



# Komplexitätstheorie

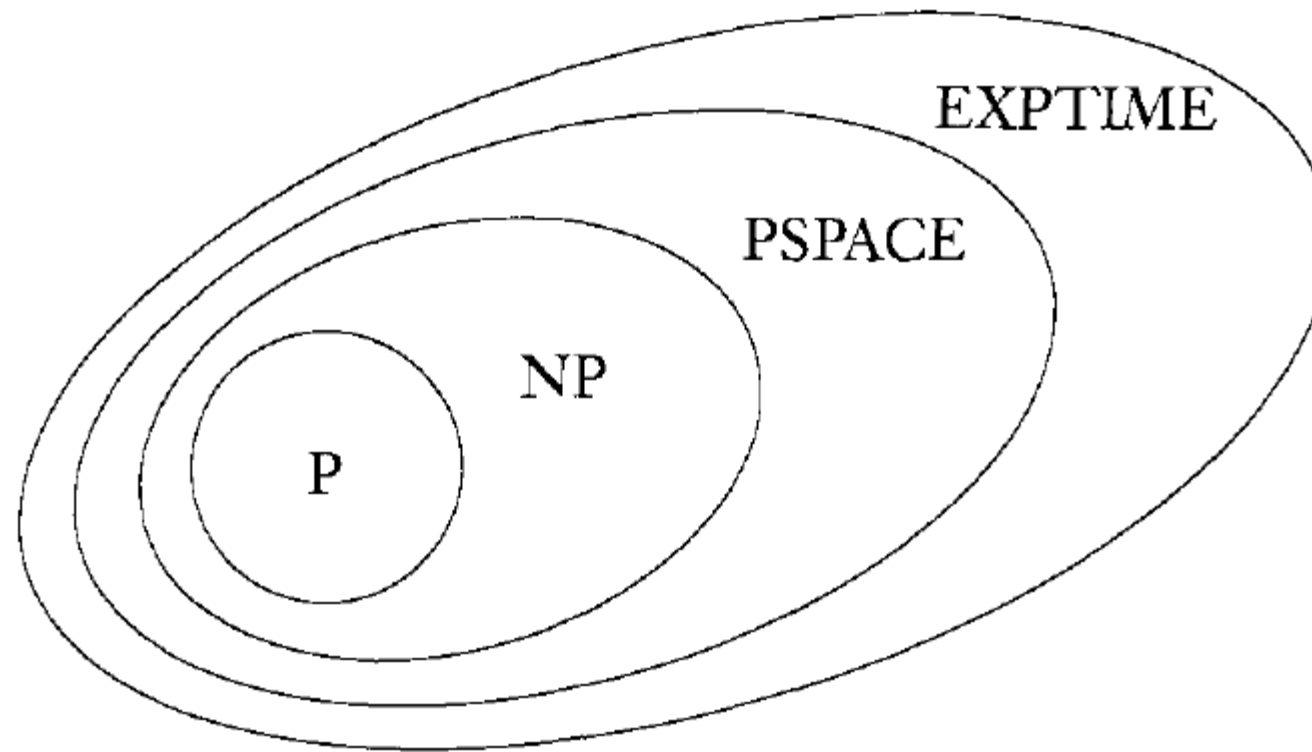
Prof. Dr. Björn Grohmann



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law



# Time & Space



$$P \subseteq NP \subseteq PSPACE = NPSpace \subseteq EXPTIME = \bigcup_k \text{TIME}(2^{n^k})$$

# PSPACE-Vollständigkeit



A language  $B$  is **PSPACE-complete** if it satisfies two conditions:

1.  $B$  is in PSPACE, and
2. every  $A$  in PSPACE is polynomial time reducible to  $B$ .

If  $B$  merely satisfies condition 2, we say that it is **PSPACE-hard**.

# True Quantified Boolean Formulas



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

$TQBF = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

## **THEOREM**

$TQBF$  is PSPACE-complete.

# True Quantified Boolean Formulas



- 1) TQBF ist in PSPACE
- 2) Für ein Wort  $w$  und eine Sprache  $A$  in PSPACE, konstruieren wir eine Formel, welche genau dann erfüllbar ist, falls  $w$  in  $A$

Die Idee ist, eine Formel  $\phi_{c_1, c_2, t}$  zu konstruieren, welche genau wahr ist, falls von der Konfig.  $c_1$  zur Konfig.  $c_2$  in  $t$  Schritten gelangt werden kann. Die Formel  $\phi_{c_{\text{start}}, c_{\text{accept}}, h}$  mit  $h = 2^{O(n^k)}$  wäre dann die gesuchte Formel.

Ein erster (rekursiver) Ansatz wäre

$$\phi_{c_1, c_2, t} = \exists m_1 \left[ \phi_{c_1, m_1, \frac{t}{2}} \wedge \phi_{m_1, c_2, \frac{t}{2}} \right]$$

$$\exists x_1, \dots, x_l$$

wird exponentiell groß

# True Quantified Boolean Formulas



- 1) TQBF ist in PSPACE
- 2) Für ein Wort  $w$  und eine Sprache  $A$  in PSPACE, konstruieren wir eine Formel, welche genau dann erfüllbar ist, falls  $w$  in  $A$

Die Idee ist, eine Formel  $\phi_{c_1, c_2, t}$  zu konstruieren, welche genau wahr ist, falls von der Konfig.  $c_1$  zur Konfig.  $c_2$  in  $t$  Schritten gelangt werden kann. Die Formel  $\phi_{c_{\text{start}}, c_{\text{accept}}, h}$  mit  $h = 2^{O(n^k)}$  wäre dann die gesuchte Formel.

Verbesserter Ansatz

$$\phi_{c_1, c_2, t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} [\phi_{c_3, c_4, \frac{t}{2}}]$$

$\forall x \in \{y, z\} [\dots]$  steht für  $\forall x [(x = y \vee x = z) \rightarrow \dots]$

muss in der Formel durch  
„Äquivalenz“ ersetzt werden

# True Quantified Boolean Formulas



- 1) TQBF ist in PSPACE
- 2) Für ein Wort  $w$  und eine Sprache  $A$  in PSPACE, konstruieren wir eine Formel, welche genau dann erfüllbar ist, falls  $w$  in  $A$

Die Idee ist, eine Formel  $\phi_{c_1, c_2, t}$  zu konstruieren, welche genau wahr ist, falls von der Konfig.  $c_1$  zur Konfig.  $c_2$  in  $t$  Schritten gelangt werden kann. Die Formel  $\phi_{c_{\text{start}}, c_{\text{accept}}, h}$  mit  $h = 2^{O(n^k)}$  wäre dann die gesuchte Formel.

**Verbesserter Ansatz** 
$$\phi_{c_1, c_2, t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} [\phi_{c_3, c_4, \frac{t}{2}}]$$

Jeder Rekursionslevel vergrößert die Formel um  $O(n^k)$ . Die Rekursionstiefe beträgt ebenfalls  $O(n^k)$  und damit insgesamt  $O(n^{2k})$ .



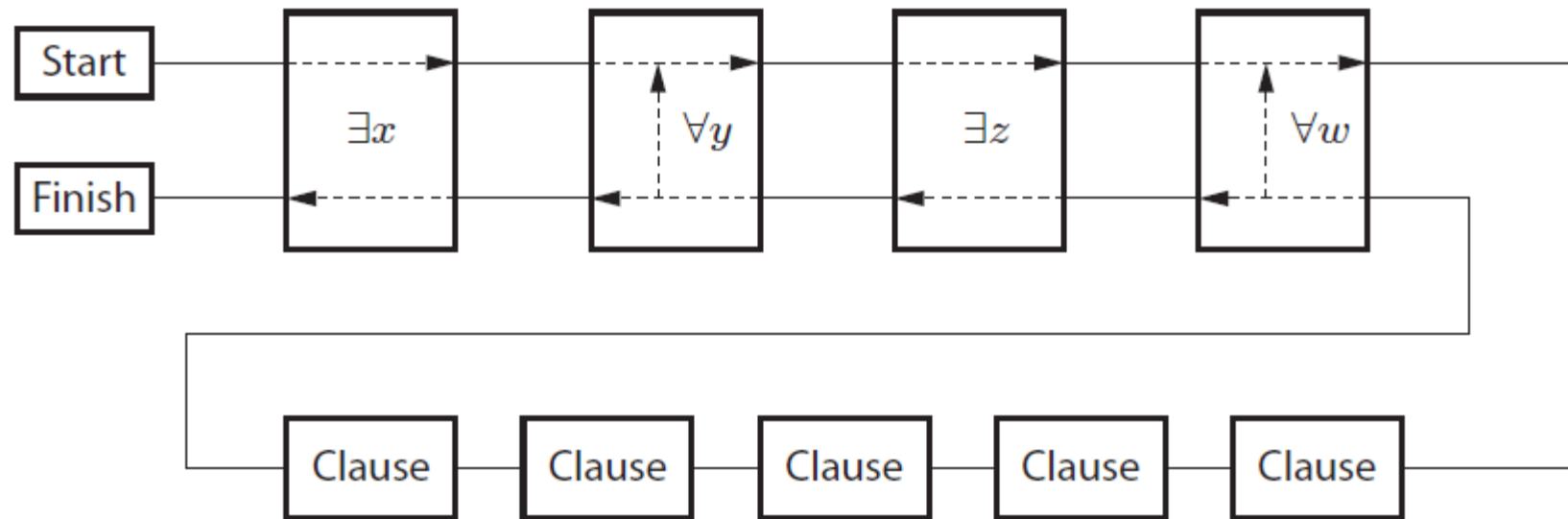


# „Gaming is hard“, Part II

**Metatheorem 4!** *If a game features doors and pressure plates, and the avatar has to reach an exit location in order to win, then:*

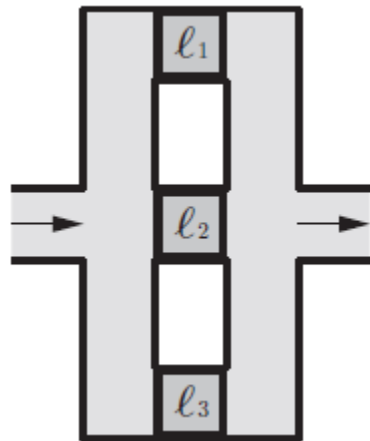
*If each door may be controlled by two pressure plates, then the game is **PSPACE-hard**.*

# „Gaming is hard“, Part II

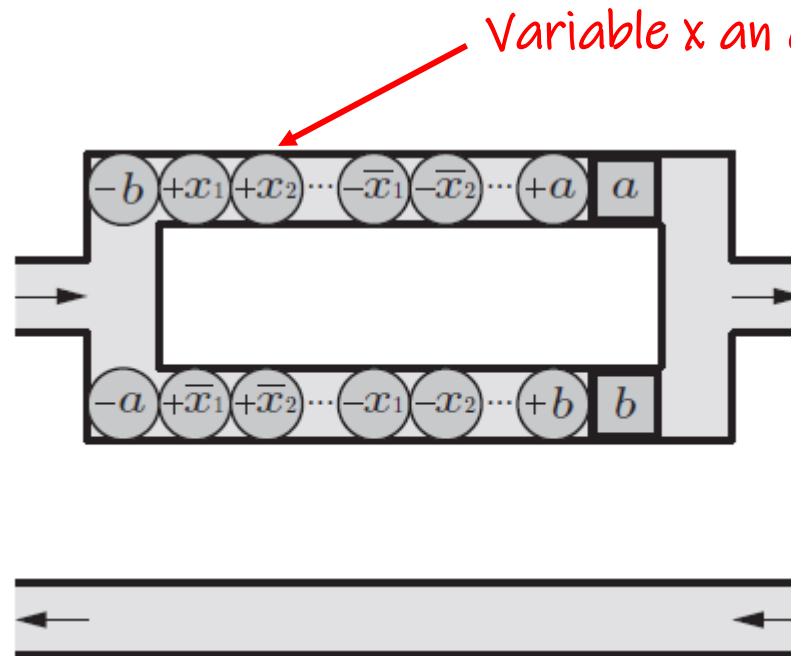


(Aufbau eines Levels, welcher eine TQBF simuliert)

