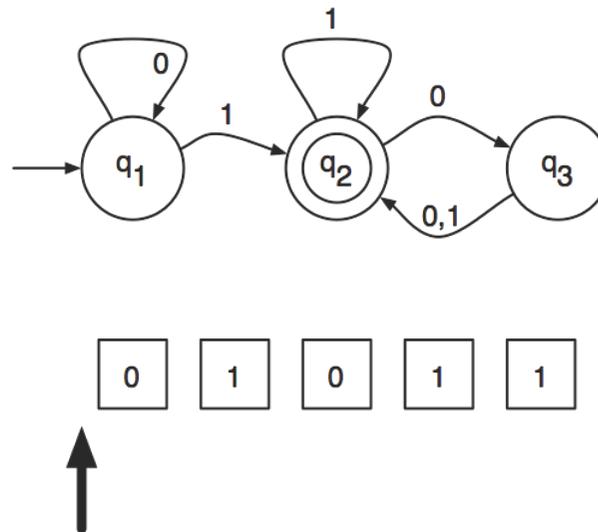


Es fängt ganz harmlos an

- Ein **Alphabet** Σ besteht aus einer **endlichen Menge von Zeichen**, z.B.
 - $\Sigma_1 = \{a,b,c\}$
 - $\Sigma_2 = \{0,1\}$
 - $\Gamma = \{0,1,x,y,z\}$
- Eine **Zeichenkette (String/Wort)** ist eine **endliche Folge (Tupel) von Zeichen**, z.B.
 - $w = abba$
 - Notation: $w_1=a, w_2=b, w_3=b, w_4=a$
 - Die Länge eines Worts wird mit $|w|$ beschrieben: $|w| = 4$
- Σ^* bezeichnet die Menge aller Zeichenketten über Alphabet Σ
 - z.B.: “abba” $\in \{a,b\}^*$
 - Die leere Zeichenkette wird mit ε bezeichnet.
 - Es gilt: $|\varepsilon| = 0$
- Eine **Teilmenge von Σ^*** wird als **Sprache** bezeichnet

- Eine Sprache $L \subseteq \Sigma^*$ muss nun irgendwie beschrieben werden.
 - z.B. durch einen **regulären Ausdruck**: (0^*10^*)
 - \emptyset ist ein regulärer Ausdruck.
 - ε ist ein regulärer Ausdruck.
 - $\forall a_i \in \Sigma$ ist a_i ein regulärer Ausdruck.
 - Sind x und y reguläre Ausdrücke, so auch $x \cup y$, (xy) und x^* .
 - Es gibt keine weiteren regulären Ausdrücke.
 - z.B. durch eine **Problembeschreibung**:
 - **Definition**: Ein *Entscheidungsproblem* ist ein input-output Tupel mit
 - geg.**: Kodierung eines Inputs einer Instanz, mittels Alphabet Σ
 - ges.**: ja/nein
 - Die Teilmenge aller Inputs, für die die Antwort “ja” ist, ist offenbar eine Sprache

- Die Frage, ob ein $w \in \Sigma^*$ ein Wort aus einer Sprache $L \subseteq \Sigma^*$ ist, kann unterschiedlich schwierig zu lösen sein
 - Bsp. 1: In einem sehr einfachen Fall durch einen endlichen Automaten:
 $0^*1(1|0(0|1))^*$



Formal ist ein deterministischer endlicher Automat (DEA) ein 5-Tupel

Def: Ein (deterministischer) endlicher Automat (**DFA**) ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei

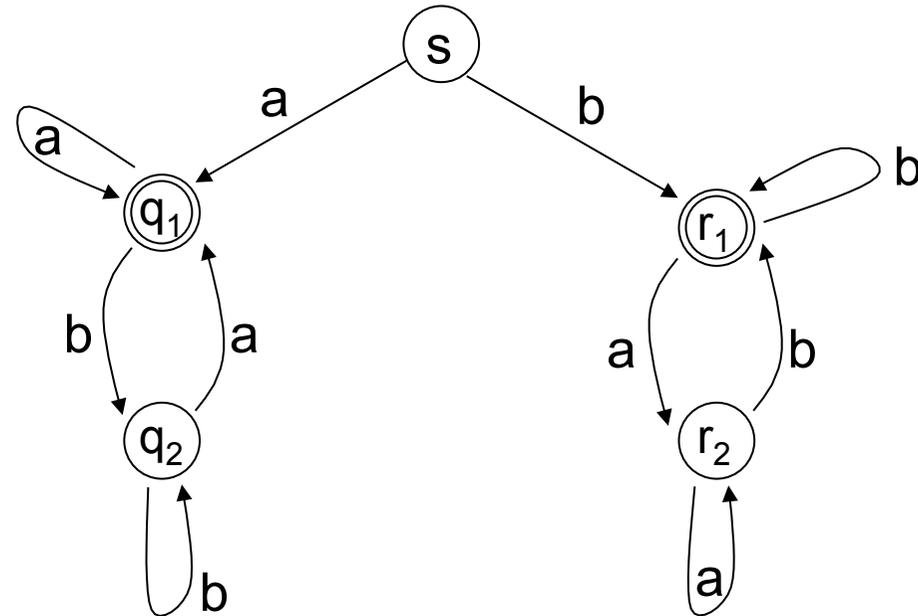
- Q eine endliche Menge von Zuständen ist,
- Σ ein endliches Alphabet
- $\delta: Q \times \Sigma \rightarrow Q$ die Übergangsfunktion,
- q_0 der Startzustand und
- $F \subseteq Q$ die Menge akzeptierender Endzustände.

Sprachbeschreibungen und Maschinen

$A := (Q, \Sigma, \delta, q_0, F)$,
 $Q := \{s, q_1, q_2, r_1, r_2\}$,
 $\Sigma := \{a, b\}$,
 $F := \{q_1, r_1\}$
 $q_0 = s$

$L(A) = \{w_1 w_2 \dots w_n : w_i \in \{a, b\}, w_1 = w_n\}$

δ	s	q_1	q_2	r_1	r_2
a	q_1	q_1	q_1	r_2	r_2
b	r_1	q_2	q_2	r_1	r_1

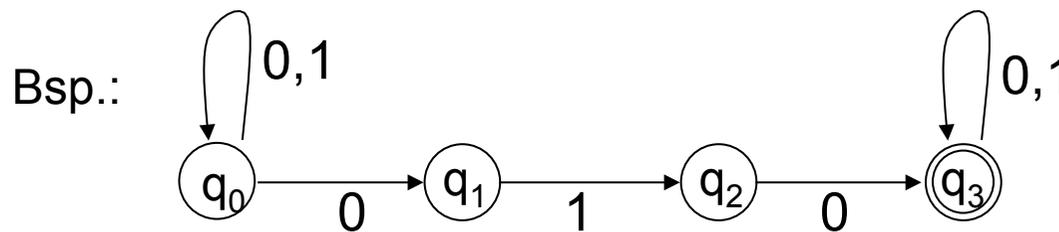


Sprachbeschreibungen und Maschinen

**Formal ist ein nichtdeterministischer endlicher Automat (NEA)
(ohne ϵ -Übergänge) ein 5-Tupel**

Def: Ein (nichtdeterministischer) endlicher Automat (**NFA**) ist ein 5-Tupel
 $(Q, \Sigma, \delta, q_0, F)$, wobei

- Q eine endliche Menge von Zuständen ist,
- Σ ein endliches Alphabet
- $\delta: Q \times \Sigma \rightarrow 2^Q$ die Übergangsfunktion,
- q_0 der Startzustand und
- $F \subseteq Q$ die Menge akzeptierender Endzustände.



	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	-	$\{q_2\}$
q_2	$\{q_3\}$	-
q_3	$\{q_3\}$	$\{q_3\}$

Sprachbeschreibungen und Maschinen

Satz: Sei N ein NFA und $L = L(N)$. Dann gibt es einen DFA A mit $L(A) = L$

Beweis: Sei $N = (Q, \Sigma, \delta, q_0, F)$. Die folgende Konstruktion heißt auch “Potenzmengenkonstruktion”. Um $A = (Q', \Sigma, \delta', q'_0, F')$ zu definieren setzen wir:

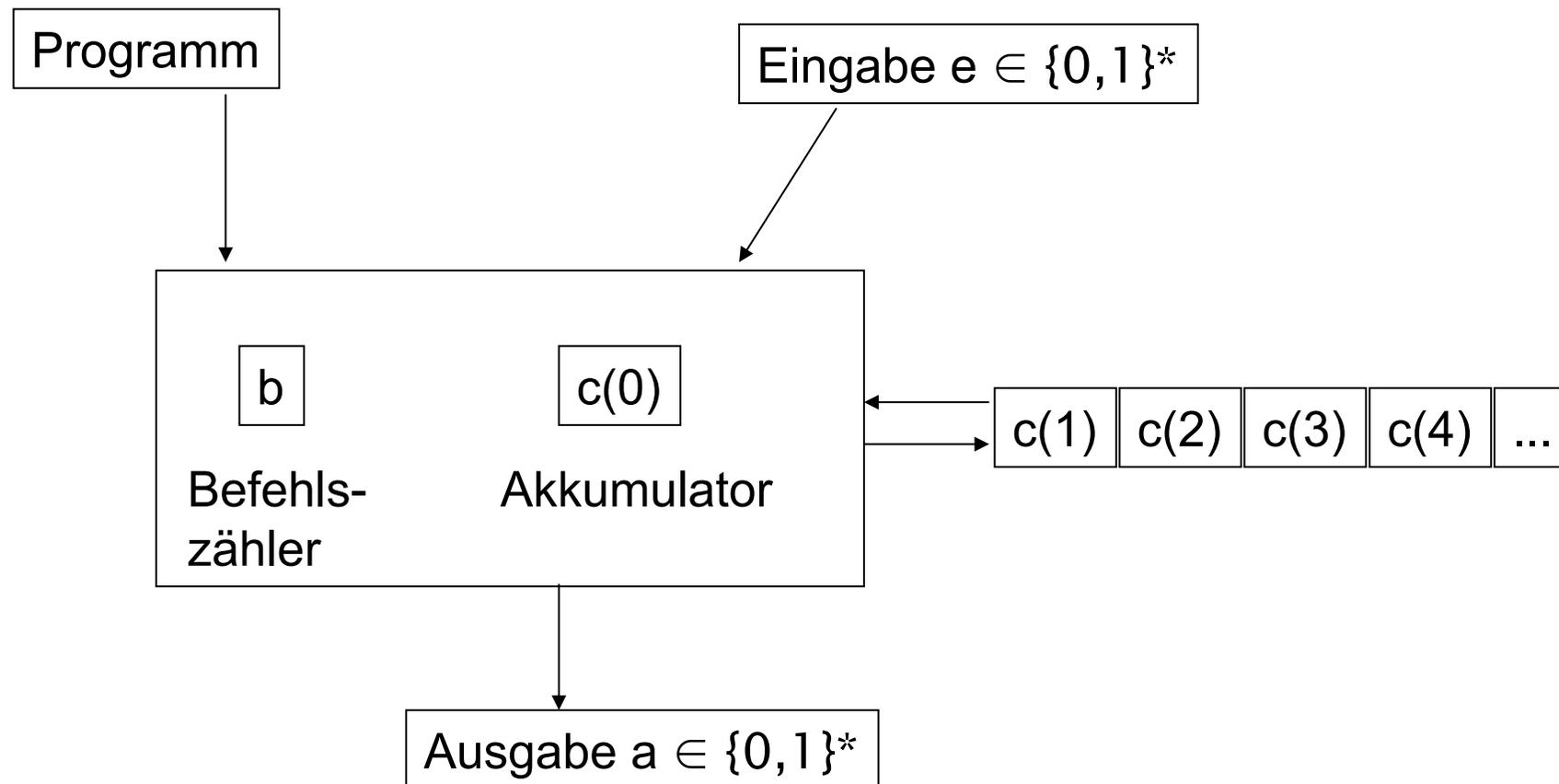
- $Q' = 2^Q$ (Potenzmenge von Q)
- $q'_0 = \{q_0\}$
- $F' = \{R \in Q' \mid R \cap F \neq \{\}\}$
- $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a) = \{q \in Q \mid \text{es gibt ein } r \in R \text{ mit } q \in \delta(r, a)\}$

Dann gilt:

$$\begin{aligned} w \in L(N) &\Leftrightarrow \delta(q_0, w) \cap F \neq \{\} \\ &\Leftrightarrow \delta(q'_0, w) \in F' \\ &\Leftrightarrow w \in L(A) \end{aligned}$$

Hierbei bedeutet $\delta(q, w)$, dass die Übergangsfunktion δ mehrfach auf das Wort w angewendet wird, Buchstabe für Buchstabe und startend bei Zustand q .

Random Access Maschinen



Random Access Maschinen

Ein/Ausgabe

read $c(0) := \text{head}(e); e := e \setminus \text{head}(e); b := b+1; \text{ falls } |e| > 0$
 $c(0) := \text{EOF}; b := b + 1; \qquad \qquad \qquad \text{sonst}$

write $a := a \ c(0); b := b + 1;$

Arithmetik

add x $c(0) := c(0) + c(x); b := b + 1;$

sub x $c(0) := c(0) - c(x); b := b + 1; \text{ falls } c(x) < c(0)$
 $c(0) := 0; \qquad \qquad \qquad b := b + 1; \text{ sonst}$

c add x $c(0) := c(0) + x; \quad b := b + 1;$

c sub x analog: Operation mit Konstante c

Random Access Maschinen

Sprünge

goto j $b := j;$

if ($c(0) R i$) *then goto* j; $b :=$ $\left\{ \begin{array}{ll} j & \text{falls } c(0) R i \\ b+1 & \text{sonst} \end{array} \right.$, $R \in \{<, >, =, \leq, \geq\}$

end Programm hält

Speicherzugriffe

direkt

load x $c(0) := c(x); b := b + 1;$

store x $c(i) := c(0); b := b + 1;$

indirekt

iload x $c(0) := c(c(x)); b := b + 1;$

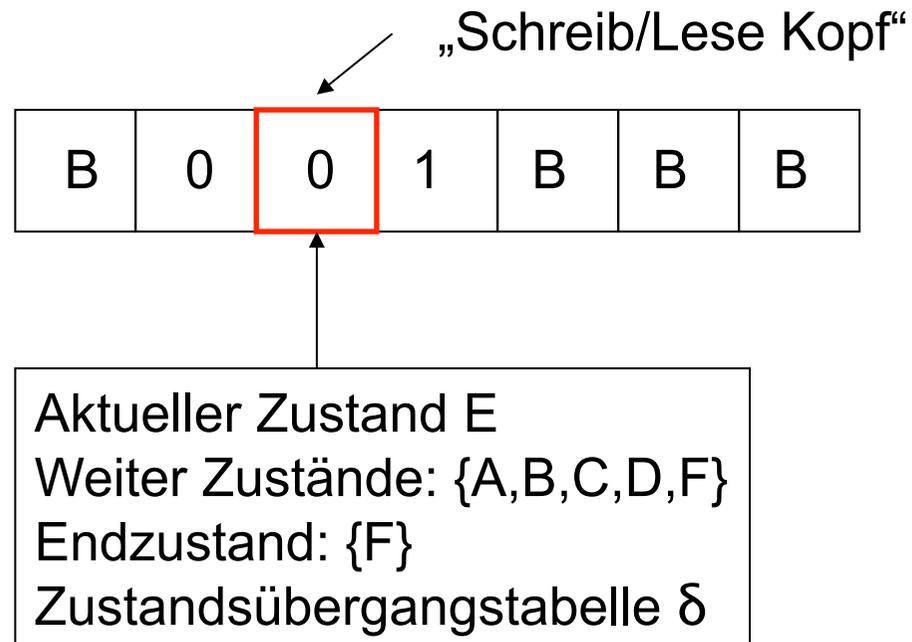
istore x $c(c(i)) := c(0); b := b + 1;$

- Die Frage, ob ein $w \in \Sigma^*$ ein Wort aus einer Sprache $L \subseteq \Sigma^*$ ist, kann unterschiedlich schwierig zu lösen sein
 - **Bsp 2.: In einem sehr komplizierten Fall ist sie nicht entscheidbar:**
 - **geg:** Codierung einer Random Access Machine (RAM, das entspricht in etwa einem herkömmlicher Computer mit unendlich viel Speicher), sowie ein $w \in \Sigma^*$
Frage: Hält die RAM bei Eingabe w ?

“nicht entscheidbar” heisst: es gibt keinen Algorithmus, der für alle Instanzen das Problem lösen kann. (faszinierende Nebeneffekte, Busy Beaver)

- **Formal ist eine (1-Band) Turingmaschine ein 6-Tupel**
- **Def:** Eine (deterministische 1-Band) Turingmaschine ist ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, F)$, wobei
 - Q eine endliche Menge von Zuständen ist,
 - Σ ein endliches Alphabet
 - $\Gamma := \Sigma \cup \{B\}$, B das so genannte Blank-Symbol
 - $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, N, L\}$ die (partielle) Übergangsfunktion,
 - q_0 der Startzustand und
 - $F \subseteq Q$ die Menge akzeptierender Endzustände.

Turingmaschinen



„Programm“: Falls die Turingmaschine im Zustand q ist und das Zeichen a liest,

dann gehe in den Zustand q' , überschreibe a durch a' , und bewege den Kopf nach rechts, links oder gar nicht.

Schreibweise: $\delta(q, a) = (q', a', R)$

Turingmaschinen, Beispiel

Geg: $\#x\$y$ mit $x,y \in \Sigma^*$ **Frage:** Ist x Präfix von y ?

$A := (Q, \Sigma, \delta, q_0, F),$
 $Q := \{q_0, q_1, q_2, q_3, q_4, q_5, q_e, q_f\},$
 $\Sigma := \{0, 1, \#, \$\},$
 $\Gamma := \Sigma \cup \{B\}$
 $F := \{q_f\}$
 q_0

δ	0	1	B	#	\$
q_0	$(q_0, 0, L)$	$(q_0, 1, L)$	(q_1, B, R)	$(q_0, \#, L)$	$(q_0, \$, L)$
q_1	$(q_2, \#, R)$	$(q_4, \#, R)$	(q_f, B, N)	$(q_1, \#, R)$	$(q_f, \$, N)$
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$			$(q_3, \$, R)$
q_3	$(q_0, \$, L)$	$(q_e, 1, N)$	(q_e, B, N)	$(q_e, \#, N)$	$(q_3, \$, R)$
q_4	$(q_4, 0, R)$	$(q_4, 1, R)$			$(q_5, \$, R)$
q_5	$(q_e, 1, N)$	$(q_0, \$, L)$	(q_e, B, N)	$(q_e, \#, N)$	$(q_5, \$, R)$
q_e					
q_f					

q_0 : Laufe nach links, bis ein B kommt, gehe dann 1 nach rechts

q_1 : lauf nach rechts, bis kein # mehr kommt. B oder \$ sind gut, gehe dann in q_f .

sonst merke das nächste Zeichen und ersetze es durch #. Gehe nach q_2 oder q_4 .

q_2 : das gemerkte Zeichen war eine 0. Lauf nach rechts zum ersten \$. Gehe dann in q_3 .

q_3 : lauf durch die \$-Zeichen. Wenn am Ende der \$ eine 0 steht, ist das ok., gehe nach q_0 , sonst gehe nach q_e .

q_4 : das gemerkte Zeichen war eine 1. Lauf nach rechts zum ersten \$. Gehe Dann in q_5 .

q_5 : lauf durch die \$-Zeichen. Wenn am Ende der \$ eine 1 steht, ist das ok., gehe nach q_0 , sonst gehe nach q_e .

