

Ludwig-Maximilians-Universität München

WS 2015/2016

MARTIN HOFMANN, ULRICH SCHÖPP

# Komplexitätstheorie

Vorlesungsmitschrieb von

Philipp Moers

<p.moers@campus.lmu.de>

<soziflip@gmail.com>

Stand: 28. Januar 2016, 11:54

## **Zusammenfassung**

Die Komplexitätstheorie beschäftigt sich mit der Klassifikation von Algorithmen und Berechnungsproblemen nach ihrem Ressourcenverbrauch, z. B. Rechenzeit oder benötigtem Speicherplatz. Probleme mit gleichartigem Ressourcenverbrauch werden zu Komplexitätsklassen zusammengefasst. Die bekanntesten Komplexitätsklassen sind sicherlich P und NP, die die in polynomieller Zeit deterministisch bzw. nicht-deterministisch lösbaren Probleme umfassen.

P und NP sind jedoch nur zwei Beispiele von Komplexitätsklassen. Andere Klassen ergeben sich etwa bei der Untersuchung der effizienten Parallelisierbarkeit von Problemen, der Lösbarkeit durch zufallsgesteuerte oder interaktive Algorithmen, der approximativen Lösung von Problemen, um nur einige Beispiele zu nennen.

## **Anmerkung**

Dies ist ein inoffizieller Vorlesungsmitschrieb. Als solcher erhebt er keinen Anspruch auf (NP-)Vollständigkeit oder Korrektheit. Nutzung, Anmerkungen und Korrekturen sind jedoch durchaus erwünscht!

Website der Vorlesung: <http://www.tcs.ifi.lmu.de/lehre/ws-2015-16/kompl>

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	Literatur . . . . .	5
<b>2</b>	<b>Turingmaschinen, Berechenbarkeit und Komplexität</b>	<b>8</b>
2.1	Turingmaschinen . . . . .	8
2.2	Halteproblem . . . . .	12
2.3	Rechenzeit . . . . .	12
2.4	Komplexitätsklassen . . . . .	14
2.5	polynomielle Verifizierbarkeit . . . . .	18
<b>3</b>	<b>NP und P</b>	<b>20</b>
3.1	Padding . . . . .	20
3.2	Was wenn $P = NP$ . . . . .	22
3.3	Sogenannte Effizienz . . . . .	23
3.4	Polynomialzeitreduktionen . . . . .	24
3.5	NP-Härte und NP-Vollständigkeit . . . . .	28
3.6	Etwas zwischen P und NP . . . . .	31
3.7	Orakel-Turingmaschinen . . . . .	35
3.8	Polynomielle Hierarchie . . . . .	41
3.9	Kollabieren der polynomiellen Hierarchie . . . . .	43
<b>4</b>	<b>Platzkomplexität</b>	<b>49</b>
4.1	Platzkonstruierbare Funktionen . . . . .	49
4.2	Platzverbrauch einer Turingmaschine . . . . .	49

4.3	Platzhierarchiesatz . . . . .	51
4.4	Zusammenhänge von Platzkomplexitätsklassen . . . . .	52
4.5	Noch ein Satz von Cook . . . . .	59
4.6	Logarithmischer Platz . . . . .	63
4.7	Mehr Härte und Vollständigkeit . . . . .	64
4.8	Horn . . . . .	65
4.9	Quantifizierte Boolesche Formeln . . . . .	70
4.10	Noch zwei PSPACE-vollständige Probleme . . . . .	74
4.11	Jumping Automata on Graphs . . . . .	78
<b>5</b>	<b>Probabilistische Algorithmen</b>	<b>84</b>
5.1	Miller-Rabin Primzahltest Einführung . . . . .	84
5.2	Fermat-Primzahltest . . . . .	85
5.3	Miller-Rabin Primzahltest . . . . .	86
5.4	Testen von polynomiellen Identitäten . . . . .	86
5.5	Chernoff Schranke . . . . .	93
5.5.1	Anwendung . . . . .	94
5.6	Interaktive Beweissysteme . . . . .	96
5.6.1	Graphisomorphismus . . . . .	99
5.6.2	Unerfüllbarkeit von 3KNF . . . . .	100
5.6.3	Probabilistically Checkable Proof . . . . .	104

# 1 Einführung

## 1.1 Motivation

Theoretische Informatik, Berechenbarkeit und insbesondere Komplexitätstheorie ist *der* Informatiker-Shit schlechthin. Let's do it!

## 1.2 Literatur

Die Vorlesung basiert hauptsächlich auf folgendem Buch:

- Bovet, Crescenzi. Introduction to the Theory of Complexity. Prentice Hall. New York. 1994.

Weiterhin ist folgende Literatur gegeben:

- C. Papadimitriou. Computational Complexity. Addison-Wesley. Reading. 1995.
- I. Wegener. Komplexitätstheorie: Grenzen der Effizienz von Algorithmen. Springer. 2003.
- S. Arora und B. Barak. Complexity Theory: A Modern Approach.

Zur Motivation:

- Heribert Vollmer. Was leistet die Komplexitätstheorie für die Praxis? Informatik Spektrum 22 Heft 5, 1999.
- Stephen Cook: The Importance of the P versus NP Question. Journal of the ACM (Vol. 50 No. 1)

---

Vorlesung vom 12.10.15

---

## 2 Turingmaschinen, Berechenbarkeit und Komplexität

### 2.1 Turingmaschinen

#### Definition

Eine **Turingmaschine**  $T$  mit  $k$  Bändern ist ein 5-Tupel

$$T = (Q, \Sigma, I, q_0, F)$$

- $Q$  ist eine endliche Menge von Zuständen
- $\Sigma$  ist eine endliche Menge von Bandsymbolen,  $\square \in \Sigma$
- $I$  ist eine Menge von Quintupeln der Form  $(q, s, s', m, q')$  mit  $q, q' \in Q$  und  $s, s' \in \Sigma^k$  und  $m \in \{L, R, S\}^k$
- $q_0 \in Q$  Startzustand
- $F \subseteq Q$  Endzustände



$\square$  ist das Leerzeichen oder **Blanksymbol**.

$T$  heißt **deterministisch** genau dann, wenn für jedes  $q \in Q$  und  $s \in \Sigma^k$  genau ein Quintupel der Form  $(q, s, \_, \_, \_) \in I$  existiert. Sonst heißt  $T$  **nichtdeterministisch**.

Eine Turingmaschine heißt **Akzeptormaschine** genau dann, wenn zwei Zustände  $q_A, q_R \in F$  speziell markiert sind.  $q_A$  signalisiert Akzeptanz,  $q_R$  signalisiert Verwerfen der Eingabe.

Eine Turingmaschine heißt **Transducermaschine** genau dann, wenn ein zusätzliches Band ausgezeichnet ist (das Ausgabeband).

### Beispiel

Akzeptormaschine  $T$  für Sprache  $L = \{0^n 1^n \mid n \geq 0\}$  wobei  $\Sigma = \{0, 1\}, Q = \{q_0, \dots, q_4\}$

$T$  wird deterministisch sein.  $T = (Q, \Sigma, I, q_0, F), q_A = q_1, q_R = q_2, F = \{q_1, q_2\}, k = 2$

$q$	$s_1$	$s_2$	$s'_1$	$s'_2$	$m_1$	$m_2$	$q'$
$q_0$	$\square$	$\square$	$\square$	$\square$	$S$	$S$	$q_1$
$q_0$	$0$	$\square$	$0$	$0$	$R$	$R$	$q_3$
$q_0$	$1$	$\square$	$1$	$\square$	$S$	$S$	$q_2$
$q_3$	$\square$	$\square$	$\_$	$\_$	$\_$	$\_$	$q_2$
$q_3$	$0$	$\square$	$0$	$0$	$R$	$R$	$q_3$
$q_3$	$1$	$\square$	$1$	$\square$	$S$	$L$	$q_4$
$q_4$	$0$	$0$	$\_$	$\_$	$\_$	$\_$	$q_2$
$q_4$	$1$	$0$	$1$	$0$	$R$	$L$	$q_4$
$q_4$	$0$	$\square$	$\square$	$\square$	$S$	$S$	$q_1$
$\_$	$\_$	$\_$	$\_$	$\_$	$\_$	$\_$	$q_2$

Die **globale Konfiguration** (oder der **Zustand**) einer Turingmaschine beinhaltet die Beschriftung aller Bänder, den internen Zustand ( $\in Q$ ) und die Positionen aller  $k$  Lese-/Schreibköpfe. Globale Konfigurationen können als endliche Wörter über einem geeigneten Alphabet (z. B.  $\{0, 1\}$ ) codiert werden.

Eine Turingmaschine **akzeptiert** eine Eingabe genau dann, wenn eine Berechnungsfolge ausgehend von dieser Eingabe existiert und in einem Zustand aus  $F$  endet.

Eine Turingmaschine **akzeptiert** eine Sprache  $L \subseteq (\Sigma \setminus \{\square\})^*$  falls gilt:

$$\text{Die Turingmaschine akzeptiert } w \Leftrightarrow w \in L$$

Eine Turingmaschine **entscheidet** eine Sprache  $L \subseteq (\Sigma \setminus \{\square\})^*$  genau dann, wenn sie sie akzeptiert und eine/die Berechnung in  $q_A$  endet.

### Zu Mehrband-Turingmaschinen:

Bisher waren die Bänder beidseitig unendlich. Ab jetzt und im Buch sind sie nur noch einseitig unendlich.

### Satz

Eine Mehrband-Turingmaschine mit  $k$  Bändern kann durch eine Einband-Turingmaschine simuliert werden.

Dies benötigt quadratischen Mehraufwand.

### Beweis

Die Beweisidee nutzt für das alte Alphabet  $\Sigma$  das neue Alphabet  $\Sigma^k \times \{0,1\}^k$ , das die Zeichen auf den Bändern und, ob der Lese-/Schreibkopf an dieser Position steht, speichert.

*q.e.d.*

### Definition

Eine **universelle Turingmaschine** erhält als Eingabe  $(M, x)$ , wobei  $M$  die Beschreibung einer Turingmaschine in geeignetem Binärformat und  $x$  die Eingabe für  $M$  ist. Sie berechnet dann die Ausführung von  $M$  auf  $x$ .

## 2.2 Halteproblem

### Definition

Gegeben Turingmaschine und Eingabe  $(M, x)$ . Das Problem, zu entscheiden, ob  $M$  angewendet auf  $x$  hält oder nicht, heißt **Halteproblem**.

### Satz

Das Halteproblem ist unentscheidbar.

### Beweis

Angenommen, es gäbe eine Turingmaschine  $M_{HALT}$ , die das Halteproblem entscheidet.

Dann könnten wir auch eine neue Turingmaschine  $M_D$  konstruieren:

Simuliere Eingabe  $M$  auf  $M$  selbst und schaue, ob sie hält. Falls ja, dann gehe in Endlosschleife. Falls nicht, halte an.

Für  $M = M_D$  ergibt sich nun ein Widerspruch: Falls sie hält, hält sie nicht. Falls sie nicht hält, hält sie.

*q.e.d.*

## 2.3 Rechenzeit

### Definition

Die **Rechenzeit** definiert man wie folgt:

Gegeben eine Turingmaschine  $M$  und Eingabe  $x$ .

$TIME_M(x)$  ist die Dauer (Anzahl der Schritte) der Berechnung von  $M$  auf  $x$ .

Im Beispiel der Maschine für  $L = \{0^n 1^n \mid n \geq 0\}$  ist  $TIME_M(x) = |x|$  (Länge des Strings).

### Satz

Das **Speedup-Theorem** besagt, dass zu jeder Turingmaschine  $M$  eine äquivalente Turingmaschine  $M'$  konstruiert werden kann, sodass

$$TIME_{M'}(x) \leq \frac{1}{k} * TIME_M(x)$$

wobei  $k \in \mathbb{N} \setminus \{0\}$  fest gewählt ist.

Zum Beispiel ist bei  $k = 7$  die neue Turingmaschine siebenmal so schnell.

### Beweis

Gegeben  $M$  mit Alphabet  $\Sigma$ .

Dann wird  $M'$  mit Alphabet  $\Sigma^k$  konstruiert. Ein Symbol von  $M'$  repräsentiert  $k$  aufeinanderfolgende Symbole von  $M$ , d.h.  $M'$  kann  $k$  Schritte von  $M$  in einem einzigen ausführen.

*q.e.d.*

### **Anmerkung:**

Die Schritte werden in der Praxis also schon aufwändiger, die definierte Metrik  $TIME$

erfasst das nur nicht. No Magic here.

### Definition

Sei  $f : \mathbb{N} \rightarrow \mathbb{N}$ .

Dann definieren wir  $DTIME(f)$  als Menge aller Entscheidungsprobleme (oder Berechnungsprobleme)  $A$ , zu denen eine deterministische Turingmaschine  $M$  existiert, sodass  $M$   $A$  entscheidet und die Rechenzeit in  $\mathcal{O}(f(n))$  liegt.

$$DTIME(f) = \{A \mid \exists M : M \text{ entscheidet } A \text{ und } \forall x \in \Sigma^* : TIME_M(x) = \mathcal{O}(f(|x|))\}$$

### Satz

Matrixmultiplikation liegt in  $\mathcal{O}(n^3)$ , also in  $DTIME(\sqrt{n}^3)$ , wenn die Länge der Matrix auf dem Band  $n$  ist. Sie liegt sogar in  $\mathcal{O}(n^{2.78})$

Offen ist die Frage, ob sie in  $DTIME(\sqrt{n}^2)$  liegt.

## 2.4 Komplexitätsklassen

### Definition

Eine Menge der Form  $DTIME(f(n))$  heißt **deterministische Zeitkomplexitätsklasse**. Analog heißt  $NTIME(f(n))$  für nichtdeterministische Turingmaschinen **nicht-deterministische Zeitkomplexitätsklasse**.

Wir betrachten zu gegebener Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  durch Turingmaschine  $M$  folgenden Algorithmus:

Rechne  $M$  auf Eingabe  $M$  selbst für  $f(M)$  Schritte. Falls  $M$  sich bis dahin akzeptiert, verwerfe die Eingabe. Falls sie sich verwirft oder bis dahin nicht gehalten hat, akzeptiere die Eingabe.

Das durch diesen Algorithmus beschriebene Problem

$$K_f = \{M \mid M \text{ akzeptiert sich selbst nicht in höchstens } f(|M|) \text{ Schritten.}\}$$

ist "offensichtlich" entscheidbar. Die Rechenzeit für diese Entscheidung muss aber im allgemeinen  $f(|M|)$  übersteigen.

Wäre  $M_f$  eine Turingmaschine, die  $K_f$  entscheidet und außerdem  $TIME_{M_f} \leq f(|x|)$  für alle  $x$ , dann führt die Anwendung von  $M_f$  auf  $M_f$  selbst zum Widerspruch (wie beim Halteproblem).

$f$  muss dazu selbst in Zeit  $\mathcal{O}(f(n))$  berechenbar und monoton steigend sein. Man nennt  $f$  dann **zeitkonstruierbar**.

Durch geschickte Ausnutzung dieses Arguments erhält man den **Zeit-Hierarchie-Satz**:

### Satz

Falls  $f : \mathbb{N} \rightarrow \mathbb{N}$  zeitkonstruierbar ist, dann gilt:

$$DTIME(f(n)) \subset DTIME(f(n) * \log^2(f(n)))$$

wobei  $\subset$  eine echte Teilmengenbeziehung bezeichnet.

**Anmerkung:**

“Vernünftige” Funktionen wie  $2^n$ ,  $\log(n)$ ,  $\sqrt{n}$  etc. sind zeitkonstruierbar.

### Satz

Nach Borodin und Trakhtenbrot gilt das **Gap-Theorem**:

Für eine totale, berechenbare Funktion  $g : \mathbb{N} \rightarrow \mathbb{N}$  mit  $g(n) \geq n$  gibt es immer eine totale, berechenbare Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$ , sodass gilt:

$$DTIME(f) = DTIME(g \circ f)$$

Es gibt also in der Hierarchie der Komplexitätsklassen beliebig große Lücken.

### Definition

**Wichtige Komplexitätsklassen:**

$$P = \bigcup_{k \geq 1} DTIME(n^k)$$

$$E = \bigcup_{k \geq 1} DTIME(2^{kn})$$

$$EXP = \bigcup_{k \geq 1} DTIME(2^{n^k})$$

Nach dem **Zeit-Hierarchie-Satz** gilt:

$$P \subset E \subset EXP$$



### Definition

**Nichtdeterministische Zeitkomplexität** Sei  $T$  eine nichtdeterministische Turingmaschine.

Für  $x \in \Sigma$  ist  $NTIME_T(x)$

1. definiert genau dann, wenn alle Berechnungen von  $T$  auf  $x$  halten
2. Falls definiert und  $x \in L(T)$  (d.h. es gibt eine akzeptierende Berechnung von  $T$  auf  $x$ ) definiert als Länge der kürzesten akzeptierenden Berechnungen von  $T$  auf  $x$ .
3. Falls überhaupt definiert  $x \notin L(T)$  so ist  $NTIME_{(x)}$  die Länge der kürzesten Berechnung.