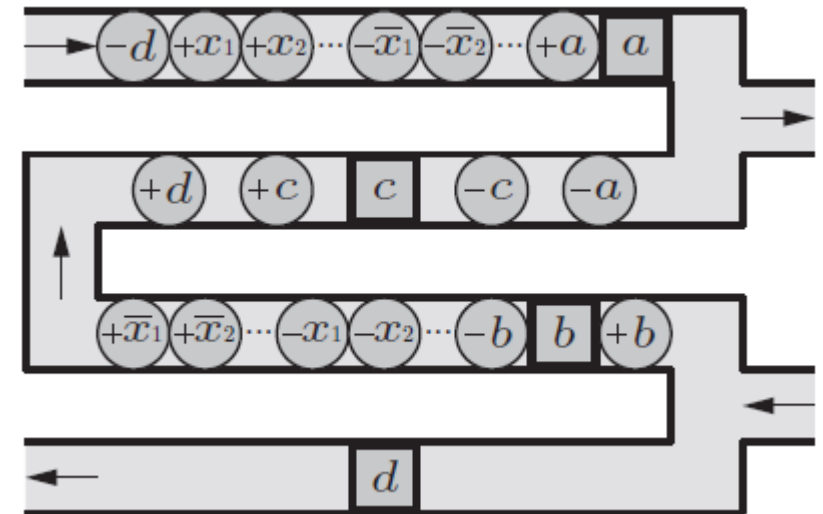
# „Gaming is hard“, Part II



Klausel, realisiert  
mit Türen



„Existenzquantor“



„Allquantor“



# „Gaming is hard“, Part II

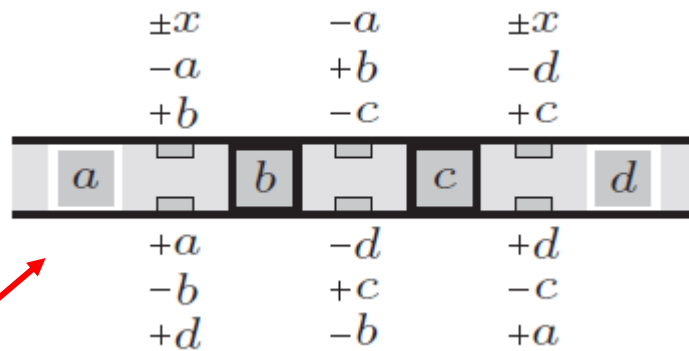
**Metatheorem 5!** *If a game features doors and  $k$ -buttons, and the avatar has to reach an exit location in order to win, then:*

*If  $k \geq 3$ , then the game is **PSPACE**-hard.*

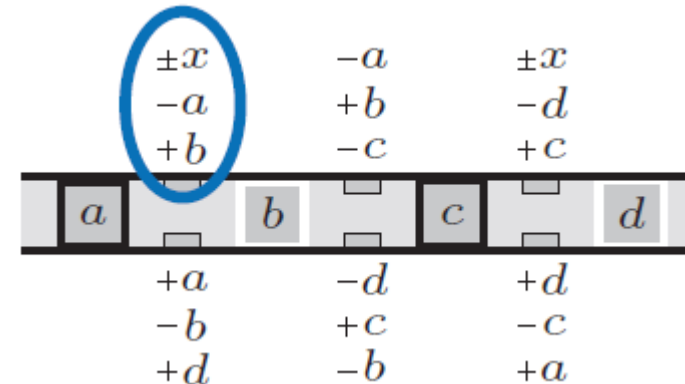
# „Gaming is hard“, Part II



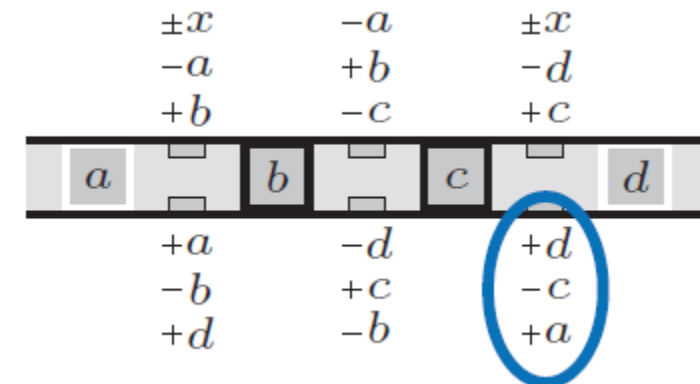
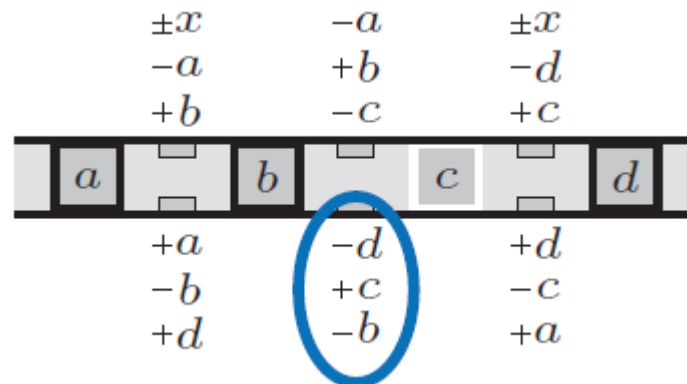
Simulation einer Druckplatte mit „3-buttons“



Startkonfiguration



Spieler kommt von links



# Hierarchy Theorems



A function  $t: \mathcal{N} \rightarrow \mathcal{N}$ , where  $t(n)$  is at least  $O(n \log n)$ , is called **time constructible** if the function that maps the string  $1^n$  to the binary representation of  $t(n)$  is computable in time  $O(t(n))$ .

A function  $f: \mathcal{N} \rightarrow \mathcal{N}$ , where  $f(n)$  is at least  $O(\log n)$ , is called **space constructible** if the function that maps the string  $1^n$  to the binary representation of  $f(n)$  is computable in space  $O(f(n))$ .

# Hierarchy Theorems



## THEOREM

**Space hierarchy theorem** For any space constructible function  $f: \mathcal{N} \rightarrow \mathcal{N}$ , a language  $A$  exists that is decidable in  $O(f(n))$  space but not in  $o(f(n))$  space.

Der folgende Algorithmus  $D$  entscheidet eine Sprache  $A$  mit maximal  $O(f(n))$  Speicher:

$D =$  “On input  $w$ :

1. Let  $n$  be the length of  $w$ .
2. Compute  $f(n)$  using space constructibility, and mark off this much tape. If later stages ever attempt to use more, *reject*.
3. If  $w$  is not of the form  $\langle M \rangle 10^*$  for some TM  $M$ , *reject*.
4. Simulate  $M$  on  $w$  while counting the number of steps used in the simulation. If the count ever exceeds  $2^{f(n)}$ , *reject*.
5. If  $M$  accepts, *reject*. If  $M$  rejects, *accept*.”

Falls nun, angenommen, eine Maschine  $M$  die Sprache  $A$  mit  $g(n)$  Speicher entscheidet und  $g(n)$  in  $o(f(n))$ , dann ergibt sich hier ein Widerspruch.



# Hierarchy Theorems



## THEOREM

**Space hierarchy theorem** For any space constructible function  $f: \mathcal{N} \rightarrow \mathcal{N}$ , a language  $A$  exists that is decidable in  $O(f(n))$  space but not in  $o(f(n))$  space.

## COROLLARY

For any two real numbers  $0 \leq \epsilon_1 < \epsilon_2$ ,

$$\text{SPACE}(n^{\epsilon_1}) \subsetneq \text{SPACE}(n^{\epsilon_2}).$$

## COROLLARY

$$\text{PSPACE} \subsetneq \text{EXPSPACE} = \bigcup_k \text{SPACE}(2^{n^k})$$



# Hierarchy Theorems



## THEOREM

**Time hierarchy theorem** For any time constructible function  $t: \mathcal{N} \rightarrow \mathcal{N}$ , a language  $A$  exists that is decidable in  $O(t(n))$  time but not decidable in time  $o(t(n)/\log t(n))$ .

Ähnlich wie beim Space Hierarchy Theorem entscheidet  $D$  die Sprache  $A$  in  $O(t(n))$  Schritten:

$D =$  “On input  $w$ :

1. Let  $n$  be the length of  $w$ .
2. Compute  $t(n)$  using time constructibility, and store the value  $\lceil t(n)/\log t(n) \rceil$  in a binary counter. Decrement this counter before each step used to carry out stages 3, 4, and 5. If the counter ever hits 0, *reject*.
3. If  $w$  is not of the form  $\langle M \rangle 10^*$  for some TM  $M$ , *reject*.
4. Simulate  $M$  on  $w$ .
5. If  $M$  accepts, then *reject*. If  $M$  rejects, then *accept*.”

Das Counterupdate kostet pro Schritt jeweils  $O(\log t(n))$ .  
Damit also insgesamt  $O(t(n))$  Schritte.

Sollte eine Maschine  $M$  in weniger (wie im Theorem) Schritten entscheiden, so ergäbe sich hier ein Widerspruch.

# Hierarchy Theorems



## THEOREM

**Time hierarchy theorem** For any time constructible function  $t: \mathcal{N} \rightarrow \mathcal{N}$ , a language  $A$  exists that is decidable in  $O(t(n))$  time but not decidable in time  $o(t(n)/\log t(n))$ .

## COROLLARY

For any two real numbers  $1 \leq \epsilon_1 < \epsilon_2$ ,

$$\text{TIME}(n^{\epsilon_1}) \subsetneq \text{TIME}(n^{\epsilon_2}).$$

## COROLLARY

$P \subsetneq \text{EXPTIME}$ .

# Hierarchy Theorems



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

Warum ist die Voraussetzung, dass eine Funktion  
„time-constructible“ sein muss notwendig?

# Gap Theorem



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

...weil es auch „seltsame“ Funktionen gibt:

**Theorem** (Gap Theorem, Trakhtenbrot, Borodin) *There exists a computable function  $f$  such that  $\text{TIME}(f(n)) = \text{TIME}(2^{f(n)})$ .*

# Gap Theorem



**Theorem** (Gap Theorem, Trakhtenbrot, Borodin) *There exists a computable function  $f$  such that  $\text{TIME}(f(n)) = \text{TIME}(2^{f(n)})$ .*

Zum Beweis sei  $\{M_e\}$  eine Aufzählung aller (unbeschränkten) det. TMs. Wir konstruieren eine Funktion  $f$  in Schritten, wobei  $f$  in jedem Schritt  $e$  folgende Bedingung erfüllen soll:

$$\forall x (|x| = n \geq e \wedge M_e(x) \downarrow \text{ in } t \text{ steps} \implies t \notin (f(n), 2^{f(n)}])$$

Falls nun eine Sprache  $A \in \text{TIME}(2^{f(n)})$ , was durch eine Maschine  $M_e$  bezeugt wird, gilt nach Voraussetzung für alle  $x$  der Länge größer als  $e$ , dass  $M_e$  die Eingabe  $x$  in höchstens  $f(n)$  Schritten akzeptiert und damit  $A \in \text{TIME}(f(n))$ .

# Gap Theorem



In Schritt  $n$  wird  $f(n)$  nun wie folgt definiert:

Betrachte folgende Sequenz von Zahlen:  $k_0 = 0, k_{l+1} = 2^{k_l}$

Für alle Berechnungen  $M_e(x)$  mit  $e \leq n = \text{Länge von } x$ , so dass  $M_e(x)$  in  $t_{e,x}$  Schritten hält, gibt es dann ein kleinstes  $k_l$ , so dass keines der  $t_{e,x}$  in dem Intervall  $(k_l, 2^{k_l}]$  liegt.

Definiere  $f(n) = k_l$ .



# Speed-Up Theorem

...ebenso gibt es auch „seltsame“ Sprachen:

**Theorem** (Speed-Up Theorem, M. Blum ) *There exists a computable set  $A$  such that for every index  $e$  for  $A$  there is another index  $i$  for  $A$  such that*

$$\forall^\infty x (\Phi_i(x) \leq \log \Phi_e(x)).$$

Anzahl der Berechnungsschritte,  
falls  $M_e(x)$  hält.

*That is, the program  $M_i$  computes  $A$  exponentially faster than  $M_e$ .*

(„ $\forall^\infty$ “ bedeutet hier „fast alle“, also alle, bis auf endlich viele)



# Speed-Up Theorem

Zunächst setzen wir  $g(x) = 2^x$ ,  $g^{(1)}(x) = g(x)$  und  $g^{(n+1)}(x) = g(g^{(n)}(x))$ . Für  $x > e + 1$  ist dann  $h_e(x) = g^{(x-e)}(0)$  damit ist  $h_e$ , wegen  $g(h_{e+1}(x)) = h_e(x)$ , eine Familie abnehmender Funktionen.