- **Eine Mehrband- Turingmaschine ist ein 6-Tupel**
- **Def:** Eine (deterministische Mehrband) Turingmaschine ist ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, F)$, wobei
 - Q eine endliche Menge von Zuständen ist,
 - Σ ein endliches Alphabet
 - $\Gamma := \Sigma \cup \{B\}$, B das so genannte Blank-Symbol
 - $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{R, N, L\}^k$ die (partielle) Übergangsfunktion,
 - q_0 der Startzustand und
 - $F \subseteq Q$ die Menge akzeptierender Endzustände.

Nichtdeterministische Turingmaschinen



- Eine nichtdeterministische Turingmaschine (NTM) ist definiert, wie eine deterministische Turingmaschine, nur dass δ eine Übergangsrelation und keine Funktion ist.
- $\delta: Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{R,N,L\}}$ ist die Übergangsrelation.
- Bsp.: Wenn die TM in Zustand q ist, und ein a liest, und $\delta(q,a) = \{(q',b,R), (q'', a, L)\}$ ist, dann ist die nichtdeterministische TM im nächsten Schritt entweder in Zustand q' , nachdem sie ein b geschrieben hat, oder sie ist in Zustand q'' nachdem sie ein a schrieb.
- Die Laufzeit einer NTM ist definiert als die Länge des kürzesten Berechnungsweges, der in einem akzeptierenden Endzustand endet.

Theorem 1: RAM und Turingmaschine können sich gegenseitig simulieren.

Theorem 2:

Church-Turing Hypothese:

Die von jeglicher Maschine berechenbaren
Funktionen sind genau die,
die von Turingmaschinen berechenbar sind.

Turingmaschinen

Def.:

Eine **Sprache** L heißt **entscheidbar**, wenn es eine Turingmaschine gibt, die zu jeder Eingabe $w \in \Sigma^*$ nach endlicher Zeit anhält, und genau dann in einem akzeptierenden Zustand endet, wenn $w \in L$ gilt.

Eine **Sprache** L heißt **semi-entscheidbar**, wenn es eine Turingmaschine gibt, die zu jeder Eingabe $w \in L$ nach endlicher Zeit in einem akzeptierenden Endzustand anhält.

Eine **Funktion** f heißt **berechenbar**, wenn es eine Turingmaschine gibt, die für alle Eingaben x , die aus dem Definitionsbereich von f stammen nach endlich vielen Schritten anhält und $f(x)$ auf das Band schreibt.

Unentscheidbarkeit

Gibt es unentscheidbare Sprachen?

Ja, denn es gibt nur abzählbar unendlich viele Turingmaschinen, aber überabzählbar viele Sprachen $L \subseteq \{0,1\}^*$

Begründung mit Hilfe des Cantor'schen Diagonalisierungsverfahrens:

	M_1	M_2	M_3	M_4	...	→
0	n	j	j	n		
1	n	n	n	j		
01	j	j	j	n		
...						
x_i						

Eintrag $(M_i, x_k) = „j“$ bedeutet, dass x_k aus der Sprache $L(M_i)$ ist. Sei nun L die Sprache, die genau aus den Wörtern besteht, bei denen beim Eintrag (M_i, x_i) „n“ steht. L gehört zu keiner der aufgeführten TMs.

Berechenbarkeit

Gibt es Funktionen, die nicht von einer Turingmaschine berechnet werden können?

Ja.

Die Busy-Beaver Funktion $\Sigma(n)$ ist definiert als die Anzahl der Einsen, die eine Champion-Turingmaschine auf ein zu Beginn leeres Band ausgibt, wobei n die Anzahl der erlaubten Zustände darstellt. Die TM muss irgendwann halten. Wir gehen weiterhin davon aus, dass diese Einsen alle zusammenhängend sein müssen.

Beweis:

Berechenbarkeit

Annahme: Die Busy-Beaver Funktion $\Sigma(n)$ ist berechnbar, und $\text{EVAL}\Sigma$ ist die TM, die $\Sigma(n)$ berechnet. Bei einer Eingabe von n Einsen schreibt sie $\Sigma(n)$ Einsen auf das Band und hält dann an.

Im folgenden definieren wir 4 Hilfs-TMs.

Sei **INC** eine TM, die bis zum ersten B nach rechts läuft, dort eine 1 schreibt und dann hält.

DOUBLE sein eine andere TM, die die Anzahl Einsen, die sich auf dem Band befinden verdoppelt. **DOUBLE** berechnet also zu Eingabe n $n+n$.

Wir bilden nun eine neue TM: **DOUBLE | EVAL Σ | INC**
Die Anzahl der Zustände dieser Maschine sei n_0

Berechenbarkeit

Sei $CREATE_{n_0}$ eine weitere TM, welche n_0 Einsen auf ein leeres Band schreibt. Diese TM gibt es, trivialerweise eine mit n_0 vielen Zuständen.

Sei nun $N := n_0 + n_0$

Das Finale: Sei BAD_{Σ} folgende TM:

$$\underbrace{CREATE_{n_0}}_{n_0} \mid \underbrace{DOUBLE \mid EVAL_{\Sigma}(N) \mid INC}_{n_0}$$

Diese Maschine hat N Zustände. Sie startet auf leerem Band, schreibt n_0 Einsen, verdoppelt diese, berechnet $\Sigma(N)$ und schreibt eine weitere 1.

BAD_{Σ} hat also eine 1 mehr als $\Sigma(N)$ geschrieben! Es folgt, dass die Annahme falsch war.

Busy Beaver

Interessanterweise sind einige Busy-Beaverwerte bekannt. Z.B. für TMs mit 2 Symbolen :

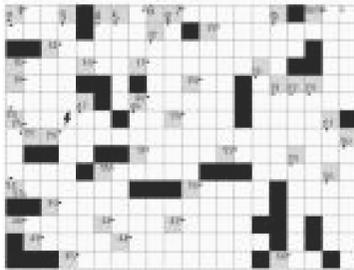
#Zustände	Anzahl Einsen des Siegers
1	1
2	4
3	6
4	13
5	≥ 4098
6	$\geq 95.524.079$

Damit beenden wir den Ausflug in die Turingmaschinen und gehen Wieder zur RAM über.

Probleme des täglichen Lebens

Im folgenden sind die Probleme lösbar. Die Frage ist nur in welcher Zeit und mit wieviel Speicherplatz.

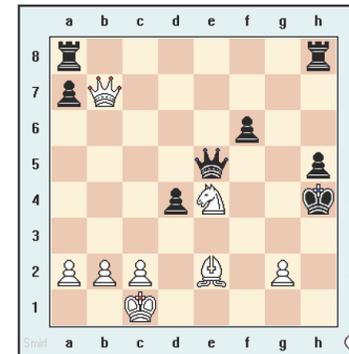
- Was ist schwieriger?



- Kopfrechnen

- Kreuzworträtsel

- Schach



- Sokoban

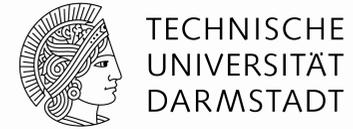


- Puzzle



- Eine Sprache $L \subseteq \Sigma^*$ muss nun irgendwie beschrieben werden.
 - z.B. durch einen *regulären Ausdruck*: (0^*10^*)
 - \emptyset ist ein regulärer Ausdruck.
 - ε ist ein regulärer Ausdruck.
 - $\forall a_i \in \Sigma$ ist a_i ein regulärer Ausdruck.
 - Sind x und y reguläre Ausdrücke, so auch $x \cup y$, (xy) und x^* .
 - Es gibt keine weiteren regulären Ausdrücke.
 - z.B. durch eine **Problembeschreibung**:
 - **Definition**: Ein *Entscheidungsproblem* ist ein input-output Tupel mit
 - geg.**: Kodierung eines Inputs einer Instanz, mittels Alphabet Σ
 - ges.**: ja/nein
 - Die Teilmenge aller Inputs, für die die Antwort “ja” ist, ist offenbar eine Sprache

Algorithmus und Rechenmodell für RAM



Zusätzliche Unterscheidung: unit-cost vs. log-cost Modell

Unit-cost Modell: *jeder Befehl der RAM wird in einem Schritt abgearbeitet*

Typischer Befehlssatz:

+, -, *, /, vergleichen, löschen, schreiben und lesen von rationalen Zahlen, Programmfluß mittels if ... else Verzweigung, Schleifen

Dieses Modell werden wir vorwiegend benutzen.

Log-cost Modell: *jeder Befehl benötigt $\Theta(k)$ Zeit, wobei k die Anzahl der Bits der Operanden ist.*

Typischer Befehlssatz:

laden, speichern, goto, branch on zero, addiere, subtrahiere, bitweises und, bitweises oder, bit-Komplement

Dieses Modell ist realistischer und wird u.a. in der Optimierung relevant. Z.B. bei der so genannten Ellipsoidmethode zur Lösung linearer Programme

Effizienzmaße (Algorithmus A): worst-case, average-case, best-case

$T_A(x)$ = Anzahl Befehle, die A bei Eingabe x ausführt.