### Definition

Nichtdeterministische Komplexitätsklassen

$$NTIME(f(n)) = \{L \mid \exists T \text{ mit } L(T) = L \text{ und } NTIME_T(x) = \mathcal{O}(f(|x|))\}$$

Es gibt einen nichtdeterministischen Zeithierarchiesatz.

$$NP = \bigcup_{k \geq 1} NTIME(n^k)$$

$$NE = \bigcup_{k \geq 1} NTIME(2^{kn})$$

$$NEXP = \bigcup_{k \geq 1} NTIME(2^{n^k})$$

Nichtdeterminismus kann durch erschöpfende Suche deterministisch simuliert werden.  
z.B.  $NP \subseteq EXP$

Allgemein:

$$NTIME(f(n)) \subseteq DTIME(2^{O(f(n))})$$

(zeitkonstruierbar)

## 2.5 polynomielle Verifizierbarkeit

### Definition

Charakterisierung von  $NP$  durch **polynomielle Verifizierbarkeit**  $PV$ .

$$L \subseteq \Sigma^*. L \in PV \text{ genau dann, wenn}$$

$$\exists L' \in P \text{ sodass gilt } x \in L \Leftrightarrow \exists z \text{ "Loesung" mit } |h| \leq L'$$

wobei  $p$  ein Polynom ist

### Satz

$$NP = PV$$

### Beweis

“ $\subseteq$ ”:

$L \in NP$ . Sei  $T$  eine nichtdeterministische Turingmaschine für  $L$  mit Laufzeit  $p(n)$ .

$$L' = \{(x, y) \mid y \text{ codiert eine akzeptierende Berechnung von } T \text{ auf } x\}$$

$$1. x \in L \Leftrightarrow \exists y : |y| \leq (p(|x|))^2. (x, y) \in L'$$

$$2. L' \in P$$

“ $\supseteq$ ”:

Gegeben  $L, L' \in P$ .

Eine nichtdeterministische Turingmaschine  $T$  für  $L$  rät zunächst  $y$  und prüft dann  $(x, y) \in L'$

*q.e.d.*

$EXP$  im Gegensatz zu  $NP$  bzw.  $PV$  umfasst auch Probleme mit exponentiell großen Lösungen bzw solchen wo die Verifikation einer Lösung einen exponentiellen Aufwand macht.

## 3 NP und P

### 3.1 Padding

#### Definition

Sei  $L \subseteq \Sigma^*$  eine Sprache.

$$\text{padd}(L) = \{1^l O x \mid x \in L, l = 2^{|x|}\}$$

#### Satz

Es gilt:  $L \in \text{DTIME}(f(s^n))$ , dann ist

$$\text{padd}(L) \in \text{DTIME}(f(n))$$

Blase  $f$  zeitkonstant und insbesondere  $f(n) \geq n$

#### Beweis

Sei  $T$  eine deterministische Turingmaschine für  $L$  und  $\text{DTIME}_T(x) \leq c x f(2^n)$

Die folgende Maschine  $T'$  entscheidet  $padd(L)$ :

Gegeben Eingabe  $y$ , schreibe  $y + 1^l O x$  und prüfe ob  $l = 2^{|x|}$  geht in Zeit  $\mathcal{O}(|y|)$

Aufwand:  $cf(2^{|x|}) \leq c * f(|y|)$

Gesamtaufwand:  $\leq c * f(|y|) + |y| = \mathcal{O}(f(|y|))$

falls  $f(n) \geq n$  (zeitkonstruierbar)

*q.e.d.*

### Satz

Umgekehrt gilt auch:

Wenn  $padd(L) \in DTIME(f(n))$  dann  $L \in DTIME(f(s^{n+1}))$

### Beweis

Sei  $T$  eine Turingmaschine für  $padd(L)$  mit  $DTIME_T(y) \leq cf(|y|)$

Wir bauen eine Maschine für  $L$ : Gegeben Eingabe  $x$ , bilde  $y = 1^{2^{|x|}} O x$ .

Aufwand:  $\mathcal{O}(2^{|x|})$  Setze  $T$  auf  $y$  an. Aufwand:  $c * f(|y|) = c * f(2^{|x|} + |x| + 1)$

Gesamtaufwand:  $\mathcal{O}(f(2^{|x|+1}))$

*q.e.d.*

## 3.2 Was wenn $P = NP$

### Satz

Folgerung:

$$P = NP \Rightarrow E = NE$$

### Beweis

Sei  $P = NP$  und  $L \in NE$  und  $T$  eine Maschine mit Aufwand  $n^{kk}$  wobei  $k$  fest,  $n$  Länge der Eingabe.

$L \in NTIME(n^{nk}) = NTIME((2^n)^k)$  Also  $padd(L) \in NTIME(2^k)$  also  $padd(L) \in NP$  und nach Annahme  $padd(L) \in P$

Also  $padd(L) \in DTIME(n^{k'})$  also  $L \in DTIME((2^{n+1})^{k'}) = DTIME(2^{k'n+k'}) = DTIME(2^{k'n}) \subseteq E$

*q.e.d.*

Slogan: Gleichheit von Komplexitätsklassen vererbt sich nach oben.

Mit anderen Paddingfunktion zeigt man ebenso:

$$P = NP \Rightarrow EXP = NEXP$$

$$E = NE \Rightarrow EXP = NEXP$$

Kontrapositiv ausgedrückt:

$$E \neq NE \Rightarrow P \neq NP$$

etc.

Es koennte sein, dass  $P \neq NP$  aber doch  $E = NE$ .

Slogan: Trennung von Komplexitätsklassen vererbt sich (durch Padding) von oben nach unten.

### 3.3 Sogenannte Effizienz

$P$  wird gemeinhin gleichgesetzt mit effizienter Lösbarkeit.

Wachstumsverhalten:

$p$  Polynom.  $\Rightarrow$

$$\exists c > 0 : p(2n) \leq c * p(n)$$

Bei Verdopplung des Inputs wird der Output also ver- $c$ -facht.

Häufig hat stures Durchprobieren, auch Bruteforcing genannt, exponentiellen und eine echte algorithmische Lösung hat polynomiellen Aufwand.

## 3.4 Polynomialzeitreduktionen

### Definition

$f : \Sigma^* \rightarrow \Sigma^*$  beziehungsweise für Binärkodierung  $f : \mathbb{N} \rightarrow \mathbb{N}$

ist in *FP* (eine **in Polynomialzeit berechenbare Funktion**) genau dann, wenn eine polynomialzeitbeschränkte (deterministische) Transducer-Maschine existiert, die  $f$  berechnet.

Gegeben Input  $x \in \Sigma^*$ , dann hält die Maschine nach  $\leq p(|x|)$  Schritten mit Ergebnis  $f(x)$ , wobei  $p$  ein Polynom ist.

### Beispiel

#### Musterbeispiele:

- Alle Polynome sind in *FP*, zum Beispiel  $f(x) = x^3 + 10x^2 + x$
- Charakteristische Funktionen aller Probleme in *P* sind in *FP*.
- Matrixmultiplikation ist in *FP*.

#### Gegenbeispiele:

- $f(x) = 2^x$  ist nicht in *FP*, denn  $|2^x| = x + 1 \geq 2^{|x|+1} + 1 =$



$$\Omega(2^{|x|})$$

- Charakteristische Funktionen von Problemen in  $EXP \setminus P$  sind nicht in  $FP$ .

### Wahrscheinliches Gegenbeispiel:

$f(n)$  = größter Teiler von  $n$  außer  $n$  selbst. Algorithmus: Alle Zahlen von 1 bis  $n$  durchlaufen und testen, immer merken, wenn größerer Teiler gefunden.

Laufzeit:  $\Omega(n) = \Omega(2^{|n|})$  (Länge der Eingabe statt Zahl selbst)

### Satz

$FP$  ist unter Komposition (Hintereinanderausführung) abgeschlossen.

### Beweis

Seien  $f : \Sigma^* \rightarrow \Sigma^*, g : \Sigma^* \rightarrow \Sigma^* \in FP$

Wir definieren  $h(x) = g(f(x))$

Programm für  $h$ :  $y = f(x); z = g(y); \text{return } z;$

Gesamtaufwand:  $O(p_f(|x|) + p_g(p_f(|x|)))$ , wobei  $p_f$  und  $p_g$  Polynome sind, die die Laufzeit für Algorithmen für  $f$  bzw.  $g$  beschränken.

*q.e.d.*

### Definition

### Polynomielle Reduzierbarkeit

Seien  $L_1, L_2 \subseteq \Sigma^*$ .

Wir sagen  $L_1$  ist polynomiell auf  $L_2$  reduzierbar (**FP-reduzierbar**) genau dann, wenn  $f \in FP$  existiert, sodass  $x \in L_1 \Leftrightarrow f(x) \in L_2$ .

Aus Algorithmus für  $L_2$  erhält man einen für  $L_1$ , indem man  $f(x)$  berechnet und prüft, ob das Ergebnis in  $L_2$  liegt.

In Zeichen:  $L_1 \leq_p L_2$  oder  $L_1 \leq L_2$ , auch  $f : L_1 \leq L_2$

### Beispiel

$3COL = \{G = (V, E) \mid G \text{ kann mit 3 Farben gefärbt werden, d.h. } \exists c : V \rightarrow \{r, g, b\} \text{ sodass } \forall (u, u') \in E : c(u) \neq c(u')\}$

$SAT = \{\phi \mid \phi \text{ aussagenlogische Formel die erfüllbar ist}\}$

Behauptung:  $3COL \leq SAT$

$$f((V, E)) = \left( \bigwedge_{v \in V} \bigvee_{y \in \{r, g, b\}} x_{v,y} \right) \wedge$$

$$\left( \bigwedge_{v \in V} \bigwedge_{c \in \{r, g, b\}} \bigwedge_{c' \in \{r, g, b\}} x_{v,c} \rightarrow \neg x_{v,c'} \right) \wedge$$

$$\left( \bigwedge_{(v,v') \in E} \bigwedge_{y \in \{r, g, b\}} x_{v,y} \rightarrow \neg x_{v',y} \right)$$

Offensichtlich ist  $f \in FP$  und  $G \in 3COL \Leftrightarrow f(G) \in SAT$ ,

also  $3\text{COL} \leq \text{SAT}$ .

### Beispiel

$\text{KNFSAT} :=$  wie SAT, aber auf konjunktive Normalform eingeschränkt.

Es gilt trivialerweise  $\text{KNFSAT} \leq \text{SAT}$ , aber auch  $\text{SAT} \leq \text{KNFSAT}$  (Durch Einführung von Abkürzungen von Teilformeln).

### Beispiel

$3\text{SAT} :=$   $\text{KNFSAT}$  eingeschränkt auf Klauseln mit 3 Literalen.

Es gilt  $\text{KNFSAT} \leq 3\text{SAT}$ .

Man kennt keine Reduktion von  $3\text{SAT}$  auf  $2\text{SAT}$ .

### Beispiel

$\text{NODE-COVER} := \{G = (V, E), n \mid \exists U \subseteq V : |U| \leq n \text{ und } \forall (v, v') \in E : v \in U \vee v' \in U\}$

Es gilt  $\text{NODE-COVER} \leq \text{KNFSAT}$ .

und – was schwieriger zu zeigen ist –  $\text{KNFSAT} \leq \text{NODE-COVER}$ :

Gegeben: KNF  $\phi$  mit  $m$  Variablen  $x_1 \dots x_m$  und  $k$  Klauseln  $C_1 \dots C_k$  wobei

$$C_j = l_{j,1} \vee \dots \vee l_{j,k}$$

Falls  $3\text{SAT}$ , so sind alle  $k_j = 3$ .

Die  $l_{j,i}$  sind Literale, d. h. negierte oder nicht-negierte Variablen.

Wir konstruieren Graphen  $G = (V, E)$  wie folgt:

- Für jede Variable  $x_t$  zwei Knoten  $x_t, \neg x_t$
- Für jede Klausel  $C_j$   $k_j$  Knoten  $(l_{j,1}) \dots (l_{j,k})$ ,  
Insgesamt  $2m \sum_{j=1}^k k_j$  Knoten
- Kanten:
  - $(x_t, \neg x_t)$
  - Vollständiger Graph für  $l_{j,1}$  bis  $l_{j,k}$
  - $(x_t, l_{j,i})$  bzw.  $(\neg x_t, l_{j,i})$ , falls  $l_{j,i} = x_t$  bzw.  $l_{j,i} = \neg x_t$

### 3.5 NP-Härte und NP-Vollständigkeit

#### Definition

$L \subseteq \Sigma^*$  ist **NP-hart** (NP-schwer, NP-schwierig) genau dann, wenn

$$\forall L' \in NP : L' \leq_p L$$

#### Satz

HALT (Halteproblem) ist NP-hart.

#### Beweis

Gegeben:  $L' \in NP$ . Baue deterministische Turingmaschine  $M$ , sodass  $M(x)$  hält genau dann, wenn  $x \in L'$  Brute-force Suche, Laufzeit exponentiell.

$x \in L' \Leftrightarrow (M, x) \in \text{HALT}$

also ist  $f(x) = (M, x)$  eine Reduktion von  $L'$  auf  $\text{HALT}$   $f : L' \leq \text{HALT}$ .

*q.e.d.*

### Definition

$L \subseteq \Sigma^*$  ist **NP-vollständig** genau dann, wenn  $L$  NP-hart ist und  $L \in \text{NP}$ .

### Satz

$\text{HALT} \notin \text{NP}$ .

### Satz

#### **Satz von Cook**

$\text{SAT}$  ist NP-vollständig.

### Beweis

$\text{SAT} \in \text{NP}$ : trivial.

Sei  $L \in \text{NP}$  gegeben und o. B. d. A.  $M$  eine nichtdeterministische Turingmaschine für  $L$  mit einem Band  $M = (\Sigma, Q, q_0, F, I)$  und  $p$  ein Polynom, das die Laufzeit von  $M$  beschränkt. Gegeben weiterhin  $x = x_1 \dots x_n$  Input.

Gesucht: aussagenlogische Formel  $\phi = f(x)$ , sodass  $\phi$  erfüllbar ist genau dann, wenn  $M$  akzeptiert  $x$ .  $f$  muss aus  $x$  in polynomieller Zeit berechenbar sein, d. h.  $f \in \text{FP}$ .

$M$  akzeptiert  $x$  genau dann, wenn eine akzeptierende Berechnung von  $M$  auf  $x$  existiert. Solch eine Berechnung hat höchstens  $p(n)$  Schritte und o. B. d. A. genau  $p(n)$

Schritte. Die Bandbeschriftung zu jedem dieser  $p(n)$  Schritte besteht aus höchstens  $p(n)$  Symbolen und o. B. d. A. genau  $p(n)$  Symbolen.

Die Formel  $\phi$  verwendet die Variablen

- $Q_t^i$ : Zur Zeit  $t$  ist  $M$  im Zustand  $i$ .
- $P_{s,t}^i$ : Zur Zeit  $t$  enthält Bandposition  $s$  das  $i$ -te Symbol.
- $S_{s,t}$ : Zur Zeit  $t$  ist der Kopf in Position  $s$ .

$$q = A \wedge B \wedge C \wedge D \wedge E \wedge F$$

*Details im Buch...*

*q.e.d.*

3SAT, NODE-COVER sind auch NP-vollständig.

Allgemein gilt:  $L$  NP-vollständig und  $L' \in NP, L \leq L'$  so folgt  $L'$  NP-vollständig.

**Anmerkung:**  $\leq$  ist transitiv, da  $FP$  unter Komposition abgeschlossen ist.

**Anmerkung:** 3COL, TRAVELINGSALESMAN, SUBSETSUM etc. sind auch NP-vollständig.

## 3.6 Etwas zwischen P und NP

### Satz

#### Satz von Ladner

Falls  $P \neq NP$ , dann

$$\exists A \in NP \setminus P : A \text{ nicht NP-vollst\"andig.}$$

$A$  liegt also echt zwischen  $P$  und NP-vollst\"andig.

### Definition

#### Diagonalisierung

Um zu zeigen, dass eine Sprache  $A$  nicht in einer Klasse  $\mathcal{C}$  ist, beziehungsweise um solch ein  $A$  zu konstruieren, kann man eine effektive (FP) Aufz\"ahlung von Turingmaschine  $(M_i)_i$  verwenden, sodass  $\mathcal{C} = \{L(M_i) \mid i \geq 0\}$  und dann daf\"ur sorgen, beziehungsweise zeigen, dass  $\forall i : A \neq L(M_i)$  beziehungsweise  $\forall i : A \triangle L(M_i) \neq \emptyset$ .

Das hei\"uft  $\forall i \exists x : (x \in A \wedge x \notin L(M_i)) \vee (x \notin A \wedge x \in L(M_i))$

### Lemma

Es existiert eine FP-Funktion  $i \mapsto M_i$ , sodass  $DTIME_{M_i}(x) \leq (|x| + 2)^2$  und  $P =$

$\{L(M_i) \mid i \geq 0\}$ .

### Lemma

Es existiert eine FP-Funktion  $i \mapsto f_i$  wobei  $f_i$  eine Übersetzermaschine ist und  $FP = \{f_i \mid i \geq 0\}$  und  $DTIME_{f_i}(x) \leq (|x| + 2)^i$ . Insbesondere  $|f_i(x)| \leq (|x| + 2)^i$ .

