine TM, die ein bekanntes unentscheidbares Problem entscheidet. Dies ist ein Widerspruch, es kann also keine TM geben, die N entscheidet.

3.2.6. Postsches Korrespondenzproblem

Beim Postschen Korrespondenzproblem (PCP oder PKP) geht es darum, für eine Menge von oben und unten beschrifteten Dominos eine nichtleere Folge der Dominos zu finden, so dass sich oben und unten dasselbe Wort ergibt. Dabei darf jeder Stein in der Folge beliebig oft verwendet werden.

Beim modifizierten PCP (MPCP oder MPKP) ist die Aufgabenstellung gleich wie beim PCP, nur dass der erste Stein der Folge vorgegeben ist.

Es lässt sich zeigen, dass MPCP auf PCP reduzierbar ist und dass das Halteproblem H auf MPCP reduzierbar ist. Damit ist auch H auf PCP reduzierbar. Insgesamt folgt, dass sowohl MPCP als auch PCP unentscheidbar sind. Achtung: Das bedeutet nicht, dass sich zu einer gegebenen PCP-Instanz keine Lösung finden lässt, es bedeutet nur, dass es keinen allgemeinen Algorithmus für beliebige Instanzen gibt.

Es gibt einige Varianten des PCP, wobei manche dieser Varianten entscheidbar sind: Wenn die Wörter auf den Dominos alle die Länge 1 haben, ist das PCP für solche Instanzen entscheidbar. Außerdem ist das PCP entscheidbar, wenn es nur einen oder nur zwei Dominos gibt. Für 7 oder mehr Dominos sowie für genau 5 Dominos ist das PCP hingegen unentscheidbar. Ungeklärt ist, ob das PCP für 3 oder 4 Steine entscheidbar ist.

3.3. Der Satz von Rice

Der Satz von Rice besagt, dass (nicht-trivale) Aussagen über von TM-berechneten Funktionen nicht entscheidbar sind.

Formal: Sei $\emptyset \subsetneq \mathcal{S} \subsetneq \mathcal{R}$, wobei $\mathcal{R}$ die Menge der von TMen berechenbaren partiellen Funktionen bezeichnet. Dann ist die Sprache

$$L(\mathcal{S}) = \{\langle M \rangle \mid M \text{ berechnet eine Funktion aus } \mathcal{S}\}$$

nicht entscheidbar.

3.4. Semi-Entscheidbarkeit und Rekursive Aufzählbarkeit

3.4.1. Semi-Entscheidbarkeit

Eine Sprache L heißt semi-entscheidbar, falls es eine TM M gibt, welche diese Sprache erkennt. Diese muss die Sprache allerdings nicht entscheiden, d.h. sie darf auf Eingaben $w \notin L$ auch nicht halten.

Ein Beispiel für eine semi-entscheidbare Sprache ist das Halteproblem. Eine TM M, welche H erkennt, lässt sich grob wie folgt beschreiben: Simuliere die eingegebene Turingmaschine auf dem eingegebenen Wort und gebe dann 1 aus. Denn wenn die simulierte Maschine hält, akzeptiert M die Eingabe, ansonsten läuft die simulierte TM ewig und die Eingabe wird nie akzeptiert.

3.4.2. Rekursive Aufzählbarkeit

Ein *Aufzähler* ist eine TM mit einem angeschlossenen Drucker, also ein zusätzliches Ausgabeband, auf dem der Kopf sich nur nach rechts bewegen darf. Ein Aufzähler für eine Sprache L druckt nun alle Worte aus L aus, wobei sich Wörter wiederholen dürfen.

Eine Sprache, für welche ein Aufzähler existiert heißt **rekursiv aufzählbar**. Es lässt sich relativ leicht zeigen (gute Übung!), dass eine Sprache genau dann **rekursiv aufzählbar** ist, wenn sie **semi-entscheidbar** ist.

Wenn eine Sprache L und ihr Komplement $\overline{L}$ rekursiv aufzählbar sind, dann ist L entscheidbar.

Eine Sprache L hat genau eine der folgenden Eigenschaften:

1. L ist entscheidbar. Dann ist auch $\overline{L}$ entscheidbar.
2. L ist rekursiv aufzählbar, $\overline{L}$ jedoch nicht (H, H_ε bspw.)
3. $\overline{L}$ ist rekursiv aufzählbar, L jedoch nicht (Diagonalsprache D bspw.)
4. Weder L noch $\overline{L}$ sind rekursiv aufzählbar (Allgemeines Halteproblem H_{all} bspw.)