Wir bestimmen nun für jedes  $x$  den Wert  $A(x)$ , d.h. ob  $x$  zu  $A$  gehört oder nicht, indem wir für alle  $e \leq x$  testen, ob  $e$  noch nicht **markiert („cancelled“)** und ob gilt  $\Phi_e(x) < h_e(x)$ . Für das kleinste  $e$  dieserart definieren wir  $A(x) = 1 - M_e(x)$  und markieren  $e$ .

↖  
Diagonalisierung





# Speed-Up Theorem

Zunächst setzen wir  $g(x) = 2^x$ ,  $g^{(1)}(x) = g(x)$  und  $g^{(n+1)}(x) = g(g^{(n)}(x))$ . Für  $x > e + 1$  ist dann  $h_e(x) = g^{(x-e)}(0)$  damit ist  $h_e$ , wegen  $g(h_{e+1}(x)) = h_e(x)$ , eine Familie abnehmender Funktionen.

Wir bestimmen nun für jedes  $x$  den Wert  $A(x)$ , d.h. ob  $x$  zu  $A$  gehört oder nicht, indem wir für alle  $e \leq x$  testen, ob  $e$  noch nicht **markiert („cancelled“)** und ob gilt  $\Phi_e(x) < h_e(x)$ . Für das kleinste  $e$  dieserart definieren wir  $A(x) = 1 - M_e(x)$  und markieren  $e$ .

Damit gilt für jedes  $e$ , dass, falls für unendlich viele  $x$  gilt  $\Phi_e(x) < h_e(x)$ , folgt  $M_e \neq A$  ( $M_e$  steht hier auch für die Sprache, welche die Maschine akzeptiert). Oder anders geschrieben:  $\forall e (M_e = A \implies \forall^\infty x \ h_e(x) \leq \Phi_e(x))$ .

← (bedenke aber, dass  $h_e$  mit wachsendem  $e$  immer kleiner wird)



# Speed-Up Theorem

Um nun  $A(x)$  zu berechnen, kann man für jedes  $e \leq x$   $M_e(x)$  für  $h_e(x)$  Schritte laufen lassen. Mit Hilfe der endlichen Menge  $F_u = \{(e, x, A(x)) : e < u \wedge e \text{ cancelled at stage } x\}$ , für eine natürliche Zahl  $u$ , genügt es aber auch, dies nur für  $u \leq e \leq x$  zu tun (da die benötigte Information für  $e < u$  aus  $F_u$  abgelesen werden kann).

Diese Berechnung dauert aber nur  $h_u(x) + \dots + h_x(x)$  Schritte, und damit ist die Laufzeit, für die ersten  $x$  Stufen, von oben beschränkt durch  $x \cdot (h_u(x) + \dots + h_x(x)) \leq h_{u-1}(x)$  (für fast alle  $x$ ).

Da  $u$  beliebig groß gewählt werden kann gilt somit  $\forall e \exists i (M_i = A \wedge \forall^\infty x \Phi_i(x) \leq h_{e+1}(x))$  und damit insgesamt

$$\Phi_i(x) \leq h_{e+1}(x) \leq \log h_e(x) \leq \log \Phi_e(x).$$

# Orakel Turing Maschine



An **oracle** for a language  $A$  is a device that is capable of reporting whether any string  $w$  is a member of  $A$ . An **oracle Turing machine**  $M^A$  is a modified Turing machine that has the additional capability of querying an oracle. Whenever  $M^A$  writes a string on a special **oracle tape** it is informed whether that string is a member of  $A$ , in a single computation step.

Let  $P^A$  be the class of languages decidable with a **polynomial time oracle Turing machine** that uses oracle  $A$ . Define  $NP^A$  similarly.

# Relativierung



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

Kann Relativierung dabei helfen  
Komplexitätsklassen zu separieren?

# Relativierung



## THEOREM

1. An oracle  $A$  exists whereby  $P^A \neq NP^A$ .
2. An oracle  $B$  exists whereby  $P^B = NP^B$ .



# Relativierung

2. Wähle für  $B$  etwa  $TQBF$  und damit  $NP^{TQBF} \subseteq NPSpace \subseteq PSpace \subseteq P^{TQBF}$ .

1. Wir konstruieren ein Orakel  $A$ , so dass die Sprache

$$L_A = \{w \mid \exists x \in A [ |x| = |w| ]\}$$

nicht in  $P^A$  liegt (offensichtlich liegt sie in  $NP^A$ ).

Dazu betrachten wir die Liste  $M_1, M_2, M_3, \dots$  aller polynomial-Zeit-Orakel-TMs.

Wir konstruieren  $A$  in Schritten:

Schritt 1: Wähle endliche viele beliebige Strings und packe sie in  $A$ .

Schritt  $i$ : Wähle  $n$ , so dass  $2^n$  größer als die Laufzeit  $p_i(n)$  von  $M_i$  und größer als die Länge aller bisherigen Strings in  $A$ .

$p_i$  ist ein Polynom



# Relativierung

2. Wähle für  $B$  etwa  $TQBF$  und damit  $NP^{TQBF} \subseteq NPSPACE \subseteq PSPACE \subseteq P^{TQBF}$ .

1. Wir konstruieren ein Orakel  $A$ , so dass die Sprache

$$L_A = \{w \mid \exists x \in A [ |x| = |w| ]\}$$

nicht in  $P^A$  liegt (offensichtlich liegt sie in  $NP^A$ ).

Dazu betrachten wir die Liste  $M_1, M_2, M_3, \dots$  aller polynomial-Zeit-Orakel-TMs.

Wir konstruieren  $A$  in Schritten:

Schritt 1: Wähle endliche viele beliebige Strings und packe sie in  $A$ .

Schritt  $i$ : Wähle  $n$ , so dass  $2^n$  größer als die Laufzeit  $p_i(n)$  von  $M_i$  und größer als die Länge aller bisherigen Strings in  $A$ . Lasse  $M_i$  mit Eingabe  $1^n$  laufen.

also eine Eingabe der Länge  $n$

# Relativierung



2. Wähle für  $B$  etwa  $TQBF$  und damit  $NP^{TQBF} \subseteq NPSpace \subseteq PSpace \subseteq P^{TQBF}$ .

1. Wir konstruieren ein Orakel  $A$ , so dass die Sprache

$$L_A = \{w \mid \exists x \in A [ |x| = |w| ]\}$$

nicht in  $P^A$  liegt (offensichtlich liegt sie in  $NP^A$ ).

Dazu betrachten wir die Liste  $M_1, M_2, M_3, \dots$  aller polynomial-Zeit-Orakel-TMs.

Wir konstruieren  $A$  in Schritten:

Schritt 1: Wähle endliche viele beliebige Strings und packe sie in  $A$ .

Schritt  $i$ : Wähle  $n$ , so dass  $2^n$  größer als die Laufzeit  $p_i(n)$  von  $M_i$  und größer als die Länge aller bisherigen Strings in  $A$ . Lasse  $M_i$  mit Eingabe  $1^n$  laufen. Falls  $M_i$  das Orakel nach  $y$  fragt und der Status von  $y$  feststeht, antworte gemäß dem Status. Falls der Status nicht feststeht antworte mit „Nein“ und setze den Status von  $y$  als „nicht in  $A$ “. Lass  $M_i$  solange laufen bis es hält. Falls  $M_i$  akzeptiert, erhalten alle verbleibenden Strings der Länge  $n$  den Status „nicht in  $A$ “.

Diagonalisierung





# Relativierung

2. Wähle für  $B$  etwa  $TQBF$  und damit  $NP^{TQBF} \subseteq NPSPACE \subseteq PSPACE \subseteq P^{TQBF}$ .

1. Wir konstruieren ein Orakel  $A$ , so dass die Sprache

$$L_A = \{w \mid \exists x \in A [ |x| = |w| ]\}$$

nicht in  $P^A$  liegt (offensichtlich liegt sie in  $NP^A$ ).

Dazu betrachten wir die Liste  $M_1, M_2, M_3, \dots$  aller polynomial-Zeit-Orakel-TMs.

Wir konstruieren  $A$  in Schritten:

Schritt 1: Wähle endliche viele beliebige Strings und packe sie in  $A$ .

Schritt  $i$ : Wähle  $n$ , so dass  $2^n$  größer als die Laufzeit  $p_i(n)$  von  $M_i$  und größer als die Länge aller bisherigen Strings in  $A$ . Lasse  $M_i$  mit Eingabe  $1^n$  laufen. Falls  $M_i$  das Orakel nach  $y$  fragt und der Status von  $y$  feststeht, antworte gemäß dem Status. Falls der Status nicht feststeht antworte mit „Nein“ und setze den Status von  $y$  als „nicht in  $A$ “. Lass  $M_i$  solange laufen bis es hält. Falls  $M_i$  akzeptiert, erhalten alle verbleibenden Strings der Länge  $n$  den Status „nicht in  $A$ “. Falls  $M_i$  nicht akzeptiert, so wähle ein Element, nach welchem  $M_i$  das Orakel nicht gefragt hat und füge es zu  $A$  hinzu.

das geht, da  $2^n$  größer  $p_i(n)$



# Relativierung

2. Wähle für  $B$  etwa  $TQBF$  und damit  $NP^{TQBF} \subseteq NPSpace \subseteq PSpace \subseteq P^{TQBF}$ .

1. Wir konstruieren ein Orakel  $A$ , so dass die Sprache

$$L_A = \{w \mid \exists x \in A [ |x| = |w| ]\}$$

nicht in  $P^A$  liegt (offensichtlich liegt sie in  $NP^A$ ).

Dazu betrachten wir die Liste  $M_1, M_2, M_3, \dots$  aller polynomial-Zeit-Orakel-TMs.

Wir konstruieren  $A$  in Schritten:

Schritt 1: Wähle endliche viele beliebige Strings und packe sie in  $A$ .

Schritt  $i$ : Wähle  $n$ , so dass  $2^n$  größer als die Laufzeit  $p_i(n)$  von  $M_i$  und größer als die Länge aller bisherigen Strings in  $A$ . Lasse  $M_i$  mit Eingabe  $1^n$  laufen. Falls  $M_i$  das Orakel nach  $y$  fragt und der Status von  $y$  feststeht, antworte gemäß dem Status. Falls der Status nicht feststeht antworte mit „Nein“ und setze den Status von  $y$  als „nicht in  $A$ “. Lass  $M_i$  solange laufen bis es hält. Falls  $M_i$  akzeptiert, erhalten alle verbleibenden Strings der Länge  $n$  den Status „nicht in  $A$ “. Falls  $M_i$  nicht akzeptiert, so wähle ein Element, nach welchem  $M_i$  das Orakel nicht gefragt hat und füge es zu  $A$  hinzu. Fahre mit Schritt  $i+1$  fort.



# Polynomial Hierachy

Für eine Klasse  $C$  definieren wir  $\exists C = \{x : \exists^{p(|x|)} y \langle x, y \rangle \in B\}$  (und analog für den Allquantor).

„es existiert ein String der  
Länge  $\leq p(|x|)$ “

Demnach gilt also z.B.  $\exists P = NP$ , bzw.  $\forall P = \text{co-NP}$ .

Für eine Klasse  $C$  ist ein Problem in  
 $\text{Co-}C$ , falls das *duale Problem* in  $C$  ist

Wir setzen jetzt  $\Sigma_0^P = \Pi_0^P = P$ , und dann für alle  $n$ :

$$\Sigma_{n+1}^P = \exists \Pi_n^P$$

$$\Pi_{n+1}^P = \forall \Sigma_n^P$$

# Polynomial Hierachy



For every  $i \geq 1$ , a language  $L$  is in  $\Sigma_i^p$  if there exists a polynomial-time TM  $M$  and a polynomial  $q$  such that

$$x \in L \Leftrightarrow \exists u_1 \in \{0, 1\}^{q(|x|)} \forall u_2 \in \{0, 1\}^{q(|x|)} \dots Q_i u_i \in \{0, 1\}^{q(|x|)} M(x, u_1, \dots, u_i) = 1,$$

die  
Quantoren  
alternieren  
hier

where  $Q_i$  denotes  $\forall$  or  $\exists$  depending on whether  $i$  is even or odd respectively.

