Diskrete Algorithmische Mathematik (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Größenordnungen, die O-Notationen („Groß-O Notation“)

TECHNISCHE UNIVERSITÄT DARMSTADT

$$Sei\ g : IN \rightarrow IR_{\geq 0}.$$

$$O(g) := \{f : IN \rightarrow IR_{\geq 0} : \exists c > 0, n_0 \in IN, so\ dass\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

bezeichnet dann die Menge aller Funktionen f: IN → IN, für die zwei positive Konstanten c ∈ $IR_{\geq 0}$ und n_0 ∈IN existieren, so dass für alle n ≥ n_0 gilt: f(n) ≤ c*g(n)

Bemerkung: Die asymptotische Notation vernachlässigt Konstanten und Terme niedrigerer Ordnung.
(Man sagt: falls f∈O(g) ist, wächst f asymptotisch höchstens so schnell wie *g*.)

Satz: Für ein Polynom $f(n) = a_m n^m + ... + a_0$ vom Grad m mit positivem Koeffizienten a_m gilt: f ∈O(n^m) ⟵ [Anmerkung: eigentlich O(n → n^m)]

Beweis:
$$f(n) \leq |a_m| n^m + ... + |a_1| n + |a_0|$$
$$\leq (|a_m| + |a_{m-1}| / n + ... + |a_0| / n^m) \cdot n^m$$
$$\leq (|a_m| + |a_{m-1}| + ... + |a_0|) \cdot n^m$$

Setzen wir nun $c = |a_m| + |a_{m-1}| + ... + |a_0|$ und n_0=1, so folgt die Behauptung.

Größenordnungen, weitere Bezeichnungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Weitere Definitionen: Wieder seien $f, g: IN \rightarrow IR_{\geq 0}$

- $f \in \Omega(g) \Leftrightarrow g \in O(f)$
  (Man sagt: *f* wächst asymptotisch mindestens so schnell wie *g*.)
- $f \in \Theta(g) \Leftrightarrow f \in O(g)$ und $g \in O(f)$
  (Man sagt: *f* und *g* wachsen asymptotisch gleich schnell.)
- $o(g) := \{f : IN \rightarrow IN : \forall c > 0\, \exists n_0 \in IN, so\ dass\ \forall n \geq n_0 : f(n) < c \cdot g(n)\}$
  (Man sagt: *f* wächst asymptotisch langsamer als *g*.)

- $F \in \omega(g) \Leftrightarrow g \in o(f)$
  (Man sagt: *f* wächst asymptotisch schneller als *g*.)

Statt $f \in O(g)$ wird oft auch f = O(g) geschrieben. Dasselbe gilt für o, ω, Ω, Θ.

Größenordnungen, Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sei f(n) die Anzahl der Vergleiche bei einer sequentiellen Suche nach dem Maximum in einer Zahlenfolge mit n Elementen. Dann gilt $f(n) \in O(n)$, da einmaliges Durchlaufen der Eingabe mit n Vergleichen auskommt.

  Andererseits muss jeder Algorithmus zumindest einmal jedes Element der Eingabe ansehen. Deshalb gilt auch $f(n) \in \Omega(n)$.

- Matrixmultiplikation: Seien A und B quadratische n x n Matrizen. Die Einträge c_{ij} von $C = A \cdot B$ ergeben sich aus $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. Also n Multiplikationen und n Additionen. Da n^2 viele Einträge von C berechnet werden, ergibt sich einerseits für den Gesamtaufwand des „offensichtlichen“ Algorithmus: $n^2(n+n-1) = 2n^3 - n^2 \in O(n^3)$, und andererseits wird jeder Algorithmus $\Omega(n^2)$ Operationen benötigen.

  Der schnellste zur Zeit bekannte Algorithmus benötigt $O(n^{2.376})$ Operationen.

Größenordnungen, Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

- $n \in o(n^2)$, $n^2 \in O(n^2)$, $n^2 \notin o(n^2)$

-
```
for i = 1 to n do
    for j = 1 to n do
        führe eine Operation aus
    end do
end do
```

$O(n^2)$ Operationen

```
for i = 1 to n do
    for j = i+1 to n do
        führe f(n) Operationen aus
    end do
end do
```

$O(n^2 \cdot f(n))$ Operationen

Größenordnungen, Bemerkung

TECHNISCHE UNIVERSITÄT DARMSTADT

a) Die Relation o(...) ist transitiv

$$f(n) = o(g(n) \text{ und } g(n)=o(h(n)) \Rightarrow f(n) = o(h(n))$$

b) Die Relation o(...) kann zur Klassifikation von Funktionen benutzt werden.
Z.B. gilt für $0 < \varepsilon < 1 < c$:

1	$= o(\log \log n)$	konstante Funktionen
$\log \log n$	$= o(\log n)$	doppelt logarithmische Funktionen
$\log n$	$= o(n^{\varepsilon})$	logarithmische Funktionen
n^{ε}	$= o(n^{c})$	Wurzelfunktionen
n^{c}	$= o(n^{\log n})$	Polynome
$n^{\log n}$	$= o(c^{n})$	subexponentielle Funktionen
c^{n}	$= o(n^{n})$	exponentielle Funktionen
n^{n}	$= o(c^{c^{n}})$	überexponentielle Funktionen

Größenordnungen, Beispiele

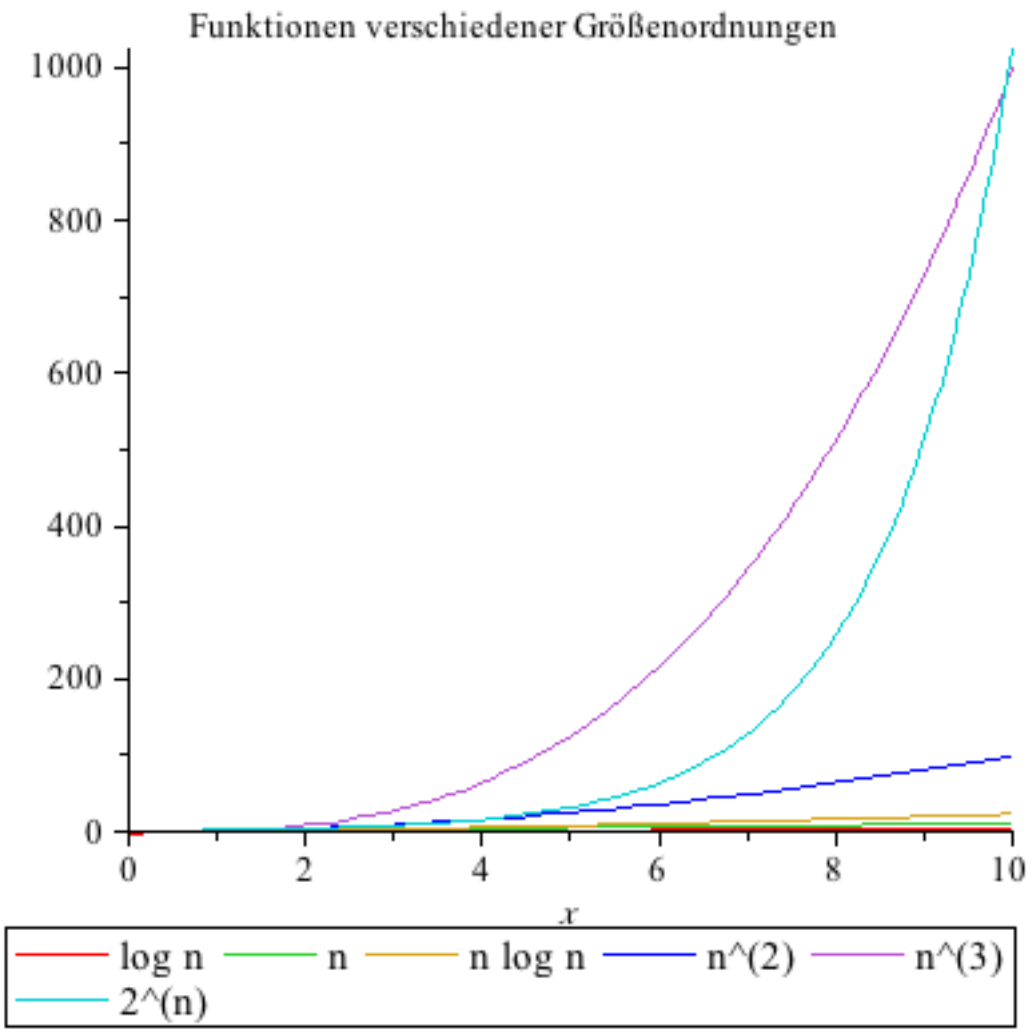
TECHNISCHE UNIVERSITÄT DARMSTADT

Die folgende Tabelle verdeutlicht das Wachstum von Funktionen verschiedener Größenordnungen:

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Größenordnungen, Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT



Größenordnungen, Rechenregeln

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für eine Konstante c gilt $\mathbf{c \in O(1)}$
- $\mathbf{c \cdot f(n) \in O(f(n))}$, klar mit Def. Von O-Notation
- $\mathbf{O(f)+O(f) \subseteq O(f)}$. Seien g und h Funktionen aus O(f). Dann gibt es c_g, c_h, n_g und n_h so dass ... (Übung)
- **O(O(f))=O(f)** mit Def.
- $\mathbf{O(f) \cdot O(g) \subseteq O(f \cdot g)}$ (Übung)
- **O(f+g)=O(max{f(n),g(n)})**.
  Sei $h \in O(f+g)$. Dann gibt es positive Konstanten c und n_0, so dass für alle $n \geq n_0$ gilt: $h(n) \leq c \cdot (f+g)(n) \leq c \cdot 2 \cdot \max\{f,g\}(n)$. Also ist $h(n) \in O(\max\{f,g\})$.

