Algorithmen und Datenstrukturen

Prof. Marc Fischlin, SS 2024

08

NP

Folien beruhen auf gemeinsamer Veranstaltung mit Christian Janson aus dem SS 2020

Leichte und (nicht zu) schwierige Probleme

Ansatz: Problem ist leicht, wenn es in Polynomialzeit lösbar ist

(Worst-Case-)Laufzeit des Algorithmus ist also $\Theta(\sum_{i=0}^{k} a_i n^i) = poly(n)$

a_i, k konstant

leicht zu lösen	**Lösung leicht zu prüfen**	**unentscheidbar**
Sortieren eines Arrays	TSP	Halteproblem
Breitensuche im Graphen	Faktorisieren	Code-Erreichbarkeit
Minimale Spannbäume berechnen	…	…
…		

Halteproblem

Gesucht: Programm H, so dass $H(P) = \begin{cases} 1 & falls\ P(P)\ anhält \\ 0 & sonst \end{cases}$

Unentscheidbarkeit des Halteproblems:
Es gibt kein Programm H, das das Halteproblem löst.

Sonst betrachte H^* mit $H^*(P) = \begin{cases} hält\ an & falls\ H(P) = 0 \\ hält\ nicht\ an & sonst \end{cases}$

Dann: $H(H^*) = 1 \underset{\text{Definition } H}{\Leftrightarrow} H^*(H^*)\ anhält \underset{\text{Definition } H^*}{\Leftrightarrow} H(H^*) = 0$

Widerspruch

Berechnungsprobleme
vs.
Entscheidungsprobleme

Berechnungs- vs. Entscheidungsprobleme (I)

Berechnungsproblem:

Gegeben: Problem $\mathbf{P}$
Gesucht: Lösung $\mathbf{S}$

Beispiel:
Berechne kürzeste
Pfade im Graphen

Entscheidungsproblem:

Gegeben: Problem $\mathbf{P}$
Gesucht: Hat $\mathbf{P}$ Eigenschaft $\mathbf{E}$?
(0/1-Antwort)

Beispiel:
Ist gerichteter Graph
stark zusammenhängend?

Wir betrachten im Folgenden nur Entscheidungsprobleme!

Berechnungs- vs. Entscheidungsprobleme (II)

Man kann jedes Berechnungs- in ein Entscheidungsproblem überführen, so dass Polynomialzeit-Lösung für Entscheidungsproblem auch Polynomialzeit-Lösung für Berechnungsproblem ergibt.

Faktorisierungsproblem:

Gegeben: n-Bit-Zahl $N \geq 2$

Gesucht: Primfaktoren von N

↓

Entscheidungsproblem:

Gegeben: n-Bit-Zahl $N \geq 2$, Zahl B

Gesucht: Ist kleinster Primfaktor von N maximal B?

Beispiel: Faktorisieren (I)

```
computeFactor(N) //use decideFactor(N,B) as sub
                 //N>1, computes prime factor of N
L=1; U=N;
WHILE L!=U DO
   M=L+floor((U-L)/2);
   IF decideFactor(N,M)==1 THEN U=M ELSE L=M+1;
return L;
```

```
decideFactor(N,B)

…
return d; // d==0 or 1
```

```
Factorize(N) // N>1

WHILE N>1 DO
   p=computeFactor(N);
   print p;
   N=N/p;
```

Entscheidungsproblem:

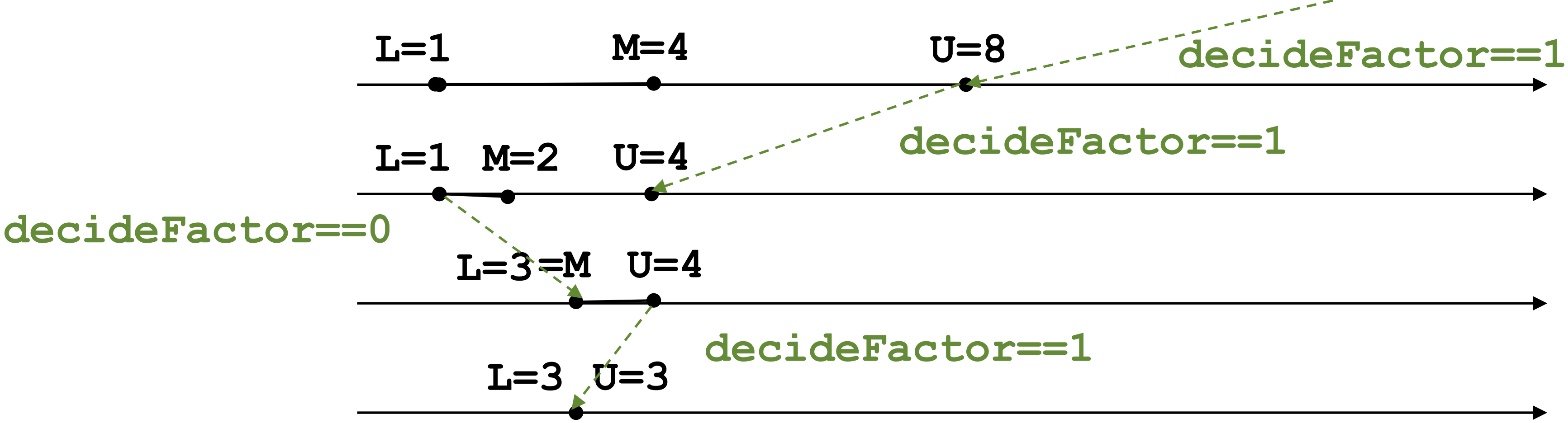
Gegeben: n-Bit-Zahl $N \geq 2$, Zahl B

Gesucht: Ist kleinster Primfaktor von N maximal B?

Beispiel: Faktorisieren (II)

```
computeFactor(N) //use decideFactor(N,B) as sub
                 //N>1, computes prime factor of N
L=1; U=N;
WHILE L!=U DO
   M=L+floor((U-L)/2);
   IF decideFactor(N,M)==1 THEN U=M ELSE L=M+1;
return L;
```

Beispiel:
$N = 15$



Beispiel: Faktorisieren (III)

```
computeFactor(N) //use decideFactor(N,B) as sub
                 //N>1, computes prime factor of N
1  L=1; U=N;
2  WHILE L!=U DO
3     M=L+floor((U-L)/2);
4     IF decideFactor(N,M)==1 THEN U=M ELSE L=M+1;
5  return L;
```

In jeder Iteration wird Suchintervall um Hälfte reduziert (wir ignorieren Runden)

Zu Beginn Intervalllänge N, also nach $\Theta(\log_2 N) = \Theta(n)$ Iterationen fertig

Laufzeit: $\Theta(\log_2 N) = \Theta(n)$ Iterationen von **`decideFactor`**

(und in jeder Iteration zusätzlich konstanter Aufwand)

Beispiel: Faktorisieren (IV)

```
computeFactor(N) //use decideFactor(N,B) as sub
                 //N>1, computes prime factor of N
L=1; U=N;
WHILE L!=U DO
   M=L+floor((U-L)/2);
   IF decideFactor(N,M)==1 THEN U=M E
return L;
```

Terminierung +
Schleifeninvariante
zeigen Korrektheit

Korrektheit (unter Annahme, dass **`decideFactor`** korrekt):

Schleifeninvariante: Zwischen **`L`** und **`U`** liegt stets ein Primfaktor von **`N>1`**

Induktionsbasis: Gilt zu Beginn wegen **`L=1`** und **`U=N`**

Induktionsschritt: Nach Voraussetzung Faktor zwischen **`L`** und **`U;`**

Wenn ein Faktor =<**`M,`** dann wird **`U=M`** gesetzt;

Wenn kein Faktor =<**`M,`** dann wird **`L=M+1`** gesetzt;

Beispiel: Faktorisieren (V)

```
computeFactor(N) //use decideF
                 //N>1, comput
L=1; U=N;
WHILE L!=U DO
   M=L+floor((U-L)/2);
   IF decideFactor(N,M)==1 THEN U=M ELSE L=M+1;
return L;
```

Laufzeit: $\Theta(\log_2 N) = \Theta(n)$
Iterationen von **decideFactor**

```
decideFactor(N,B)

…
return d;
```

(Annahme)
Laufzeit: $poly(n)$

```
Factorize

WHILE N>1 DO
   p=computeFactor(N);
   print p;
   N=N/p;
```

Gesamtlaufzeit:
$\Theta(n^2 \cdot poly(n))$

In jeder Iteration wird Primfaktor **p>=2** abgespalten, also maximal $\Theta(\log_2 N) = \Theta(n)$ Iterationen

Berechnung durch Entscheidung (I)

Berechnungsproblem:
Gegeben: Problem **P**
Gesucht: Lösung **S**

Entscheidungsproblem:
Gegeben: Problem **P**, String **s**
Gesucht: Ist **s** Präfix der Binärdarstellung einer Lösung **S**?

```
decide(P,s)

…
return d; // d==0/1
```

```
compute(P) //use decide(P,s) as sub

s=““; // empty string
IF decide(P,s)==0 THEN return “no solution“;
done=false;
WHILE !done DO
   szero=decide(P,s+“0“);
   sone =decide(P,s+“1“);
   IF szero==0 AND sone==0 THEN
      done=true
   ELSE IF szero==1 THEN s=s+“0“ ELSE s=s+“1“;
return “solution “ + s;
```

Lösung gefunden

sucht bit-weise in richtige Richtung

Berechnung durch Entscheidung (II)

Sofern Bitlänge der Lösungen polynomiell beschränkt und `decide` in Polynomialzeit, läuft `compute` auch in Polynomialzeit

Laufzeit: $\Theta\left(2 \cdot \max_{S} |\mathbf{S}| + 1\right)$ Iterationen von `decide`

```
compute(P) //use decide(P,s) as sub

s=““; // empty string
IF decide(P,s)==0 THEN return “no solution“;
done=false;
WHILE !done DO
    szero=decide(P,s+“0“);
    sone =decide(P,s+“1“);
    IF szero==0 AND sone==0 THEN
        done=true
    ELSE IF szero==1 THEN s=s+“0“ ELSE s=s+“1“;
return “solution “ + s;
```

Überlegen Sie sich, was mit **`computeFactor`** passiert, wenn die Unterroutine **`decideFactor`** manchmal eine falsche Antwort zurückgibt. Lösungsideen?