Es ist klar, dass  $A \in NP$  aber  $A \notin P$  und  $A$  nicht NP-vollständig, wenn

- $A \in NP$
- $\forall i \exists x : x \in A \Delta L(M_i)$
- $\forall i \exists x : x \in SAT \wedge f_i(x) \notin A$  oder  $x \notin SAT \wedge f_i(x) \in A$

Das heißt  $f_i$  ist keine Reduktion von  $SAT$  auf  $A$ .

Wir konstruieren  $A$  in der folgenden Form:

$$A = \{x \mid x \in SAT \wedge f(|x|) \text{ gerade.}\}$$

$f$  wird sogleich rekursiv definiert derart, dass dieses  $A$  die Bedingungen 1, 2 und 3 erfüllt.

Man sollte also versuchen sicherzustellen, dass

- $f(n)$  in Zeit  $p(n)$  berechenbar für Polynom  $p$  (Bedingung 1).
- Für alle  $i$  existiert  $x$  mit  
 $x \in SAT$  und  $f(|x|)$  gerade und  $x \notin L(M_i)$   
oder



$(x \notin SAT \text{ oder } f(|x|) \text{ ungerade}) \text{ und } x \in L(M_i)$

- Für alle  $i$  existiert  $x$ , sodass  $x \in SAT$  und  $(f(|f_i(x)|))$  ungerade oder  $f_i(x) \notin SAT$   
oder  
 $x \notin SAT$  und  $f(|f_i(x)|)$  gerade und  $f_i(x) \in SAT$

$f$  wird jetzt rekursiv definiert.

Wir schreiben  $A_f = \{x \mid x \in SAT \wedge f(|x|) \text{ gerade}\}$

$f(n+1) = IF \ (2 + \log \log n)^{f(n)} \geq \log n$

$THEN \ f(n)$

$ELIF \ \exists x : |x| \leq \log \log n \text{ und } x \in L(M_i) \wedge x \notin A_f \text{ oder } x \notin L(M_i) \wedge x \in A_f$

$THEN \ f(n) + 1 \text{ ebe } f(n)$

$ELIF \ \exists x : |x| \leq \log \log n$

$THEN \ f(n) + 1$

Um  $f(n+1)$  zu berechnen wird rekursiv nur auf Werte  $f(m)$  mit  $m \leq n$  zugegriffen, also ist  $f$  eine totale Funktion.

Es genügt, ein Polynom  $p(n)$  zu finden, sodass in Zeit  $p(n)$  der Wert  $f(n+1)$  aus  $f(0), f(1), \dots, f(n)$  bestimmt werden kann.

Die Laufzeit für  $f(n)$  ist nämlich dann  $\mathcal{O}(\sum_{m < n} p(m)) = \text{poly}(n)$ .

Offensichtlich ist  $A = A_f \neq L(M_i)$ , falls  $f(n) = 2i + 1$  für ein  $n$ , denn dann war  $f(n') = 2i$  für ein  $n' < n$  und  $f(n' + 1) = 2i + 1$  also die Suche in Fall 2 erfolgreich.

Ebenso ist  $f_i$  keine Reduktion:  $SAT \leq A_f$ , falls  $f(n) = (2i + 1) + 1$  für ein  $n$ .

Das heißt wir müssen zeigen, dass  $f$  surjektiv ist, d.h. dass jeder Fall irgendwann erfolgreich abgeschlossen wird.

*Details dazu auf der Website.*

## 3.7 Orakel-Turingmaschinen

Orakel-Turingmaschinen als Mittel zu zeigen, dass die Beweismethoden Diagonalisierung<sup>1</sup> und Simulation<sup>2</sup> nicht helfen, um  $P = NP$  zu entscheiden.

### Definition

Eine **Orakel-Turingmaschine**  $T$  hat ein zusätzliches Band (Orakelband) und drei zusätzliche Zustände  $q_Q$  (Frage),  $q_{yes}$ ,  $q_{no}$  (Antwort).

Ist  $A \subseteq \Sigma^*$ , dann definiert man Berechnungen  $T^A(x)$  von  $T$  auf  $x$  mit Orakel  $A$  wie folgt:

- Wie üblich mit der zusätzlichen Regel:  
Falls  $T$  in  $q_Q$  so wird  $T$  in Zustand  $q_{yes}$ ,  $q_{no}$  versetzt und zwar in einem Schritt, je nach dem, ob die aktuelle Beschriftung  $z \in \Sigma^*$  des Orakelbands (Anfrage/Query) in  $A$  ist ( $q_{yes}$ ) oder nicht ( $q_{no}$ ).

Man schreibt  $L^A(T)$  oder  $L(T^A)$  für die von  $T$  akzeptierte Sprache, falls Anfragen gemäß  $A$  beantwortet werden.

### Beispiel

Sei  $STCONN = \{(G, s, t) \mid \exists \text{ Pfad von } s \text{ nach } t \in G\}$

---

<sup>1</sup>zum Beispiel benutzt für  $NP \subseteq EXP$

<sup>2</sup>zum Beispiel benutzt für  $P \subset EXP$  (Ladner)

Feststellen, ob ein Graph  $G$  einen nichttrivialen Zyklus enthält. Zähle alle Paare  $(u, v)$  auf und frage jeweils  $(G, u, v)$  und  $(G, v, u)$  ab.

Damit ist eine Maschine  $T$  beschrieben, sodass  $CYCLE = L^{STCONN}(T)$ . Die Laufzeit von  $T$  ist  $|G|^2$  (insbesondere polynomiell). Es ist also  $CYCLE \in P^{STCONN}$ . (Definition unten)

Nachdem nun  $STCONN \in P$  folgt  $CYCLE \in P$ .

### Definition

Sei  $A \subseteq \Sigma^*$ .

Man definiert  $P^A$  als die Menge aller Sprachen  $L$  sodass eine deterministische Turingmaschine  $T$  existiert mit  $L = L^A(T)$  und  $DTIME(T^A(x)) \leq p(|x|)$  für ein Polynom  $p$ .

Analog  $NP^A$ .

**Beobachtung:**  $A \in P \Rightarrow P^A = P \wedge NP^A = NP$ .

### Beispiel

$$SAT \in P^{SAT}$$

$$NODE - COVER \in P^{SAT}$$

$$NP \in P^{SAT}$$

$$TAUT = \{\phi \mid \phi \text{ allgemeingültig}\} \in P^{NP}$$

$$IMPL = \{(\phi, \psi) \mid \phi \text{ allgemeingültig} \Rightarrow \psi \text{ allgemeingültig}\} \in P^{NP}$$

$$CIRCUIT - MIN = \{(\text{Schaltkreis } C, A) \mid \text{Schaltkreis } C' \text{ der Größe } \leq k \text{ und } C \equiv C'\} \in NP^{SAT}$$

Wir zeigen jetzt die folgenden zwei Sätze, aufgrund derer kein Beweis für  $P = NP$  oder  $P \neq NP$  existieren kann, welcher in Gegenwart von Orakeln auch funktioniert:

### Satz

$$\exists A \in \Sigma^* : P^A = NP^A$$

### Beweis

$A = \{(T, x, 0^k) \mid \text{deterministische Turingmaschine } T \text{ akzeptiert } x \text{ und benutz dabei } \leq k \text{ Bandzellen}\}$

$A$  ist offensichtlich entscheidbar.

Sei  $T$  eine nichtdeterministische Orakel-Turingmaschine und  $p(n)$  ein Polynom, das die Laufzeit von  $T$  beschränkt und somit auch die Größe aller Orakelanfragen.

Wir müssen eine deterministische polynomiell zeitbeschränkte Turingmaschine  $T'$  mit Orakel  $A$  bauen, sodass  $L^A(T') = L^A(T)$ .

Zunächst konstruieren wir eine deterministische Turingmaschine  $T_{HILF}$ , die ohne Orakelbenutzung die Sprache  $L^A(T)$  entscheidet, indem alle Orakelanfragen "mit Bordmitteln" (also selbst) beantwortet werden unter Verwendung der Entscheidbarkeit von  $A$ .

Vollständige Berechnungssequenzen einer Berechnung, deren Bandplatz  $\leq k$  ist und  $t$  Schritte lang ist, benötigen Platz  $\mathcal{O}(k * t)$ .

Orakelanfragen einer Berechnung von  $T$  auf  $x$  (Eingabe) haben die Form  $(S, y, 0^k)$  wobei  $|S|, |y|, k \leq p(|x|)$ .

Wir verwenden also jetzt 3 Hilfsbänder, eines für die nichtdeterministische Berechnung von  $T$  auf  $x$ , die wir der Reihe nach alle simulieren, eines für Orakelanfragen, eines für die Beantwortung der Orakelanfragen.

Diese Maschine ist deterministisch und benötigt auf ihren Bändern höchstens  $q(|x|)$  Platz, wobei  $q$  ein von  $p$  abgeleitetes Polynom ist (in etwa  $q(n) = \mathcal{O}(p(n)^2)$ ).

Die eigentliche Maschine  $T'$  arbeitet jetzt wie folgt:

Gegeben Eingabe  $x$ , schreibe  $(T_{HILF}, x, O^{q(|x|)})$  auf das Orakelband. Falls  $q_{yes}$ , dann akzeptiere. Falls  $q_{nein}$ , dann verwirfe.

*q.e.d.*

### Satz

$$\exists B \in \Sigma^* : P^B \neq NP^B$$

### Beweis

Falls  $B \subseteq \Sigma^*$ , definiere  $L_B = \{0^k \mid \exists x \in \Sigma^* : |x| = k \wedge x \in B\}$

Offensichtlich ist  $L_B \in NP^B$ , egal was  $B$  ist.

Es gilt jetzt,  $B$  so zu wählen, dass für jede polynomiell zeitbeschränkte deterministische Orakel-Turingmaschine gilt:  $L_B \neq L^B(T)$ .

Sei  $i \mapsto T_i$  eine effektive Aufzählung von Orakel-Turingmaschinen, sodass  $DTIME(T_i^x(x)) \leq |x|^i + i$  (alternativ  $(|x| + 2)^i$  wie letztes Mal).

Für alle deterministischen Orakel-Turingmaschinen  $S$  und alle Orakel  $x$  muss  $i$  existieren, sodass  $L(S^x) = L(T_i^x)$ .  $T_i$  ist die durch  $i$  beschriebene Orakel-Turingmaschine künstlich auf Laufzeit  $n^i + i$  beschränkt. Jetzt muss also für jedes  $i$  ein  $n_i$  existieren, sodass  $T_i^B(0^{n_i})$  akzeptiert und  $B$  enthält kein Wort der Länge  $n_i$  (dann ist nämlich

$0^{n_i} \notin L_B$ ), oder aber  $T_i^B(0^{n_i})$  verwirft und  $B$  enthält ein Wort  $x_i$  mit  $|x_i| = n_i$ , denn dann ist  $0^{n_i} \in L_B$ .

Dann ist in der Tat  $L_B \notin P^B$ .

**Beobachtung:** Wenn  $T_i^X(x)$  nach  $t$  Schritten hält und  $U$  aus Wörtern  $y$  mit  $|y| > t$  besteht, dann gilt  $T_i^{X \cup U}(x)$  akzeptiert  $\Leftrightarrow T_i^X(x)$  akzeptiert.

Wir definieren rekursiv  $n_i \in \mathbb{N}, B(i) \subseteq \Sigma^*, \Sigma = \{0, 1\}, n_0 = 0, B(0) = \emptyset$

Falls  $B(0), \dots, B(i-1)$  schon definiert, definiere  $n_i, B(i)$  so, dass gilt  $\exists x : |x| = n_i \in B(i) \Leftrightarrow T_i^{B(i)}(0^{n_i})$  akzeptiert nicht.

Außerdem sollte  $T_i(0^{n_i})$  keine Elemente von  $B(j) j > i$  anfragen.

$$B = \bigcup_{i \geq 0} B(i)$$

Es gilt dann  $T_i^{B(i)}(0^{n_i})$  akzeptiert nicht  $\Leftrightarrow T_i^{B(i)}(0^{n_i})$  akzeptiert nicht  $\Leftrightarrow 0^{n_i} \in L_{B(i)} \Leftrightarrow 0^{n_i} \in L_b$  (Falls wir zusätzlich dafür sorgen, dass  $B(j)$  mit  $j \geq i$  keine Elemente der Länge  $n_i$  enthält.)

$$\text{Allgemein: } B(i) = \begin{cases} B(i-1) & , \text{ falls } \dots \\ B(i-1) \cup \{x\} & , \text{ falls } \dots \end{cases}$$

$n_i B(i) \in n_{i-1}(i-1)$   $n_i$  sodass

- $n_i > n_{i-1}^{i-1} + i - 1$  (Größer als die Laufzeit von  $T_{i-1}(0^{n_{i-1}})$  und  $T_j(0^{n_j})$  für  $j < i$ .)
- $2^{n_i} > n_i^i + i$  (da  $2^x$  schneller wächst als  $x^i + i$ .)

Es gibt also mehr Wörter  $x$  mit  $|x| = n_i$  als die Laufzeit von  $T_i(0^{n_i})$ .

Halbkonkrete Ausführung dieser Aufzählung für die ersten drei Schritte:

$$n_0 = 0, B(0) = \emptyset, n_1 > 0^0 + 0, 2^{n_1} > n_1 + 1$$

Rechne  $T_1^{B(0)}(000)$ . Wir nehmen an, dass nicht akzeptiert wird. Also  $B(1) = B(0) = \emptyset$

$n_2 > n_1^1 = 4, 2^{n_2} > n_2^2 + 2, \rightsquigarrow n_2 = 5$  Rechne  $T_2^{B(1)}(00000)$ . Wir nehmen an, dass akzeptiert wird. Diese Rechnung dauerte  $\leq 5^2 + 2 = 27$  Schritte. Es gibt 32



Wörter der Länge 5. Sei  $x$  eines, das nicht abgefragt wurde.  $x = 10110$ .  $B(2) = B(1) \cup \{10110\} = \{10110\}$

$n_3 > 27, 2^{n_3} > n_3^3 + 3, \rightsquigarrow n_3 = 28$  Rechne  $T_3^{B(2)}(0^{28})$ . Wir nehmen an, dass nicht akzeptiert wird. Also  $B(1) = B(0) = \emptyset$

...

Invariante  $B \cap \{x \mid |x| \leq n_i\} = B(i)$

Rechenzeit von  $T_i(0^{n_i}) < n_{i+1}$  ( $> n_i^i + i$ )

$B$  unterscheidet sich von  $B(i-1)$  nur durch Wörter die von  $T_i^{B(i-1)}(0^{n_i})$  nicht angefragt werden. Entweder, da sie länger sind als die Rechenzeit oder von der Länge  $n_i$  sind, aber so gewählt wurden, dass sie nicht gefragt werden.

$T_i^B(0^{n_i})$  akzeptiert

$\Leftrightarrow T_i^{B(i-1)}(0^{n_i})$  akzeptiert

$\Leftrightarrow \neg \exists x \in B(i) : |x| = n_i$

$\Leftrightarrow 0^{n_i} \notin L_B$

Das heißt  $L(T_i^B) \neq L_B$

$L_B \notin P^B$

*q.e.d.*

## 3.8 Polynomielle Hierarchie

### Definition

$co-\phi = \{L \subseteq \Sigma^* \mid \Sigma^* \setminus L \in \phi\}$

$$co-P = P$$

$co-NP$  ist nicht offensichtlich gleich  $NP$  wegen der asymmetrischen Akzeptanzbedingungen bei Nichtdeterminismus.

### Definition

Die **polynielle Hierarchie** (PH):

$$\Delta_0^P = P$$

$$\Sigma_0^P = \Pi_0^P = P$$

$$\Sigma_1^P = NP$$

$$\Pi^P = co-NP$$

Das soll andeuten, dass von der PH die Rede ist. Es hat nicht mit einem Exponenten oder Orakel zu tun.

### Definition

#### **relativierte Quantoren**

Notation:  $\exists_n x. A(x) \equiv \exists x \in \Sigma^*. |x| \leq n \wedge A(x)$

Ebenso:  $\forall_n x. A(x) \equiv \forall x \in \Sigma^*. |x| \leq n \Rightarrow A(x)$

Außerdem:  $\neg \exists_n x. A(x) \Leftrightarrow \forall_n x. \neg A(x)$

und:  $\neg \forall_n x. A(x) \Leftrightarrow \exists_n x. \neg A(x)$

### Definition

Die weitere **polynielle Hierarchie** (PH):

$$\Sigma_2^P = NP^{SAT} = NP^{NP}$$

$$\Pi_2^P = co-NP^{SAT} = co-NP^{NP}$$

$$\Delta_2^P = P^{SAT} = P^{NP}$$

$$\Sigma_{n+1}^P = NP^{\Sigma_n^P}$$

$$\Pi_{n+1}^P = co-NP^{\Sigma_n^P}$$

$$\Delta_{n+1}^P = P^{\Sigma_n^P}$$

*Anmerkung:*

$$NP^\phi = \bigcup_{X \in \phi} NP^X$$

## 3.9 Kollabieren der polynomiellen Hierarchie

Falls  $P = NP$ , so folgt  $\Sigma_n^P = P$ .

Im allgemeinen gilt: Falls  $\Sigma_{n+1}^P = \Sigma_n^P$  für ein bestimmtes  $n$ , dann auch  $\Sigma_{n'}^P = \Sigma_n^P$  für alle  $n' \geq n$ . Man sagt dann PH kollabiere auf der  $n$ . Stufe.