We say that  $L$  is in  $\Pi_i^p$  if there exists a polynomial-time TM  $M$  and a polynomial  $q$  such that

$$x \in L \Leftrightarrow \forall u_1 \in \{0, 1\}^{q(|x|)} \exists u_2 \in \{0, 1\}^{q(|x|)} \dots Q_i u_i \in \{0, 1\}^{q(|x|)} M(x, u_1, \dots, u_i) = 1,$$

where  $Q_i$  denotes  $\exists$  or  $\forall$  depending on whether  $i$  is even or odd respectively.

The *polynomial hierarchy* is the set  $\mathbf{PH} = \cup_i \Sigma_i^p$ .

# Polynomial Hierachy



Sei  $NP^C = NP(\mathcal{C}) = \bigcup \{NP^C : C \in \mathcal{C}\}$ . Dann gilt für jedes  $n > 0$ :  $\Sigma_{n+1}^P = NP(\Sigma_n^P)$ .

Für jedes  $S$  in  $\Sigma_{n+1}^P$  gibt es nach Definition ein  $S'$  aus  $\Pi_n^P$ , sodass  $S = \{x, \text{ex-poly. } y \text{ mit } (x,y) \text{ in } S'\}$ .  
Damit ist  $S$  aber in  $NP(\Pi_n^P) = NP(\Sigma_n^P)$ .

Sei umgekehrt  $S$  in  $NP(\Sigma_n^P)$ . Dann existiert zu einer Eingabe  $x$  also eine Reihe von Queries  $q_i(x,y)$  an ein Orakel  $S'$  aus  $\Sigma_n^P$ . Da diese adaptiv sein können nehmen wir an, dass die TM die Antworten  $a_i(x,y)$  schon im Vorfeld rät. Es gilt dann, dass  $x$  in  $S$ , genau dann wenn

$$\exists y \left( A(x, y) \wedge \bigwedge_{i=1}^{q(x,y)} \left( (a^{(i)}(x, y) = 1) \Leftrightarrow (q^{(i)}(x, y) \in S') \right) \right) \in \Sigma_{n+1}^P$$

es existiert eine Reihe von Queries

so dass die TM  $x$  akzeptiert

und die geratenen Antworten mit den Antworten des Orakels übereinstimmen

# Polynomial Hierachy



Vollständiges Problem für Level i:

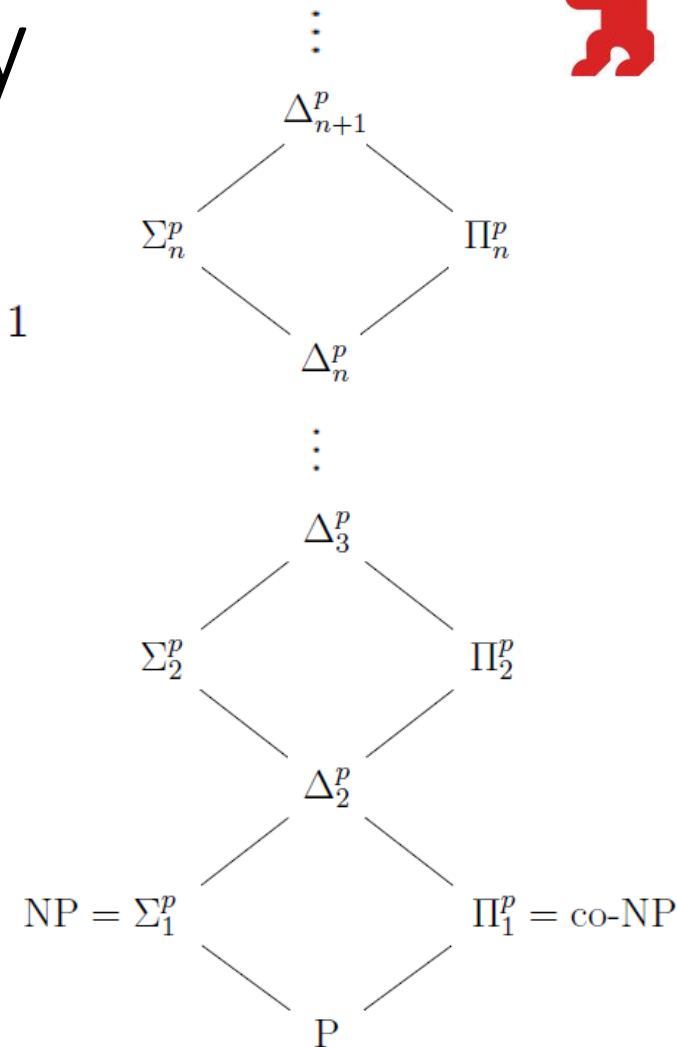
$$\Sigma_i \text{SAT} = \exists u_1 \forall u_2 \exists \dots Q_i u_i \varphi(u_1, u_2, \dots, u_i) = 1$$

Falls es ein vollständiges Problem für PH  
geben sollte, so kollabiert PH.

es ist leicht einzusehen, dass

$$\text{PH} \subseteq \text{PSPACE}$$

Falls jedoch  $\text{PH} = \text{PSPACE}$   
gelten sollte, so kollabiert PH.



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

$$\Sigma_{n+1}^P = \text{NP}(\Sigma_n^P).$$

$$\Pi_{n+1}^P = \text{co-}\Sigma_{n+1}^P.$$

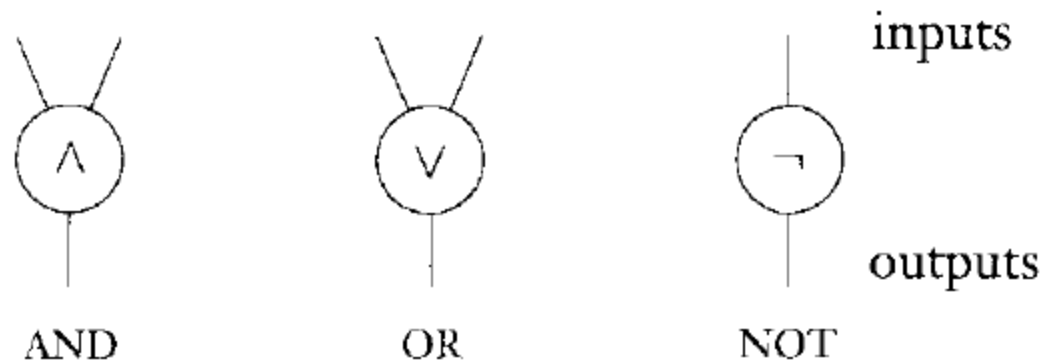
$$\Delta_{n+1}^P = P(\Sigma_n^P).$$

$$\text{PH} = \bigcup_{n \geq 0} \Sigma_n^P.$$

# Circuit Complexity



A **Boolean circuit** is a collection of **gates** and **inputs** connected by **wires**. Cycles aren't permitted. Gates take three forms: AND gates, OR gates, and NOT gates, as shown schematically in the following figure.

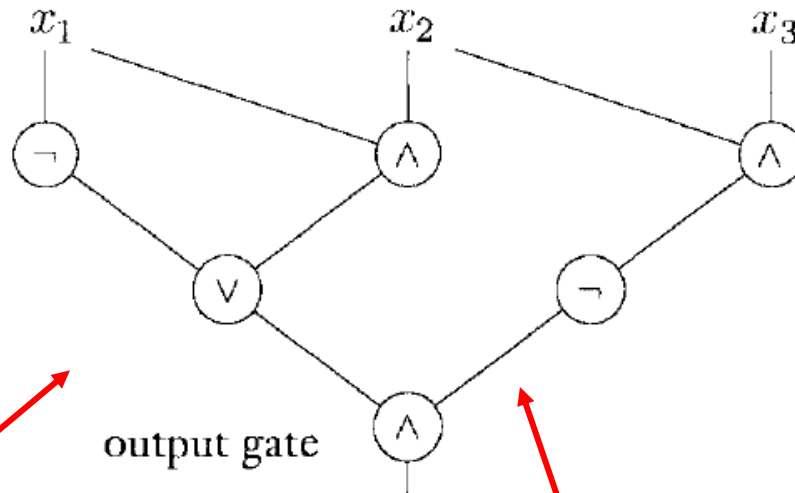




# Circuit Complexity



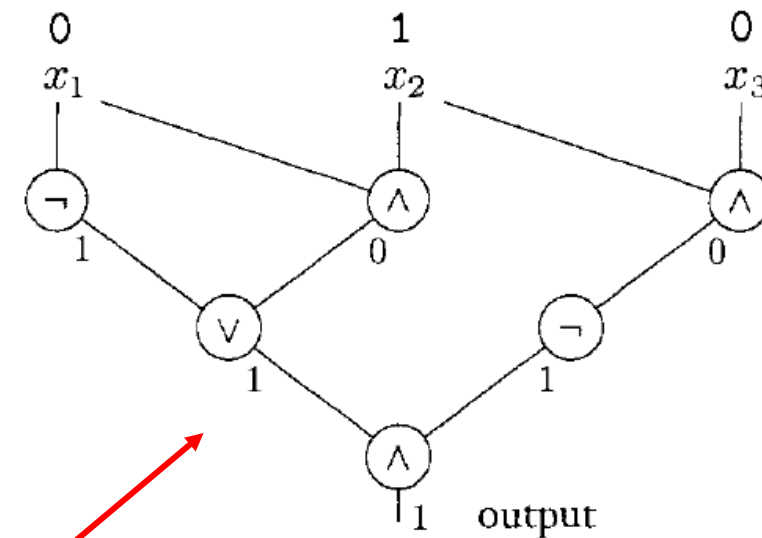
input  
variables



Anzahl der  
„gates“  
heißt die  
„size“

Länge des längsten  
Pfads heißt „depth“

inputs



Auswertung



# Circuit Complexity



A **circuit family**  $C$  is an infinite list of circuits,  $(C_0, C_1, C_2, \dots)$ , where  $C_n$  has  $n$  input variables. We say that  $C$  decides a language  $A$  over  $\{0,1\}$  if, for every string  $w$ ,

$$w \in A \quad \text{iff} \quad C_n(w) = 1,$$

where  $n$  is the length of  $w$ .

# Circuit Complexity



The *circuit size complexity* of a language is the *size* complexity of a *minimal circuit family* for that language. The *circuit depth complexity* of a language is defined similarly, using depth instead of size.