$S_A(x)$ = größte Adresse im Speicher, die A bei Eingabe x benutzt.

- **Worst Case Laufzeit:** $T_A^{wc}(n) := \max \{T_A(x) \mid \langle x \rangle \leq n\}$
- **Average Case Laufzeit:** $T_A^{ac}(n) := \sum_{\{x \mid \langle x \rangle = n\}} p_x T_A(x)$,
erfordert die Kenntnis von Auftrittswahrscheinlichkeiten, bzw.
Annahme von Gleichverteilung
- **Best Case Laufzeit:** $T_A^{bc}(n) := \min \{T_A(x) \mid \langle x \rangle \leq n\}$
- **Platzbedarf:** $S_A^{wc}(n) := \max \{S_A(x) \mid \langle x \rangle \leq n\}$

Ein Zeit-Komplexitätsmaß

- Definition: Komplexität eines Algorithmus

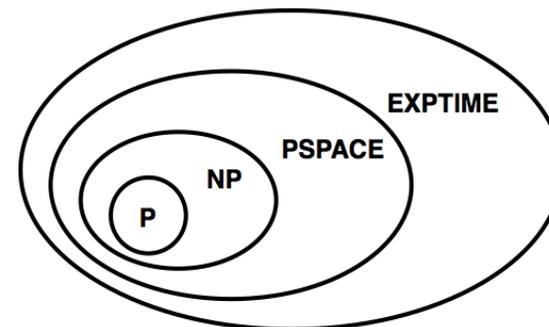
- Sei A ein deterministischer (RAM-)Algorithmus, der auf allen Eingaben hält.
- Die **Laufzeit (Zeitkomplexität)** von A ist eine Funktion $f_A: \mathbb{N} \rightarrow \mathbb{N}$,
 - wobei $f_A(n)$ die maximale Anzahl von Schritten von A beschreibt, **über alle Eingaben der Länge n**.
 - Linear-Zeit-Algorithmus: $f_A(n) \leq c n$ für eine Konstante c
 - Polynom-Zeit-Algorithmus: $f_A(n) \leq c n^k$ für Konstanten c und k

- Definition: Komplexität eines Problems

- Die Zeit- (Platz-) Komplexität eines Problems p ist die Laufzeit des schnellsten (am wenigsten Speicherplatz benötigenden) Algorithmus, der Problem p löst.
- Ein Problem p ist “in Polynomzeit lösbar”, wenn es Algorithmus A, Polynom Π und $n_0 \in \mathbb{N}$ gibt, so dass für alle $n > n_0$ **gilt : $f_A(n) \leq \Pi(n)$**

P, NP, PSPACE

- **P**: Klasse aller Probleme, die von einer deterministischen RAM in Polynomzeit gelöst werden können
- **NP**: Klasse aller Probleme, die von einer nichtdeterministischen TM in Polynomzeit gelöst werden können.
- **PSPACE** : Klasse aller Probleme, die von einer deterministischen RAM mit polynomiell viel Platz gelöst werden können
- Man weiß nur, dass $P \neq \text{EXPTIME}$ und **$P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$**
- $\text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$
- Allgemein wird aber vermutet, dass alle Inklusionen echt sind, d.h.



Beispiele

- **Definition: *HAMPATH***

- Das Hamiltonsche Pfadproblem

- Geg.:

- ein gerichteter Graph

- Zwei Knoten s, t

- Ges.: existiert ein Hamiltonscher Pfad von s nach t

- d.h. ein gerichteter Pfad, der alle Knoten besucht, aber keine Kante zweimal benutzt

- **Algorithmus für Hamiltonscher Pfad:**

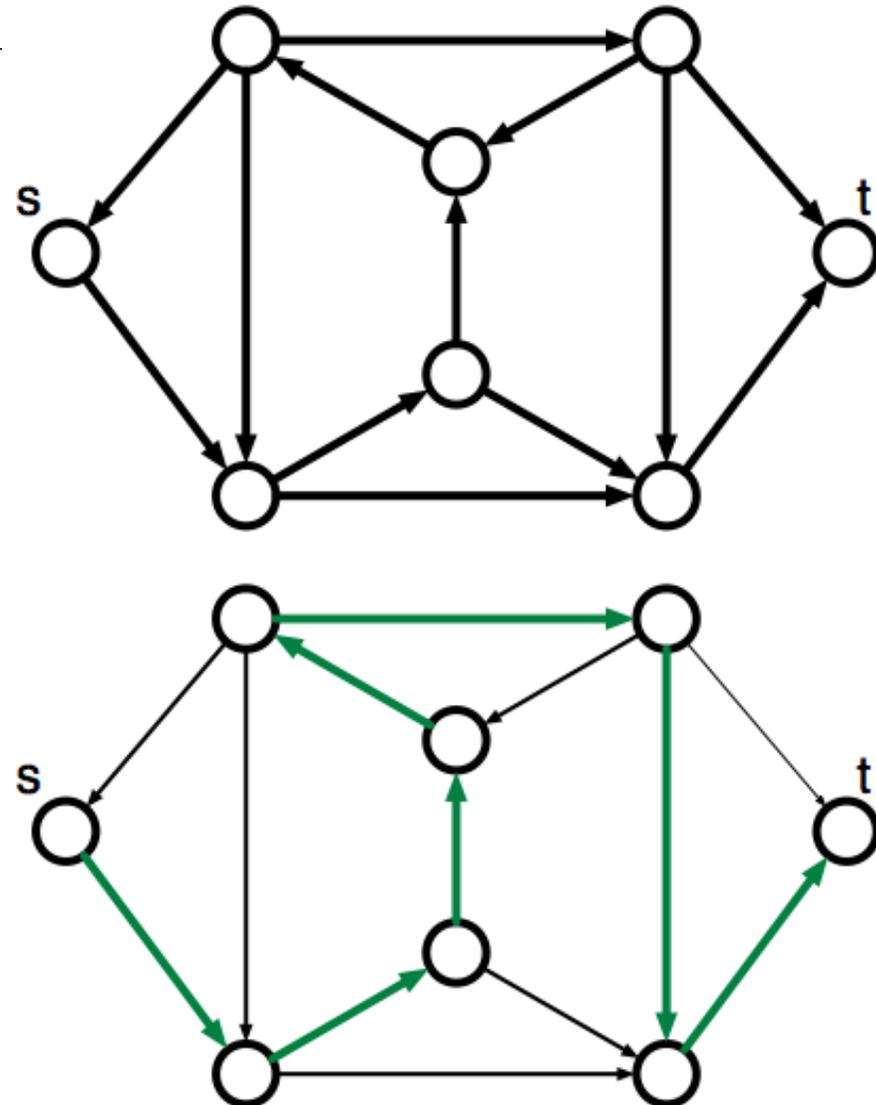
- Rate eine Permutation $(s, v_1, v_2, \dots, v_{n-2}, t)$

- Teste, ob Permutation ein Pfad ist

- falls ja, akzeptiere

- falls nein, verwerfe

- **Also: $\text{HamPath} \in \text{NP}$**



Das SAT Problem

- Eine Boolesche Funktion $f(x_1, x_2, \dots, x_n)$ ist erfüllbar, wenn es eine Wertebelegung für x_1, x_2, \dots, x_n gibt, so dass $f(x_1, x_2, \dots, x_n) = 1$
 - $(x \vee y) \wedge (z \vee \neg x \vee \neg y) \wedge (x \vee \neg z)$ ist erfüllbar, da
 - die Belegung $x = 1, y = 0, z = 0$
 - $(1 \vee 0) \wedge (0 \vee 0 \vee 1) \wedge (1 \vee 1) = 1 \wedge 1 \wedge 1 = 1$ liefert.
- Definition (SAT Problem, die Mutter aller NPc Probleme)
 - **Gegeben:**
 - Boolesche Funktion ϕ
 - **Gesucht:**
 - Gibt es x_1, x_2, \dots, x_n so dass $\phi(x_1, x_2, \dots, x_n) = 1$
- SAT ist in NP. Man vermutet, dass SAT nicht in P ist.

Das QSAT Problem

- Eine quantifizierte Boolesche Formel (QBF) besteht aus
 - Einer Folge von Quantoren $\exists x, \forall y$ mit daran gebundenen Variablen; obdA seien genau alle x_i mit ungeradem i existenzquantifiziert
 - Einer Booleschen Funktion $F(x_1, x_2, \dots, x_m)$
 - Jede Variable der Funktion ist genau einmal an einem Quantor gebunden
- Die quantifizierte Boolesche Formel ist erfüllbar falls
 - Im Falle eines Existenzquantors: $\exists x F(x) \Leftrightarrow F(0) \vee F(1)$
 - Im Falle eines Allquantors: $\forall x F(x) \Leftrightarrow F(0) \wedge F(1)$
- Definition (QSAT Problem, die Mutter aller PSPACEc Probleme)
 - **Gegeben:** Quantifizierte Boolesche Funktion ϕ
 - **Frage:** Gibt es x_1 , so dass es für alle x_2 ein x_3 gibt, so dass ... so dass $\phi(x_1, x_2, \dots, x_n) = 1$
- QSAT ist in PSPACE. Man vermutet, dass QSAT nicht in NP ist. QSAT ist PSPACE-schwer.