  Andererseits sei $h \in O(\max\{f,g\})$. Dann gibt es positive Konstanten c und n_0, so dass für alle $n \geq n_0$ gilt: $h(n) \leq c \cdot \max\{f,g\}(n) \leq c \cdot (f+g)(n)$,
  und damit $h \in O(f+g)$.

Größenordnungen, die O-Notationen („Master Theorem“)

TECHNISCHE UNIVERSITÄT DARMSTADT

$Seien\ a \geq 1, b > 1\ Konstanten\ und\ sei\ T(n) : IN_0 \rightarrow IR_{\geq 0}.$

Sei

$$T(n) = aT(n/b) + f(n)$$

$(wobei\ n/b\ entweder\ für\ \lfloor n/b \rfloor\ oder\ \lceil n/b \rceil\ stehe.)$

$\rightarrow falls\ \exists \varepsilon > 0\ mit\ f(n) = O(n^{\log_b a - \varepsilon}), dann$

$T(n) = \Theta(n^{\log_b a})$

$\rightarrow falls\ f(n) = \Theta(n^{\log_b a}), dann$

$T(n) = \Theta(n^{\log_b a} \cdot \log n)$

$\rightarrow falls\ \exists \varepsilon > 0\ mit\ f(n) = \Omega(n^{\log_b a + \varepsilon}), und\ falls\ \exists c > 0\ gibt$

$mit\ a \cdot f(n/b) \leq c \cdot f(n)\ für\ genügend\ große\ n, dann$

$T(n) = \Theta(f(n))$

Beachte:
Die 3 Fälle decken nicht alles ab!

Größenordnungen, die O-Notationen („Master Theorem“)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

$T(n) = 9T(\lceil n/3 \rceil) + n$

dann ist a=9, b=3, f(n)=n, und somit $n^{\log_b a} = n^{\log_3 9} = n^2$

Damit ist $f(n)=O(n^{\log_3 9-\varepsilon})$, und wir schließen mit Fall 1:

$T(n) = \Theta(n^2)$

$T(n) = T(\lceil 2n/3 \rceil) + 1$

dann ist a=1, b=3/2, f(n)=1 und $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Fall 2, weil $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$

also: $T(n) = \Theta(\log n)$

Komplexitätstheorie

TECHNISCHE UNIVERSITÄT DARMSTADT

- Was ist ein Problem?
  - **Problem:** binäre Relation zwischen einer Menge *I* von Instanzen und einer Menge *S* von Lösungen.

    Beispiel Max-Summenproblem:

    Eingabe: Folge $a_1,...,a_n$ ganzer Zahlen. Sei $f(i,j) := a_i+a_{i+1}+...+a_j$, für $1 \le i \le j \le n$.

    Gesucht: Das maximale $f(i,j)$.

    Die Menge der Instanzen besteht hier aus allen Folgen ganzer Zahlen mit endlicher Länge. Die Menge der Lösungen ist die Menge der ganzen Zahlen. Durch die Definition von f und der Forderung nach einem maximalen f(i,j), werden den Zahlenfolgen endlicher Länge maximale Summen zugeordnet.

- Was ist ein Problem (2)?

  Man unterscheidet das abstrakte Problem und die Beschreibung des Problems und seiner Instanzen.

  Wenn man ein Problem kommunizieren möchte, muss man es kodieren.

  Kodierung eines Problems und seiner Instanzen, Problem- und Instanz-beschreibung:

  – Ein Alphabet ist eine Menge von Symbolen, auf die man sich zur Beschreibung von Problemen einigt. z.B:
    - {A,..,Z,a,..,z,0,…,9} genügt für die meiste Schriftkommunikation
    - Laute, zur sprachlichen Kommunikation
    - {0,1} eignet sich besonders, um Probleme Computern verständlich zu machen

TECHNISCHE UNIVERSITÄT DARMSTADT

– Um ein Problem zu beschreiben, genügt im Allgemeinen nicht ein Alphabet, man definiert auch Regeln, welche Bedeutung Verknüpfungen der Symbole/Buchstaben haben; so genannte Kodierungsschemata.

- **Ganze Zahlen** werden **binär** dargestellt (durch **Bits**), d.h., wir schreiben

$$n = \pm \sum_{i=0}^{k} x_i \cdot 2^i, \quad x_i \in \{0,1\} \; und \; k = \lfloor \log_2(|n|) \rfloor$$

d.h., die **Kodierungslänge <n>** einer ganzen Zahl n ist gegeben durch

$$\langle n \rangle = \lceil \log_2(|n|+1) \rceil + 1 = \lceil \log_2 |n| \rceil + 2$$

- **Rationale Zahlen:** Sei r eine rationale Zahl. Dann gibt es eine ganze Zahl p und eine natürliche Zahl q mit r = p/q.

<r> = <p> + <q>

Komplexitätstheorie

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Vektoren**

für $x = (x_1,\ldots,x_n) \in \mathbb{Q}^n$ ist

$$\langle x \rangle = \sum_{i=1}^{n} \langle x_i \rangle$$

- **Matrizen**

für $A \in \mathbb{Q}^{mxn}$ ist

$$\langle A \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle$$

Inputlänge: Die Anzahl der Bits, die benötigt werden, um eine Instanz *I* vollständig zu beschreiben wird Inputlänge <*I*> genannt.

Komplexitätstheorie

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Problem zu lösen, also beliebigen Instanzen des Problems Lösungen zuzuordnen, kann unterschiedlich schwierig sein
  - **Bsp.: In einem sehr komplizierten Fall ist dies 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?

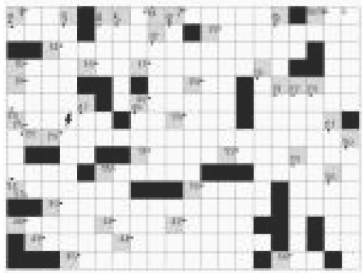
“nicht entscheidbar” heisst: es gibt keinen Algorithmus, der für alle Instanzen des Problems die richtige Antwort geben kann.

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

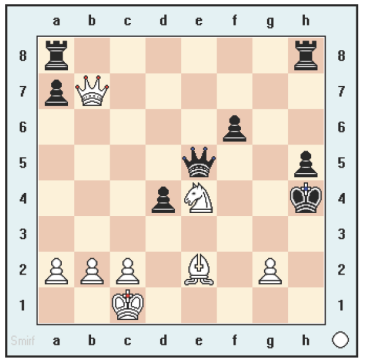
Probleme des täglichen Lebens

TECHNISCHE UNIVERSITÄT DARMSTADT

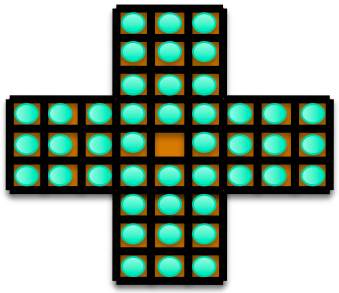
- Was ist schwieriger?
  - Kopfrechnen



  - Kreuzworträtsel
  - Schach



  - Sokoban



  - Solitär

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Was ist ein Algorithmus?**
  - Ein **Algorithmus** ist eine Anleitung zur schrittweisen Lösung eines Problems. Wir sagen, ein Algorithmus A löst ein Problem Π, falls A für alle Instanzen $I \in \Pi$ des Problems eine Lösung in einer endlichen Anzahl von Schritten findet. Die Anleitung hat konstante Kodierungslänge.
  - Ein Schritt ist eine elementare Operation. (was ist das?)