# Circuit Complexity



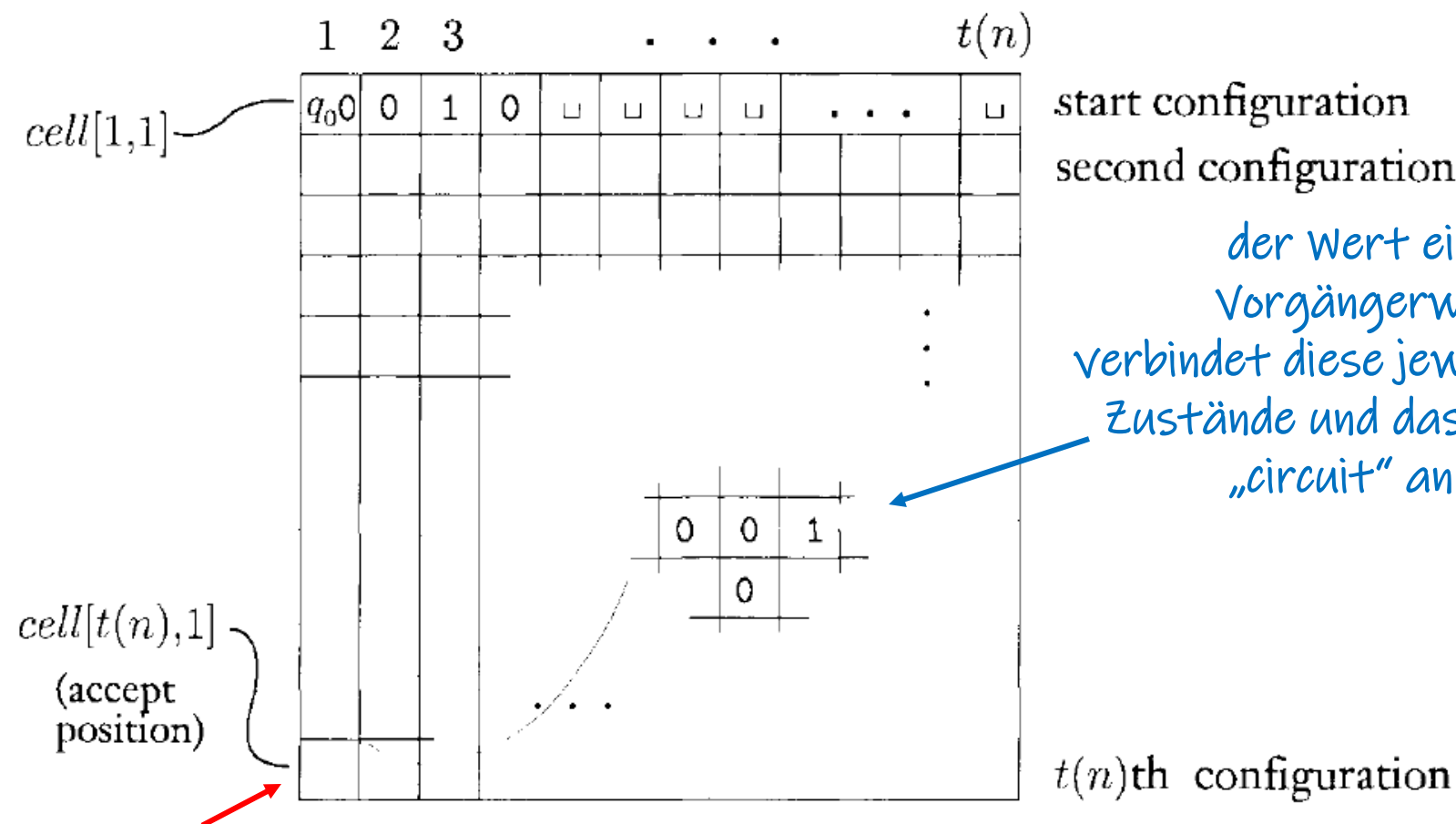
Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

## THEOREM

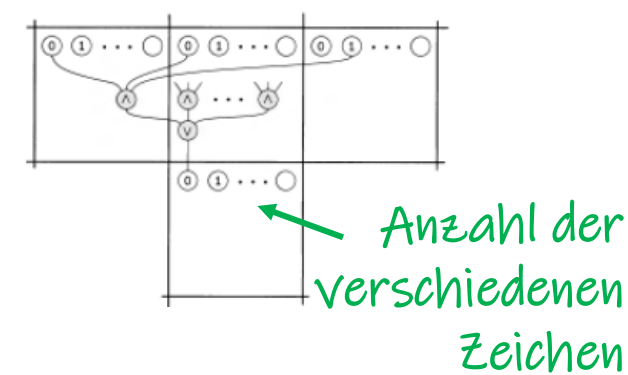
Let  $t: \mathcal{N} \rightarrow \mathcal{N}$  be a function, where  $t(n) \geq n$ . If  $A \in \text{TIME}(t(n))$ , then  $A$  has circuit complexity  $O(t^2(n))$ .



# Circuit Complexity



der Wert einer Zelle, hängt von den 3 Vorgängerwerten ab, d.h. der „circuit“ verbindet diese jeweils. Da eine Zelle auch die Zustände und das „blank“ kodiert, sieht der „circuit“ an dieser Stelle etwa so aus:



Hier wird angenommen, dass man an diesem Feld sehen kann, ob die TM akzeptiert.

# Circuit Complexity



For a given function  $s : \mathbb{N} \rightarrow \mathbb{N}$ ,  $\text{SIZE}(s(n))$  denotes the class of languages  $L$  that are computed by a circuit family such that the size of  $C_n$  is bounded by  $s(n)$ .

**Proposition**      *Any language  $L \subseteq \{0, 1\}^*$  is in  $\text{SIZE}(O(2^n))$ .*

**Proposition**      (Shannon) *There are languages that do not have subexponential circuit size: For any  $s$  such that  $s(n) = 2^{o(n)}$  there is a language  $L \notin \text{SIZE}(s(n))$ .*

# Circuit Complexity



**Definition 7.3.1.** Let  $\mathcal{C}$  be a **complexity class** and let  $\mathcal{F}$  be a class of **advice functions** of the form  $s : \mathbb{N} \rightarrow \{0, 1\}^*$ . We think of  $s$  as giving an advice string for every length  $n$ . Then  $\mathcal{C}/\mathcal{F}$  is the class of all sets of the form

$$\{x : \langle x, s(|x|) \rangle \in B\}$$

for some  $B \in \mathcal{C}$  and some  $s \in \mathcal{F}$ .

# Circuit Complexity



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

**Theorem**  $P/\text{poly} = \text{SIZE}(n^{O(1)}).$

# Circuit Complexity



**Theorem**  $P/\text{poly} = \text{SIZE}(n^{O(1)})$ .

Die Richtung  $P/\text{poly} \subseteq \text{SIZE}(n^{O(1)})$  haben wir schon gezeigt, da für eine Sprache  $A$ , welche in  $O(t(n))$  Schritten entschieden werden kann, für Elemente fester Länge  $n$  auch immer ein „circuit“ der Größe  $O(t(n)^2)$  existiert, welcher diese entscheidet.

Die Richtung  $\text{SIZE}(n^{O(1)}) \subseteq P/\text{poly}$  gilt, da die „circuits“ in polynomialer Zeit ausgewertet können, d.h. die Menge  $B = \{\langle x, C \rangle : C \text{ circuit} \wedge C(x) = 1\}$  ist in  $P$ . Damit gilt dann für eine Sprache  $A$ :  $x \in A \Leftrightarrow \langle x, C_{|x|} \rangle \in B$  und somit  $A \in P/\text{poly}$ .



# Circuit Complexity



**Theorem**  $P/\text{poly} = \bigcup \{P(S) : S \text{ sparse}\}.$

↖  
Eine Menge  $S$  ist „sparse“, falls ein Polynom  $s$  existiert, so dass für alle  $n$   
 $|S \cap \{0, 1\}^{\leq n}| \leq s(n)$

# Circuit Complexity



Theorem  $P/poly = \bigcup \{P(S) : S \text{ sparse}\}.$

Angenommen, dass  $A$  in  $P/poly$  mit einer polynomial beschränkten „advice“-Funktion  $f$ , so können wir eine „sparse“ Menge definieren, durch

$$S = \{\langle 0^n, z \rangle : z \sqsubseteq f(n)\}$$

„ $z$  ist Präfix von  $f(n)$ “

Gilt umgekehrt  $A = M^S$ , mit einer „sparsen“ Menge  $S$  und  $M$  läuft  $p(n)$  Schritte. Dann können für jede Länge von  $x$  die Elemente  $S \cap \{0, 1\}^{\leq p(|x|)}$  in einen „advice“-String  $f(|x|)$  geschrieben werden und dieser statt des Orakels verwendet werden.

# Circuit Complexity



Die Klasse  $P/poly$  ist insofern etwas „seltsam“ als dass z.B. gilt:

**Theorem**  $P/poly$  contains non-recursive languages.

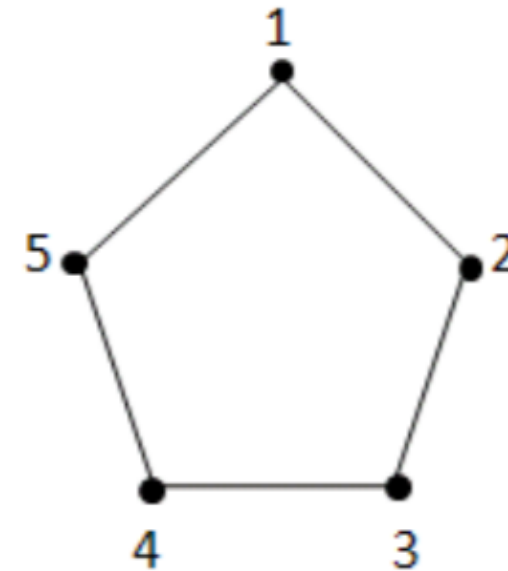
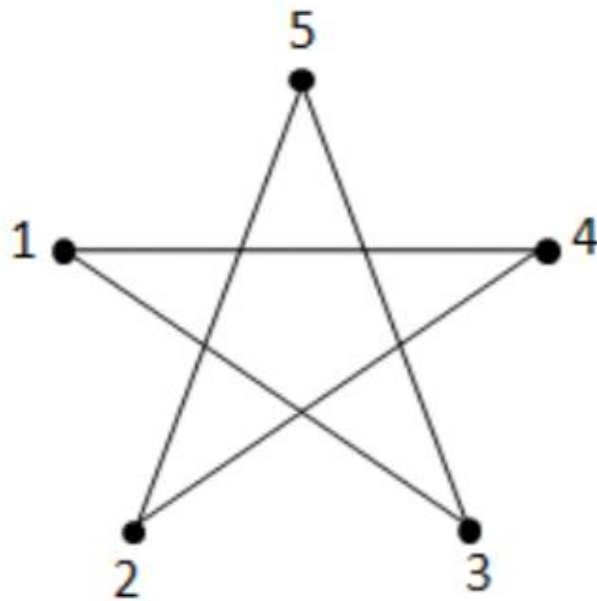
← (d.h. „unentscheidbar“)

Nichtsdestotrotz ist ein möglicher Ansatz um „ $P \neq NP$ “ zu zeigen, zu beweisen, dass  $NP$  nicht in  $P/poly$  enthalten ist ( $P$  ist natürlich trivialerweise in  $P/poly$ ). Sollte dies dennoch der Fall sein, so kann man z.B. zeigen, dass  $P \#$  auf die zweite Stufe kollabiert.

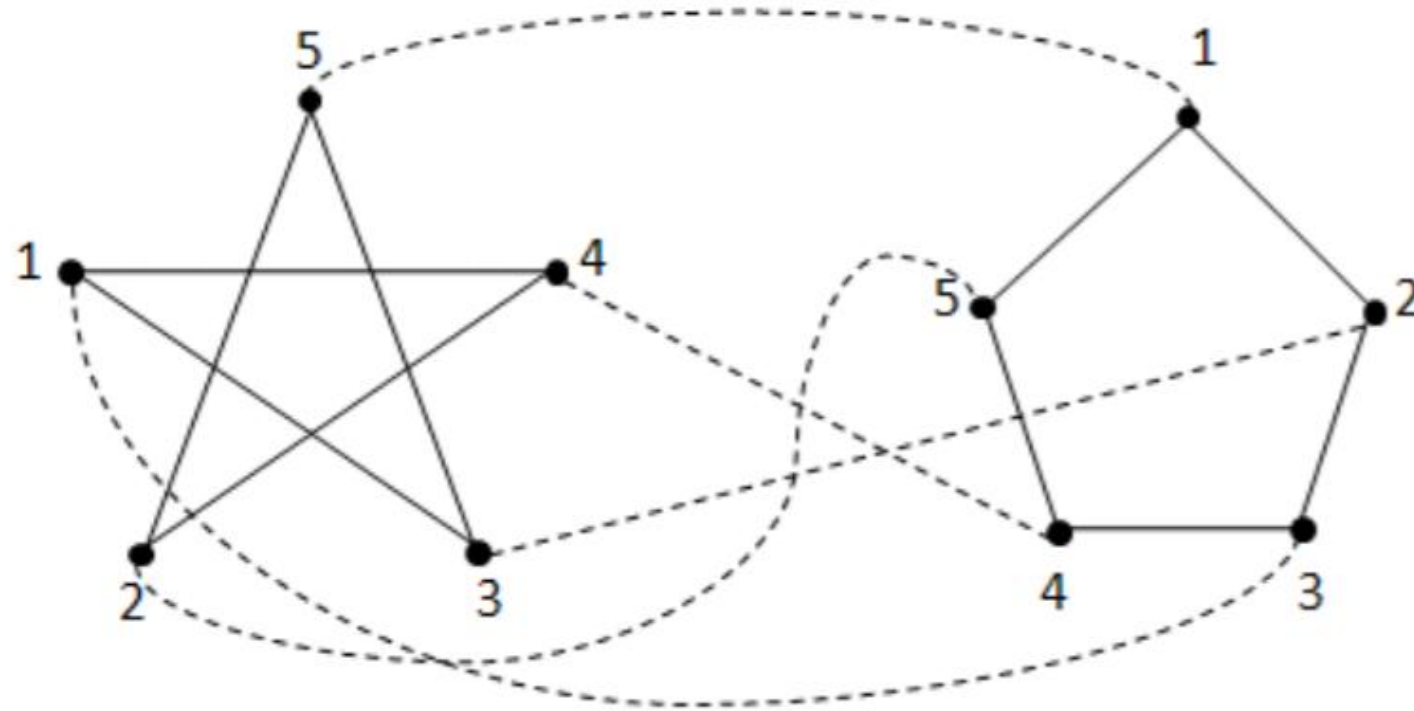
# Interactive Proofs



Sind diese beiden Graphen  
isomorph?