Satz

$$L \in \Sigma_2^P \Leftrightarrow$$

$$\exists L' \in P, \text{ Polynom } p : x \in L \Leftrightarrow \exists_{p(|x|)} y. \forall_{p(|x|)} z. (x, y, z) \in L'$$

Beweis

- “ $\Leftarrow$ ”:

Seien  $L' \in P$ ,  $p$  Polynom vorgegeben.

Gesucht: nichtdeterministische Polynomialzeit-Orakel-Turingmaschine  $M$  sodass  $M$  mit geeignetem  $NP$ -Orakel die Sprache  $L$  entscheide.

$$L'' = \{(x, y) \mid \exists_{p(|x|)} z. (x, y, z) \notin L'\}$$

$$L'' \in NP \text{ (Rate } z \text{ und prüfe } (x, y, z) \notin L')$$

$$x \in L \Leftrightarrow \exists_{p(|x|)} y. (x, y) \notin L''$$

Die Maschine  $M$  rät also  $y$  und prüft  $(x, y) \notin L''$  mit Orakel für  $L''$ .

- “ $\Rightarrow$ ”:

Sei  $M$  eine nichtdeterministische durch  $p(n)$  laufzeitbeschränkte Turingmaschine mit Orakel  $X \in NP$ , z.B.  $X = SAT$ ,  $L = L(M)$

Es ist  $x \in L = L(M)$  genau dann, wenn

$\exists$  Lauf  $y$  von  $M$  auf  $x$  :  $M$  akzeptiert  $x \wedge |y| \leq q(|x|)$

wobei  $q$  ein Polynom ist mit  $q(n) = \mathcal{O}(n^2)$

Um zu prüfen, ob  $y$  tatsächlich ein LAuf ist und noch dazu akzeptierend, muss neben allem möglichen, was in polynomieller Zeit geht, z.B.

- Folgekonfigurationen jeweils gemäß der Maschinentafel ( $\delta_M$ ) aus Vorgängerkonfigurationen enthalten,
- Am Anfang Startzustand, am Ende akzeptierender Zustand,
- Input okay in Startkonfiguration kopiert,

auch geprüft werden, dass alle Orakelanfragen richtig beantwortet wurden.

...

und

$\exists \eta. \eta_i$  erfüllt die  $i$ . positiv beantwortete Orakelanfrage  $q_i \in SAT$ .

und

$\forall \rho. \rho_j$  erfüllt die  $j$ . negativ beantwortete Orakelanfrage  $\rho_j \in SAT$  nicht.

Es gibt also ein Polynom  $q(n)$  sodass  $x \in L(M) \Leftrightarrow \exists_{q(|x|)} y. L_1(x, y) \wedge \exists_{p(|x|)} \eta. L_2(y, \eta) \wedge \forall_{q(|x|)} \rho. L_3(y, \rho)$

wobei  $L_1, L_2, L_3 \in P$

- $L_1(x, y) \iff y$  ist akzeptierender Lauf von  $M$  auf  $x$  bis auf Orakelanfragen

- $L_2(y, \eta) \iff$  Die in  $y$  positiv beantworteten Orakelanfragen sind  $\rho_1 \dots \rho_k, \eta = y_1 \dots y_k$  und  $\eta_i \models \psi_i$  für  $i = 1 \dots k$
- $L_3(y, \rho) \iff$  Die in  $y$  negativ beantworteten Orakelanfragen sind  $\psi_1 \dots \psi_k, \rho = \rho_1 \dots \rho_k$  und  $\rho \not\models \psi_i$  für  $i = 1 \dots k$

*q.e.d.*

### Korollar

$$L \in \Pi_2^P \iff$$

$$\exists \text{ Polynom } p \wedge L' \in P : x \in L \iff \forall_{p(|x|)} y. \exists_{p(|x|)} z. (x, y, z) \in L'$$

### Satz

Ist  $L \in \Sigma_n^P$ , so existiert ein Polynom  $p(n)$  und  $L' \in P$  sodass

$$x \in L \iff y_1 \forall_{p(|x|)} y_2 \exists_{p(|x|)} y_3 \dots Q_{p(|x|)} y_n. (x, y_1, \dots y_n) \in L'$$

$n$  gerade:  $Q = \forall$

$n$  ungerade:  $Q = \exists$

Beweis durch Induktion über  $n$ .

### Satz

#### **Satz von Kamp-Lipton**

Falls  $SAT$  polynomiell große Schaltkreise hat, so kollabiert die PH auf der zweiten Stufe.

Das heißt  $\forall n \geq 2 : \Sigma_n^P = \Sigma_2^P$ .

Dass  $SAT$  polynomielle Schaltkreise hat soll heißen: Es gibt ein Polynom  $p(n)$  und für jedes  $n \in \mathbb{N}$  einen booleschen Schaltkreis  $C_n$  mit  $n$  Inputs und Größe  $\leq p(n)$

$(|C_n| \leq p(n))$ .

Für alle aussagenlogische Formeln  $\phi$  mit  $|\phi| = n$  gilt  $\phi \in SAT \Leftrightarrow C(\phi) = TRUE$ ,  
 $C(\phi)$  die Bitkodierung von  $\phi$  an die  $n$  Inputs von  $C$  anlegen.

Kurznotation:  $SAT \in P/poly$

Der Satz von Kamp-Lipton sagt also  $SAT \in P/poly \Rightarrow PH = \Sigma_2^P$

Hilfsmittel: Selbstreduzierbarkeit von SAT

Ein Schaltkreis  $C_n$  wie oben beschrieben kann so umgebaut werden in einem Schaltkreis  $D_n$  polynomielle Größe, dass bei Antwort TRUE eine erfüllende Belegung zurückgeliefert wird.

Das heißt  $D_n$  hat  $n$  Ausgänge, die eine Belegung kodieren sollen. Spezifiziere

- $\phi \in SAT.D_n(\phi)(\eta, TRUE)$  mit  $\eta \models \phi$
- $\phi \notin SAT.D_n(\phi)(\_, FALSE)$

$D_n$  ruft  $C_n$  insgesamt  $m \leq n$  mal auf, wobei  $m$  die Zahl der Variablen plus eins ist.

---

Vorlesung vom 26.11.15

---



## 4 Platzkomplexität

### 4.1 Platzkonstruierbare Funktionen

#### Definition

Eine Funktion  $s(n)$  heißt **platzkonstruierbar** genau dann, wenn eine deterministische Turingmaschine existiert, die bei Eingabe  $0^n$  genau  $s(n)$  Bandfelder beschreibt und dann hält.

#### Beispiel

Alle Polynome mit Koeffizienten  $\in \mathbb{Q}^+$ , die Wurzelfunktion, die Logarithmusfunktion, die Potzenfunktion usw. sind platzkonstruierbar.

### 4.2 Platzverbrauch einer Turingmaschine

#### Definition

Der **Platzverbrauch einer Turingmaschine** (deterministisch oder nichtdeterministisch) bei Eingabe  $x$  ist

- **erste Definition**

die Größe des beschriebenen Teils aller Bänder am Ende der Berechnung. (Mit dieser Definition ist der Platzverbrauch stets  $\geq |x|$ ).

- **zweite Definition**

die Endgröße aller anderen Bänder, wobei das Eingabeband nicht überschrieben werden darf.

Die zweite Definition ist Standard, wenn sublineare Platzschranken betrachtet werden, zum Beispiel  $\log(n)$ . Oberhalb von  $\mathcal{O}(n)$  sind die beiden Definitionen äquivalent.

Notation:  $DSPACE_M(x)$  und  $NSPACE_M(x)$

$$DSPACE_M(s(n)) = \{L \mid \exists DTMM : L = L(M) \wedge DSPACE_M(x) = \mathcal{O}(s(|x|))\}$$

$$NSPACE_M(s(n)) = \{L \mid \exists DTMM : L = L(M) \wedge NSPACE_M(x) = \mathcal{O}(s(|x|))\}$$

$$PSPACE = \bigcup_{k \geq 0} DSPACE(n^k) \text{ (polynomieller Platz)}$$

$$LSPACE = DSPACE(n)$$

$$LOGSPACE = DSPACE(\log n) \text{ (auch als } L \text{ bezeichnet)}$$

$$NLOGSPACE = NSPACE(\log n) \text{ (auch als } NL \text{ bezeichnet)}$$

### Beispiel

STCONN (Erreichbarkeit in gerichteten Graphen) ist  $\in NLOGSPACE$  (rate Pfad) und  $\in LSPACE$  (Tiefensuche/Breitensuche)

Es gibt eine triviale, aber wissenswerte Beziehung zwischen Zeit- und Platzkomplexität:

$$DSPACE(s(n)) \subseteq DTIME(2^{\mathcal{O}(s(n))})$$

Hat die Berechnung nach  $2^{c*s(n)}$  Schritten nicht geendet, so kann abgebrochen werden wegen Wiederholung einer globalen Konfiguration. (Tatsächlicher Platzverbrauch  $\leq c * s(n)$ )

$$DTIME(t(n)) \subseteq DSPACE(t(n))$$

Mehr Platz als Laufzeit kann nicht angefordert werden.

### Satz

Für deterministische Einband-Turingmaschinen  $T$  gilt:

$$DTIME_T(x) = \mathcal{O}(t(|x|)) \Rightarrow L(T) \in DSPACE(\sqrt{t(n)})$$

Für Mehrband-Turingmaschinen gibt es einen ähnlichen Satz, bei dem der Platz allerdings etwas größer ist. Dass er für Einband-Turingmaschinen so gut ist, ist gewissermaßen kurios.

## 4.3 Platzhierarchiesatz

### Satz

“Echt mehr Platz hilft auch mehr.”

*Für genaue Aussage und Beweis siehe z.B. Papadimitrion*

Wichtige Konsequenz:

$$LOGSPACE \subset PSPACE$$

$$LOGSPACE \subseteq NLOGSPACE \subseteq P \subseteq NP \subseteq PH \subseteq PSPACE \subseteq EXP$$

Von jeder dieser Inklusionen ist unbekannt, ob sie echt sind. Mindestens eine muss aber echt sein.

## 4.4 Zusammenhänge von Platzkomplexitätsklassen

### Satz

#### Satz von Savitch

Für eine platzkonstruierbare Funktion  $s(n) \geq \log(n)$  ist  
 $NSPACE(s(n)) \subseteq DSPACE(s(n)^2)$

(Vergleiche  $NTIME(t(n)) \subseteq DTIME(2^{\mathcal{O}(t(n))})$  )

### Beweis

Sei eine nichtdeterministische Turingmaschine  $T$  gegeben mit Platzbedarf  $S = c * s(|x|)$  bei Eingabe  $x$ . Wir betrachten eine Kodierung der globalen Konfigurationen von  $T(x)$  durch Wörter der Länge  $S$  und o. B. d. A. gebe es exakt eine akzeptierende Endkonfiguration  $s_{ACC}$ . (Alle Bänder am Ende löschen, d.h. mit 0 überschreiben.)

$x \in L(T) \iff s_{ACC}$  von  $s_{INI}$  aus in  $\leq 2^S$  Schritten erreichbar

Hier steht  $s_{INI}$  für die Startkonfiguration bei Eingabe  $x$ .  $2^S$  ist die Gesamtzahl der Konfigurationen.

Das heißt  $s_{ACC}$  ist von  $s_{INI}$  aus im Graphen der Konfigurationen erreichbar (Spezialfall von STCONN).

*q.e.d.*

### Notation:

$s \rightarrow_T s'$ :  $s'$  ist 1-Schritt-Folgekonfiguration von  $s$  in  $T$  und kann in *LOGSPACE* entschieden werden.

$REACH(s, s')$ :  $s \rightarrow^* s'$  ( $s'$  ist von  $s$  erreichbar)

$REACH(s, s', i)$ :  $s \rightarrow^{\leq 2^i} s'$  ( $s'$  ist von  $s$  in weniger als  $2^i$  Schritten erreichbar)

$x \in L(T) \iff REACH(s_{INI}, s_{ACC}, S)$

Es gilt

$$REACH(s, s', 0) \iff s = s' \vee s \rightarrow_T s'$$

$$(2^0 = 1)$$

$$REACH(s, s', i+1) \iff \exists \check{s} : REACH(s, \check{s}, i) \wedge REACH(\check{s}, s', i)$$

$$(2^{i+1} = 2 * 2^i)$$

Dies liefert eine rekursive Implementierung von  $REACH(s, s', i)$

( $\exists \check{s} \rightsquigarrow$  for  $\check{s} \in$  globale Konfigurationen)

Der Rekursionsstack hat Tiefe  $S$  (Toplevel-Aufruf  $REACH(s_{INI}, s_{ACC}, S)$ ).

Jeder Activationrecord hat Größe  $\mathcal{O}(S)$  genauer gesagt  $2S$  für die beiden Parameter  $s, s', \log(S)$  für die Parameter  $i$ . Wenn gewünscht noch ein weiteres  $S$  für die for-Schleife.

Die Gesamtgröße des Stacks ist beschränkt durch  $S * \mathcal{O}(S) = \mathcal{O}(S^2)$ .

Historisch wurde zunächst gezeigt, dass STCONN in  $DSPACE(\log(n)^2)$  liegt. Der Satz von Savitch kann auch hieraus abgeleitet werden.

## Satz

### Satz von Immerman-Szelepcsényi

Sei  $s(n) \geq \log(n)$ .

Dann ist  $NSPACE(s(n)) = co-NSPACE(s(n))$

Wichtiger Spezialfall:  $co-STCONN \in NSPACE(\log(n)) = NL = NLOGSPACE$   
Die allgemeine Behauptung kann aus diesem Spezialfall leicht gefolgert werden (durch den Graph der globalen Konfiguration).

## Beweis

Es sei eine nichtdeterministische Turingmaschine  $T$  vorgelegt und  $NSPACE_T(x) \leq c * s(|x|)$ . Wir müssen eine nichtdeterministische Turingmaschine  $T'$  konstruieren, so dass  $NSPACE_{T'}(x) = \mathcal{O}(s(|x|))$  und  $x \in L(T') \Leftrightarrow x \notin L(T)$ .

Es existiert akzeptierende Berechnung von  $T'$  auf  $x$  genau dann, wenn alle Berechnungen von  $T$  auf  $x$  verwerfen.

Sei  $x$  fixiert und o. B. d. A.  $\Sigma = \{0, 1\}$ .

Schreibe  $s_{INI}$  für die globale Startkonfiguration von  $T$  auf  $x$  und  $s_{ACC}$  für die (o. B. d. A.. einzige) akzeptierende globale Konfiguration. Weiter sei  $S = c' * s(|x|)$  so gewählt, dass alle globalen Konfigurationen durch 0/1-Strings der Länge  $S$  kodiert werden.

$s \rightarrow_T s'$  bedeute, dass  $s'$  in einem Schritt aus  $s$  hervorgehen kann (Das kann in Platz  $\log(S)$  entschieden werden).

$T'$  soll nun  $x$  akzeptieren genau dann, wenn kein Pfad (der Länge  $2^S$ ) von  $s_{INI}$  zu  $s_{ACC}$  existiert. (Anzahl der globalen Konfigurationen ist kleiner als  $2^S$ . Ein einziger Pfad braucht, wenn er voll ausgeschrieben wird, schon Platz  $S * 2^S \notin \mathcal{O}(s(|x|))$ .)

### Vorbemerkung:

Nehmen wir an, dass die Anzahl  $N$  der von  $s_{INI}$  aus erreichbaren globalen Konfigurationen bekannt ist bzw. berechnet werden kann.

Wir zählen der Reihe nach alle globalen Konfigurationen auf (geht mit Platz  $\mathcal{O}(S)$ ) und raten für jede von denen einen Pfad von  $S_{INI}$  dorthin. Durch Mitführen eines Zählers haben wir am Ende der Aufzählung die Anzahl derjenigen Knoten, für die das gelungen ist.

```
1 function A(...) {
2   cnt = 0;
3   for (s in globalConfigs) {
4     pfad = guessPath();
5     if (pfad.endsAt(s))
6       cnt++;
7     if (s = s_acc)
8       return "reject";
9   }
10  if (cnt == N)
11    return "accept";
12  else
13    return "reject";
14 }
15
16 function guessPath() {
17   s = s_ini;
18   for (i = 1; i <= 2^S; i++) {
19     sX = guessNonDet({0,1}^S);
20     if (s2 -> sX) {
21       s2 = sX;
22     } else {
23       return "reject";
24     }
25     b = guessNonDet({0,1});
26     if (b) {
27       break;
28     }
29   }
30 }
```

Falls  $N$  die Anzahl der von  $s_{INI}$  aus erreichbaren globalen Konfigurationen ist, so kann A akzeptieren genau dann, wenn  $s_{ACC}$  von  $s_{INI}$  unerreichbar ist.

Begründung " $\Rightarrow$ ": Falls A akzeptiert, dann ist  $s_{ACC}$  tatsächlich unerreichbar, weil alle



$N$  erreichbaren Konfigurationen in der for-Schleife als solche erkannt wurden und  $s_{ACC}$  nicht unter ihnen war.

Begründung " $\Leftarrow$ ": Falls  $s_{ACC}$  unerreichbar ist, so kann A akzeptieren, indem bei jedem der von  $s_{INI}$  aus erreichbaren  $s$  tatsächlich ein entsprechender Pfad geraten wird.