3.4.3. Abschlusseigenschaften

Die folgenden Abschlusseigenschaften gelten für entscheidbare bzw. semi-entscheidbare Sprachen L_1 und L_2:

- Sind L_1 und L_2 entscheidbar, ist auch $L_1 \cap L_2$ entscheidbar.
- Sind L_1 und L_2 semi-entscheidbar, ist auch $L_1 \cap L_2$ semi-entscheidbar.
- Sind L_1 und L_2 entscheidbar, ist auch $L_1 \cup L_2$ entscheidbar.
- Sind L_1 und L_2 semi-entscheidbar, ist auch $L_1 \cup L_2$ semi-entscheidbar.

3.5. Die Technik der Reduktion

Die Sprache L_1 heißt auf die Sprache L_2 reduzierbar, falls es eine (berechenbare!) Funktion f gibt, so dass:

$$x \in L_1 \iff f(x) \in L_2$$

Wir schreiben $L_1 \leq L_2$. Falls $L_1 \leq L_2$, so gelten folgende Eigenschaften:

1. Falls L_2 entscheidbar ist, ist auch L_1 entscheidbar.
2. Falls L_2 rekursiv aufzählbar ist, so auch L_1. Wir können statt $x \in L_1$ auch einfach $f(x) \in L_2$ semi-entscheiden.
3. Im Umkehrschluss: Falls L_1 nicht entscheidbar/rekursiv aufzählbar ist, dann ist auch L_2 nicht entscheidbar/rekursiv aufzählbar.

Reduzierbarkeit ist transitiv, für drei Sprachen L_1, L_2, L_3 mit $L_1 \leq L_2$ und $L_2 \leq L_3$ gilt also auch $L_1 \leq L_3$.

3.6. Mächtigkeit von Programmiersprachen

Eine Programmiersprache sowie ein Rechnermodell wird als *Turing-mächtig* bezeichnet, wenn man mit ihr jedes von einer TM berechenbare Problem berechnen kann.

3.6.1. WHILE-Programme

Ein WHILE-Programm ist ein Programm, welches nur aus Zuweisungen und (geschachtelten, hintereinanderausgeführten) `WHILE`-Schleifen besteht. Als Konstanten kennen WHILE-Programme nur $-1, 0, 1$.

Wir können WHILE-Programme auch induktiv definieren:

- Für jedes $c \in \{-1, 0, 1\}$ ist auch x_i `:=` x_j `+` c ein WHILE-Programm.
- Für P_1, P_2 WHILE-Programme ist auch P_1`;` P_2 ein WHILE-Programm.
- Für P ein WHILE-Programm ist auch `WHILE` $x_i \neq 0$ `DO` P `END` ein WHILE-Programm.

Aufpassen: Das letzte WHILE-Programm in einem „Block“ (etwa innerhalb einer Schleife) hat kein Semikolon am Ende, wie man es von vielen anderen Programmiersprachen gewohnt ist.

Das Ergebnis einer Programmausführung wird in der Variable x_0 hinterlegt. Alle Variablen (außer die Eingabevariablen $x_1, \ldots, x_k$) sind standardmäßig mit 0 initialisiert. Es lässt sich zeigen, dass WHILE-Programme Turing-mächtig sind.

Ein Beispielprogramm in WHILE:

```
WHILE x0 ≠ 0 DO
  x1 := x0 + 1;
  x0 := x0 - 1
END
```

3.6.2. LOOP-Programme

LOOP-Programme sind WHILE-Programme mit der folgenden Änderung: Anstatt einer `WHILE`-Anweisung gibt es nun eine `LOOP`-Anweisung: `LOOP` x_i `DO` P `END`. Innerhalb von P darf nun x_i nicht vorkommen.

LOOP-Programme können sogenannte *primitiv-rekursive* Funktionen berechnen, sind jedoch *nicht* Turing-mächtig. Dies lässt sich schön mithilfe der Ackermann-Funktion zeigen, die schneller wächst, als jede Variable in LOOP-Programmen es jemals könnte. Damit ist etwa die Ackermann-Funktion nicht mit LOOP-Programmen berechenbar.

4. Komplexität

Anstatt uns zu fragen, was wir überhaupt berechnen können, wollen wir nun betrachten, wie schnell wir berechenbare Probleme lösen können.

4.1. Komplexitätsklasse P

Probleme der Komplexitätsklasse P (**P**olynomiell) definieren wir dadurch, dass es für jedes Problem in P eine TM gibt, die dieses Problem mit polynomieller worst-case Laufzeit entscheidet.

4.2. Komplexitätsklasse NP

Analog zur Definition von P ist NP (**N**ichtdeterministisch-**P**olynomiell) die Klasse aller Probleme, für welche es eine NTM gibt, die dieses Problem mit polynomieller worst-case Laufzeit entscheidet.