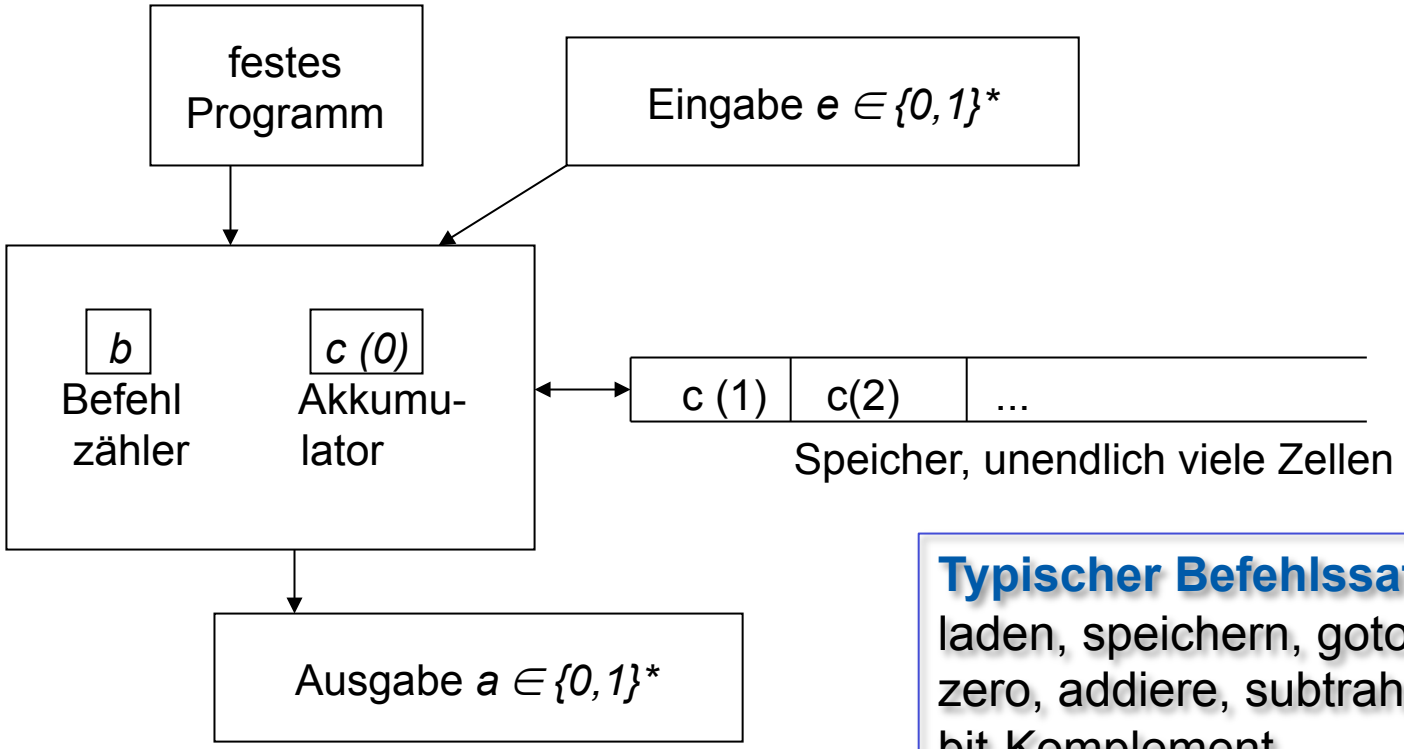
Offenbar hängt die Definition einer elementaren Operation von einer Maschine ab, die unseren Algorithmus A ausführt!

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

→ Rechenmodell, Effizienzmaß.

Registermaschine (Random Access Machine RAM)



Typischer Befehlssatz:
laden, speichern, goto, branch on zero, addiere, subtrahiere, und, oder, bit-Komplement

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

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 Θ(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

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

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:** $T_A^{wc}(n) := max\{T_A(x) \mid <x> \leq n\}$

- **Average Case Laufzeit:** $T_A^{ac}(n) := \sum_{\{x \mid <x> = n\}} p_x \, T_A(x)$,
  erfordert die Kenntnis von Auftrittswahrscheinlichkeiten, bzw. Annahme von Gleichverteilung

- **Best Case Laufzeit:** $T_A^{bc}(n) := min\{T_A(x) \mid <x> \leq n\}$

  (In unserem bisherigen Beispiel war $T_A^{bc}(n) \approx T_A^{wc}(n)$ *.)*

- **Platzbedarf**: $S_A^{wc}(n) := max\{S_A(x) \mid <x> \leq n\}$

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Definition: (worst-case) Komplexität eines Algorithmus
  - Sei A ein deterministischer (RAM-)Algorithmus, der auf allen Eingaben hält.
  - Die **Laufzeit (Zeitkomplexität)** von A ist eine Funktion f: N→N,
    - wobei f(n) die maximale Anzahl von Schritten von A, **auf einer Eingabe der Länge n ist**.
    - Linear-Zeit-Algorithmus: $f(n) \leq c\, n$ für eine Konstante c
    - Polynom-Zeit-Algorithmus: $f(n) \leq c\, n^k$ für Konstanten c und k (und jeweils *n* hinreichend groß)

- 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(n) <= p(n)**

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel: Addition um 1 im Binärsystem

Eingabe: Binärdarstellung $x_{n-1} \ldots x_0$ von x
Ausgabe: Binärdarstellung von $x + 1$

Algorithmus:

Falls $x_{n-1} \ldots x_0 = (1 \ldots 1)$ ist,
gib $y_n \ldots y_0 = (1\ 0 \ldots 0)$ aus,
sonst *suche kritische Position*, d.h. das kleinste i mit $x_i = 0$.
gib $(x_{n-1} \ldots x_{i+1}\ 1\ 0 \ldots 0)$ aus.

Laufzeit: # veränderte Bits

worst case : $n + 1$ (bei Eingabe $1 \ldots 1$)
best case : 1 (z.B. bei Eingabe $1 \ldots 10$)

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

Average Case:

1 Operation mit Wahrscheinlichkeit ½
2 Operationen mit Wahrscheinlichkeit ¼
...
n Operationen mit Wahrscheinlichkeit $1/2^n$

1*1/2 + 2*1/4+3*1/8+... = $\sum_{i=0}^{n-1} \frac{1}{2^{i+1}}(i+1) \leq \sum_{i=0}^{\infty} \frac{1}{2^{i+1}}(i+1) = 2$

- Average case nahe bei best case!

- Es gibt auch Beispiele, wo average case nahe bei worst case, bzw. weit weg von worst case und best case ist.

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beispiel, das die Abhängigkeit der Laufzeit von der Größe der Eingabe verdeutlicht:**
  - Def. Fibonacci-Zahlen: $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$
  - Sehr langsamer Algorithmus:

```
fib(n)
  falls n <= 1 gib F_n aus
  sonst gib fib(n-1)+fib(n-2) aus
```

    Laufzeit: O(2^n), aber n ist ein Index in die Fibonacci-Folge, und dessen Kodierung hat logarithmische Länge in n. Sei also k = <n>. Dann ist die Laufzeit in k: O((2^(2^k))

  - Langsamer Algorithmus:

```
f0=0; f1=1
for i = 2 to n do
    tmp=f1;
    f1=f1+f0;
    f0=tmp;
falls n=0, gib f0 aus, sonst gib f1 aus
```

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

– Def. Fibonacci-Zahlen: $F_0=1$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$

Schneller Algorithmus:

betrachte $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ *sowie* $F = \begin{pmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{pmatrix}$

$$A^1 = \begin{pmatrix} f_2 & f_1 \\ f_1 & f_0 \end{pmatrix}, A^2 = \begin{pmatrix} f_3 & f_2 \\ f_2 & f_1 \end{pmatrix}, \textit{und}$$

$$F \cdot A = \begin{pmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} f_n + f_{n-1} & f_n \\ f_{n-1} + f_{n-2} & f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

Offenbar ist $A^n_{1,2}$ die n-te Fibonacci-Zahl. Nur, was bringt uns das?

Algorithmus und Rechenmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Berechnung von A^n
- Betrachte die Binärdarstellung von

$$n = \sum_{i=0}^{k} x_i \cdot 2^i, \quad x_i \in \{0,1\} \text{ und } k = \lfloor \log_2(|n|) \rfloor \text{ und damit}$$

$$A^n = A^{\sum_{i=0}^{k} x_i \cdot 2^i} = \prod_{i=0}^{k} A^{x_i \cdot 2^i} = \prod_{\substack{x_i = 1 \\ 0 \le i \le k}} A^{2^i}$$

- z.B. m = 13 = 1101_2. Bilde A, A^4, A^8 und bilde $A^*A^4*A^8 = A^{1+4+8}$

  Dabei ist $A^{(2^i)} = (A^{\overbrace{2 \cdots 2}^{O(k)-mal}}) = \overbrace{(\ldots(A^2)^2\ldots)^2}^{O(k)-mal\ quadrieren}$

- Aufwand für Potenzenbildungen: O(k)
  Aufwand für A^n berechnen: O(k)

Die Klassen P und NP

TECHNISCHE UNIVERSITÄT DARMSTADT

Entscheidungsproblem

- Problem, das nur zwei mögliche Antworten besitzt. „ja“ oder „nein“
- Beispiele: Ist n eine Primzahl? Gibt es einen Lösungsweg beim Solitärbrettspiel?