Grobe Struktur des Algorithmus für  $T'$ :

```
1 N = 1;
2 for (i = 1 ... 2^S) {
3     // Invariante: Anzahl der von s_ini aus in weniger als i-1 Schritten
4     // erreichbaren Konfigurationen ist gleich N
5     updateN();
6 }
7 A();
```

Der Block `updateN()` wird selbst Nichtdeterminismus enthalten, in dem Sinne, dass die gesamte Berechnung verwerfend abgebrochen werden kann. Passiert das nicht, dann ist  $N$  korrekt aktualisiert und bei passender Wahl der nichtdeterministischen Entscheidungen, passiert das auch.

```
1 function updateN() {
2     cnt = 0;
3     for (s in globalConfigs) {
4         reachable = false;
5         cnt2 = 0;
6         // alle N Stück die von s_ini aus in weniger als i-1 Schritten erreichbar
7         // sind, aufzählen
8         for (sCheck in globalConfigs) {
9             pfad = guessPath();
10            if (pfad.endsAt(sCheck)) {
11                cnt2++;
12                if (sCheck == s || sCheck -> s)
13                    reachable = true;
14            }
15        }
16        if (cnt2 != N)
17            return "reject";
18        else if (reachable)
19            cnt++;
20    }
21    N = cnt;
22 }
```

Am Ende von `updateN` hat entweder N den korrekten Wert oder es wurde verworfen. Es ist möglich, die nichtdeterministischen Entscheidungen so zu treffen, dass der korrekte Wert geliefert wird, sodass nicht verworfen wird.

*q.e.d.*

## 4.5 Noch ein Satz von Cook

### Definition

$NSPACE(s(n)) + STACK$

Intuitiv ist das alles, was mit  $\mathcal{O}(s(n))$  platzbeschränkten Arbeitsbändern und einem unbeschränkten Stack berechnet werden kann.

Eine mögliche Formalisierung: Turingmaschine mit Stack hat zusätzlich ein Stackalphabet  $\Gamma$  und ein besonderes Symbol  $A \in \Gamma$ . Jedes "Quintupel" enthält eine zusätzliche Komponente, die Stack-Aktion, eine der folgenden drei:

- IDLE (Keller bleibt unverändert)
- POP (oberstes Kellersymbol wird entfernt)
- PUSH ( $x \in \Gamma$  auf den Keller legen)

Außerdem ist das jeweils oberste Kellersymbol lesbar.

Die Maschine wird mit Kellerinhalt  $\boxed{A}$  gestartet und hält per Definition, wenn der Keller leer wird, d.h. wenn  $A$  gePOPt wird.

Formal:

$$\delta = Q \times (\Sigma^k \times \Gamma) \times Q \times \Sigma^k \times S$$

wobei  $S$  die Menge der Stack-Aktionen ist.

## Beispiel

Der Beweis des Satzes von Savitch zeigt, dass  $STCONN \in DSPACE(\log(n) + STACK)$  und  $NSPACE(s(n)) \subseteq DSPACE(s(n) + STACK)$  ist.

## Satz

$$s(n) \geq \log(n) \Rightarrow NSPACE(s(n)) + STACK = DTIME(2^{\mathcal{O}(s(n))})$$

## Beweis

- “ $\subseteq$ ”:

Sei  $T$  eine zunächst deterministische durch  $s(n)$  platzbeschränkte Turingmaschine mit Stack.

Wir fixieren eine Kodierung der “Bandkonfigurationen” von  $T$ . Diese Bandkonfigurationen beinhalten den Zustand und gesamten Inhalt aller Arbeitsbänder und Kopfpositionen. Bei Eingabe der Größe  $n$  haben die so kodierten Bandkonfigurationen die Größe  $\mathcal{O}(s(n))$ , also konkret  $c * s(n)$  für ein festes  $c$ .

Sei nun eine Eingabe fixiert.  $B$  sei die Menge der Bandkonfigurationen der Größe  $c * s(|x|)$  und  $\Gamma$  das Stackalphabet.

Wir definieren die partielle Funktion  $f : B \times \Gamma \rightarrow B : f(b, x) = b'$

$b'$  ist die Bandkonfiguration, die erreicht wird, wenn man  $T$  in  $b$  und mit Kellereinhalt  $\boxed{x}$  startet und so lange rechnet, bis  $X$  zum ersten Mal gePOPt wird.

### **Bemerkung:**

$x \in L(T) \iff f(b_{INI}, A) = b'$  und  $b'$  akzeptierende Endkonfiguration ist.  
o. B. d. A. ist  $b'$  eindeutig.

### **Bemerkung:**

Streng genommen sollte man schreiben  $f_x(b, X)$  oder  $f(x, b, X)$ , da die Eingabe  $x$  in die Definition von  $f$  einfließt.

$f$  gestattet folgende rekursive Definition:

$$f(b, X) = \begin{cases} b' & , \text{POP} \\ f(b', X) & , \text{IDLE} \\ f(f(b', Y), X) & , \text{PUSH}(Y) \end{cases}$$

wobei  $b'$  die auf  $(b, X)$  unmittelbar folgende Bandkonfiguration ist.

Statt nun  $f(b_{INI}, A)$  durch Rewriting auszuwerten, was letztendlich der Berechnung von  $T$  entspräche, tabulieren wir die gesamten  $f$ -Werte systematisch mit dynamischer Programmierung. Die Anzahl der Bandkonfigurationen ist  $2^{c*s(|x|)}$ . Damit gibt es insgesamt  $\mathcal{O}(2^{c*s(|x|)})$ , o. B. d. A.  $2^{c*s(|x|)}$  viele  $f$ -Aufrufe. Eine Wertetabelle für  $f$  kann also in Zeit  $2^{\mathcal{O}(s(|x|))}$  komplett ausgefüllt werden. Gesamtrechnzeit:  $\mathcal{O}((2^{c*s(|x|)})^2) = 2^{\mathcal{O}(s(|x|))}$

Wenn die Maschine nichtdeterministisch ist, dann nehmen wir statt  $f$  die Funktion  $F : B \times \Gamma \times B \rightarrow \text{bool}$  :

$$F = \begin{cases} \text{true} & \text{falls es Berechnung von } b \text{ nach } b' \text{ gibt, an deren Ende } X \text{ erstmals gePOPt wird} \\ \text{false} & \text{sonst} \end{cases}$$

$F$  kann ganz genauso mit dynamischer Programmierung ausgewertet werden.

Sei umgekehrt  $T$  eine deterministische Turingmaschine mit Zeitschranke  $2^{c*s(n)}$ . Die Kopfpositionen aller Köpfe können durch String der Länge  $\mathcal{O}(s(n))$  kodiert werden (Binärkodierung der numerischen Positionen).  $a$  mit  $|a| = c' * s(n)$  kodiere diese Positionen.

Ebenso können Zeitpunkte in der Berechnung durch String dieser Länge kodiert werden.

Wir definieren bei fester Eingabe  $x$  folgende rekursive Funktionen:

- $\text{band}(t, a) \in \Sigma^k$  als Bandinhalt zur Zeit  $t$  an den Positionen  $a$ .

$band(0, a) =$  Inhalt der initialisierten Bänder an den geforderten Positionen

$band(t + 1, a) = \dots zustand(t) \dots band(t, a) \dots kopf(t)$

–  $zustand(t) \in Q$

$zustand(0) = q_0$

–  $kopf(t) =$  Positionen aller Köpfe zur Zeit  $t$

$kopf(t + 1) = \dots kopf(t) \dots band(t, a) \dots zustand(t)$

Die klassische rekursive Auswertung dieser Funktionen beziehungsweise des Toplevel-Aufrufs  $zustand(2^{c \cdot s(|x|)}) \in \{q_{ACC}, q_{REG}\}$  kann auf einer  $DSPACE(s(n)) + STACK$  Maschine simuliert werden.

*q.e.d.*

## 4.6 Logarithmischer Platz

Eine Funktion  $f : \Sigma^* \Rightarrow \Sigma^*$  ist in  $FL$  und heißt **in logarithmischem Platz berechenbar**, wenn es eine deterministische Turingmaschine gibt die bei Eingabe  $x$ :

- Den Funktionswert  $f(x)$  auf ein gesondertes Ausgabeband schreibt.
- Das Ausgabeband nicht mehr ließt.
- Das Eingabeband (wie immer bei Space) nicht mehr beschreibt.
- $\mathcal{O}(\log|x|)$  platzbeschränkte Arbeitsbänder besitzt.

**Äquivalente Charakterisierung:** Maschine schreibt das  $n$ -te Symbol auf ein besonderes Band (oder signalisiert es durch Einnehmen eines bestimmten Zustandes), wenn  $n$  in Binärkodierung auf ein besonderes Band geschrieben wird.

### Bemerkung

- $2^{c \cdot \log(n)} = n^c$
- $f \in FP \Rightarrow |f(x)| = \mathcal{O}(n^c)$  für ein  $c$ , d. h. polynomiell beschränkt und auch  $DTIME_T(x) \in \mathcal{O}(n^c)$  bei passenden  $T$ , welches  $f$  berechnet.

### Bemerkung

$FL \subseteq FP$

### Satz

$FL$  ist unter Komposition abgeschlossen.  $f, g \in FL \Rightarrow g \times f \in FL$

Gegeben Turingmaschinen  $T_f$  und  $T_g$  im zweiten Format ( $n$ -tes Symbol von  $f(x)$  wird bei Eingabe von  $x$  und  $n$  "on demand" berechnet).

Turingmaschine  $T_{g \times f}$  für  $g \times f$  wird ebenfalls im 2. Format konstruiert. Sie simuliert auf der äußeren Ebene die Maschine  $T_g$ . Jede Leseanfrage der Eingabe wird in eine Berechnung von  $T_f$  auf  $x$  an der entsprechenden Ausgabeposition übersetzt.

## 4.7 Mehr Härte und Vollständigkeit

- $X$  ist **NL-hart**, wenn für jedes  $X' \in NL$  gilt  $X' \leq_L X$
- $X$  ist **P-hart**, wenn für jedes  $X' \in P$  gilt  $X' \leq_L X$
- $X$  ist **NL- bzw. P-vollständig**, wenn  $X$  NL-hart bzw. P-hart ist und  $X \in NL$  bzw.  $X \in P$

Hierbei bedeutet  $X' \leq_L X$ , dass  $f \in FL$  existiert, mit  $x \in X' \Leftrightarrow f(x) \in X$

### **Bemerkung:**

$X' \leq_L X \Rightarrow x \leq_P X$ , bzw.  $X \leq X'$

### Satz

Das Problem STCONN (Erreichbarkeit im gerichteten Graphen) ist NL-vollständig.

### Beweis



$STCON \in NL \checkmark$

**NL-hart:** Sei  $T$  eine NL-Maschine.  $x \in ((T)E)$  es existiert ein Pfad von  $a_{INI}$  nach  $a_{ACC}$  im Graphen, dessen Knoten die globale Konfiguration von  $T$  bei Eingabe  $x$  sind,  $a_{INI}$  die Startkonfiguration und  $a_{ACC}$  die (o. B. d. A. eindeutige) akzeptierende Endkonfiguration ist. Die Übersetzung von  $x$  in dem Graphen kann offensichtlich durch eine "FI-Maschine" im 1. Format geschehen.

*q.e.d.*

## 4.8 Horn

### Definition

Eine Klausel (Disjunktion von Literalen) mit höchstens einem positiven Literal heißt **Hornklausel**. Hornklauseln können logisch äquivalent in den Formaten:

- $P_1, P_2, \dots, P_k \rightarrow q$
- d. h.  $\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_k \vee q$
- oder  $P_1, P_2, \dots, P_k \rightarrow \perp$
- d. h.  $\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_k$

geschrieben werden.

### Definition

Gegeben eine Menge von Hornklauseln  $H$ . Das Problem, ob  $H$  unerfüllbar ist, heißt

## Hornproblem.

### Definition

Eine Hornklausel der Form  $\rightarrow q \text{ k} = 0$  heißt **Faktum**. Eine Hornklausel der Form  $\rightarrow \perp \text{ k} = 0$  heißt **Goal**.

In der Regel entfällt  $H$  genau ein "Goal",  $p \rightarrow \perp$ .

$H$  ist dann unerfüllbar, genau dann, wenn  $p$  aus den übrigen Klausen hergeleitet werden kann.  $H \vdash p$  hierbei ist die Herleitbarkeit wie folgt definiert.

- $\rightarrow q \in H$  dann  $H + g$  alle Fakten sind herleitbar
- $P_1, P_2, \dots, P_k \rightarrow q \in H$  und  $H + P_1, H + P_2, \dots, H + P_k$  dann auch  $H + g$

### **Bemerkung:**

Eine Horn-Klausel Menge ohne Goals ist immer erfüllbar.

### **Bemerkung:**

Eine Hornklauselmenge mit mehreren Goals ist unerfüllbar genau dann, wenn mindestens ein Goal herleitbar ist.

### Satz

$Horn \in P$  Wir benutzen eine dynamische Menge  $S$ , die am Ende alle herleitbaren Variablen enthalten soll.  $S := \emptyset$

```
1 while (!done)
2   s' = s
3   for p_1, \dots, p_k \rightarrow q \in H
4     if {p_1, \dots, p_k} \subseteq S' then
5       S' := S' \cup {q}
6   done := S' = S''; S := S'
7 return ''es existiert P \rightarrow \bot \in H mit q \subseteq S''
```

Dieses Vorgehen läuft in polynomieller Zeit, da die Zahl der Durchläufe durch die Zahl der Variablen beschränkt ist, aber nicht offensichtlich in logarithmischem Platz, da die dynamische Menge linearen Platz benötigt.

### Beispiel

- $\rightarrow A$
- $A \rightarrow B$
- $F \rightarrow F$
- $B, C \rightarrow D$
- $A, B \rightarrow C$
- $A, D \rightarrow E$
- $E \rightarrow \perp$

$$S : \emptyset, \{A\}, \{A, B\}, \{A, B, C\}, \{A, B, C, D\}, \{A, B, C, D, E\}$$

### Satz

Das Hornproblem ist P-vollständig.

### Beweis

Genau wie Satz Cook (SAT ist NP-vollständig), im Fall einer deterministischen Maschine entstehen bei der Übersetzung nur Horn-Klauseln.

**Details** Sei eine deterministische Turingmaschine  $T$  gegeben, Eingabe  $x$   $DTIME_T(x) \subseteq p(|x|)$

## Variablen

$zust(t, g)$  Zustand zu Zeit  $t$  ist  $g$

$band(t, i, x)$  Bandinhalt zur Zeit  $t$  an Position  $i$  ist

$kopf(t, i)$  Kopf an Pos  $i$  zur Zeit  $t$

Klauseln z.B.

- $band(t, i, x), kopf(t, i') \rightarrow band(t + 1, i, x)$
- $band(t, i, x), kopf(t, i), zust(t, q) \rightarrow band(t + 1, i, x')$
- $band(t, i, x), kopf(t, i)$
- $Zustand(t, q) \rightarrow zust(t + 1, g') \dots$

falls jeweils  $S(g, x) = (g', x', m)$

Eingabe  $x$  wird akzeptiert genau dann, wenn  $zust(P|x|), q_{ACC}$  herleibar ist um Goal  $zust(P(|x|), q_{ACC} \rightarrow \perp$  hinzunehmen. Die Übersetzung von  $T, x$  in diese Klauselmenge kann mit FL-Maschine durchgeführt werden. Also  $L(T) \leq_L HORN$ .

*q.e.d.*

Das bereits genannte offene Problem  $NL \subsetneq P$  NL echt in  $P$  enthalten? kann also umformuliert werden, in die Frage, ob  $HORN \in NL$ .

### Markieren von Klauseln

Auf ein Fakt darf jederzeit eine Marke gelegt werden. Liegen auf allen Prämissen  $p_1, p_2, \dots, p_k$  einer Klausel  $p_1, p_2, \dots, p_k \rightarrow$  schon Marken, dann darf  $q$  mit einer Marke beehrt werden.

## 4.9 Quantifizierte Boolesche Formeln

### Satz

$QBF$  ist PSPACE-vollständig.

### Quantifizierte Boolesche Formeln ( $QBF$ )

Gegeben ist eine boolesche Formel mit Quantoren, die über boolesche Werte rangieren.

$$\forall x \exists y. x \leftrightarrow \neg y$$

o. B. d. A. kann man diese Formeln als geschlossen voraussetzen (ggf.  $\exists, \forall$  davorschreiben)

Gesucht ist die Antwort auf die Frage, ob diese Formel wahr ist.

Gegeben ist eine boolesche Formel  $\phi$ .

$$\phi \text{ erfüllbar} \iff \exists x_1 \exists x_2 \dots \exists x_n. \phi \in QBF$$

wobei  $\{x_1 \dots x_n\}$  die Variablen von  $\phi$  sind.

$$TAUT \leq QBF$$

$$\phi \text{ Tautologie} \iff \forall x_1 \forall x_2 \dots \forall x_n. \phi \in QBF$$

$$AEQSAT = \{(\phi, \psi) \mid \phi \text{ erfüllbar} \Leftrightarrow \psi \text{ erfüllbar} \}$$

$$(\phi, \psi) \in AEQSAT \iff (\exists x.\phi) \Leftrightarrow (\exists y.\psi) \in QBF$$

$$\exists x_1 \exists y_1 \forall x_2 \forall y_2. (\phi(x_2) \rightarrow \psi(y_1)) \wedge (\psi(y_2) \rightarrow \phi(x_1))$$

### Spezialfall: $QBF_n$

Gegeben ist eine  $QBF$ -Formel in Pränexform (alle Quantoren ganz außen) mit maximal  $n$  Quantorenwechseln beginnend mit  $\exists$ .