4.3. Komplexitätsklasse EXP

EXP oder EXPTIME ist die Klasse der Entscheidungsprobleme L, für die es ein Polynom p gibt, sodass sich L auf einer (deterministischen) TM mit worst-case Laufzeit $\mathcal{O}(2^{p(n)})$ berechnen lässt. n ist die Eingabelänge.

4.4. P, NP und EXP

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$$

Wir wissen auch, dass $\mathsf{P} \neq \mathsf{EXP}$ (Zeithierarchiesatz).

Es ist $\mathsf{P} \subseteq \mathsf{NP}$, weil eine deterministische TM als Spezialfall einer NTM gesehen werden kann.

$\mathsf{NP} \subseteq \mathsf{EXP}$ gilt, weil wir NTMs auf TMs in exponentieller Zeit simulieren können, indem wir den Berechnungsbaum der NTM durchlaufen.

Es ist aber völlig offen, ob $\mathsf{P} = \mathsf{NP}$ (es winken $> 10^6$ Euro für einen Beweis/Widerlegung!).

4.5. Zertifikate & Verfizierer

Man kann die Komplexitätsklasse NP auch anders charakterisieren:

Eine Sprache L ist genau dann in NP, falls gilt

$$x \in L \iff \exists y \in \{0,1\}^*, |y| \leq p(|x|) : V \text{ akzeptiert } y\#x$$

Hierbei muss der deterministische Verifizierer V in polynomieller Zeit das Zertifikat y prüfen können. Das Zertifikat ist durch die Eingabe polynomiell (p) in der Länge begrenzt.

Anders ausgedrückt: Lösungen für Probleme in NP sind bei geeignetem Zertifikat leicht zu überprüfen.

4.6. Polynomielle Reduktion

Ähnlich der Reduktion im Kapitel Berechenbarkeit (3.5) können wir hier die polynomielle Reduktion einführen. Wir sagen eine Sprache L_1 ist auf eine Sprache L_2 polynomiell reduzierbar (geschrieben $L_1 \leq_p L_2$), falls:

$$x \in L_1 \iff f(x) \in L_2$$

für eine Funktion f, welche in polynomieller Zeit (auf einer TM) ausgewertet werden kann.

4.7. NP-Schwere und NP-Vollständigkeit

Eine Sprache L ist NP-schwer, falls $\forall L' \in \text{NP} : L' \leq_p L$ gilt, also wenn sich jedes Problem aus NP polynomiell auf L reduzieren lässt.

Eine Sprache L ist NP-vollständig, falls $L \in \text{NP}$ gilt und L NP-schwer ist. Die Menge aller Probleme, die diese Eigenschaft haben, bezeichnet man als NPC.

4.7.1. Kochrezept für NPC-Beweise

Es dient als gute Zusammenfassung, sich noch mal das aus der Vorlesung bekannte „Kochrezept“ für Beweise der NP-Vollständigkeit von Problemen vor Augen zu führen. Um zu zeigen, dass eine Sprache L in NPC liegt, zeigt man:

1. Zeige, dass $L \in \mathsf{NP}$.
2. Wähle eine Sprache $L' \in \mathsf{NPC}$, welche auf L reduziert wird.
3. Beschreibe die Reduktion, mit der Instanzen von L' auf L abgebildet werden.
4. Zeige, dass diese Reduktion in polynomieller Zeit berechnet werden kann.
5. Zeige die Korrektheit der Reduktion f: $x \in L' \iff f(x) \in L$.

4.8. NP-Vollständigkeit verschiedener Probleme

Ein großer Teil der Vorlesung beschäftigt sich mit Reduktionen verschiedener Probleme, um NP-Vollständigkeit zu beweisen. Wir wollen die Beweise hier nicht wiederholen, merken aber an, dass ein Verständnis dennoch klausurrelevant sein könnte. Einige NP-vollständige Probleme sind im Anhang Komplexitätstabelle aufgelistet.

4.8.1. SAT: Der Satz von Cook und Levin

Die Aussage des Satz von Cook und Levin ist, dass SAT NP-vollständig ist. Es war das erste Problem, dessen NP-Vollständigkeit bewiesen wurde, und zwar indem die beiden ein *beliebiges* Problem aus NP auf SAT reduziert haben... Holy heck. Der Beweis ist zum Glück nicht klausurrelevant.

4.9. Pseudo-polynomiell lösbare und stark NP-schwere Probleme

Für Entscheidungsprobleme, in denen Gewichte, Kosten, Abstände o.ä. wichtig sind, bezeichnen wir für eine Instanz I eines solchen Problems den größten Zahlenwert in I mit $Number(I)$. Für eine Instanz I von Partition wäre das beispielsweise das Maximum der Zahlen $a_1, \ldots, a_n$.