Optimierungsproblem

- geg.: möglicherweise implizit beschriebene Menge Ω von möglichen Lösungen und eine Bewertungsfunktion $f : \Omega \rightarrow IR$.
  ges: eine Lösung x mit $f(x) = \max\{ g(x) \mid x \in \Omega \}$
- Beispiele: Finde einen bestmöglichen Flugplan.

Entscheidungs- und Optimierungsprobleme lassen sich ineinander umformulieren.

Die Klassen P und NP

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Klasse *P*: informelle Beschreibung

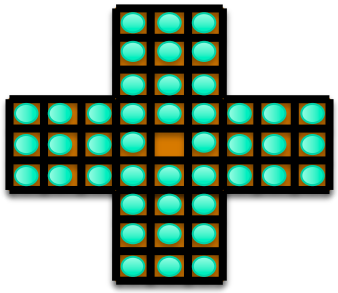
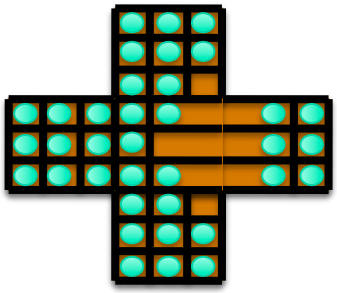
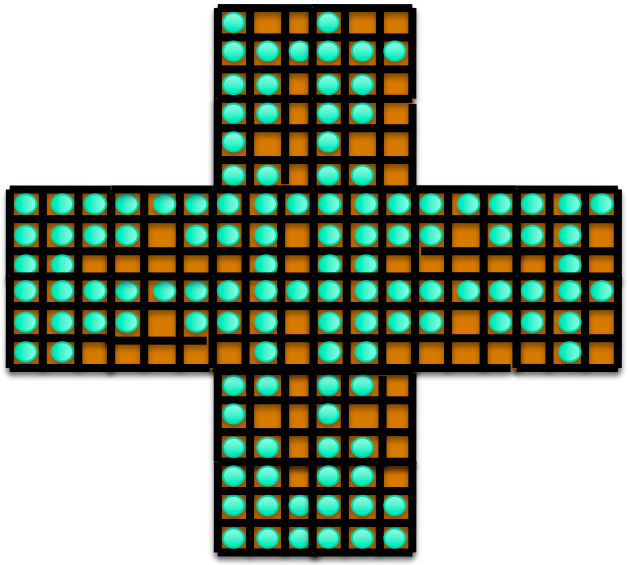
- Menge derjenigen Entscheidungsprobleme, für die es einen Algorithmus gibt, der worst-case polynomielle Laufzeit besitzt und der das Entscheidungsproblem löst.

Die Klasse P: formale Definition

- Gegeben sei ein Kodierungsschema *E* und ein Rechnermodell *M*.
- *Π* sei ein Entscheidungsproblem, wobei jede Instanz aus *Π* durch das Kodierungsschema *E* kodiert sei
- *Π* gehört zur Klasse *P* (bzgl. *E* und *M*), wenn es einen auf *M* implementierbaren Algorithmus zur Lösung alle Probleminstanzen aus *Π* gibt, dessen worst-case Laufzeitfunktion auf M polynomiell ist.

Die Klassen P und NP, Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

Geg: Eine beliebige Startposition im $n \times n$-Solitär
Ges: ja/nein mit ja, wenn schon mehr als die Hälfte der Steine vom Brett sind.
→ einfach
→ in P

Geg: Eine beliebige Startposition im $n \times n$-Solitär
Ges: ja/nein mit ja, wenn es einen Weg gibt, Steine so zu schlagen, , dass genau ein Stein in der Mitte übrigbleibt.
→ intuitiv nicht so einfach.
→ in „NP“

Die Klassen P und NP

TECHNISCHE UNIVERSITÄT DARMSTADT

_NP_, Definition 1:

Ein Entscheidungsproblem Π gehört in die Klasse *NP*, wenn gilt:

- Für jedes Problembeispiel $I \in \Pi$, für das die Antwort „ja“ lautet, gibt es (mindestens) ein Objekt *Q*, mit dessen Hilfe die Korrektheit der Antwort „ja“ überprüft werden kann.
- Es gibt einen Algorithmus, der Problembeispiele $I \in \Pi$ und Zusatzobjekte *Q* als Input akzeptiert und der in der Laufzeit polynomiell in $<I>+<Q>$ ist, überprüft, ob *Q* ein Objekt ist, aufgrund dessen eine „ja“-Antwort gegebene werden muss.

- Es wird keine Aussage darüber gemacht, wie *Q* berechnet wird. *Q* kann geraten werden.
- Die einzige Aussage, die über „nein“-Antworten gemacht wird, ist, dass es einen Algorithmus geben muss, der korrekt „ja“ oder „nein“ entscheiden kann.

Die Klassen P und NP

TECHNISCHE UNIVERSITÄT DARMSTADT

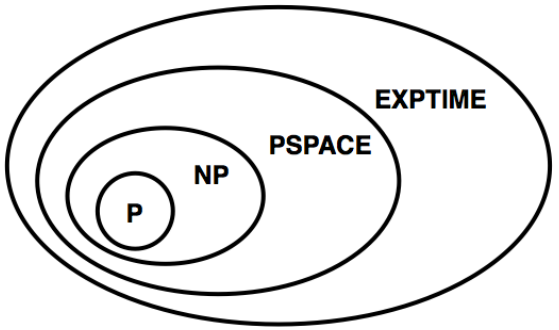
_NP_, Definition 2 (gleichwertig zur vorigen Definition):
Hier wird die Klasse *NP* über eine so genannte nicht-deterministische RAM definiert. Eine solche Maschine ist eine RAM mit dem zusätzlichen Befehl „goto L1 or goto L2;“.

Ein Problem Π ist in *NP*, wenn es einen (nicht-deterministischen) Algorithmus *A* (für die nicht-deterministische RAM) gibt, so dass es für jedes Problembeispiel $I \in \Pi$, für das die Antwort „ja“ lautet, einen Berechnungsweg von *A* gibt, der polynomielle Länge in $<I>$ besitzt. Für Instanzen, für die die Antwort „nein“ ist, darf *A* auf keinem Berechnungsweg die Antwort „ja“ ausgeben. Für alle Instanzen muss A irgendwann anhalten.

P, NP, PSPACE

TECHNISCHE UNIVERSITÄT DARMSTADT

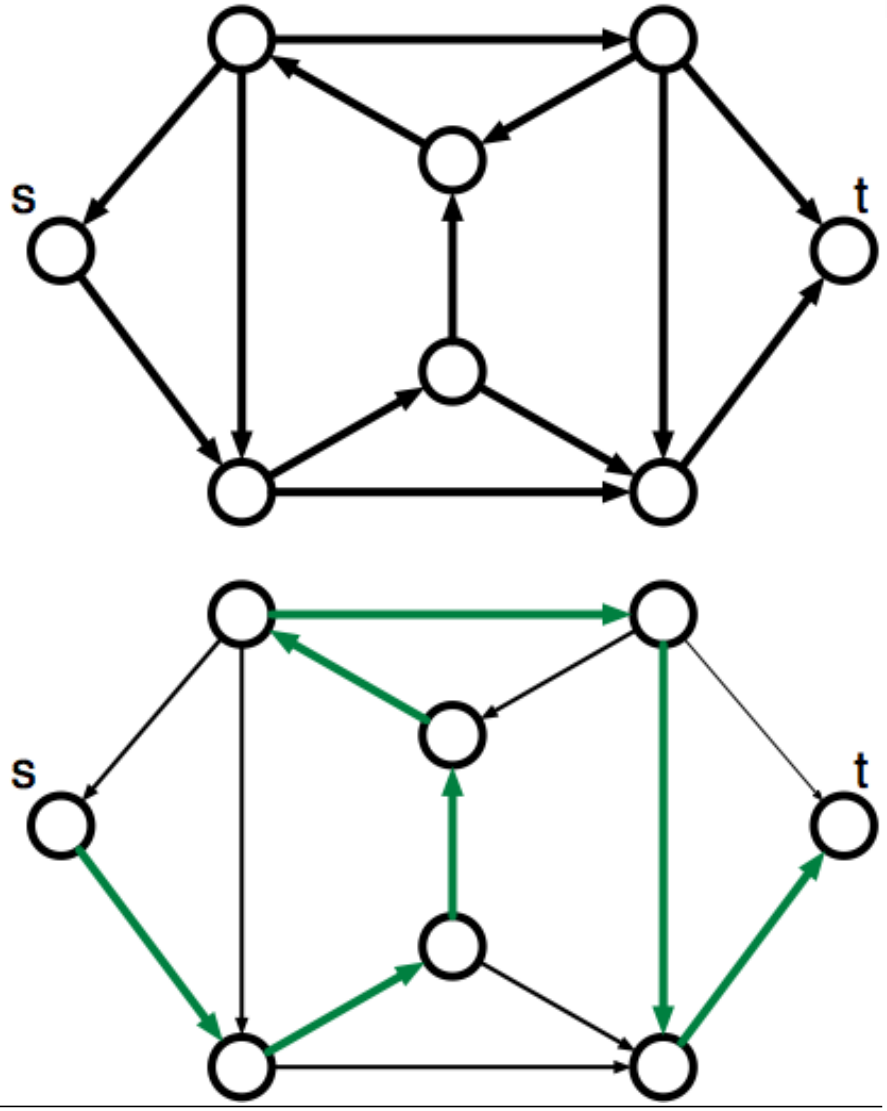
- **P:** Klasse aller Probleme, die von einer deterministischen RAM in Polynomzeit gelöst werden können
- **NP:** Klasse aller Probleme, die von einer nichtdeterministischen RAM in Polynomzeit gelöst werden können.
- **PSPACE :** Klasse aller Probleme, die von einer deterministischen RAM mit polynomiell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