Gegeben:  $\exists x_1 \forall x_2 \exists x_3 \dots \exists / \forall x_n. \phi$  wobei  $\phi$  eine boolesche Formel ohne Quantoren ist.

$QBF_n$  ist vollständig für  $\Sigma_n^P$  (polynomielle Hierarchie) falls  $n \geq 1$ .

(Lezteres ist unmittelbare Konsequenz aus der logischen Charakterisierung der PH und Satz von Cook.)

Rekursiver Algorithmus für QBF:

```

1  check(exists x.phi, eta) =
2      check(phi, eta[x -> true]) or
3      check(phi, eta[x -> false])
4  check(forall x.phi, eta) =
5      check(phi, eta[x -> true]) and
6      check(phi, eta[x -> false])
7  check(phi1 and phi2, eta) =
8      check(phi1, eta) and
9      check(phi2, eta)
10 check(x, eta) = eta(x)
11 ...

```

Wir zeigen jetzt  $L \leq QBF$  für beliebiges  $L \in PSPACE$ :

## Beweis

Sei  $T$  eine deterministische Turingmaschine mit  $L(T) = L$  und  $DSPACE_T(x) \leq p(|x|)$  und  $DTIME_T(x) \leq 2^{p(|x|)}$  wobei  $p$  ein Polynom ist. Außerdem habe  $T$  zu jeder Eingabe  $x$  genau eine Startkonfiguration  $a_{INI}$  und eine akzeptierende Endkonfiguration  $a_{ACC}$ . sodass  $x \in L(T) \Leftrightarrow \exists a_1, a_2 \dots a_{2^{p(|x|)}}$  mit  $a_1 = a_{INI}, a_{2^{p(|x|)}} = a_{ACC}$   $a_i \rightarrow_T a_{i+1}$  wobei  $\rightarrow_T$  ein Schritt der Auswertung durch  $T$  ist.

$REACH(a, a', k) := a'$  ist in  $2^k$  Schritten von  $a$  aus erreichbar.

$$x \in L \Leftrightarrow REACH(a_{INI}, a_{ACC}, p(|x|))$$

$$REACH(a_{INI}, a', 0) \Leftrightarrow a \rightarrow_T a'$$

$$REACH(a_{INI}, a', k+1) \Leftrightarrow \exists \check{a}. REACH(a, \check{a}, k) \wedge REACH(\check{a}, a', k)$$

Kodiere Konfiguration  $a$  durch  $p(|x|)$  viele boolesche Variablen.  $a \rightarrow_T a'$  durch quantorenfreie boolesche Formel.

Dies, zusammen mit Abwicklung der Rekursion, liefert eine  $QBF_1$ -Formel (nur Existenzquantoren), also wäre  $PSPACE \subseteq NP$ . Das ist natürlich falsch, denn die so erhaltene  $QBF_1$ -Formel hat die Größe  $\Omega(2^{p(|x|)})$ .

Die Aufblähung findet nicht statt, wenn man folgende Rekurrenz verwendet:

$$REACH(a, a') \Leftrightarrow a \rightarrow_T a'$$

$$REACH(a, a') \Leftrightarrow \exists \check{a} \forall b \forall b'. (b = a \wedge b' = \check{a} \vee b = \check{a} \wedge b' = a') \Rightarrow REACH(b, b', k)$$

Die Größe von  $REACH(a, a', k)$  ist  $\mathcal{O}(k * p(|x|))$  lalalalala  $\mathcal{O}(p(|x|)^2)$

*q.e.d.*

Die Charakterisierung von  $PSPACE$  durch QBF liefert einen direkten Zusammenhang zu Gewinnstrategien in endlichen 2-Personen-Spielen. Die Existenz einer Gewinnstrategie kann in offensichtlicher Weise als QBF-Formeln können selbst als 2-Personen-Spiel verstanden werden.

Man kann ein entsprechendes Maschinenmodell definieren, die **alternierenden Turingmaschinen**.

## Beispiel



Gegeben ein Graph  $G = (V, E)$ .

Knoten entsprechen beliebiger Belegung von booleschen Variablen. Kanten durch boolesche Formeln repräsentiert. Die Nachbarknoten eines Knoten können aufgezählt werden.  $s, t \in V$  vorgegeben.

Es werden abwechselnd Marken auf den Graphen gesetzt. Spieler 1 hat weiße Marken, Spieler 2 hat schwarze Marken. Ein Spieler hat gewonnen, wenn es einen Pfad seiner Farbe von  $s$  nach  $t$  gibt.

Gefragt ist, ob Spieler 1 gewinnen kann. Man zeige, dass dieses Problem PSPACE-vollständig ist.

## 4.10 Noch zwei PSPACE-vollständige Probleme

### Definition

#### GENERALIZED GEOGRAPHY (GGEO)

Gegeben: Gerichteter Graph  $G = (V, E)$  und Startknoten  $s \in V$ .

Folgendes Zwei-Parteien-Spiel wird auf  $G$  gespielt: Die Parteien ziehen abwechselnd eine Marke entlang der Kanten des Graphen beginnend bei  $s$ . Einmal besuchte Knoten dürfen nicht wieder besucht werden. Wer nicht mehr ziehen kann, hat verloren.

Gefragt: Besitzt die anziehende Partei eine Gewinnstrategie?

### Satz

GGEO ist PSPACE-vollständig.

### Beweis

- “ $\in PSPACE$ ”:

Wir betrachten die Funktion

$$WIN(v \in V, B \subseteq V) = \begin{cases} true & , \text{ falls GGEO kann von } v \text{ aus gewonnen werden auf } (V \setminus B, E) \\ false & , \text{ sonst} \end{cases}$$

Es ist  $WIN(v \in V, B \subseteq V) =$

if  $v \in B \vee \forall w. (v, w) \in E \Rightarrow w \in B$   
 then *false*  
 else  $\exists w. (v, w) \in E \wedge w \notin B \wedge \forall w'. (w, w') \in E \wedge w' \notin B \Rightarrow \text{WIN}(w', B \cup \{v, w\})$

WIN kann rekursiv (mit Stack<sup>1</sup> wie bei Savitch) in *PSPACE* ausgewertet werden.

- "PSPACE"-hart:

Wir geben eine Reduktion von QBFSAT auf GGEO.

Sei eine Instanz  $\phi$  von QBFSAT gegeben. O. B. d. A. habe  $\phi$  folgende Form:  
 $\exists x_1 \forall x_2 \exists x_3 \forall \dots \exists x_n. \Phi(x_1 \dots x_n)$  (KNF).

*q.e.d.*

## Definition

### Universalität von NEA (NEA-UNIV)

Gegeben: nichtdeterministischer endlicher Automat  $A = (\Sigma, Q, \delta, q_0, F)$

Gefragt:  $L(A) = \Sigma^* s t$

#### **Bemerkung:**

Zu entscheiden, ob  $L(A) = \emptyset$  ist in  $NLOGSPACE \subseteq P$ , denn es ist äquivalent zu der Frage, ob  $E$  von  $q_0$  aus erreichbar ist im Graphen  $(Q, K)$  wobei  $K = \{(q, q') \mid \exists a \in \Sigma : (q, a, q') \in \delta\}$

Sei  $B$  der zu  $A$  äquivalente deterministische endliche Automat<sup>2</sup> und  $\bar{B}$  dessen Kom-

<sup>1</sup>Tiefe beschränkt durch Anzahl der Knoten

<sup>2</sup>Das kennen wir ja, mit der Teilmengenkonstruktion...

plement. Es ist  $L(A) = \Sigma^* \iff L(\bar{B}) = \emptyset$ .

Wenn man den "Potenzautomaten"  $B$  beziehungsweise sein Komplement  $\bar{B}$  nicht explizit, sondern "on demand" aufbaut, kann die Suche in  $\bar{B}$  in  $PSPACE$  erfolgen.

### Satz

NEA-UNIV ist  $PSPACE$ -vollständig.

### Beweis

- " $\in PSPACE$ ":

Bereits gezeigt.

- " $PSPACE$ "-hart:

Sei eine normalisierte (im üblichen Sinne) deterministische Turingmaschine  $T$  mit polynomiellen Platzverbrauch und eine Eingabe  $x$  vorgelegt.

Alle globalen Konfigurationen durch Strings der Länge  $N = p(|x|)$  kodiert in vernünftiger Weise. Genau eine akzeptierende Endkonfiguration  $q_{ACC}$ .

Berechnungen von  $T$  auf  $x$  lassen sich also durch Wörter der Form  $w = a_0 a_1 a_2 \dots a_n$  repräsentieren, wobei  $a_0 = a_{INI}$  (Startkonfiguration zu Eingabe  $x$ ).

$a_i \rightarrow a_{i+1}$  für  $i = 0 \dots n-1$   
insbesondere  $|a_i| = N$

$a_n$  Endkonfiguration ( $q_{ACC}$  oder nicht).

$x \in L(T) \iff \exists w. w$  hat diese Form und  $a_n = q_{ACC}$

d. h.  $x \in L(T) \iff \forall w. w$  ist keine Berechnung beginnend bei  $x \vee w$  hört mit  $q_{ACC}$  auf.

Die Eigenschaften

$U_x \iff w$  ist keine Berechnung beginnend bei  $x$

$V_x \iff w$  hört nicht mit  $x$  auf.

können durch nichtdeterministische endliche Automaten geprüft werden, die aus  $x$  in polynomieller Zeit (sogar FL) berechnet werden können.

Skizze der Konstruktion dieser Automaten:

– Für  $U_x$ :

Prüfe, dass die ersten  $N$  Symbole der Eingabe der Konfiguration  $a_{INI}$  zu  $x$  entsprechen. Dies kann mit  $N$  Zuständen fest verdrahtet werden.

Falls nicht, so wird sofort akzeptiert.

Ansonsten muss der Automat für  $U_x$  nichtdeterministisch einen Fehler in  $w$  raten.

\* Letzte  $N$  Symbole sind keine Haltkonfiguration.

\* Eingabe hat die Form  $uw_1uw_2vw_3$  wobei  $|uw_2| = N$  und  $|u| = |v| = 7$  und  $u, v$  entsprechen einander in aufeinanderfolgenden Konfigurationen.  $v$  ist nicht entsprechender Teil der Folgekonfigurationen.

Nach Konstruktion ist also ein Automat  $A_x$  entstanden mit  $L(A_x) = U_x v V_x$  also  $L(A_x) = \Sigma^* \iff x \in L(T)$

– Für  $V_x$ :

So ähnlich.

*q.e.d.*

## 4.11 Jumping Automata on Graphs

### Definition

Gegeben ein Alphabet  $\Sigma$ .

Eine **Zeigerstruktur** über  $\Sigma$  ist ein Paar  $(V, E)$  wobei

- $V$  eine endliche Menge von Knoten und
- $E \subseteq V \times \Sigma \rightarrow V$   
Notation:  $v.l = E(v, l)$  für  $v \in V$  und  $l \in \Sigma$

Vorstellung: Knoten sind Objekte. Alphabetsymbole sind Feldnamen/Pointer.

### Beispiel

Ein regulärer ungerichteter Graph vom Grad  $d$ :

- Jeder Knoten hat genau  $d$  Nachbarn ("regulär").
- Die Nachbarn eines Knotens sind linear angeordnet (nummeriert von 1 bis  $d$ ).

Als Zeigerstruktur:  $\Sigma\{1 \dots d\}.E(v, i) = i\text{-ter Nachbar von } v$ .

### Beispiel

Gerichtete reguläre Graphen vom Grad  $d$  kann man ebenso als Zeigerstruktur über  $\Sigma = \{1 \dots d\}$ , oder besser, falls Kanten auch in umgekehrter Richtung navigiert werden können, über  $\Sigma = \{1 \dots d\} \cup \{1' \dots d'\}$  darstellen.

### Beispiel

Nicht-reguläre Graphen (Knoten haben beliebig viele Nachbarn) können durch Adjazenzlisten repräsentiert werden, z. B. zirkulär  $\Sigma = \{head, next, elem\}$ .

Dabei hat ein Knoten ein Pointer mehr als Nachbarn: Knoten  $a$  zeigt mit *head* auf  $a_1$ ,  $a_1$  mit *next* auf  $a_2$ ,  $a_2$  mit *next* auf  $a_1$ .  $a_1$  zeigt mit *elem* auf  $b$  und  $a_2$  mit *elem* auf  $b$  usw.

### Beispiel

Sogar Graphen ohne Ordnung auf den Nachbarn können so repräsentiert werden.

$$(V, E), E \subseteq V \times V$$

$$\rightsquigarrow V \times E, \mathcal{E}$$

$$\Sigma = \{s, t\}$$

$$\mathcal{E} : (v, v').s = v$$

$$(v, u').t = v'$$

$$v.s = v'$$

$$v.t = v$$

### Beispiel

Gegeben: Endliche Gruppe  $G$  mit Erzeugern  $g_1 \dots g_d$

$\rightsquigarrow$  Zeigerstruktur  $\Sigma = \{g_1 \dots g_d, g_1^{-1}, \dots, g_d^{-1}, V = G, v.l = v * l \text{ wobei}$

\* die Gruppenmultiplikation ist.

### Definition

Ein **Jumping Automata on Graphs** (JAG) ist ein Quintupel  $(\Sigma, Q, q_0, \delta, P)$  wobei

- $\Sigma$  Alphabet
- $Q$  Zustände
- $q_0 \in Q$  Startzustand
- $P$  Pebbles
- $\delta$  Menge von Quadrupeln der Form  $(q, S, q', m)$  mit  $q, q' \in Q$  und  $S$  Äquivalenzrelation auf  $P$ ,  $m : P \rightarrow P + \Sigma$ :  
Falls  $m(p) = p' \in P$  so muss  $p$  auf die Position von  $p'$  gesetzt werden.  
Falls  $m(p) = l$  so muss  $p$  auf  $v.l$  gesetzt werden, wenn  $p$  auf  $v$  liegt.

Eine Konfiguration eines JAG auf einer Zeigerstruktur  $(V, E)$  über  $\Sigma$  ist ein Paar  $C = (q, M)$  wobei  $q \in Q, M : P \rightarrow V$ . Die Ein-Schritt-Relation  $C \Rightarrow_A C' \iff C = (q, M), C' = (q', M'), \exists (q, S, q', m) \in \delta. (p, p') \in S \iff M(p) = M(p')$  Falls  $m(p) = p' \in P$  dann  $M'(p) = M(p')$   
Falls  $m(p) = l$  dann  $M'(p) = M(p).l$

$A$  deterministisch, falls zu jedem  $(q, S)$  genau ein  $(g', m)$  mit  $(q, S, q', m) \in \delta$  existiert.  
In diesem Fall gibt es zu jedem  $C = C_0$  eine eindeutige Folge  $C_0 \rightarrow C_1 \rightarrow C_2 \dots$

### Beispiel

$\Sigma = \{S, N, O, W\}$  Himmelsrichtungen



Automat soll Pebble  $b$  entlang der ersten “freien” Kante bewegen. (ver-  
sperrt  $v.X = v, X \in \{S, N, O, W\}$ )

Anfang bei Pebble  $s$ , Stehenbleiben bei Pebble  $t$ .  
 $s$  und  $t$  werden nicht bewegt.

### Satz

JAG-Berechnungen können durch *LOGSPACE*-Turingmaschinen simuliert werden.

### Definition

Ein **deterministischer JAG** besucht einen Knoten  $v$  von Knoten  $s$  aus, wenn in der Berechnungsfolge beginnend bei  $C_0 = (q_0, M_0)$ ,  $M_0(p) = s$  der Knoten  $v$  irgendwann belegt ist, d. h. existiert  $i$  sodass  $C_i = (q_i, M_i)$ ,  $M_i(p) = u$  für ein  $p \in P$

### Satz

#### **Satz von Cook Rackoff**

Sei  $A = (\Sigma, Q, q_0, \delta, P)$  (deterministischer JAG) und  $V = (\mathcal{Z}_m)^d$

$$\Sigma = \{e_1 \dots e_d\} \cup \{e_1^{-1}, e_d^{-1}\}$$

$$e_i = (0 \dots 0, 1, 0 \dots 0) \text{ (} d \text{ Stück, die 1 an Position } i \text{)}$$

$$e_i^{-1} = (0 \dots 0, -1, 0 \dots 0) \text{ (} d \text{ Stück, die 1 an Position } i \text{)}$$

Ausgehend von beliebigem  $s \in \mathcal{Z}_m^d$  kann  $A$  höchstens  $(m \cdot |Q|)^{c|P|}$  viele Knoten besuchen, wobei  $c > 0$  eine Konstante (unabhängig von  $A, m, d$ ) ist, durch die man das einfacher abschätzen kann.

$$\text{Falls } m \geq |Q|^{c^P}$$

$$(m \cdot |Q|)^{c|P|} \leq m^{c|P|+1}$$

Wenn zusätzlich  $d > c^{|P|} + 1$

$$(m \cdot |Q|)^{c^{|P|}} < m^d = |\mathcal{Z}_m^d|$$

Kein fester JAG kann alle Knoten in  $\mathcal{Z}_m^d$  besuchen.