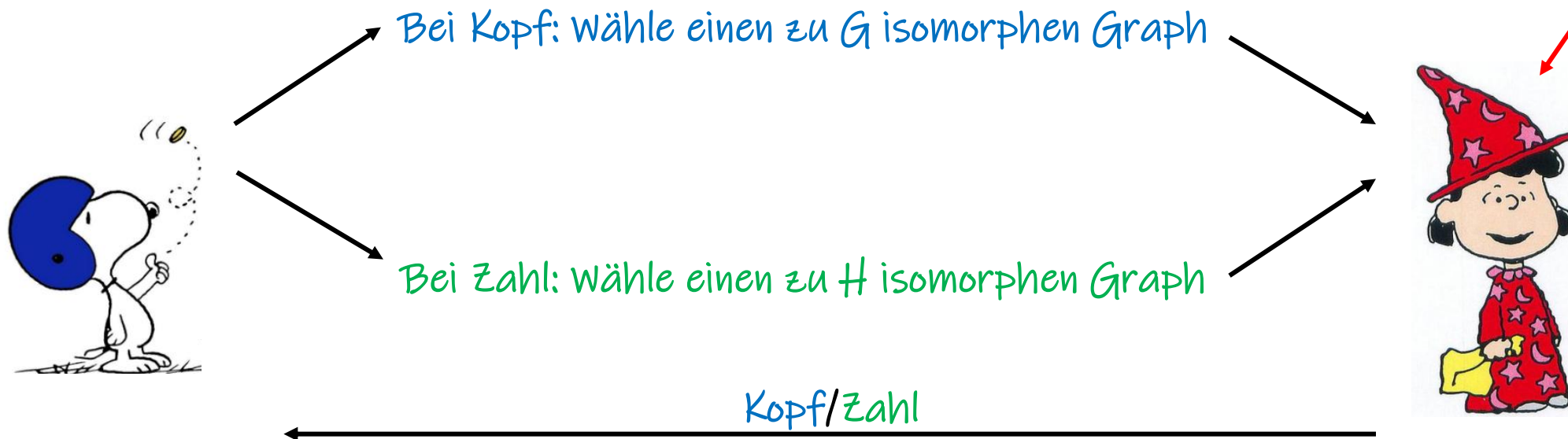
# Interactive Proofs



# Interactive Proofs



Aber wie beweise ich, dass zwei Graphen  $G$  und  $H$  nicht isomorph sind?



Lucy hat  
unbeschränkte  
Rechenpower

# Interactive Proofs



Ein Verifier ist eine Funktion  $V: \Sigma^* \times \Sigma^* \times \Sigma^* \longrightarrow \Sigma^* \cup \{\text{accept}, \text{reject}\}$

Ein Prover ist eine Funktion  $P: \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$

Prover und Verifier schicken sich eine Reihe von Nachrichten, wobei  $V$  akzeptiert (entsprechend, falls  $V$  ablehnt), falls:

1. for  $0 \leq i < k$ , where  $i$  is an even number,  $V(w, r, m_1 \# \dots \# m_i) = m_{i+1}$ ;
2. for  $0 < i < k$ , where  $i$  is an odd number,  $P(w, m_1 \# \dots \# m_i) = m_{i+1}$ ; and
3. the final message  $m_k$  in the message history is *accept*.

Wort der Länge  $n$

random string  
(der Länge  $p(n)$ )

Nachrichte

(der Länge  $\leq p(n)$ )

$k \leq p(n)$ , mit  $p$  ein Polynom

# Interactive Proofs



Say that language  $A$  is in **IP** if some **polynomial time** function  $V$  and **arbitrary function**  $P$  exist, where for every function  $\tilde{P}$  and string  $w$

d.h. der  
Prover hat  
„unbegrenzte  
Rechenpower“

1.  $w \in A$  implies  $\Pr[V \leftrightarrow P \text{ accepts } w] \geq \frac{2}{3}$ , and
2.  $w \notin A$  implies  $\Pr[V \leftrightarrow \tilde{P} \text{ accepts } w] \leq \frac{1}{3}$ .

die Wahrscheinlichkeit geht über alle random strings  
der Länge  $p(n)$ , wobei  $n$  die Länge von  $w$



# Interactive Proofs



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

**LEMMA**

$IP \subseteq PSPACE.$

# Interactive Proofs



Es genügt folgenden Wert mit Hilfe einer PSPACE-TM zu berechnen:

$$\Pr[V \text{ accepts } w] = \max_P \Pr[V \leftrightarrow P \text{ accepts } w]$$

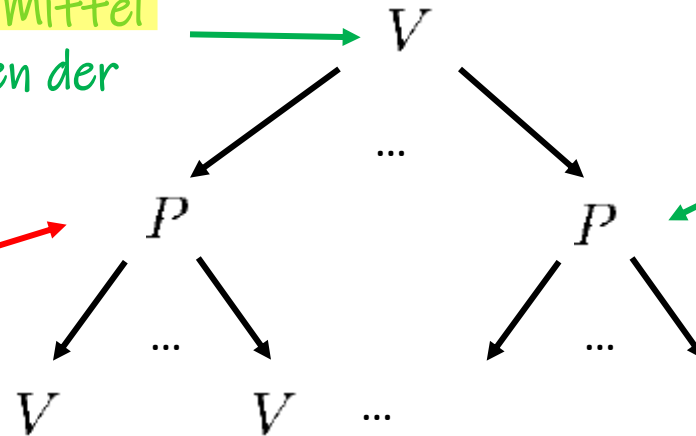
Dieser ist **mindestens 2/3**, falls  $w$  aus  $A$  und **höchstens 1/3**, falls  $w$  nicht aus  $A$  ist.

Das (simulierte) Protokoll sieht als Baum so aus:

bilde hier das **(gewichtete) Mittel**  
über die Wahrscheinlichkeiten der  
Kinder

bilde hier das  
**Maximum** über die  
Wahrscheinlichkeiten  
der Kinder

die Höhe des  
Baums ist  
höchstens  $p(n)$



# IP=PSPACE



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

**THEOREM**

$IP = PSPACE.$

# IP=PSPACE



Wie überzeugt ein Prover einen Verifier, dass eine „Quantified Boolean Function“ (QBF) erfüllbar ist?

Beispiel:

$$B = \forall x_1 [\bar{x}_1 \vee \exists x_2 \forall x_3 (x_1 \wedge x_2) \vee x_3]$$

Idee: Arithmetisierung der QBF

$$A = \prod_{z_1 \in \{0,1\}} \left[ (1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_2 + z_3) \right]$$

hier gilt  $A=2$

# IP=PSPACE



## Arithmetisierung einer QBF:

- (1) Replace each Boolean variable  $x_i$  by a new variable  $z_i$  which can range over the (positive and negative) integers  $\mathbb{Z}$ .
- (2) Replace each occurrence of  $\bar{x}_i$  by  $(1 - z_i)$ .
- (3) Replace  $\wedge$  by integer multiplication  $\cdot$ ,  $\vee$  by integer addition  $+$ , the universal quantification  $\forall x_i$  by the integer product  $\prod_{z_i \in \{0, 1\}}$ , and the existential quantifier  $\exists x_i$  by the integer sum  $\sum_{z_i \in \{0, 1\}}$ .

(wir nehmen an, dass alle Negationen in der QBF bis zu den Variablen durchgereicht wurden)

# IP=PSPACE



THEOREM     *A closed QBF  $B$  is **true** iff the value of its arithmetic form  $A$  is **nonzero**.*

# IP=PSPACE



$$B = \forall x_1 [\bar{x}_1 \vee \exists x_2 \forall x_3 (x_1 \wedge x_2) \vee x_3]$$

$$A = \prod_{z_1 \in \{0,1\}} \left[ (1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_2 + z_3) \right]$$

Wie überzeugt ein Prover einen Verifier, dass eine „arithmetisierte“ QBF einen Wert  $> 0$  hat?

Idee: bilde eine „funktionale Form“ von  $A$

$$A' = \left[ (1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_2 + z_3) \right]$$

und schreibe  $A'$  als Polynom

$$q(z_1) = z_1^2 + 1$$

Verifier  $V$   
überprüft, ob  
 $q(0) \cdot q(1) = 2$

# IP=PSPACE



Als nächstes wählt V eine Zufallszahl, z.B. „3“. Diese wird in  $A'$ , bzw  $q$  eingesetzt:

$$A'(z_1 = 3) = \left[ (1 - 3) + \sum_{z_2 \in \{0, 1\}} \prod_{z_3 \in \{0, 1\}} (3z_2 + z_3) \right]$$

$q(3) = 10$

Dieser Term wird noch abgezogen und nun muss der Prover den Verifier überzeugen, dass

$$A = \sum_{z_2 \in \{0, 1\}} \prod_{z_3 \in \{0, 1\}} (3z_2 + z_3) = 10 - (-2) = 12$$



# IP=PSPACE



$$A = \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (3z_2 + z_3)$$

bilde wie vorher

$$A' = \prod_{z_3 \in \{0,1\}} (3z_2 + z_3)$$

Zusammen mit dem Polynom

$$q(z_2) = 9z_2^2 + 3z_2$$

Verifier V  
überprüft, ob  
 $q(0)+q(1) = 12$

# IP=PSPACE



Als nächstes wählt V eine Zufallszahl, z.B. „2“. Diese wird in  $q$  eingesetzt:

$$q(2) = 9 \cdot 4 + 3 \cdot 2 = 42 = A = \prod_{z_3 \in \{0,1\}} (6 + z_3)$$

bilde wie vorher

$$A' = (6 + z_3)$$

Zusammen mit dem Polynom

$$q(z_3) = z_3 + 6$$

Verifier V überprüft, ob  $q(0) \cdot q(1) = 42$ , wählt eine Zahl, z.B. „5“ und überprüft selbst, ob

$$A'(z_3 = 5) = (6 + 5) = 11 = 5 + 6 = q(5)$$

# IP=PSPACE



Wie groß kann der „arithmetische Wert“ einer Formel werden?

Beispiel:

$$B = \forall x_1 \forall x_2 \cdots \forall x_{n-1} \exists x_n (x_n \vee \bar{x}_n)$$

**THEOREM** *Let  $B$  be a closed QBF of size  $n$ . Then the value of its arithmetic form  $A$  cannot exceed  $O(2^{2^n})$ .*

das kann ein polynomialer Verifier aber  
nicht überprüfen!

# IP=PSPACE



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

**THEOREM**     *Let  $B$  be a closed QBF of size  $n$ . Then there exists a prime  $p$  of length polynomial in  $n$  such that  $A \not\equiv 0 \pmod{p}$  iff  $B$  is true.*

# IP=PSPACE



Wie groß kann der Grad des Polynoms zu einer „funktionalen Form“ werden?