Beispiele:

$$\begin{aligned}
 & \blacksquare \exists x \forall y (x \wedge y) \vee (\neg x \wedge \neg y) \\
 & \quad = (\forall y (0 \wedge y) \vee (\neg 0 \wedge \neg y)) \vee (\forall y (1 \wedge y) \vee (\neg 1 \wedge \neg y)) \\
 & \quad = (\forall y: \neg y) \vee (\forall y: y) \\
 & \quad = (\neg 0 \wedge \neg 1) \vee (0 \wedge 1) \\
 & \quad = 0 \vee 0 \\
 & \quad = 0
 \end{aligned}$$

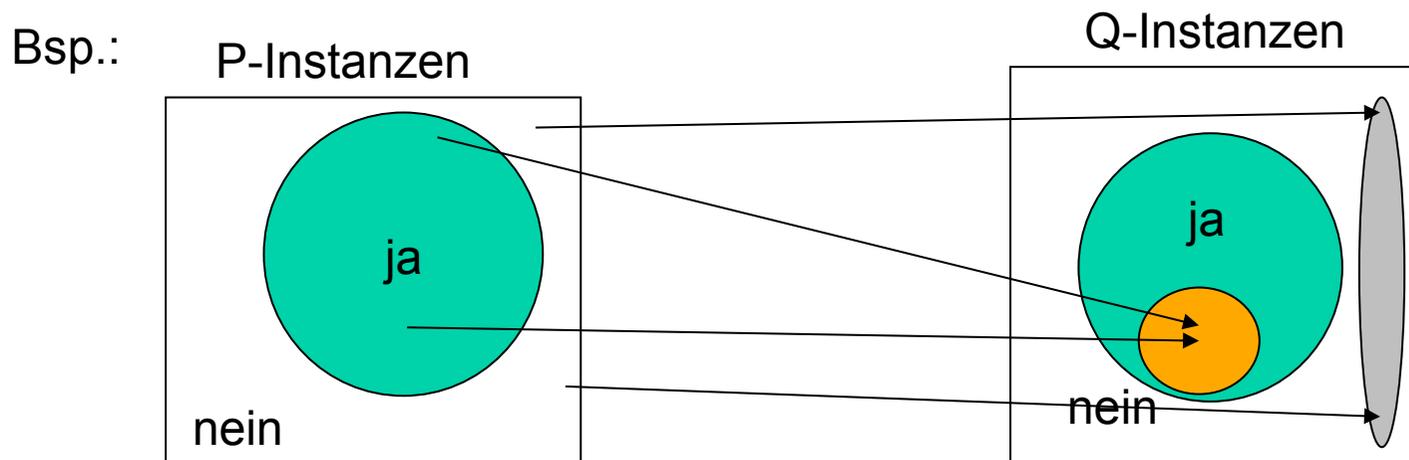
$$\begin{aligned}
 & \blacksquare \forall y \exists x (x \wedge y) \vee (\neg x \wedge \neg y) \\
 & \quad = (\exists x: (x \wedge 0) \vee (\neg x \wedge \neg 0)) \wedge (\exists x: (x \wedge 1) \vee (\neg x \wedge \neg 1)) \\
 & \quad = (\exists x: \neg x) \wedge (\exists x: x) \\
 & \quad = (\neg 0 \vee \neg 1) \wedge (0 \vee 1) \\
 & \quad = 1 \wedge 1 \\
 & \quad = 1
 \end{aligned}$$

Einordnung von Problemen in P, NP, PSPACE

- Angabe eines Algorithmus für Problem
- Reduktionstechnik steht im Zentrum

Definition: Seien P, Q Probleme. Sei L_P (L_Q) die Menge der Instanzen des Problems P (Q), für die die Antwort „ja“ ist. P heißt auf Q **polynomiell** **reduzierbar** ($P \leq_p Q$), wenn es eine von einem deterministischen Algorithmus in Polynomzeit berechenbare Funktion $f: \Sigma^* \rightarrow \Sigma^*$ gibt, so dass

$$\underline{x \in L_P \Leftrightarrow f(x) \in L_Q}$$

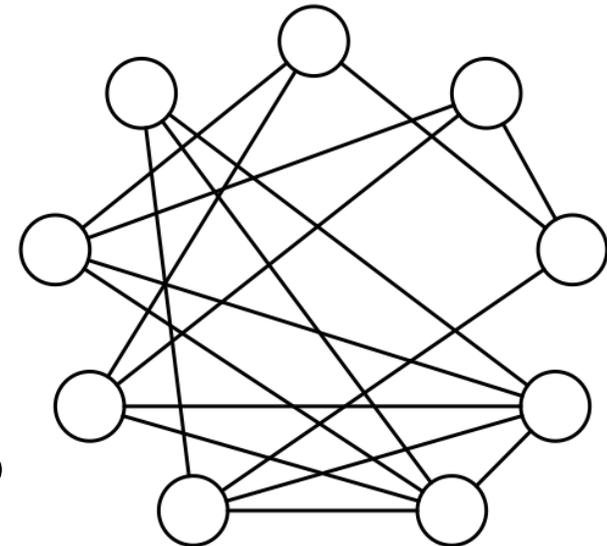


Das 3-SAT-Problem und das Clique-Problem

- 3-SAT:
 - **Gegeben:**
 - Eine Boolesche Formel in 3-CNF
 - **Gesucht:**
 - Gibt es eine erfüllende Belegung
- Definition k-Clique
 - Ein ungerichteter Graph $G=(V,E)$ hat eine k-Clique,
 - falls es k verschiedene Knoten gibt,
 - so dass jeder mit jedem anderen eine Kante in G verbindet
- CLIQUE:
 - **Gegeben:**
 - Ein ungerichteter Graph G
 - Eine Zahl k
 - **Gesucht:**
 - Hat der Graph G eine Clique der Größe k?

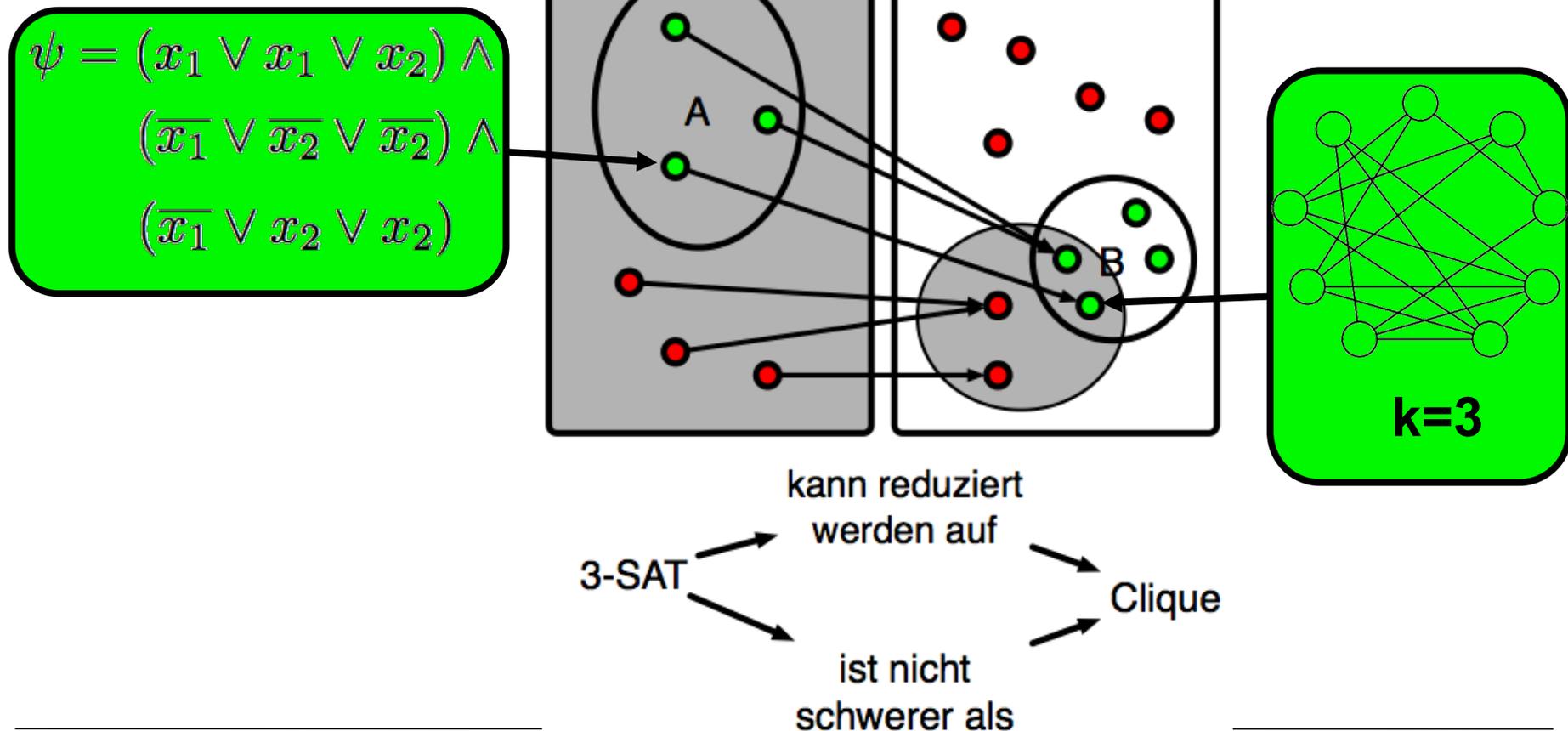
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