Beispiele aus der Klasse NP

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Definition: *HAMPATH***
  - Das Hamiltonsche Pfadproblem
    - Geg.:
      - ein gerichteter Graph
      - Zwei Knoten s,t
    - Ges.: existiert ein so genannter Hamitonscher 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 $(s,v_1,v_2,...,v_{n-2},t)$
  - Teste, ob Permutation ein Pfad ist
    - falls ja, akzeptiere
    - falls nein, verwerfe
- **Also: HamPath $\in$ NP**



Beispiele aus der Klasse NP

TECHNISCHE UNIVERSITÄT DARMSTADT

Das SAT Problem

- Eine Boolesche Funktion $f(x_1,x_2,..,x_n)$ ist erfüllbar, wenn es eine Wertebelegung für $x_1,x_2,..,x_n$ gibt, so dass $f(x_1,x_2,..,x_n) = 1$
  - $(x \vee y) \wedge (z \vee \neg x \vee \neg y) \wedge (x \vee \neg z)$ 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 $x_1,x_2,..,x_n$ so dass $\phi(x_1,x_2,..,x_n) = 1$

- SAT ist in NP. Man vermutet, dass SAT nicht in P ist.

Einordnung von Problemen in P, NP, PSPACE

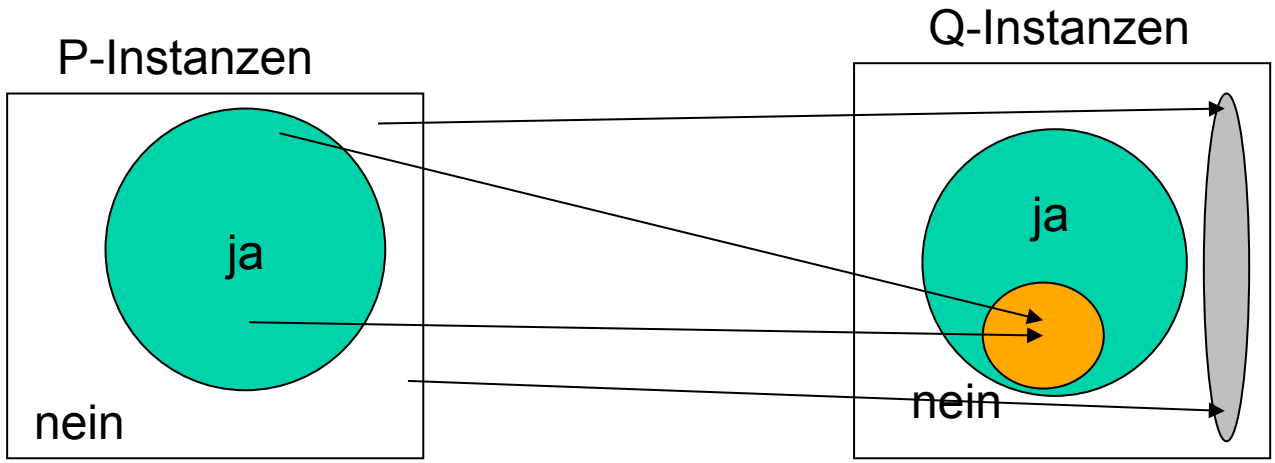
TECHNISCHE UNIVERSITÄT DARMSTADT

Die Reduktionstechnik

Definition: Seien P,Q Probleme. Sei L_P (bzw. L_Q) die Menge der Instanzen des Problems P (bzw. Q), für die die Antwort „ja“ ist. Sei zudem Σ ein Alphabet zur Problemkodierung und Σ^* bezeichne die Menge aller Symbolketten, die aus dem Alphabet erzeugt werden können. 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

$$x \in L_P \Leftrightarrow f(x) \in L_Q$$

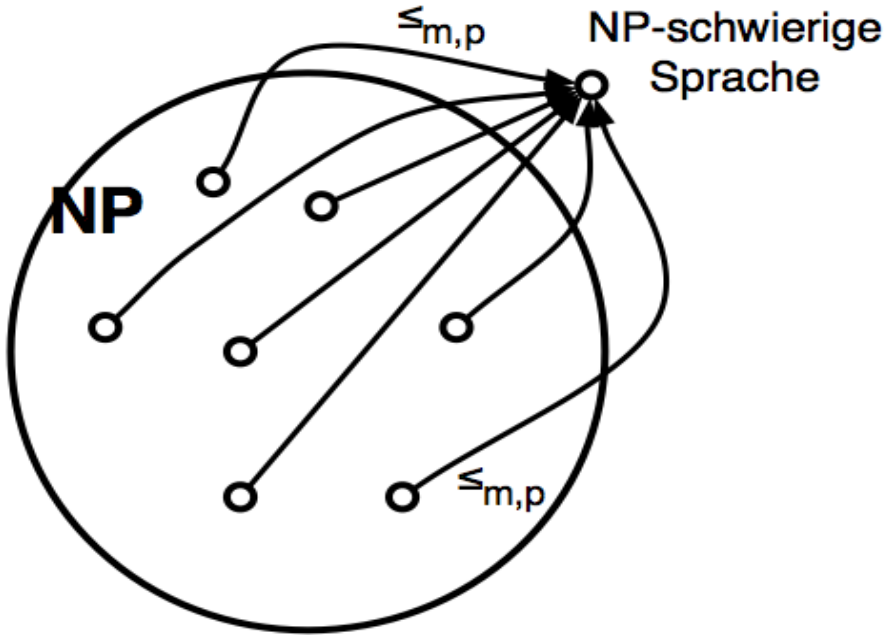
Bsp.:



NP-Schwierig

TECHNISCHE UNIVERSITÄT DARMSTADT

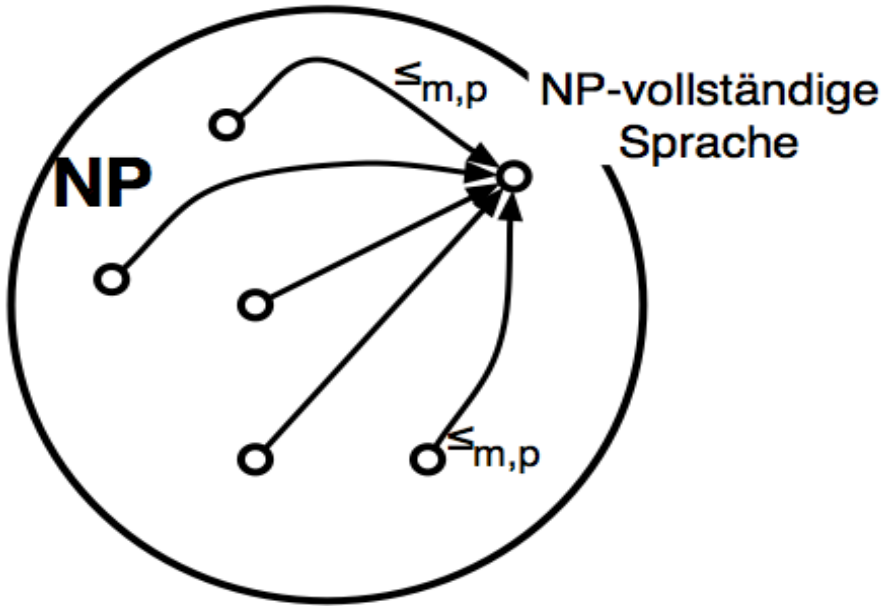
- Definition:
  - Eine Sprache S ist **NP-schwierig** (NP-hard) wenn:
    - jede Sprache aus NP mit einer Polynom-Zeit-Abbildungsreduktion auf S reduziert werden kann, d.h.
    - für alle $L \in NP$: $L \leq_p S$
- Theorem
  - Falls eine NP-schwierige Sprache in P ist, ist P=NP
- Beweis
  - Falls $S \in P$ und $L \leq_p S$ gilt $L \in P$.