4.9.1. Pseudo-polynomielle lösbare Probleme

Ein Problem ist in pseudo-polynomieller Zeit lösbar, wenn ein Algorithmus für das Problem existiert, dessen Laufzeit polynomiell in $|I|$ und $Number(I)$ auf Probleminstanzen I beschränkt ist. Solche Probleme sind u.a. Partition, Knapsack-E und Subset-Sum.

4.9.2. Stark NP-schwere Probleme

Ein Problem X ist stark NP-schwer, wenn X beschränkt auf Instanzen I mit $Number(I) \leq p(|I|)$ für ein Polynom p immer noch NP-schwer ist. Beispielsweise ist Bin-Packing-E stark NP-schwer.

Wenn es ein Problem gibt, das sowohl pseudo-polynomiell lösbar als auch stark NP-schwer ist, gilt $\mathsf{P} = \mathsf{NP}$. Unter der Annahme $\mathsf{P} \neq \mathsf{NP}$ gibt es also kein solches Problem.

4.10. Approximationsalgorithmen für NP-schwere Probleme

NP-schwere Probleme lassen sich nur langsam lösen - auf unseren Computern nicht in polynomieller Zeit (angenommen $\mathsf{P} \neq \mathsf{NP}$). Dennoch lassen sich NP-schwere Probleme approximieren, sodass es bessere zeitliche Garantien für eine Lösung gibt, die in einem bestimmten Bereich der optimalen Lösung liegt.

Für ein Optimierungsproblem Π sagen wir, dass für eine Instanz I von Π der optimale Zielfunktionswert $opt(I)$ ist.

- Für ein Minimierungsproblem Π: Ein α-Approximationsalgorithmus mit $\alpha > 1$ berechnet für jedes I eine Lösung mit Zielfunktionswert höchstens $\alpha \cdot opt(I)$.
- Für ein Maximierungsproblem Π: Ein α-Approximationsalgorithmus mit $\alpha < 1$ berechnet für jedes I eine Lösung mit Zielfunktionswert mindestens $\alpha \cdot opt(I)$.

Wir suchen in der Regel *effiziente* Approximationsalgorithmen, das heißt welche, die in polynomieller Zeit laufen. Wenn $\mathsf{P} = \mathsf{NP}$ gilt, können wir natürlich Probleme aus NP in polynomieller Zeit lösen und brauchen diese Approximationen zunächst nicht[1].

Nützlich ist auch ein *Approximationsschema*. Dies ist ein Algorithmus, der für jedes gegebene $\varepsilon > 0$ eine zulässige Lösung mit Approximationsfaktor $1+\varepsilon$ bzw. $1-\varepsilon$ berechnet.

4.10.1. Approximation von BPP

Für BPP gibt es einen effizienten 2-Approximationsalgorithmus. Es ist außerdem bewiesen, dass wenn $\mathsf{P} \neq \mathsf{NP}$ gilt, es keinen effizienten α-Approximationsalgorithmus mit $\alpha < \frac{3}{2}$ gibt.

[1] Wenn der Algorithmus, der NP-Probleme in polynomieller Zeit löst, dennoch langsam ist, sind die Approximationen natürlich trotzdem sinnvoll.

4.10.2. Approximation von TSP

Wenn $\mathsf{P} \neq \mathsf{NP}$: Das allgemeine TSP ist nicht effizient approximierbar.

Für metrisches TSP dagegen, das heißt TSP, wo die Knoten des zu durchwandernden Graphen in einem metrischen Raum liegen, gibt es aber einen effizienten 2-Approximationsalgorithmus.

4.11. coNP

Die Komplexitätsklasse coNP besteht aus den Entscheidungsproblemen, für die wir in polynomieller Zeit Nein-Instanzen mit einem passenden Zertifikat mit polynomieller Länge entscheiden können.

4.11.1. coNP-Vollständigkeit

Ein Problem in coNP heißt coNP-vollständig, wenn sich alle Probleme in coNP auf das Problem reduzieren lassen.

Für jedes NP-vollständige Entscheidungsproblem X ist das Komplement $\overline{X}$ coNP-vollständig. Dabei werden Ja-Instanzen in X zu Nein-Instanzen in $\overline{X}$ und umgekehrt. Dabei müssen die Zertifikate nicht geändert werden, ein Zertifikat y für ein $x \in X$ ist ebenfalls ein Zertifikat für $x \notin \overline{X}$.