k=3



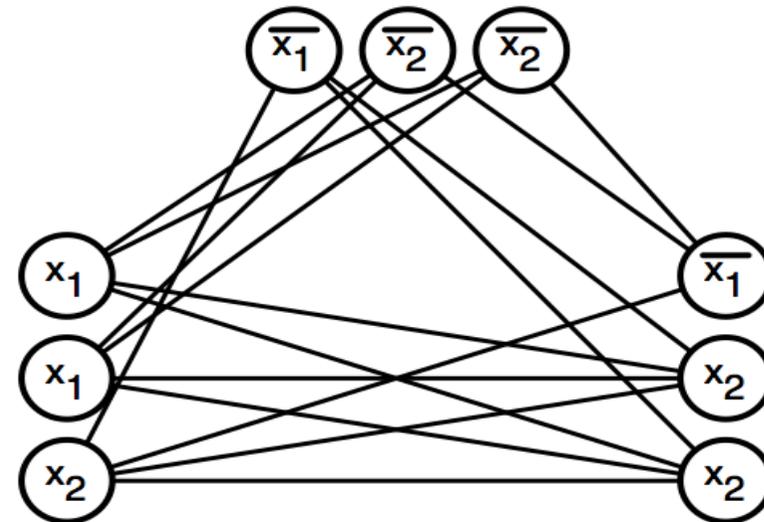
3-SAT lässt sich auf Clique reduzieren

- Theorem: $3\text{-SAT} \leq_p \text{CLIQUE}$

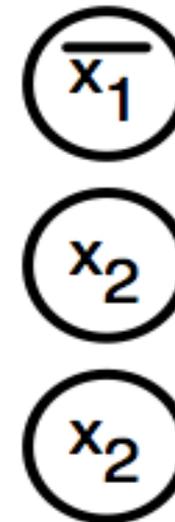
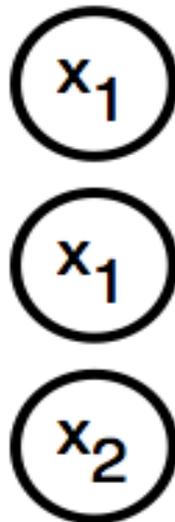


3-SAT lässt sich auf Clique reduzieren

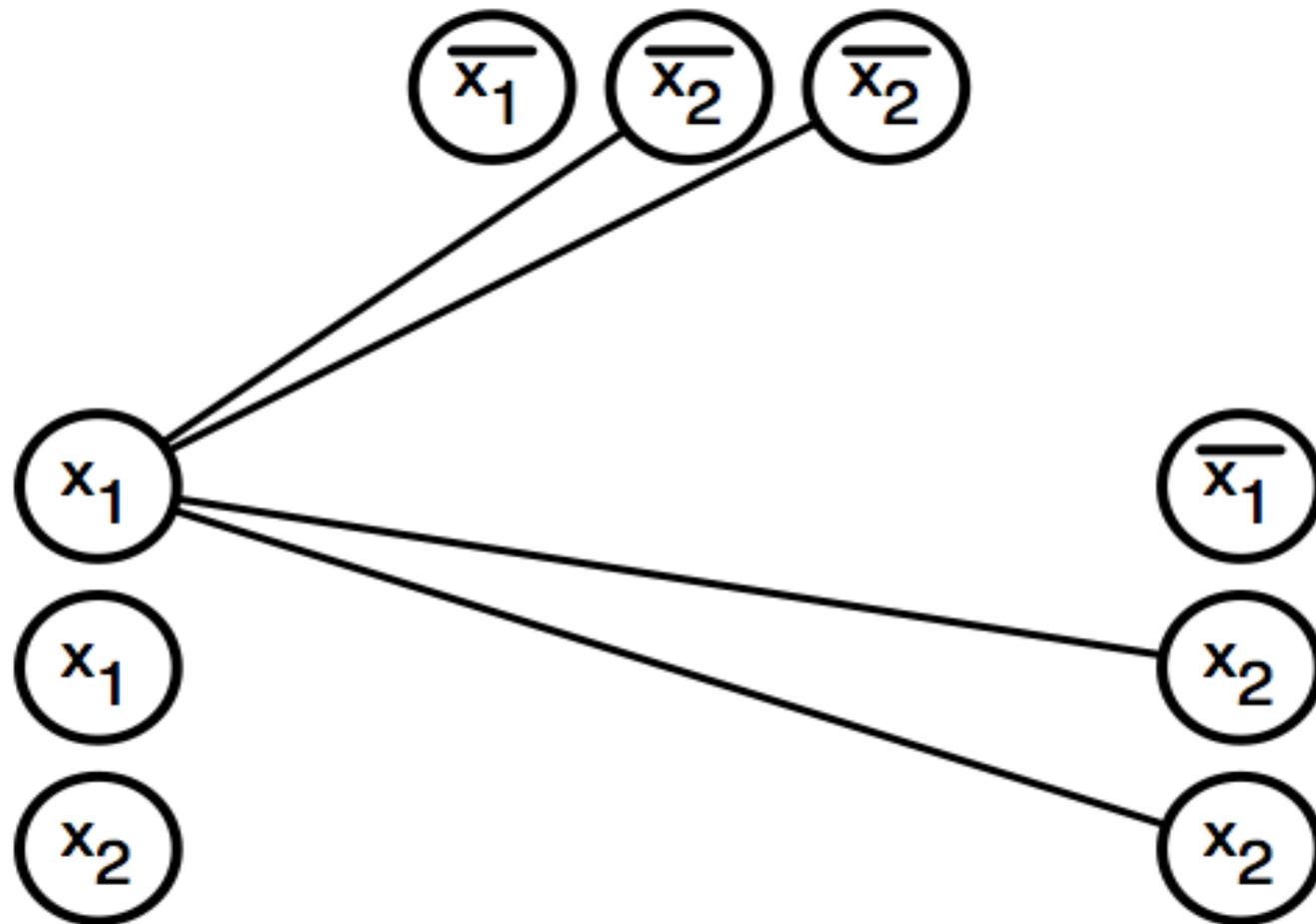
- Theorem: $3\text{-SAT} \leq_p \text{CLIQUE}$
- Beweis
 - Konstruiere Reduktionsfunktion f wie folgt:
 - $f(\phi) = \langle G, k \rangle$
 - k = Anzahl der Klauseln
 - Für jede Klausel C in ϕ werden drei Knoten angelegt, die mit den Literalen der Klausel bezeichnet werden
 - Füge Kante zwischen zwei Knoten ein, gdw.
 - die beiden Knoten nicht zur selben Klausel gehören und
 - die beiden Knoten nicht einer Variable und der selben negierten Variable entsprechen.



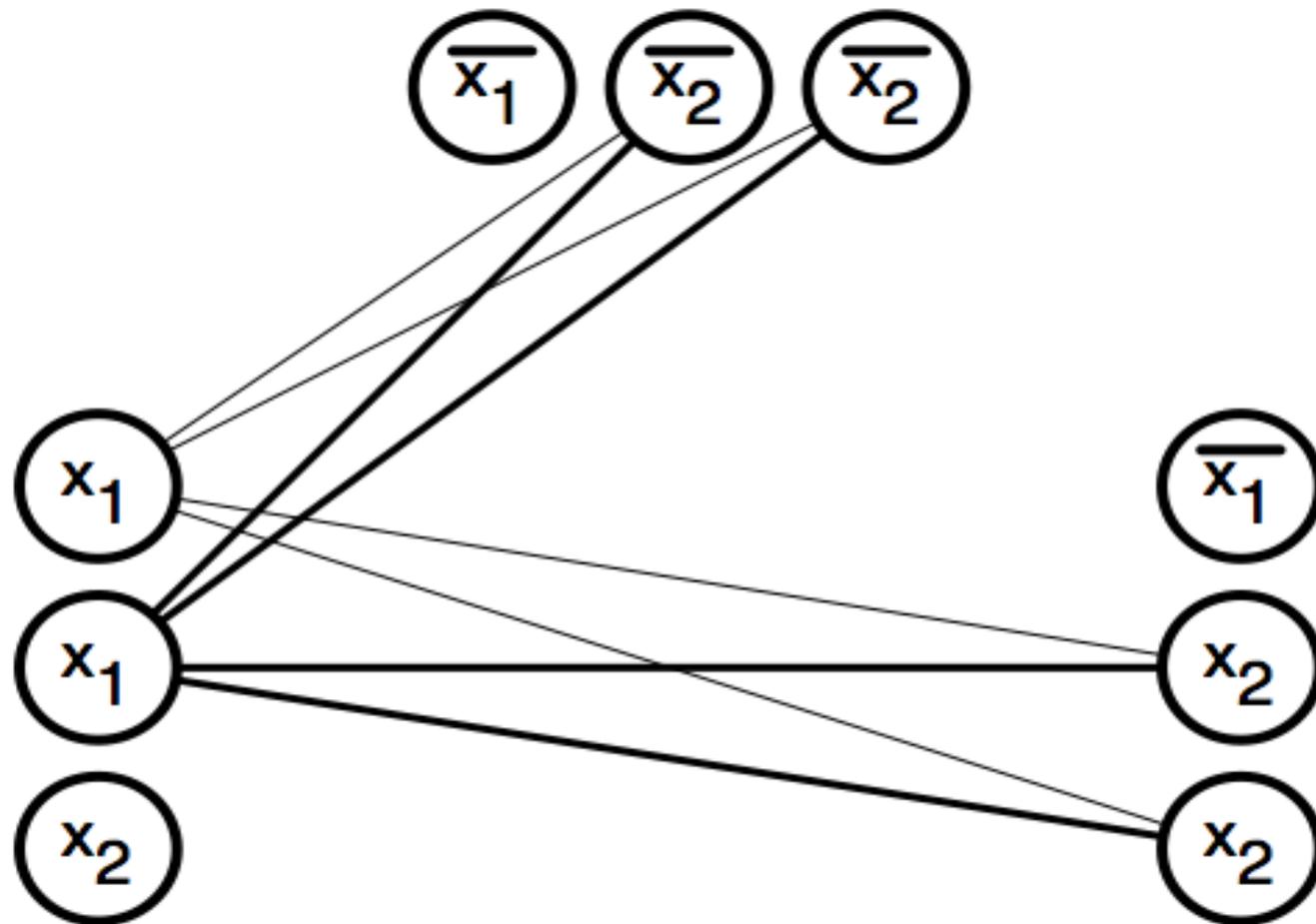
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$$