Besonderheit:  $d$  ist in dem Automaten fest eingebaut. Man kann aber durch Verwendung von Adjazenzlisten auf feste Alphabetgröße reduzieren.

### Lemma

Sei  $\alpha$  eine Folge von Moves  $\alpha = m_1, m_2 \dots m_t$  mit  $m_i : P \rightarrow P + \Sigma$

Sei  $M : P \rightarrow (\mathcal{Z}_m)^d$  eine Belegung.

Wir schreiben  $M' = \alpha(M)$  für die Belegung, die aus  $M$  durch Abarbeiten von  $\alpha$  entsteht.

Falls  $(M_i)_i$  eine Folge von Belegungen  $M_0 \rightarrow_\alpha M_1 \rightarrow_\alpha M_2 \rightarrow_\alpha \dots \rightarrow_\alpha M_{i+1} = \alpha(M_i)$  dann existiert  $k \leq |P|, n \leq |P|! * m$  sodass  $M_{k+ni} = M_k$  für alle  $i \geq 0$ .

Es wiederholt sich also irgendwann nur noch oder so.

Grund: Falls  $z \in (\mathcal{Z}_m)^d$ , dann ist  $z^n = zz \dots z = 1$ .

Falls  $\alpha$  keine Pebble-Sprünge enthält, kann man deshalb  $k = 0, n = m$  wählen.

---

Vorlesung vom 07.01.16

---

# 5 Probabilistische Algorithmen

## 5.1 Miller-Rabin Primzahltest Einführung

Gegeben:  $n \in \mathbb{N}$  binär kodiert,  $\text{Länge}(n) = \log(n)$

Gefragt: Ist  $n$  prim?

Man müsste testen bis  $\sqrt{n}$  Anzahl der Zahlen  $< \sqrt{n}$ :  $\sqrt{2^{\log(n)}} = 2^{\frac{1}{2}\log(n)}$ , d. h. exponentiell in  $\text{Länge}(n)$ . Das ist nicht effizient, insbesondere  $\notin P$ .

### Lemma

Sei  $G$  abelsche Gruppe (endlich) und  $U$  eine Untergruppe von  $G$ .

Dann ist  $|U|$  ein Teiler von  $|G|$ .

### Korollar

Falls  $\text{ggT}(x, n) = 1$ , d. h.  $x \in \Phi(n)$ , dann  $x^{\phi(n)} = 1 \pmod n$ , wobei  $\phi(n) = |\Phi(n)|$  die Anzahl der zu  $n$  teilerfremden Zahlen kleiner  $n$  ist.

### Korollar

Falls  $n$  prim ist, so gilt  $x^{n-1} = 1 \pmod n$

### Lemma

Falls  $x \notin \Phi(n)$ , so ist  $x^{n-1} \neq 1 \pmod n$ .

### Beweis

Sei  $b = \text{ggT}(x, n) > 1$ ,  $v = x^{n-2} \pmod n$ .

$x^{n-1} = v * x + u * n$  für ein  $u \in \mathbb{Z}$

$b | RHS \Rightarrow x^{n-1} \neq 1$

*q.e.d.*

## 5.2 Fermat-Primzahltest

Gegeben:  $n \in \mathbb{N}$

Wähle zufällig  $x \in \{1 \dots n-1\}$ .

Bilde  $a := x^{n-1} \pmod n$  mittels "Repeated Squaring".

Falls  $a = 1$ , so antworte "ist prim".

Falls  $a \neq 1$ , so antworte "ist zusammengesetzt".

Ist  $n$  tatsächlich prim, so wird mit Sicherheit geantwortet "ist prim" (siehe Korollar).

Ist  $n$  zusammengesetzt und  $x \notin \Phi(n)$ , dann wird korrekt geantwortet "ist zusammengesetzt".

Wenn  $x \in \Phi(n)$ , so kann trotzdem  $a = 1$  sein, also korrekt geantwortet werden.

Falls  $U \neq \Phi(n)$ , dann ist  $|U|$  ein echter Teiler von  $\phi(n)$ , also  $\leq \frac{1}{2}n$ . D. h. für mehr als 50% der  $x$  ist dann  $a \neq 1$ .

Falls aber für alle  $x \in \Phi(n)$  gilt  $x^{n-1} = 1 \pmod n$  und doch  $n$  zusammengesetzt ist, dann funktioniert der Fermat-Test nicht gut. Solche  $n$  heißen Carmichaelzahl, falls  $n$  nicht prim, aber  $x^{n-1} = 1 \pmod n \forall x \in \Phi(n)$ . Solche Carmichaelzahlen sind sehr selten.

### Satz

Ist  $n$  Carmichaelzahl, so ist der Anteil der  $x \in \{1 \dots n-1\}$ , sodass  $\exists i : \gcd(x^{\frac{n-1}{2^i}} - 1, n) > 1$  größer oder gleich 75%.

## 5.3 Miller-Rabin Primzahltest

Gegeben  $n \in \mathbb{N}$

Wähle zufällig  $x \in \{1 \dots n-1\}$ .

Bilde  $a = x^{n-1} \pmod n$ .

Falls  $a \neq 1$ , so ist  $n$  sicher zusammengesetzt.

Falls  $a = 1$ , berechne  $\gcd(x^{\frac{n-1}{2^i}} - 1, n)$  für  $i = 1 \dots \log(n)$ . Falls einmal  $> 1$ , so ist  $n$  sicher zusammengesetzt. Sonst antworte "ist prim".

Fehlerwahrscheinlichkeit  $\leq 50\%$ .

## 5.4 Testen von polynomiellen Identitäten

Gegeben: zwei Polynome  $p(x), q(x)$  mit Koeffizienten aus  $\mathbb{Z}$  und  $n \nmid x = x_1 \dots x_n$ .

Gefragt: Ist  $p(x) = q(x)$ ?

### Beispiel

$$p = (a^2 + b^2 + c^2 + d^2)(A^2 + B^2 + C^2 + D^2)$$

$$q = (aA + bB + cC + dD)^2 + \\ (aB + bA + cD + dC)^2 + \\ (aC + bD + cA + dB)^2 + \\ (aD + bC + cB + dA)^2$$

Hier ist in der Tat  $p = q$ .

Für diese Aufgabe ist kein Polynomialzeitverfahren bekannt. Ausmultiplizieren hat im Allgemeinen exponentielle Laufzeit, falls  $*$  und  $+$  geschachtelt auftreten.

### **Probabilistisches Verfahren:**

Wähle zufällig  $x \in \{-nd, \dots, 0, 1, 2, \dots, nd\}^n$ , wobei  $n = \text{Anzahl Vbl}$  und  $d \geq \text{Grad}(p), \text{Grad}(q)$ .

Wir zeigen jetzt, dass die Fehlerwahrscheinlichkeit kleiner oder gleich 50% ist.

### Lemma

#### **Lemma von Schwarz-Zippe**

Sei  $p$  ein Polynom vom Grad  $d$  in  $n$  Variablen und  $p \neq q$ .

Dann hat  $p$  im Bereich  $\{-N, \dots, N\}^n$  höchstens  $dn(2N + 1)^{n-1}$  Nullstellen.

Falls  $N = nd$ , dann ist  $\frac{(2N+1)^{n-1} * dn}{(2N+1)^n} = \frac{dn}{2N+1} = \frac{dn}{2dn+1} \leq 50\%$ .



### Definition

Ein **Perfektes Matching** zu Graph  $G = (V, E)$  ist eine Kantenauswahl  $T \subseteq E$ , sodass  $u, v \in T, u \neq v \rightarrow u \cap v = \emptyset$ . (Die ausgewählten Kanten haben keine Punkte gemeinsam.)

Ein Graph ist perfekt, wenn  $\bigcup T = V$ . (Alle Knoten werden gematcht.)

### Satz

Sei  $G = (V, E)$  ein Graph und  $A$  die zugehörige Adjazenzmatrix.  $G$  hat ein perfektes Matching genau dann, wenn  $\det(A) \neq 0$ .

### Definition

Eine **Probabilistische Turingmaschine** ist wie eine nichtdeterministische Turingmaschine aufgebaut. Zu jedem Paar  $q, a$  gibt es exakt zwei Quintupel (ggf. künstlich aufblähen).

Deutung: Passendes Quintupel wird jeweils mit Wahrscheinlichkeit 50% ausgewählt.

Die Wahrscheinlichkeit, dass  $T$  die Eingabe  $x$  akzeptiert, ist  $\frac{\text{Anzahl akzeptierender Berechnungen}}{\text{Anzahl aller Berechnungen}}$ .

Polynomialzeitbeschränkt, falls Laufzeit  $\leq p(|x|)$  unabhängig von den Zufallsentscheidungen.

### Definition

Eine Sprache  $X \subseteq \Sigma^*$  ist in der Komplexitätsklasse  $PP$ , wenn eine Probabilistische Turingmaschine  $T$  existiert, die

- polynomiell zeitbeschränkt ist,
- $x \in X \Leftrightarrow \text{Akzeptanzwahrscheinlichkeit} \geq 50\%$ .

### Satz

$$P \subseteq PSPACE$$

### Beweis

Alle Berechnungen durchprobieren (Platz jeweils wiederverwenden) und Anzahl der akzeptierenden Berechnungen mitzählen.

*q.e.d.*

### Satz

$$NP \subseteq PSPACE$$

### Beweis

Sei eine nichtdeterministische Turingmaschine  $T$  für  $X$  gegeben (O.B.d.A. im 2. Nachfolgequintupelformat). Zunächst mit 50% Wahrscheinlichkeit einfach so akzeptieren, anschließend  $T$  als Probabilistische Turingmaschine fahren.

$x \in X$ : Wahrscheinlichkeit der Akzeptanz  $\geq 50\%$

$x \notin X$ : Wahrscheinlichkeit der Akzeptanz  $= 50\%$

*q.e.d.*

### Satz

(Beispiel Spielman et al.)

$PP$  ist unter  $\cap, \cup$  abgeschlossen. Der Beweis dazu ist kompliziert.

### Definition

Komplexitätsklassen  $R, RP$

$X \in R$  genau dann, wenn eine polynomiell zeitbeschränkte Probabilistische Turingmaschine  $T$  existiert mit

- $x \in X \Rightarrow$  Wahrscheinlichkeit der Akzeptanz  $\geq 50\%$
- $x \notin X \Rightarrow$  Wahrscheinlichkeit der Akzeptanz  $= 0$

### Beispiel

$PRIM \in R$

### Definition

Komplexitätsklasse  $BPP$

$X \in BPP$  genau dann, wenn eine polynomiell zeitbeschränkte Probabilistische Turingmaschine  $T$  existiert mit

- $x \in X$ : Wahrscheinlichkeit der Akzeptanz  $> 75\%$
- $x \notin X$ : Wahrscheinlichkeit der Akzeptanz  $\leq 25\%$

### Satz

Sei  $A \in BPP$  und  $q(n)$  ein Polynom.

Es existiert eine Probabilistische Turingmaschine  $T$  für  $A$  mit Fehlerwahrscheinlichkeit

$\leq e^{-q(|x|)}$  für Eingabe  $x$ .

### Beweis

Beweisidee:  $A$  auf Eingabe  $x$  wiederholt, d. h.  $t = t(|x|)$  mal ablaufen lassen. Am Ende diejenige Antwort, die häufiger gegeben wurde, nach außen reichen.

$x \in A$  Anzahl der Akzeptanzen binomialverteilt mit Parameter  $p \geq 75\%$ . Sei  $S$  diese Anzahl (Zufallsvariable). Wir möchten  $\Pr(S \leq \frac{t}{2}) \leq e^{-q(|x|)}$ .

Es gilt allgemein die Chernoff-Schranke. (*siehe Details im Buch*)

*q.e.d.*

## 5.5 Chernoff Schranke

Seien  $X_1 \dots X_n$  0 – 1 Zufallsvariablen mit  $Pr(X_i = 1) = P_i$

Setze  $S = X_1 + \dots + X_n$

$\mu = E[S] = p_1 \dots + p_n$

Für jedes  $S > 0$

$Pr(S > (1 + \delta)\mu) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\delta$  Falls zusätzlich  $\delta < 1$

$Pr(S \leq (1 - \delta)\mu) \leq e^{-\frac{\delta^2}{2} \mu}$

In der Literatur gibt es andere Versionen.

**Anmerkung** t-fache unabhängige Ausführung eines BRP-Algorithmus für  $A \subseteq \Sigma^*$   
bei Eingabe  $x$ .  $x \in A$ :  $\mu = 0,75 * t$

Fehlerwahrscheinlichkeit:  $Pr(S \leq \frac{t}{2}) = Pr(S \leq (1 - \delta)\mu)$  Für  $\delta = \frac{2}{3}$

t ist also so zu wählen, dass

$$e^{-\frac{2^2}{3} * \frac{3}{4} * t} \leq e^{-q(|x|)}$$

Also  $q(|x|) \geq \frac{2^2}{3} * \frac{3}{4} * t$

$$q(|x|) \geq 24t$$

$$t \notin A : \mu = \frac{1}{4} * t$$

$$PR(S \geq \frac{t}{s} = PR(S'(1+1)\mu) \geq \frac{e}{4}^{\frac{t}{4}} = (e^{-0.386})^{\frac{t}{4}}$$

Es muss also  $0.386 * \frac{t}{4} \leq q(|x|)$  Wahr, wenn  $t(|x|) = 24 * g(|x|)$

### 5.5.1 Anwendung

Satz:  $BPP \subseteq \frac{P}{Poly} \frac{P}{Poly} = \{A | \forall n \exists \text{ Schaltkreis } C. |C| \leq p(n) \forall x. |x| < n : x \in A \Leftrightarrow C(x) = 1\}$

$$SAT \in \frac{P}{Poly} \Rightarrow PH \text{ kollabiert}$$

#### Beweis

Sei ein BPP Algorithmus M für A gegeben. OBdA(vorhergehender Satz) können wir annehmen, dass

$$x \in A \Rightarrow Pr(Makzeptiert.x) \geq 1 - 2^{-(|x|+1)}$$

$$x \notin A \Rightarrow Pr(Makzeptiert.x) \leq 1 - 2^{-(|x|+1)}$$

Wir nehmen außerdem an, dass  $|\Sigma| = 2$  (sonst ersetze 2 durch  $|\Sigma|$ ).

Anteil der Zufallsentscheidungen, der für ein bestimmtes  $x \in \Sigma^*$  ungünstig ist, ist  $\leq 2^{-(|x|+1)}$ , d.h.  $\leq 2^{-(n+1)}$  falls  $|x| = n$

Der Anteil der Zufallsentscheidungen, der für mindestens ein  $x \in \Sigma^*$  ungünstig ist, also  $\leq 2^n * 2^{-(n+1)} \leq \frac{1}{2}$

Der Anteil der Zufallsentscheidungen die für jedes  $x$  mit  $|x| = n$  günstig sind, ist  $> 0$ , ja sogar  $> \frac{1}{2}$ . Der Schaltkreis  $C$ , der für alle Inputs der Größe  $n$  funktioniert ergibt sich aus  $M$  in dem eine dieser universell günstigen Entscheidungen festverdrahtet wird.

*q.e.d.*

BPP, RP, co-RP ✓

ZPP : Antwort immer richtig mit  $W \leq 50$  darf "dont know" geantwortet werden. Laufzeit in jedem Fall polynomiell.

Offensichtlich  $ZPP = RP \cap co - RP$

Las Vegas Algorithmus (verallgemeinert ZPP) Liefert nur richtige Ergebnisse aber manchmal ( $W < 50\%$ ) kein Ergebnis ("dont know")

Monte Carlo Algorithmus Liefert Ergebnis, dass mit  $W \leq 25\%$  falsch ist (verallgemeinert BPP)

### Beispiel

für Las Vegas

Randomisiertes Quicksort (Laufzeitschranke  $n \log n$  mit laufen lassen "dont know" falls nicht richtig)

Allg. ist ein Las Vegas Alg. ersetzbar durch einen, der immer richtig antwortet, aber mit einer gewissen Wahrscheinlichkeit nicht terminiert. Letzteres mit Wahrscheinlichkeit 0.

(Wiederholung) Folgende formale Version:  $ZPP = \{A \mid \text{es existiert ein Entscheidungsverfahren, für } A \text{ dessen Laufzeit eine Zufallsvariable ist mit polynomiellern Erwartungswert } \}$ . beschränkt.

## 5.6 Interaktive Beweissysteme

- Polynomiell laufzeitbeschränkter Verifizierer ("wir"):  $V$
- Mit unbeschränkter Rechenkraft ausgestatteter Beweiser ("sie"):  $P$

$P$  und  $V$  haben spezielle Zustände zur Beendigung von Phasen.

Bei gegebener Eingabe  $x$  auf dem Eingabeband finden eine bestimmte Zahl von Runden statt.  $Zahl \leq poly(|x|)$  Zu Beginn jeder Runde schreitet  $V$  ein Wort aufs Frageband. Anschließend schreibt  $P$  eine Antwort aufs Antwortband.

Zum Schluss (alle Runden vorbei) entscheidet ob akzeptiert oder nicht.