NP-Vollständigkeit

TECHNISCHE UNIVERSITÄT DARMSTADT

- Definition:
  - Eine Sprache S ist **NP-vollständig** (NP-complete) wenn:
    - S ∈ NP
    - S ist NP-schwierig
- Korollar:
  - Ist eine NP-vollständige Sprache in P, dann ist P=NP
- Beweis:
  - folgt aus der NP-Schwierigkeit der NP-vollständigen Sprache.

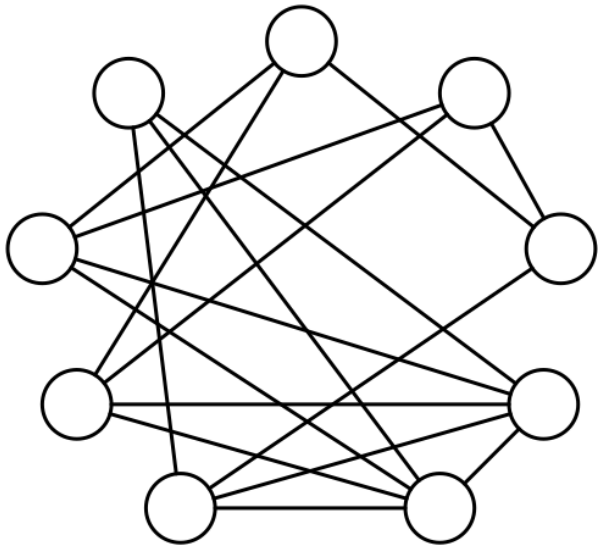


Das 3-SAT-Problem und das Clique-Problem

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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?