Beispiel:

$$B = \forall x_1 \forall x_2 \cdots \forall x_n (x_1 \vee x_2 \vee \cdots \vee x_n)$$

$$A = \prod_{z_1 \in \{0,1\}} \prod_{z_2 \in \{0,1\}} \cdots \prod_{z_n \in \{0,1\}} (z_1 + z_2 + \cdots + z_n)$$

$$q(z_1) = \prod_{z_2 \in \{0,1\}} \cdots \prod_{z_n \in \{0,1\}} (z_1 + z_2 + \cdots + z_n)$$

dieses Polynom hat den Grad  $2^{(n-1)}$ . Auch das könnte ein polynomialer Verifier nicht verarbeiten

# IP=PSPACE



*Definition.* A closed QBF is called *simple* if in the given syntactic representation every occurrence of each variable is separated from its point of quantification by *at most one universal quantifier* (and arbitrarily many other symbols).

$$\forall x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge \forall x_4 (x_2 \wedge x_3 \wedge x_4)]$$

diese QBF ist „simple“

$$\forall x_1 \forall x_2 [(x_1 \wedge x_2) \wedge \forall x_3 (\bar{x}_1 \wedge x_3)]$$

diese nicht

# IP=PSPACE



**THEOREM**     *Every QBF of size  $n$  can be transformed into an equivalent simple QBF whose size is **polynomial** in  $n$ .*

**THEOREM**     *If  $B$  is **simple**, then the degree of the polynomial  $q(z_i)$  that describes the functional form of  $A$  grows at **most linearly** with the size of  $B$ .*

# IP=PSPACE



Für  $a = A \bmod p$  geht der interaktive Beweis, dass  $a \neq 0$  ist so, dass  $P$  den Wert  $a$  und  $p$  an  $V$  schickt\*.  $A$  selbst kommt zwischenzeitlich in der Form  $A_1 + A_2$  oder  $A_1 * A_2$  vor, wobei nur  $A_2$  mit Summen- oder Produktzeichen beginnt. Ferner ist  $a_1 = A_1 \bmod p$ .

- (1) If  $A_2$  is empty,  $V$  stops and accepts the claim iff  $a = a_1$ .
- (2) If  $A_1$  is nonempty,  $V$  replaces  $A$  by  $A_2$ , and replaces  $a$  by  $a - a_1 \pmod{p}$  or  $a/a_1 \pmod{p}$  (depending on the operator that connects  $A_1$  and  $A_2$ ). If  $V$  tries to divide  $a$  by  $a_1 = 0 \pmod{p}$ , he stops and accepts the claim iff  $a = 0 \pmod{p}$ .
- (3) Otherwise,  $P$  sends the polynomial descriptions  $q(z_i)$  of  $A'$  to  $V$ .  $V$  checks that  $a = q(0) + q(1) \pmod{p}$  or  $a = q(0) \cdot q(1) \pmod{p}$  (depending on the first symbol of  $A_2$ ), sends a random  $r \in \mathbb{Z}_p$  to  $P$ , replaces  $A$  by  $A'(z_i = r) \pmod{p}$ , and replaces  $a$  by  $q(r) \pmod{p}$ .

\* $V$  kann in polynomialer Zeit überprüfen, ob  $p$  eine Primzahl ist



# IP=PSPACE



## THEOREM

- (1) *When  $B$  is true and  $P$  is honest,  $V$  always accepts the proof.*
- (2) *When  $B$  is false,  $V$  accepts the proof with negligible probability.*

# Probabilistische Turing Maschine



A **probabilistic Turing machine**  $M$  is a type of nondeterministic Turing machine in which each nondeterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch  $b$  of  $M$ 's computation on input  $w$  as follows. Define the probability of branch  $b$  to be

$$\Pr[b] = 2^{-k},$$

where  $k$  is the number of coin-flip steps that occur on branch  $b$ . Define the probability that  $M$  accepts  $w$  to be

$$\Pr[M \text{ accepts } w] = \sum_{\substack{b \text{ is an} \\ \text{accepting branch}}} \Pr[b].$$



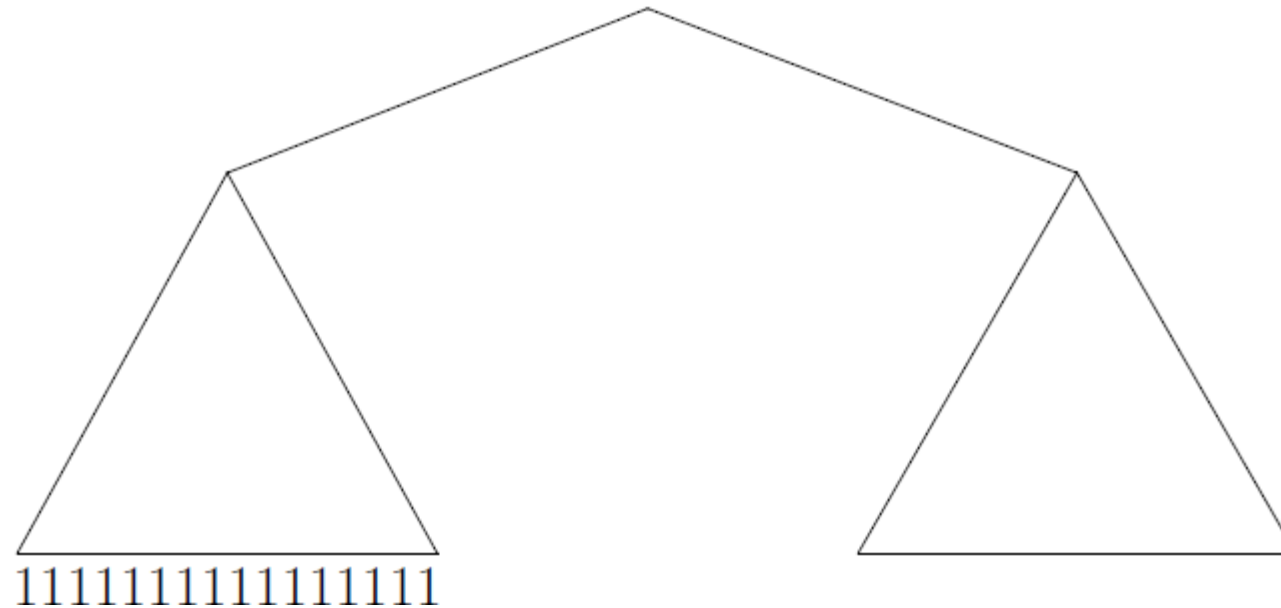
**Definition 6.1.1.** The class **PP**, for *probabilistic polynomial time*, is the class of sets  $L$  that are computed in polynomial time by a probabilistic machine such that

$$\begin{aligned} x \in L &\Leftrightarrow \text{the fraction of accepting paths is } > \frac{1}{2}, \\ x \notin L &\Leftrightarrow \text{the fraction of rejecting paths is } \geq \frac{1}{2}. \end{aligned}$$

# PP



Es gilt  $NP \subseteq PP \subseteq PSPACE$ , denn eine PSPACE Maschine kann alle Pfade „abklappen“, und eine NP Maschine kann leicht umgebaut werden, so dass, im Falle einer Akzeptanz, immer auch mehr als die Hälfte der Pfade akzeptiert:



# PP



Die Sprache  $\text{MAJ} = \{\varphi : \varphi \text{ is satisfied by more than } \frac{1}{2} \text{ of all assignments}\}$ , wobei  $\varphi$  eine Boolesche Funktion ist, ist offensichtlich in PP.

Wir zeigen, dass die Klasse PP auch die Sprache

$$\text{SA} = \{\langle \varphi, i \rangle : \varphi \text{ is satisfied by more than } i \text{ assignments}\}$$

enthält und zudem gilt:

**Theorem** *The sets MAJ and SA are PP-complete.*

Wir definieren für jede Eingabe  $\langle \varphi, i \rangle$ , für eine Formel mit  $m$  Variablen eine neue Formel mit  $m+1$  Variablen durch  $(y \wedge \varphi) \vee (\neg y \wedge \psi)$ , wobei  $\psi$  eine Formel ist, welche genau  $2^m - i$  erfüllende Belegungen hat. Genau dann hat nun  $\varphi$  mehr als  $i$  erfüllende Belegungen, falls die neue Formel mehr als  $i + 2^m - i = 2^m = \frac{1}{2}2^{m+1}$  solcher Belegungen besitzt und somit  $\text{SA} \leq_m^p \text{MAJ}$ .

Eine Sprache  $A$  aus PP, welche in  $p(n)$  entscheidet, ob ein Element  $x$  zu  $A$  gehört, hat im positiven Fall mehr als  $2^{p(n)-1}$  akzeptierende Pfade. Gemäß dem Cook-Levin-Theorem existiert zu diesem  $x$  dann auch eine Formel  $\varphi_x$ , und damit  $A \leq_m^p \text{SA}$ , mittels  $x \mapsto \langle \varphi_x, 2^{p(n)-1} \rangle$ .

# BPP



**Definition 6.1.5.** The class **BPP**, for *bounded probabilistic polynomial time*, is the class of sets  $L$  that are recognized in polynomial time by probabilistic machines with error probability bounded away from  $\frac{1}{2}$ , i.e. such that for some  $\varepsilon > 0$  and every  $x$ ,

$x \in L \Leftrightarrow$  the fraction of accepting paths is  $> \frac{1}{2} + \varepsilon$ ,

$x \notin L \Leftrightarrow$  the fraction of rejecting paths is  $> \frac{1}{2} + \varepsilon$ .

der entscheidende Unterschied  
ist hier, dass diese Konstante  
unabhängig von der Eingabe ist

# BPP



**Theorem**  $A \in \text{BPP}$  if and only if for all polynomials  $p$  there is a probabilistic Turing machine recognizing  $A$  in polynomial time with error probability  $\leq \frac{1}{2^{p(n)}}$ .

Wir nehmen an, dass die Maschine  $M$  die Elemente der Sprache  $A$  mit einer Fehlerwahrscheinlichkeit von  $\varepsilon < \frac{1}{2}$  akzeptiert. Für  $\delta = 1 - \varepsilon$  und ein Polynom  $p$ , wähle  $q(n) = c \cdot p(n)$ , mit einer Konstanten  $c$ , für die gilt

$$(4\varepsilon\delta)^c < \frac{1}{2}$$

Die Idee ist nun die Maschine  $m = 2q(n) + 1$  mal laufen zu lassen und genau dann zu akzeptieren, falls mehr als die Hälfte der Fälle akzeptiert.

Die Wahrscheinlichkeit, dass hierbei falsch entschieden wird liegt dann bei

$$\sum_{j=0}^{q(n)} \binom{m}{j} \delta^j \varepsilon^{m-j} \leq \delta^{\frac{m}{2}} \varepsilon^{\frac{m}{2}} \sum_{j=0}^{q(n)} \binom{m}{j} < \delta^{\frac{m}{2}} \varepsilon^{\frac{m}{2}} 2^m = (4\varepsilon\delta)^{\frac{m}{2}} \leq (4\varepsilon\delta)^{cp(n)} < \frac{1}{2^{p(n)}}$$

# BPP



**Theorem**  $BPP \subseteq P/poly$

Für  $L$  aus BPP dürfen wir annehmen, dass die Fehlerwahrscheinlichkeit der zugehörigen Maschine kleiner als  $2^{-n}$  ist. Damit ergibt sich dann aber:

$$Prob_r[\exists x \in \{0, 1\}^n : M(x, r) \neq \chi_L(x)] \leq \sum_{x \in \{0, 1\}^n} Prob_r[M(x, r) \neq \chi_L(x)] < 2^n \cdot 2^{-n} = 1.$$

„charakteristische Funktion“

von  $L$

Damit existiert aber mindestens ein  $r$ , so dass für alle  $x$  (der Länge  $n$ ) gilt  $M(x, r) = \chi_L(x)$ .



# BPP



Eine Sprache aus BPP besitzt sogar „ziemlich viele“ advice strings:

**Theorem** (Adleman) *The following are equivalent.*

- (i)  $A \in \text{BPP}$ ,
- (ii) *For any polynomial  $q$  there exist  $B \in \text{P}$  and a polynomial  $p$  such that for all  $n$ , among all strings of length  $p(n)$  there are at least*

$$2^{p(n)} \left( 1 - \frac{1}{2^{q(n)}} \right).$$

*correct advice strings  $y$  for  $A$  at length  $n$*

(D.h. dass  $x \in A \iff \langle x, y \rangle \in B$  für jedes  $x$  der Länge  $n$ )

# BPP



Theorem  $BPP \subseteq \Sigma_2^P \cap \Pi_2^P$

Es genügt zu zeigen, dass  $BPP \subseteq \Sigma_2^P$ , da BPP abgeschlossen gegenüber Komplementbildung ist.