### Definition

Sei  $X \subseteq \Sigma^*$ . Eine polynomiell beschränkte Turingmaschine  $V$  ist polynomiell verifizierbar für  $x$  genau dann wenn  $\exists$  Prover  $P \forall x \in A . V \leftrightarrow P$  auf  $x$  akzeptiert.  $\forall x \notin A . V \nleftrightarrow P$  akzeptiert auf  $x$  nicht.

Falls Verifizierer deterministische polynomiell beschränkte Turingmaschine sind, so spricht man von der Klasse DIP (falls Problem für die ein deterministischer polynomiellzeit Verifizierer existiert.)

### Satz

$DIP = NP$

" $\subseteq$ " Verwendet Formulierung von NP mit Polynom großen "Lösungen".

" $\supseteq$ " Sei  $V$  ein DIP Verifizierer für  $X$  und  $P$  der zugehörigem Prover. Komplettes Transcript raten bzw. als Lösung verwenden.



## Definition

$\subseteq \Sigma^*$  ist in  $TP$  falls eine probabilistischer polynomialzeit beschränkter Verifizierer  $V$  existiert so, dass  $\exists$  Prover  $P$

$x \in A : Pr(V \leftrightarrow P \text{ auf } x \text{ akzeptiert}) \geq 75\%$ .

$x \notin A : \forall \text{ Prover } P.$

$P_1(V \leftrightarrow P \text{ auf } x \text{ akzeptiert}) \leq 25\%$ .

Evidenz für  $IP \stackrel{\supseteq}{\neq} NP$ .

Sei  $G$  das Problem Graph-Isomorphismus. Offensichtlich  $GI \in NP$  Isomorphismus vorliegen.

Man weiß nicht ob  $\overline{GI} \in NP$ .

Aber:  $\overline{GI} \in IP$ .

$V$  :

- Gegeben:  $G, H$  Graphen
- Wähle zufällig (1:1)  $b \in \{0, 1\}$
- if  $b$  then  $K := G$  else  $K := H$
- Wähle zufällige Permutation  $\pi$  auf den Knoten von  $K$  und stelle an  $P$  die Fragen  $\pi(K)$   $P$  antwortet mit  $b' \in \{0, 1\}$  Akzeptiere, falls  $b = b'$

Wiederhole dies 2x (akzeptiert "wirklich" nur falls beide Male akzeptiert wurde).

Dies ist korrektes Protokoll für  $\overline{GI}$ .

## Definition

### Interaktives Beweissysteme

$L \in IP$  genau dann, wenn

$\exists$  polynomialzeitbeschränkte, probabilistische Turingmaschine Verifier  $V$  :

$\exists$  Prover  $P$  :

$x \in L \Rightarrow (P, V)$  akzeptiert auf  $x$  mit Wahrscheinlichkeit  $\geq 75\%$

$x \notin L \Rightarrow \forall P : (P, V)$  verwirft auf  $x$  mit Wahrscheinlichkeit  $\geq 75\%$

Der Prover ist eine deterministische Turingmaschine ohne Ressourcenbeschränkung. In jeder Runde schickt der Verifizierer eine Frage an den Prover und der Prover eine Antwort an den Verifizierer. Die Anzahl der Runden ist polynomiell beschränkt. Ausschließlich  $V$  entscheidet, ob akzeptiert wird.

### 5.6.1 Graphisomorphismus

Verifizierer für das Problem, ob zwei Graphen nicht isomorph sind,  $\overline{GI}$ :

Gegeben:  $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$

Falls  $|V_1| \neq |V_2|$ , dann verwerfen.

Wähle zufällig  $i \in \{1, 2\}$  und Permutation  $\phi$  auf  $V_1$ . Übermittle  $G_3 = \phi(G_i)$  an den Prover.

Prover antwortet mit einer Zahl  $j \in \{1, 2\}$ . Falls  $i = j$ , dann akzeptieren, sonst verwerfen.

Wenn  $G_1 \not\cong G_2$ , so kann der intendierte Prover so antworten, dass  $V$  mit Sicherheit akzeptiert.

Wenn  $G_1 \cong G_2$ : Falls  $G_3$  übermittelt wird und  $P$  richtig rät, so ist das in 50% der Fälle die falsche Antwort, da ja  $G_3 = \phi(G_1) = \phi(\psi^{-1}(G_2))$  wenn  $\phi_1 : G_1 \rightarrow G_2$  Isomorphismus

$\Rightarrow$  Akzeptanzwahrscheinlichkeit  $\leq 50\%$ .

## 5.6.2 Unerfüllbarkeit von 3KNF

Ziel:  $co - NP \subseteq IP$

Konstruktion eines IP-Protokolls für Unerfüllbarkeit von 3KNF.

Vorgelegt sei eine 3KNF  $\phi$ .

Gefragt ist  $\phi(\eta) = 0$  für alle Belegungen  $\eta$ .

$\eta : Vars(\phi) \rightarrow \{0, 1\}$

Lösung ist die Arithmetisierung. Man ordnet jeder KNF  $\phi$  ein Polynom  $[\phi]$  in denselben Variablen zu.

- $[x] = x$
- $[\phi \wedge \psi] = [\phi] * [\psi]$
- $[\phi \vee \psi] = 1 - (1 - [\phi]) * (1 - [\psi])$
- $[\neg \phi] = 1 - [\phi]$

Falls  $\eta : \text{Vars} \rightarrow \{0, 1\}$

$$\phi[\eta] = [\phi](\eta)$$

Durch Induktion über Aufbau von  $\phi$ .

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

$$[\phi] = (1 - (1 - x)(1 - (1 - y))) * z$$

Wenn  $\phi$  eine 3KNF mit  $n$  Variablen und  $m$  Klauseln ist, dann ist  $[\phi]$  ein Polynom in  $n$  Variablen (denselben) und vom Grad  $3m$ .

Wir erweitern die Syntax von Polynomen um den Operator  $SUM_x$  wobei  $SUM_x(p) = p(x = 0) + p(x = 1)$ .

Jetzt ist  $\phi$  unterfüllbar (mit Variablen  $x_1 \dots x_n$ ) genau dann, wenn  $SUM_{x_1}(SUM_{x_2}(SUM_{x_3}(\dots SUM_{x_n}([\phi]))))$ .

Wir entwerfen ein IP-Protokoll für das Problem  $p(a_1 \dots a_n) = b?$ , wobei  $p$  verallgemeinertes Polynom ist.

Alle Zwischenergebnisse bei der Auswertung eines erweiterten Polynoms sind durch  $2^m$  beschränkt (und größer oder gleich 0).

Wir können daher alle Berechnungen in  $\mathbb{Z}/p\mathbb{Z}(\mathbb{Z}_p)$ , ( $\mod p$ ) durchführen (statt in  $\mathbb{R}$ ), sofern  $p$  eine Primzahl  $\geq 2^m$  ist.

Ganz zu Beginn einigen sich  $P$  und  $V$  also auf solch eine Primzahl  $p$ . Anschließend finden alle Berechnungen  $\mod p$  statt. Das Protokoll läuft über so viele Runden, wie das verallgemeinerte Polynom  $p$  groß ist.

Falls  $p = x_i$ : Verifizierer prüft, ob  $a_i = b$ . Wenn ja, akzeptiere sonst verwerfen.

Falls  $p = p_1 * p_2$ : Verifizierer fragt Prover nach Zwischenergebnissen  $b_1, b_2$  und prüft, ob  $b_1 * b_2 = b$  und in folgenden Runden wird dann bestätigt, dass  $p_1(a) = b_1, p_2(a) = b_2$ .

Fülle  $p = p_1 + p_2$ ,  $p = P_1 - P_2$

$p = 1$  analog.

Fall  $p = \text{SUM}_x p_1(x_1 \dots x_n, x)$

Gefragt ist  $p_1(a_1 \dots a_n, 0) + p_1(a_1 \dots a_n, 1) = b$ .

Verifizierer bittet Prover um die Koeffizienten (in  $\mathbb{Z}_p$ ) es Polynoms (vollständig ausmultipliziert.)

$q(x) = p_1(a_1 \dots a_n, x)$  Polynom in 1 Variablen vom Grad  $3m$ .

Verifizierer prüft, ob  $q(0) + q(1) = b$ .

Verifizierer wählt zufällig  $a \in \mathbb{Z}/p\mathbb{Z}$ .

In weiteren Runden wird bestätigt, dass  $p_1(a_1 \dots a_n, a) = q(a)$

Das funktioniert intuitiv deshalb weil bei Polynomen eine zufällige Einsetzung fast so gut wie eine symbolische Variablenbelegung ist.

### IP-Protokoll:

Zu Beginn jeder Runde liegt ein verallgemeinertes Polynom  $p$  in Variablen  $x_1 \dots x_n$  vor und Werte  $a_1 \dots a_k \in \mathbb{Z}_p$ , wobei  $p$  eine Primzahl  $\geq 2^n$  ist.

$$b \in \mathbb{Z}_p$$

Es gibt dann zu bestätigen, dass  $p(a_1 \dots a_k) = b$ .

Gestartet wird mit  $SUM_{x_1} \dots SUM_{x_n}[q] \stackrel{?}{=} 0$  für  $k = 0$ .

Ist  $p(a_1 \dots a_k) \equiv SUM_x p_1(a_1 \dots a_k, x)$

\* Wählt  $V a \in \mathbb{Z}_p$  zufällig und lässt sich das Polynom  $q(x) = p_1(a_1 \dots a_k, x)$  von P geben.

$V$  prüft, ob  $q(0) + q(1) = b$  und in der nächsten Runde wird bestätigt, dass  $p_1(a_1 \dots a_k, a) = q(a)$

Falls  $\phi$  unerfüllbar ist, dann kann P alle Fragen wie gewünscht beantworten und  $V$  wird zu 100% überzeugt. Falls  $\phi$  erfüllbar ist, ist die einzige Möglichkeit, dass P den Verifier fälschlicherweise überzeugt, dass ein oder mehrere Male an der Stelle \* zufällig eine der  $\leq 3m$  Nullstellen von  $p_1(a, x) - q(x)$  wobei  $p_1(a, x) \neq q(x)$ , erwischt wurde.

Die Wahrscheinlichkeit, dass das nicht passiert, ist  $\geq (1 - \frac{3m}{p})^2$ . (Der Bruch bezeichnet die Größe des Körpers  $\mathbb{Z}_p$ )

$p \geq c^n \rightsquigarrow$  Wahrscheinlichkeit  $\geq 75\%$  wie gewünscht.

### Verallgemeinerung:

$$IP \supseteq PSPACE$$

$$QBF \in IP$$

Verallgemeinerte Polynome werden um zusätzliche Operatoren erweitert:

- $EX_x p(x) = 1 - (1 - p(0))(1 - p(1))$
- $ALL_x p(x) = p(0) * p(1)$
- $RED_x p(x) = p(0) + x * (p(1) - p(0))$

$RED_x p(x)$  stimmt mit  $p(x)$  auf  $[0, 1)$  überein.  $RED_x p(x)$  hat in  $x$  aber Grad 1.

Einer QBF-Formel  $\phi$  kann man mit diesen Operatoren ein verallgemeinertes Polynom  $[\phi]$  vom Grad  $n$  (Anzahl der Variablen in  $\phi$ ) zuordnen, sodass  $[\phi] = 1 \Leftrightarrow \phi$  wahr (keine freien Variablen).

### Satz

Zusammengefasst gilt:  $IP = PSPACE$

Es gibt aber Orakel  $A$ , sodass  $IP^A \neq PSPACE^A$

## 5.6.3 Probabilistically Checkable Proof

### Definition

Bei **Probabilistically Checkable Proof** (PCP) gibt es einen Verifier wie bei IP. Statt eines interaktiven Provers gibt es jetzt einen "Beweis"  $\Pi \in \{0, 1\}^N$ .  $V$  stellt Fragen an  $\Pi$  in Form eines Bitstrings  $q \in \{0, 1\}^{\log N}$ . Diese werden mit dem  $q$ . Bit von  $\Pi$  beantwortet.



$$L \in PHP$$

$$\Leftrightarrow \exists V.$$

$$x \in L \Rightarrow \exists \Pi. (\Pi, V) \text{ akzeptiert mit Wahrscheinlichkeit } \geq 75\%$$

$$x \notin L \Rightarrow \forall \Pi. (\Pi, V) \text{ akzeptiert mit Wahrscheinlichkeit } \leq 25\%$$

Falls Rechenzeit von  $V \leq p(n)$ , wobei  $n = |x|$  und  $p$  Polynom, dann machen nur Beweise  $\Pi$  mit  $|\Pi| \leq 2^{p(n)}$  Sinn.

Offensichtlich ist  $IP \subseteq PCP$ . Die Umkehrung ist fraglich, denn  $PCP = NEXP$ . Warum ist nicht offensichtlich  $PCP \subseteq IP$ .

Einschränkungen:

- $PCP(\psi(n), q(n)) = \{L | \exists PCP \text{ Verifier } V, \text{ der bei Eingabe } x \text{ nur } \mathcal{O}(\psi(|x|)) \text{ Zufallsbits verwendet und } \mathcal{O}(q(|x|)) \text{ Anfragen stellt.}\}$

## Satz

### PCP-Theorem

$$NP = PCP(\log(n), 1)$$

$c * \log(n)$  Zufallsbits

$d$  Fragen

Insgesamt  $2^{c * \log(n)} * d = n^c * d = \text{poly}(n)$

Der Beweis hat polynomielle Größe, kann also geraten werden.

$$\Rightarrow P(P(\log(n), 1)) \subseteq NP$$

Im Buch ist ein Beweis von  $NP \subseteq PCP(poly(n), 1)$

### **Problem MAX-CLIQUE-APPROX**

Gegeben ein Graph  $G = (V, E)$ .

Man berechne eine Clique in  $G$ , die mindestens halb so groß ist, wie die maximale Clique. Das Problem nennt man MAX-CLIQUE-APPROX (MCA).

### **Satz**

Falls  $P \neq NP$ , dann gibt es keinen polynomiellen Algorithmus für das Problem MAX-CLIQUE-APPROX.

### **Beweis**

Wir geben uns einen  $PCP(\log(n), 1)$  Verifier  $V$  für SAT vor. Außerdem einen hypothetischen Polynomialzeit-Algorithmus für MCA.

Daraus bauen wir ein Polynomialzeit-Entscheidungsverfahren für SAT.

Sei 3KNF  $x$  vorgelegt. Es werden  $d = \mathcal{O}(\log(|x|))$  Zufallsbits verwendet und  $t = \mathcal{O}(1)$  Fragen gestellt.

Transcript:  $(r, q_1, a_1, q_2, a_2, \dots, q_t, a_t)$ , wobei  $r$  die Zufallsbits und  $q_i$  die Fragen sind. Dieses Transcript beschreibt einen möglichen Lauf der Berechnung von  $V$ .

Nachdem  $V$  eine Turingmaschine ist, hängen die  $q_i$  nur ab von  $r$  und  $a_1 \dots a_t$ . Gültige Transcripts sind nur solche, die tatsächlich durch  $V$  in Erscheinung treten. Die Anzahl der Transcripts ist  $2^d * 2^t = poly(n)$ .

Bilde Graphen  $G = (V, E)$ , wobei  $V$  die Menge der Transcripts und  $E$  die Konsistenz

der Transcripts ist.

$(r, q_1, a_1, q_2, a_2, \dots, q_t, a_t)$  und  $(r', q'_1, a'_1, q'_2, a'_2, \dots, q'_t, a'_t)$  sind konsistent, wenn  $q_i = q'_i \Rightarrow a_i = a'_i$ .

Jeder funktionierende Beweis  $\Pi$  liefert eine Clique der Größe  $\geq 0.75 * 2^d$ , wobei  $d$  so in etwa  $\mathcal{O}(\log(i))$  ist oder so.

Umgekehrt gibt es keinen Beweis mit Akzeptanzwahrscheinlichkeit  $\geq 25\%$ , dann gibt es keine Clique  $\geq 0.25 * 2^d$ .

Mit dem hypothetischen Algorithmus für MCA berechnet man eine Clique, die mindestens halb so groß wie die größte Clique ist. Falls  $> 0.5 * 0.75 * 2^d = 0.375 * 2^d$ , dann akzeptieren, sonst verwerfen.

*q.e.d.*

## Schaltkreiskomplexität

Schaltkreise aus UND-, ODER-Gattern. NICHT-Gatter stellen

XOR mit NAND gebaut: <https://www.circuitlab.com/circuit/5m6zgn/xor-gate-with-nand-gates/>

UND-Gatter gefolgt von UND können verschmolzen werden. ODER-Gatter auch. (Also mehr Eingänge statt hintereinander stecken.)

Daher können wir Schaltkreise als geschichtet annehmen, wobei jede Schicht Gatter derselben Art enthält und UND, ODER abwechseln.

Die Größe eines Schaltkreises ist die Anzahl der Gatter.

Die Tiefe eines Schaltkreises ist die Anzahl der Schichten.

DNF:

Tiefe 3 genügt, wenn man exponentielle Größe in Kauf nimmt.

Satz: First-Save-Sipser oder so:

Paritätsfunktion ( $\sum \bmod 2$ ) kann nicht mit polynomieller Größe und konstanter Tiefe berechnet werden.

(Tiefe  $\log n$  leicht: Binärbaum aus XOR).