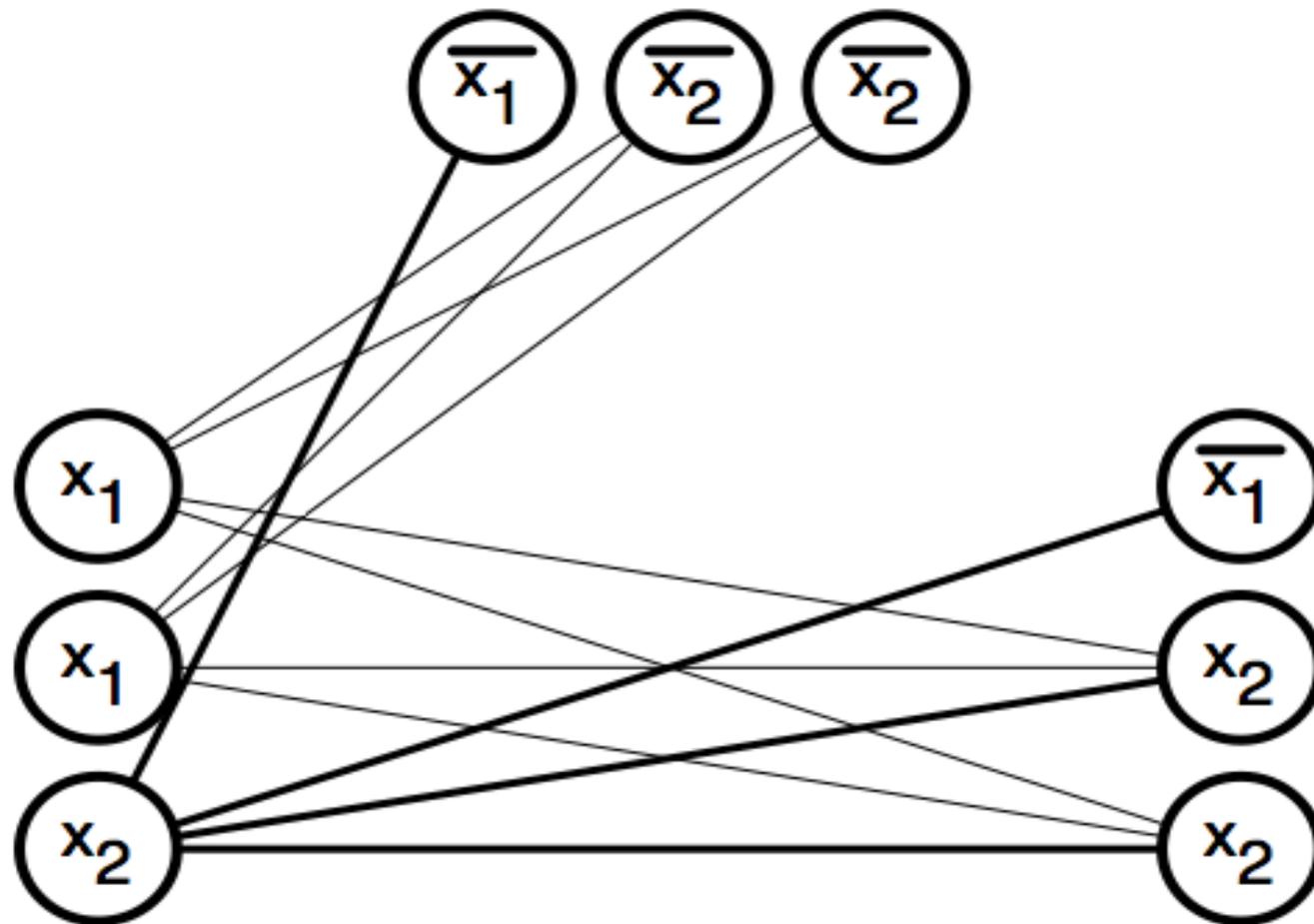
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



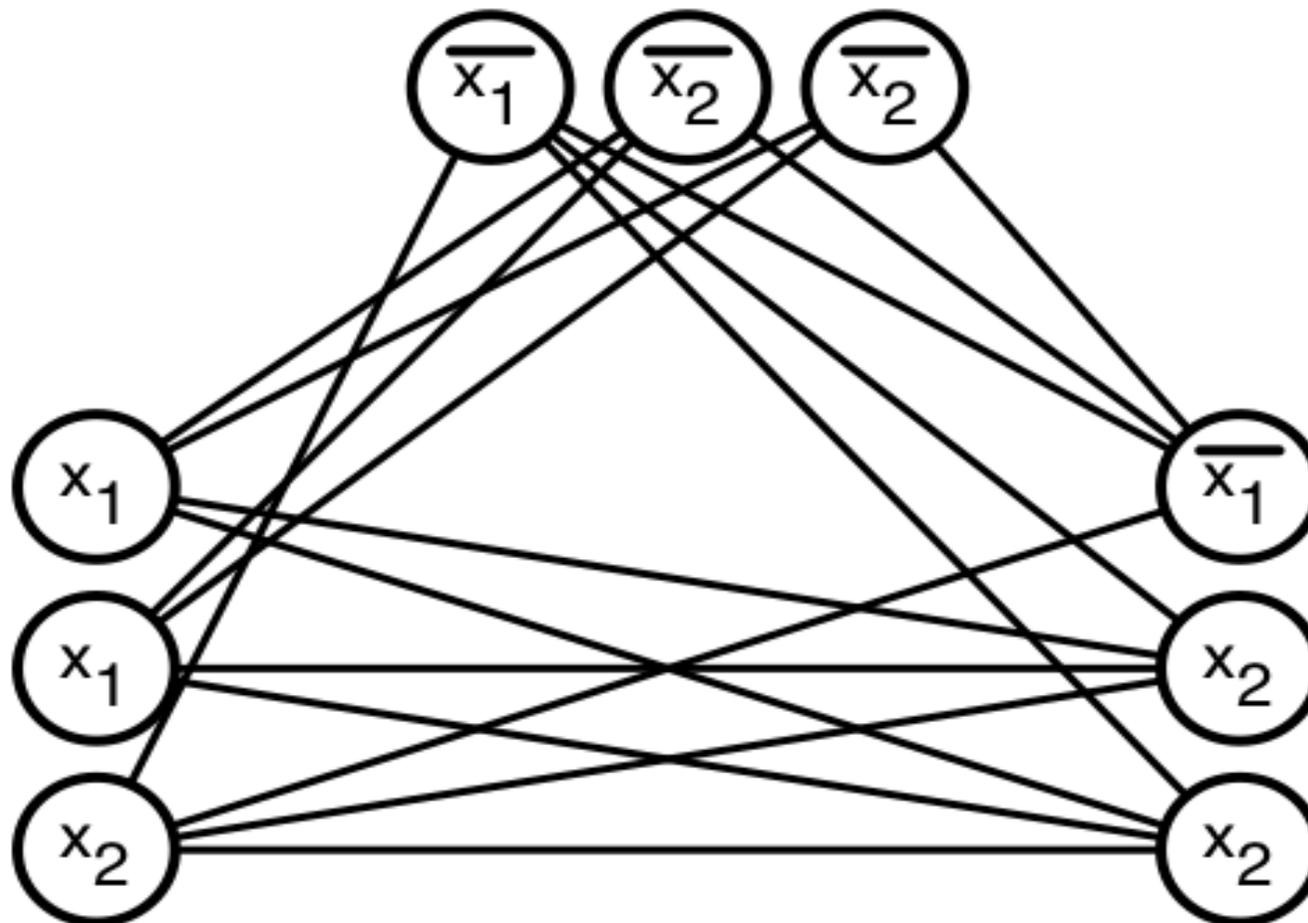
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