Es wird angenommen, dass $\mathsf{NP} \neq \mathsf{coNP}$ gilt. Wenn allerdings coNP ein NP-vollständiges Problem enthält, gilt $\mathsf{NP} = \mathsf{coNP}$.

A. Entscheidbarkeitstabelle

Problem	Beschreibung	entscheidbar?
Halteproblem H	$H = \{\langle M\rangle w \mid M \text{ hält auf } w\}$	semi
$\overline{H}$	$\overline{H} = \{\langle M\rangle w \mid M \text{ hält nicht auf } w\} \cup \overline{\text{Syn}}$	✗
spez. Halteprob. H_ε	$H_\varepsilon = \{\langle M\rangle \mid M \text{ hält auf } \varepsilon\}$	semi
$\overline{H}_\varepsilon$	$\overline{H}_\varepsilon = \{\langle M\rangle \mid M \text{ hält nicht auf } \varepsilon\} \cup \overline{\text{Syn}}$	✗
H_{all}	$H_{\text{all}} = \{\langle M\rangle \mid M \text{ hält auf jeder Eingabe}\}$	✗
$\overline{H}_{\text{all}}$	$\overline{H}_{\text{all}} = \{\langle M\rangle \mid M \text{ hält nicht auf jeder Eingabe}\} \cup \overline{\text{Syn}}$	✗
Nullstellenproblem	Für ganzzahliges Polynom: ex. Nullstelle in $\mathbb{Z}$?	semi
Postsches Korrespondenzproblem (PKP)	Dominos so wählen und sortieren, dass oben und unten das gleiche steht	semi
Modifiziertes PKP	PKP mit festem Startdomino	semi
2-PKP	PKP mit zwei Dominos	✓

Mit $\overline{\text{Syn}}$ sind hier die Elemente aus $\{0,1\}^*$ gemeint, die nicht die Form $\langle M \rangle w$ resp. $\langle M \rangle$ haben.

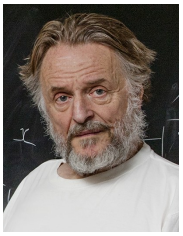
B. Komplexitätstabelle

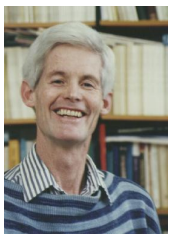
Klasse	Problem	Beschreibung
P	Primzahltest	
P	Minimaler Spannbaum	
P	Lineare Programmierung	
NP	Graph Isomorphismus	
NPC	SAT	Erfüllbarkeitsproblem der Aussagenlogik.
NPC	CLIQUE	Hat ein Graph eine Clique?
NPC	TSP-E	Gibt es eine Eulertour einer maximalen Länge?
NPC	KNAPSACK-E	Gibt es Objekte unter einer Gewichtsschranke mit mindestem Nutzen?
NPC	BIN-PACKING-E	Passen Objekte in max. Anzahl von Behältern?
NPC	COLORING	Lässt sich ein Graph mit k Farben färben?
NPC	HAMILTON	Gibt es einen Hamiltonkreis?
NPC	SUBSET-SUM	Gibt es eine Teilmenge der Eingabe mit bestimmter Summe?
NPC	PARTITION	Lässt sich die Menge in zwei Teilmengen aufteilen, sodass beide die gleiche Summe haben? Spezialfall von SUBSET-SUM.
NPC	DOMINATINGSET	Gibt es eine Teilmenge der Knoten des Eingabegraphen unter einer Schranke, sodass jeder Knoten im Graphen entweder in dieser Teilmenge oder direkt damit verbunden ist?
NPC	VERTEXCOVER-E	Gibt es eine Teilmenge der Knoten des Eingabegraphen unter einer Schranke, sodass jede Kante im Graphen mit einem der gewählten Knoten inzident ist?
NPC	INDEPENDENTSET	Gibt es eine Teilmenge der Knoten des Eingabegraphen unter einer Schranke, sodass keine zwei verbundenden Knoten ausgewählt sind?

C. Superstars der BuK


Wilhelm **Ackermann**

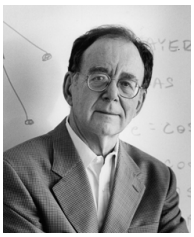

Alonzo **Church**


John **Conway**


Stephen **Cook**


Kurt **Gödel**


David **Hilbert**

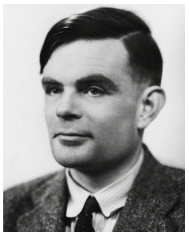

Richard **Karp**


Stephen **Kleene**


Donald **Knuth**


Leonid **Levin**


Emil **Post**


Alan **Turing**