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

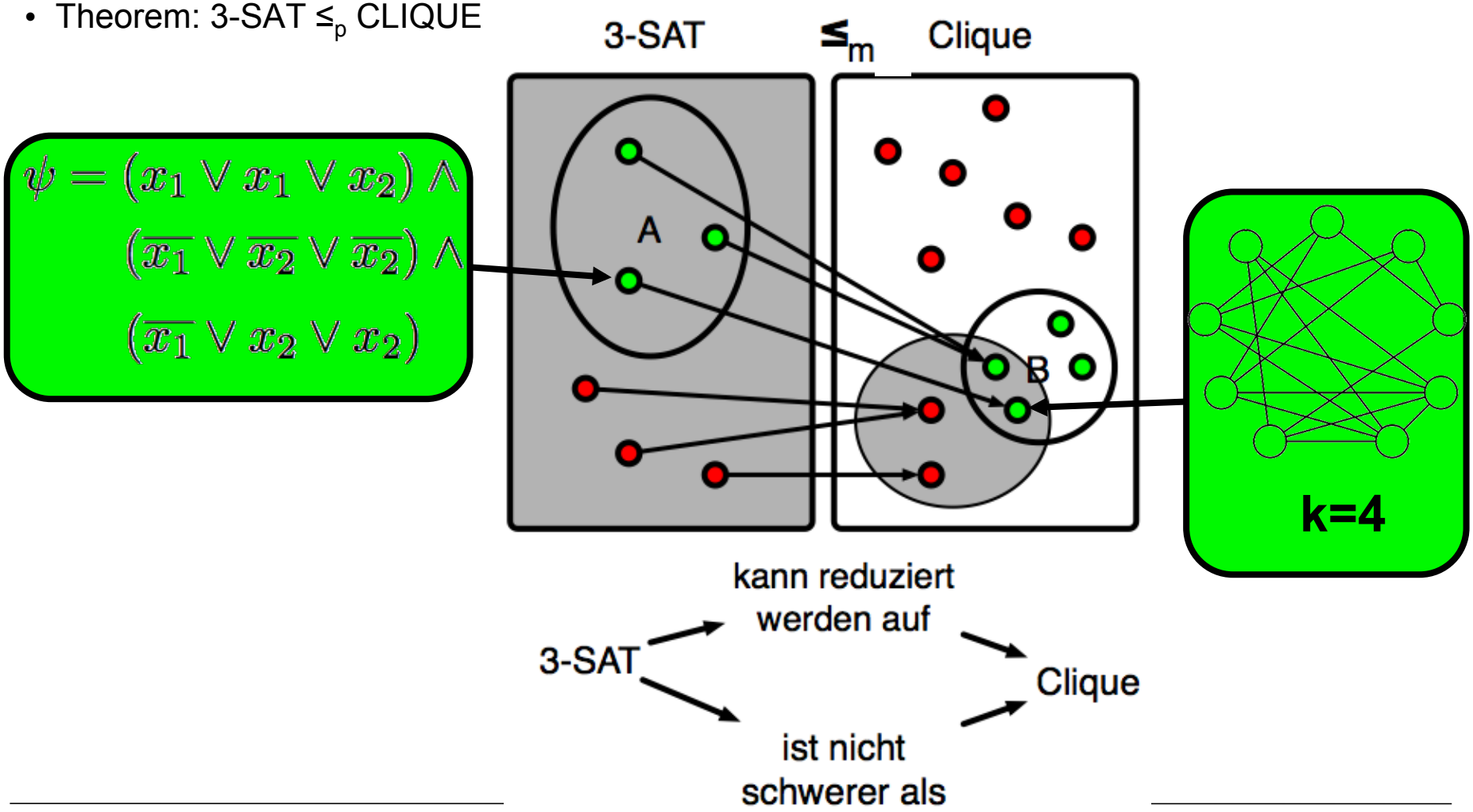


k=4

3-SAT läßt sich auf Clique reduzieren

TECHNISCHE UNIVERSITÄT DARMSTADT

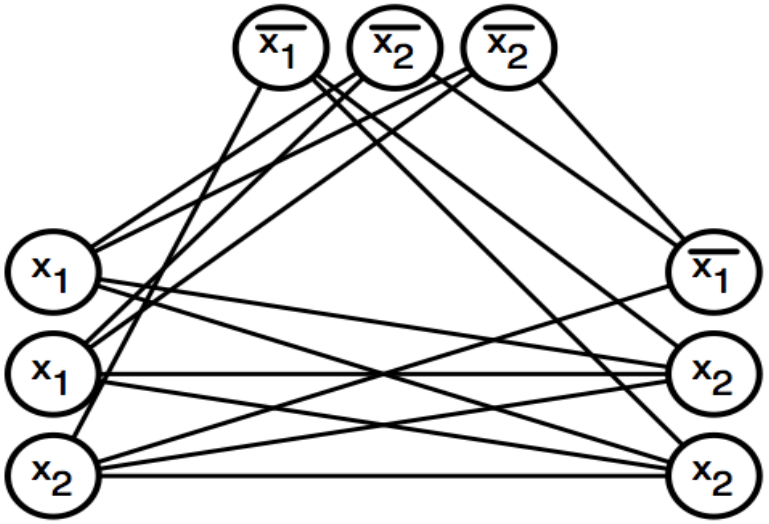
- Theorem: 3-SAT $\leq_p$ CLIQUE



3-SAT läßt sich auf Clique reduzieren

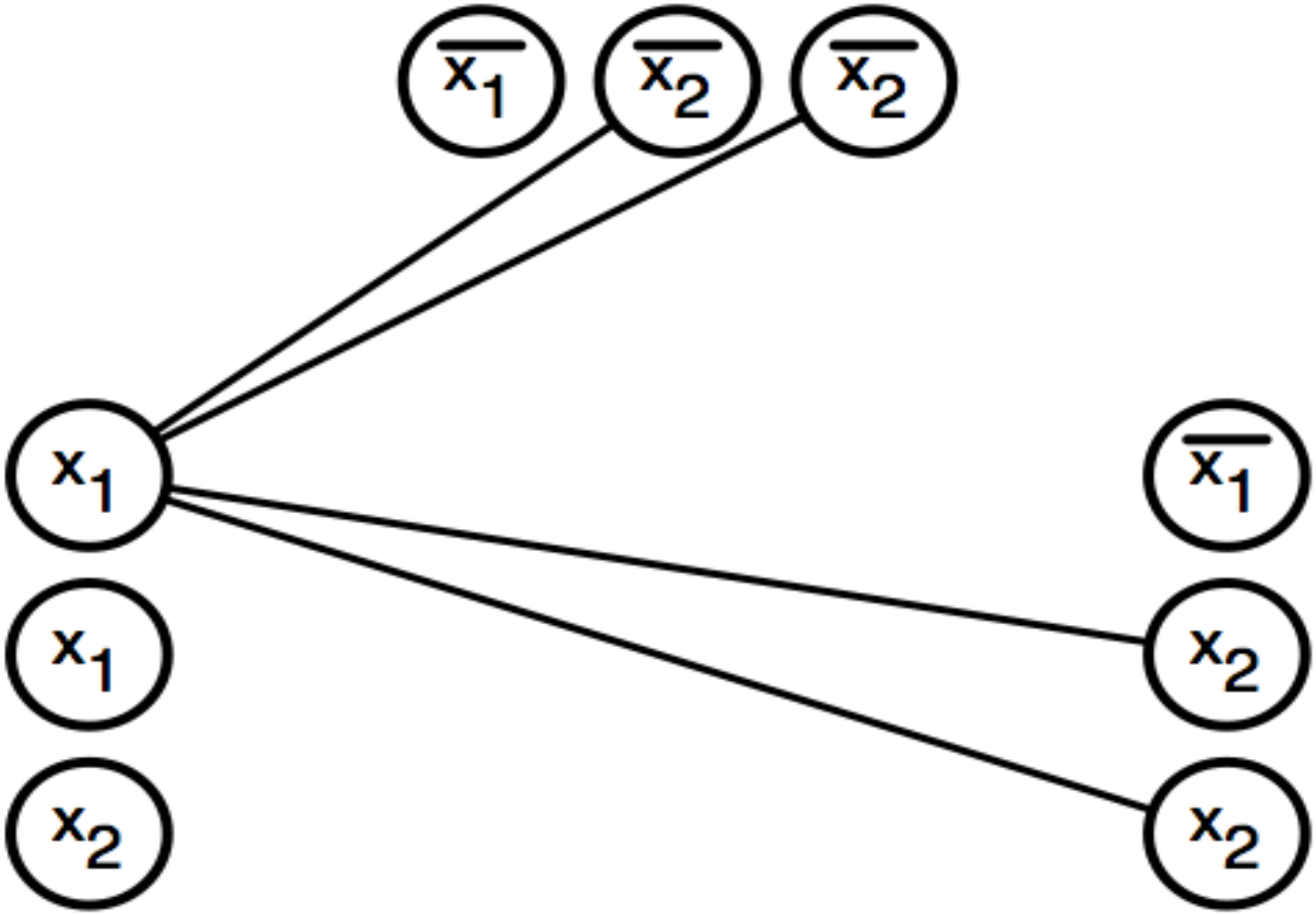
TECHNISCHE UNIVERSITÄT DARMSTADT

- Theorem: 3-SAT $\leq_{m,p}$ CLIQUE
- Beweis
  - Konstruiere Reduktionsfunktion f wie folgt:
  - f(ϕ) = <G,k>
  - 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.

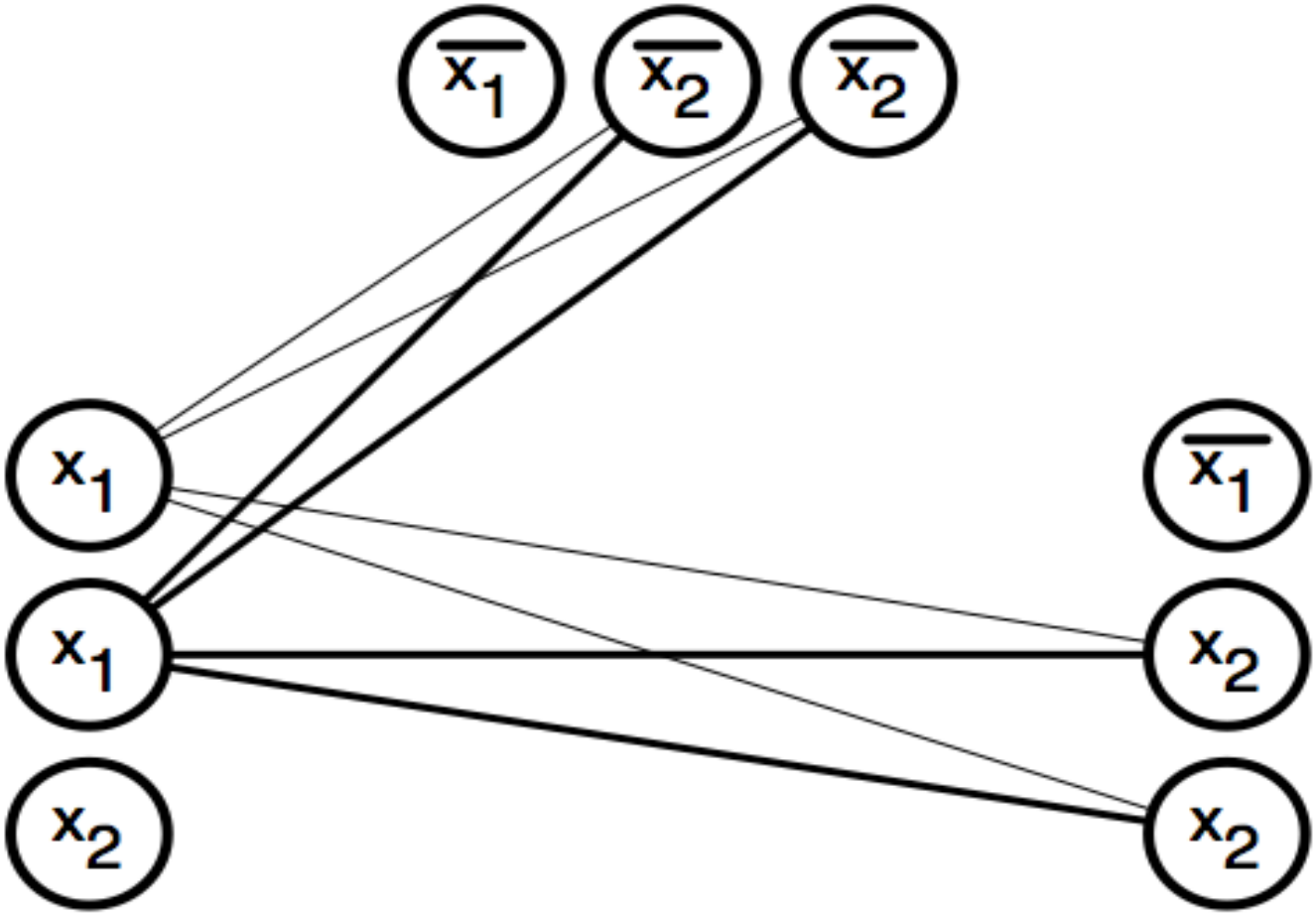


$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

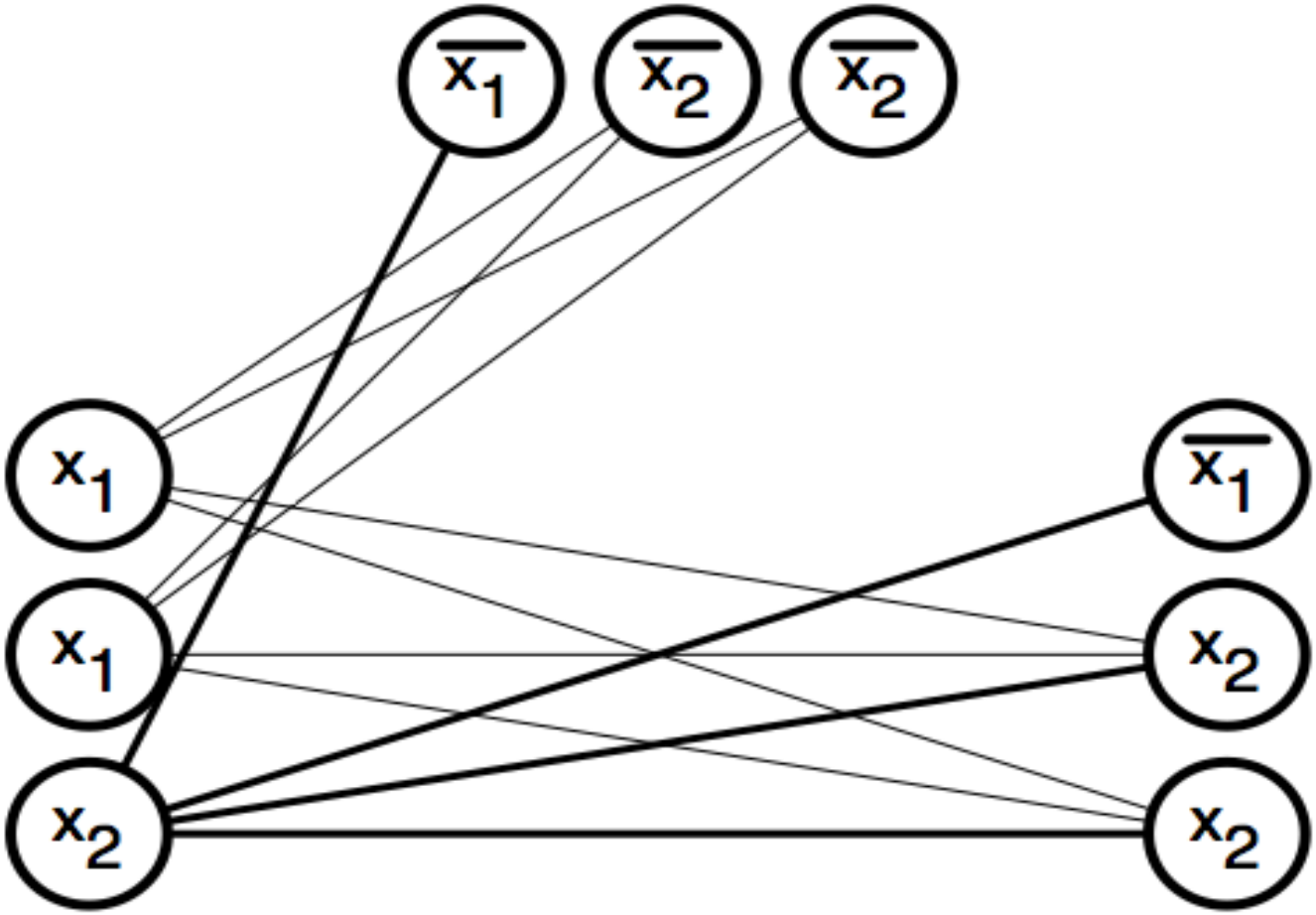
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