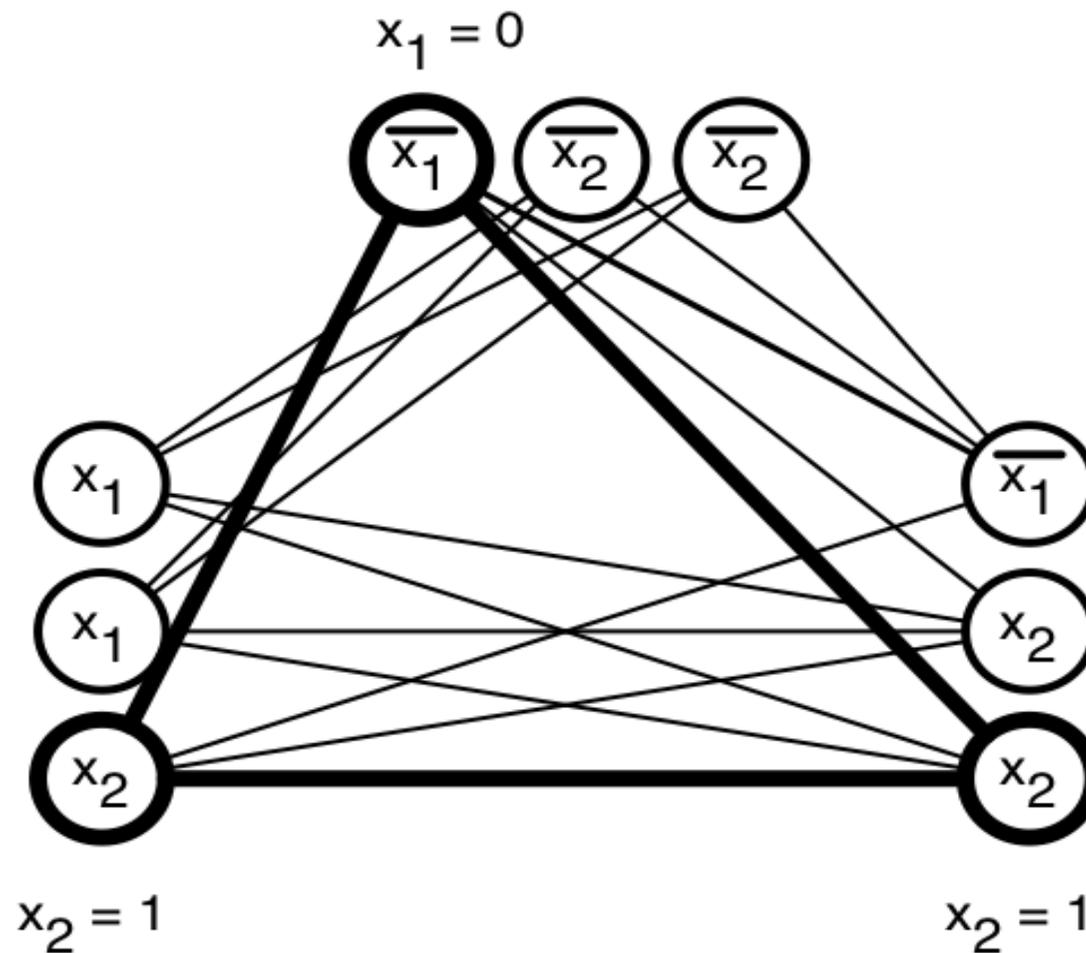
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

0	0	1	1	0	0	1	1	1
---	---	---	---	---	---	---	---	---

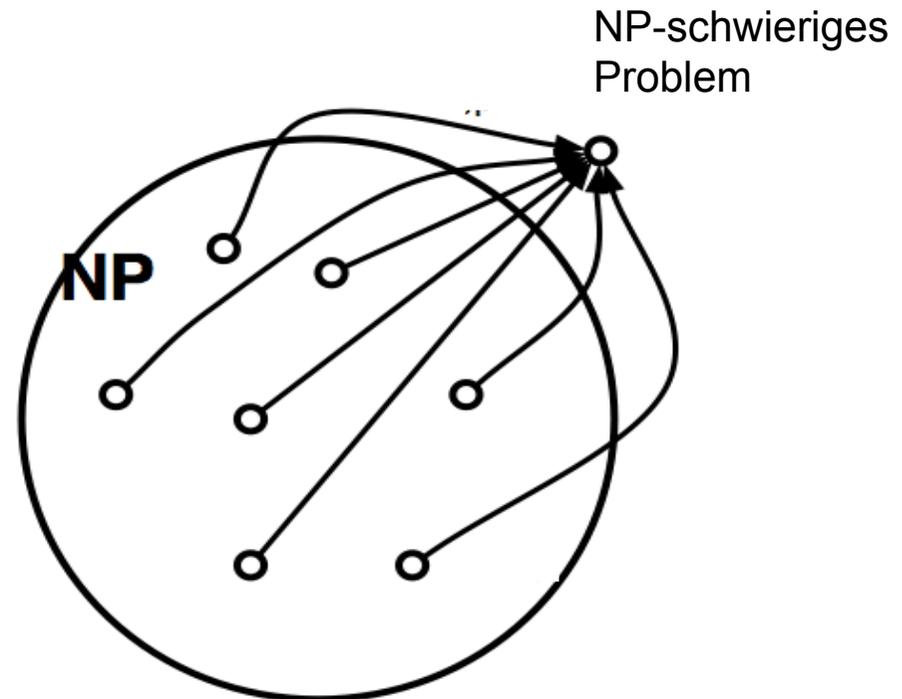
Beweis der Korrektheit der Reduktionsfunktion



- Die Reduktionsfunktion ist korrekt:
- Behauptung:
 - Eine erfüllende Belegung in ϕ existiert gdw. eine k -Clique in G existiert
- 1. Fall: eine erfüllende Belegung existiert in ϕ
 - Dann liefert die Belegung in jeder Klausel mindestens ein Literal mit Wert 1
 - Wähle aus der Knotenmenge einer Klausel ein beliebiges solches Literal
 - Die gewählte Knotenmenge besteht dann aus k Knoten
 - Zwischen allen Knoten existiert eine Kante, da Variable und negierte Variable nicht gleichzeitig 1 sein können
- 2. Fall: eine k -Clique existiert in G
 - Jeder der Knoten der Clique gehört zu einer anderen Klausel
 - Setze die entsprechenden Literale auf 1
 - Bestimme daraus die Variablen-Belegung
 - Das führt zu keinem Widerspruch, da keine Kanten zwischen einem Literal und seiner negierten Version existieren
- Laufzeit:
 - Konstruktion des Graphens und der Kanten benötigt höchstens quadratische Zeit.

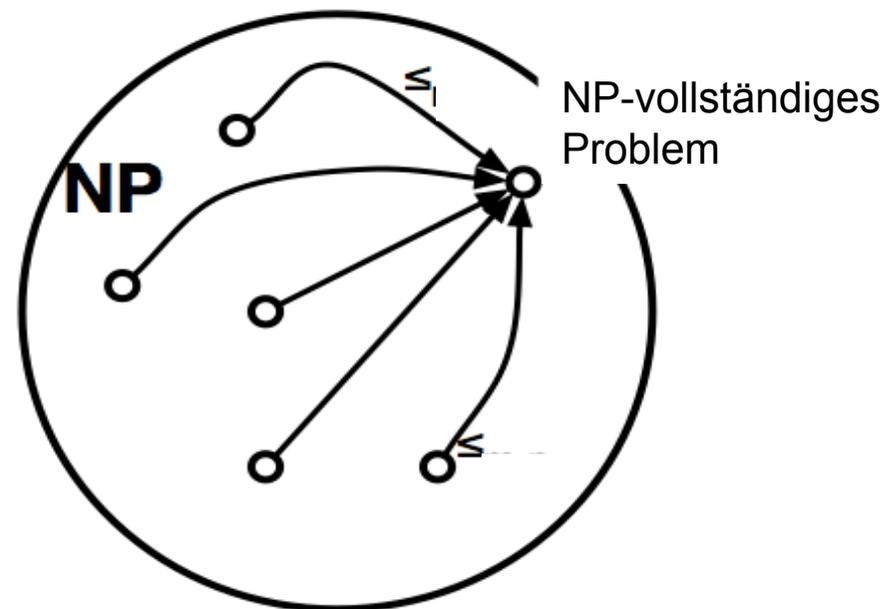
NP-Schwierig

- Definition:
 - Ein Problem S ist **NP-schwierig** (NP-hard) wenn:
 - jedes Problem aus NP mit einer Polynom-Zeit-Abbildungsreduktion auf S reduziert werden kann, d.h.
 - für alle $L \in \text{NP}$: $L \leq_p S$
- Theorem
 - Falls ein NP-schwieriges Problem in P ist, ist $P = \text{NP}$
- Beweis
 - Falls $S \in P$ und $L \leq_p S$ gilt $L \in P$.



NP-Vollständigkeit

- Definition:
 - Ein Problem S ist **NP-vollständig** (NP-complete) wenn:
 - $S \in NP$
 - S ist NP-schwierig
- Korollar:
 - Ist eine NP-vollständiges Problem in P , dann ist $P=NP$
- Beweis:
 - folgt aus der NP-Schwierigkeit der NP-vollständigen Probleme.



SAT ist NP-Vollständigkeit

- Satz von Cook, Idee
 - Für jedes NP-vollständige Problem gibt es eine NTM, die das Problem entscheidet
 - Sei nun das Problem P , eine NTM M und eine Eingabe x gegeben.
 - Dann lässt sich eine boolesche Formel in KNF angeben, deren Variablen-Belegungen die Maschine M auf x simuliert.
- Also: Die Entscheidung “akzeptiert M x ” kann rückgeführt werden auf die Erfüllbarkeit einer SAT-Formel.

Nichtdeterministische Maschinen und Verifizierer



Def. Es sei L eine Sprache. Ein Verifizierer für L ist ein deterministischer Algorithmus A , mit $L = \{w \mid \text{es gibt ein } c \text{ mit } A \text{ akzeptiert } wc\}$

Der Zeitaufwand für einen Verifizierer wird abhängig von der Länge von w gemessen. L ist polynomiell prüfbar, wenn es einen Verifizierer mit polynomiellem Zeitaufwand gibt.

Satz: NP ist die Menge aller Probleme, für die es einen Verifizierer mit polynomiellem Zeitaufwand gibt.

(ohne Beweis)

- **Komplexität wird gemessen mit Hilfe des worst-case,**
 - In der realen Welt treten aber oft „gutmütige“ Instanzen auf
 - andererseits: Haben wir dann das richtige Problem formuliert?
- **Die Eingabegröße**

Sei ein Problem p gegeben. Wenn wir nun die Kodierung der Eingabe signifikant kleiner bekommen, wird das resultierende aus p abgeleitete Problem p' möglicherweise schwerer, obwohl die Laufzeit möglicherweise sinkt.

Fazit: Die Lücke zwischen der Komplexitätstheorie und der Wirklichkeit ist größer als wir es von der Physik gewohnt sind. Trotzdem ist die Komplexitätstheorie eine großartige Theorie.