Für eine Sprache  $L$  aus BPP dürfen wir zudem annehmen, dass es eine Maschine gibt, welche  $L$  mit einer Fehlerwahrscheinlichkeit  $\leq 1/2^n$  akzeptiert. Sei  $l = p(n)$  die Laufzeit von  $M$  (die Länge des längsten Pfads),  $x$  ein binärer String der Länge  $n$  und  $r$  ein Pfad (dargestellt als binärer String der Länge  $l$ ). Wir betrachten die Menge  $A(x)$  der Pfade, welche  $x$  akzeptieren:

$$A(x) = \{r \in \{0, 1\}^l : M(x, r) \text{ accepts}\}$$

Für ein  $t$  aus  $\{0, 1\}^l$  definieren wir die Translation von  $A$ ,  $A(x) + t = \{r + t : r \in A(x)\}$ , wobei die Addition einem „xor“ entspricht.

# BPP



*Claim 1.* If  $|A(x)| \geq 2^l(1 - \frac{1}{2^n})$  then there exist  $t_1 \dots t_l$ , all of length  $l$ , such that  $\bigcup_i A(x) + t_i = \{0, 1\}^l$ .

Wir wählen  $t_1, \dots, t_l$  zufällig und zeigen, dass für die Menge  $S = \bigcup_i A(x) + t_i$  gilt:

$$\Pr[r \notin S] = \prod_{i=1}^l \Pr[r \notin A(x) + t_i] \leq \left(\frac{1}{2^n}\right)^l$$

Demnach gilt also

$$\Pr[\exists r \ r \notin S] \leq \sum_r 2^{-nl} = 2^{l-nl} \leq 2^{-n}$$

und somit  $\Pr[S = \{0, 1\}^l] \geq 1 - \frac{1}{2^n}$ , d.h. es existieren  $t_1, \dots, t_l$ , so dass gilt  $S = \{0, 1\}^l$ .

# BPP



*Claim 2.* If  $|A(x)| \leq 2^l \frac{1}{2^n}$  then there do not exist  $t_1 \dots t_l$  of length  $l$  such that  $\bigcup_i A(x) + t_i = \{0, 1\}^l$ .

Dies folgt aus der Beobachtung, dass für jedes  $i$  gilt  $|A(x) + t_i| \leq 2^{l-n}$  und damit  $l \cdot 2^{l-n} < 2^l$ .

Damit ergibt sich nun die gesuchte Darstellung in  $\Sigma_2^P$ :

$$x \in L \Leftrightarrow \exists t_1 \dots t_l \in \{0, 1\}^l \forall r \in \{0, 1\}^l \bigvee_{1 \leq i \leq l} M(x, r + t_i) \text{ accepts}$$

# BPP



Hochschule für  
Wirtschaft und Recht Berlin  
Berlin School of Economics and Law

Gibt es BPP-vollständige Probleme?

# Parity P



sprich: „Parity P“

**Definition 16.** A language  $L \in \oplus\mathbf{P}$  if there exists a poly-time deterministic TM  $M$  and polynomial  $p$  such that

$$x \in L \iff \#\{u \in \{0, 1\}^{p(|x|)} \mid M(x, u) = 1\} \text{ is odd.}$$

Hier interessiert also nur  
das niedrigste Bit



# Toda's Theorem(e)

**Theorem** (Toda's "First" Theorem).  $PH \subseteq BPP^{\oplus P}$

**Theorem** (Toda's "Second" Theorem).  $BPP^{\oplus P} \subseteq P^{PP}$

**Corollary** ("Toda's Theorem").  $PH \subseteq P^{PP}$



# Toda's Theorem(e)

Wir zeigen zumindest, dass  $NP$  in  $BPP^{\oplus P}$  enthalten ist:

Der folgende probabilistische Algorithmus entscheidet mit Hilfe eines  $\oplus P$  Orakels, ob eine (aussagenlogische) Formel  $F$  erfüllbar ist.

Eingabe:  $F$  (eine aussagenlogische Formel) mit  $n$  Variablen

1. Wähle eine zufällige Zahl  $k$  aus der Menge  $\{0, 1, \dots, n-1\}$
2. Wähle zufällige Teilmengen  $S_1, \dots, S_{k+2}$  der Menge  $\{1, \dots, n\}$
3. Entscheide mit Hilfe des Orakels, ob die Konjunktion von  $F$  und Formeln  $[S_i]$  eine ungerade Anzahl an erfüllenden Belegungen besitzt

Diese Formel ist genau dann wahr,  
falls eine ungerade Anzahl der  
gewählten  $x_j$  wahr ist, mit  $j$  aus  $S_i$





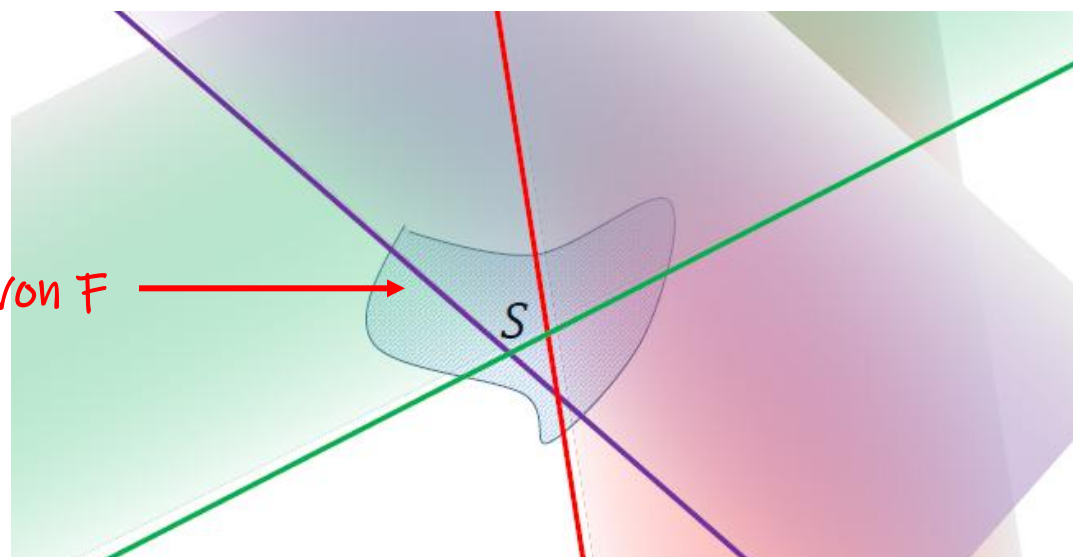
# Toda's Theorem(e)

Zunächst ist klar, dass, falls  $F$  unerfüllbar, der Algorithmus niemals „ungerade“ ausgeben kann.

Wir nehmen nun an, dass  $F$   $m$  ( $> 0$ ) erfüllende Belegungen besitzt. Mit einer Wahrscheinlichkeit von **mindestens  $1/n$**  wird  $k$  so gewählt, dass gilt  $2^k \leq m \leq 2^{k+1}$ .

Für eine erfüllende Belegung  $b$  von  $F$  (wobei  $b \neq 0 \dots 0$ ) gilt dann also, dass diese jede Formel  $[S_i]$  mit einer Wahrscheinlichkeit von  $1/2$  „übersteht“.

Erfüllende Belegungen von  $F$





# Toda's Theorem(e)

Zunächst ist klar, dass, falls  $F$  unerfüllbar, der Algorithmus niemals „ungerade“ ausgeben kann.

Wir nehmen nun an, dass  $F$   $m$  ( $> 0$ ) erfüllende Belegungen besitzt. Mit einer Wahrscheinlichkeit von **mindestens  $1/n$**  wird  $k$  so gewählt, dass gilt  $2^k \leq m \leq 2^{k+1}$ .

Für eine erfüllende Belegung  $b$  von  $F$  (wobei  $b \neq 0 \dots 0$ ) gilt dann also, dass diese jede Formel  $[S_i]$  mit einer Wahrscheinlichkeit von  $1/2$  „übersteht“.

Damit „übersteht“ die Belegung  $b$  dann die Konjunktion von  $k+2$  Formeln  $[S_1], \dots, [S_{k+2}]$  mit einer Wahrscheinlichkeit von  $1/2^{k+2}$ .

Die Wahrscheinlichkeit, dass ein bestimmtes  $b$  die einzige Belegung ist, welche „überlebt“ ist somit:

$$\frac{1}{2^{k+2}} \cdot \left(1 - \sum_{b'} \frac{1}{2^{k+2}}\right) = \frac{1}{2^{k+2}} \left(1 - \frac{m-1}{2^{k+2}}\right) \geq \frac{1}{2^{k+2}} \cdot \left(1 - \frac{2^{k+1}}{2^{k+2}}\right) = \frac{1}{2^{k+3}}$$



# Toda's Theorem(e)

Damit ergibt sich eine Wahrscheinlichkeit, dass eine einzige Belegung „überlebt“

$$\sum_b \frac{1}{2^{k+3}} = \frac{m}{2^{k+3}} \geq \frac{2^k}{2^{k+3}} = \frac{1}{8}$$

und somit insgesamt  $\frac{1}{8n}$ , da  $k$  ja „nur“ mit Wahrscheinlichkeit  $1/n$  die „richtige Wahl“ ist.

Um daraus nun einen BPP Algorithmus zu machen, müssen wir mehrere solcher „geratenen“ Formeln (aber alle mit der selben Formel  $F$  als Bestandteil) durch „oder“ verknüpfen. Im Falle von  $\oplus P$  ist sogar möglich: bei der Konjunktion von zwei Formeln multipliziert sich die Anzahl der Belegungen (ungerade mal ungerade = ungerade) und eine Negation ist möglich durch hinzufügen einer „Dummy-Belegung“:

$$(F \wedge y) \vee (x_1 \wedge \cdots \wedge x_n \wedge \neg y)$$

← Diese Formel hat genau eine Belegung mehr als  $F$

und damit auch eine Disjunktion.



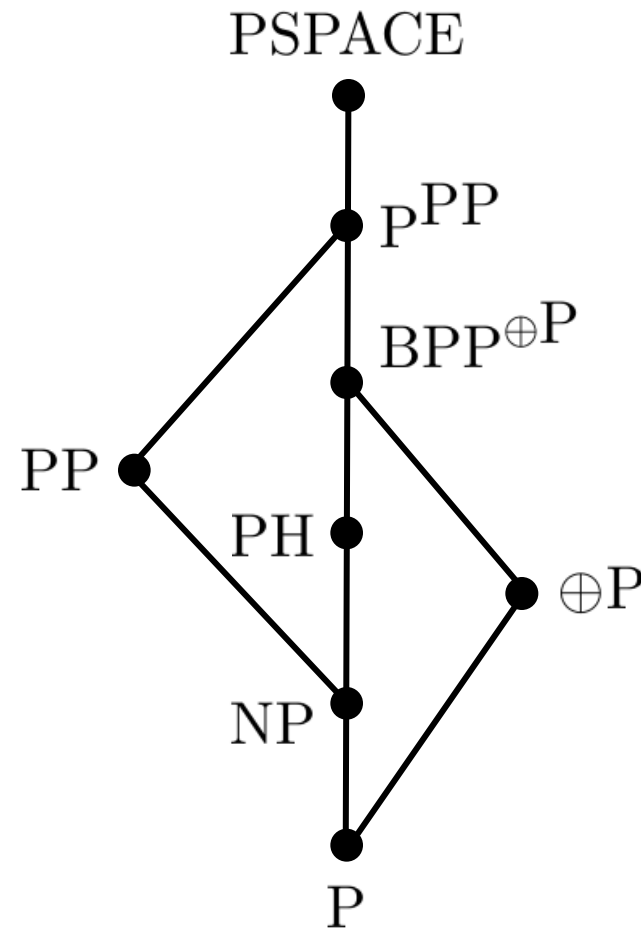
# Toda's Theorem(e)

Insgesamt lässt sich damit die Fehlerwahrscheinlichkeit des Algorithmus durch Verknüpfung von  $t$  Formeln „beliebig“ senken:

$$\left(1 - \frac{1}{8n}\right)^t = \left(\left(1 - \frac{1}{8n}\right)^{8n}\right)^{\frac{t}{8n}} \leq \left(\frac{1}{e}\right)^{\frac{t}{8n}} < \varepsilon$$

d.h. nach geeigneter Wahl von  $t$  ist als die Wahrscheinlichkeit, dass der Algorithmus korrekt entscheidet (ob  $F$  erfüllbar ist) größer als  $1 - \varepsilon$ .

# Übersicht



da bisher nicht bekannt ist, ob  
 $P \neq PSPACE$  ist, ist auch  
(noch) nicht klar, welche  
Klassen davon echte  
Teilklassen sind