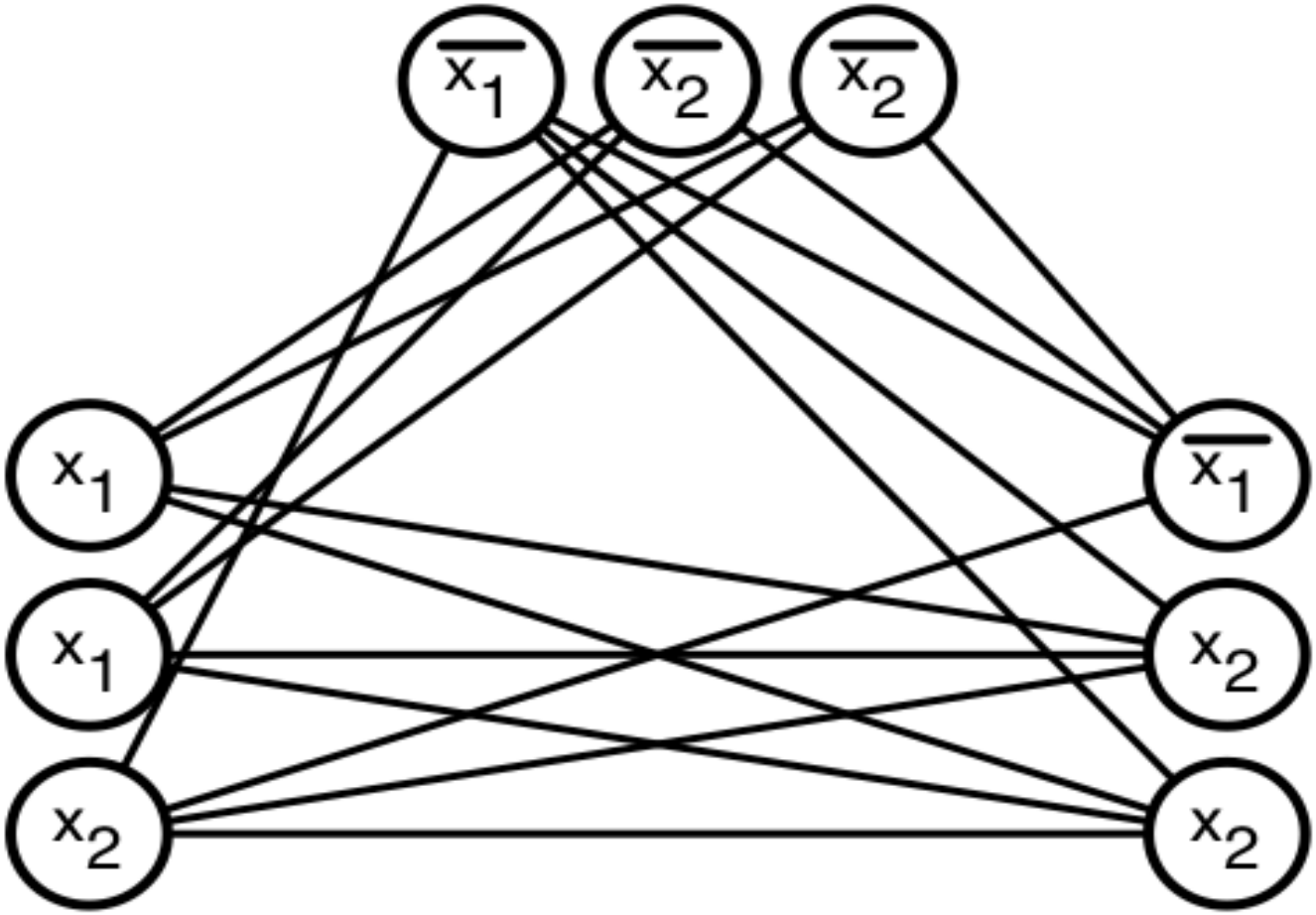
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



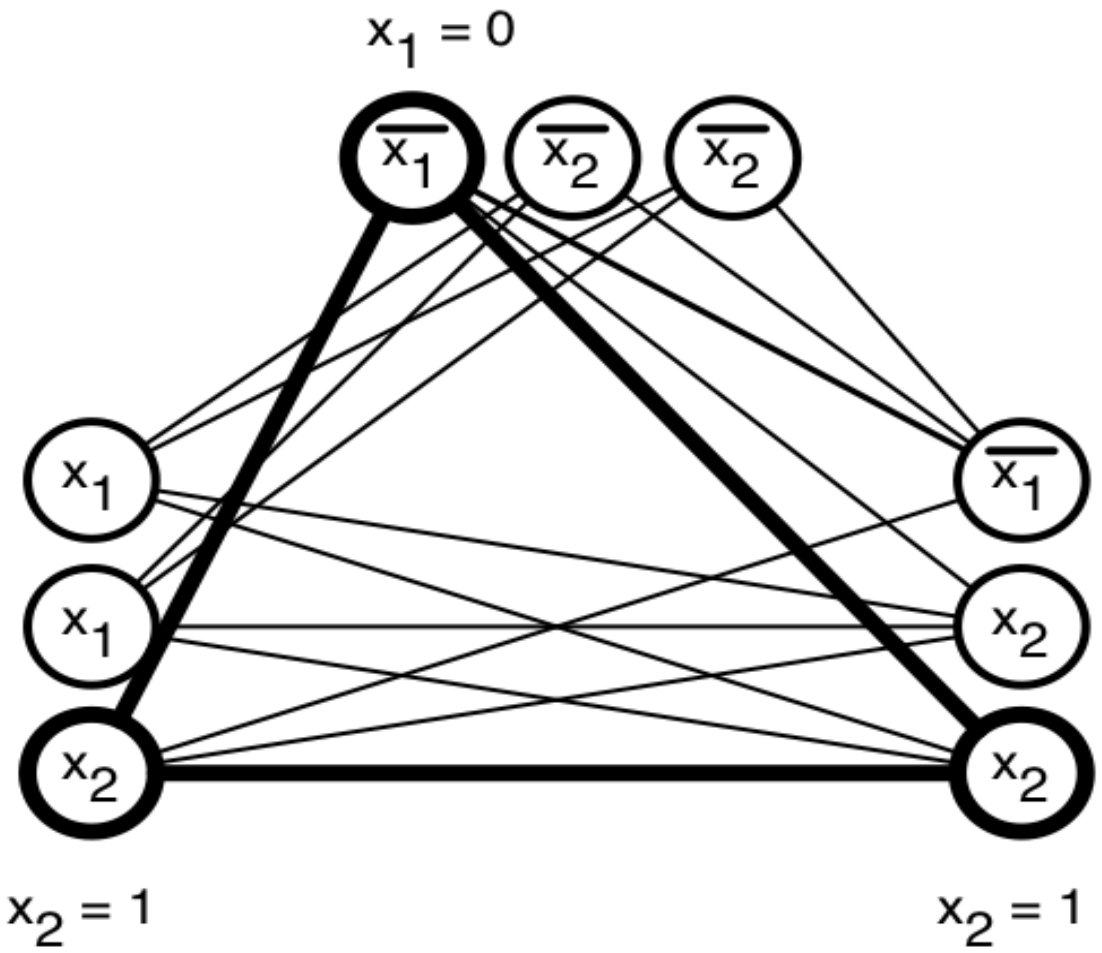
$$\psi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



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



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



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

0	0	1	1	0	0	1	1	1

Beweis der Korrektheit der Reduktionsfunktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Reduktionsfunktion ist korrekt:
- Behauptung;
  - Eine erfüllende Belegung in ϕ existert 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
  - Bestimmte 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.