Welche (Bit-)Eigenschaft hat die von **`compute`** berechnete Lösung?

Komplexitätsklassen P und NP

Komplexitätsklasse P

Betrachte Entscheidungsproblem für Eigenschaft E als Menge:

$$L_E = \{P \mid P \; hat \; Eigenschaft \; E\}$$

(L von „language“)

Beispiel: $L_{SC} = \{G \mid G \; ist \; gerichteter, stark \; zusammenhängender \; Graph\}$

Komplexitätsklasse P:
Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse **P**, wenn es einen Polynomialzeit-Algorithmus A_{L_E} mit Ausgabe 0/1 gibt, der stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also $P \in L_E \Leftrightarrow A_{L_E}(P) = 1$ für alle P gilt.

Eigentlich: Algorithmus=Turing-Maschine und Problem-Universum = $\{0,1\}^*$

Komplexitätsklasse NP (I)

Prüfen einer vermeintlichen Lösung ist einfach für L_E:

Gegeben: Problem P und vermeintliche Lösung S
Entscheide: Zeigt S, dass P Eigenschaft E hat oder nicht?

S dient als zusätzliche Entscheidungshilfe;
heißt auch „witness“, Zeuge, Zertifikat,… für P

Technische Einschränkung:
Lösungen S sind von polynomieller Komplexität in Eingabeproblem P;
meist: Lösungen S haben polynomielle Bitlänge (in Bitlänge von P)

Komplexitätsklasse NP (II)

Beispiel: $L_{Fakt} = \{(N, B) \mid N > 1 \; hat \; Primfaktor \; \leq B\}$

$(289{,}20) \in L_{Fakt}$
$(361{,}12) \notin L_{Fakt}$

Gegenwärtig unklar, wie in Polynomialzeit ohne Hilfe (und ohne Quantencomputer) zu entscheiden, ob Eingabe (N, B) in L_{Fakt} oder nicht

Mit Hilfe einfach:
Zeuge S zu $P = (N, B)$ ist Faktor p von N mit $1 < p \leq B$

```
verify(N,B,p) // check alleged solution

1  IF N>1 AND 1<p=<B and p|N THEN return 1 else return 0;
```

Hinweis: Wir prüfen nicht, dass p prim;
wenn zusammengesetzter Faktor in Schranke B, dann erst recht Primfaktor

Komplexitätsklasse NP (II)

Beispiel: $L_{Fakt} = \{(N,B) \mid N > 1 \; hat \; Primfaktor \; \leq B\}$

$(289{,}20) \in L_{Fakt}$
$(361{,}12) \notin L_{Fakt}$

Wichtig: es gibt keine „falsche“ Hilfe für nicht-zugehörige Eingaben:

Wenn $(N,B) \in L_{Fakt}$, dann gibt es S, das **`verify`** akzeptieren lässt

Wenn $(N,B) \notin L_{Fakt}$, dann gibt es **kein** S, das **`verify`** akzeptieren lässt

Entscheidung (mit Hilfe) muss in beiden Fällen richtig sein

```
verify(N,B,p) // check alleged solution

1  IF N>1 AND 1<p=<B and p|N THEN return 1 else return 0;
```

Hinweis: Wir prüfen nicht, dass p prim;
wenn zusammengesetzter Faktor in Schranke B, dann erst recht Primfaktor

Komplexitätsklasse NP (IV)

Zur Erinnerung: Komplexität der Hilfseingabe S_P polynomiell in der von P

Komplexitätsklasse NP:
Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse **NP**, wenn es einen Polynomialzeit-Algorithmus A_{L_E} mit Ausgabe 0/1 gibt, der **bei Eingabe eines Zeugen S_P für Eingabe $P \in L_E$ bzw. für jede Eingabe S_P für Eingabe $P \notin L_E$** stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also
$P \in L_E \Leftrightarrow \exists \boldsymbol{S_P}: A_{L_E}(P, \boldsymbol{S_P}) = 1$ für alle P gilt.

$P \notin L_E \Leftrightarrow \forall \boldsymbol{S_P}: A_{L_E}(P, S_P) = \mathbf{0}$ für alle P (äquivalent)

(**NP** steht für **N**icht-deterministische **P**olynomialzeit)

P vs. NP (I)

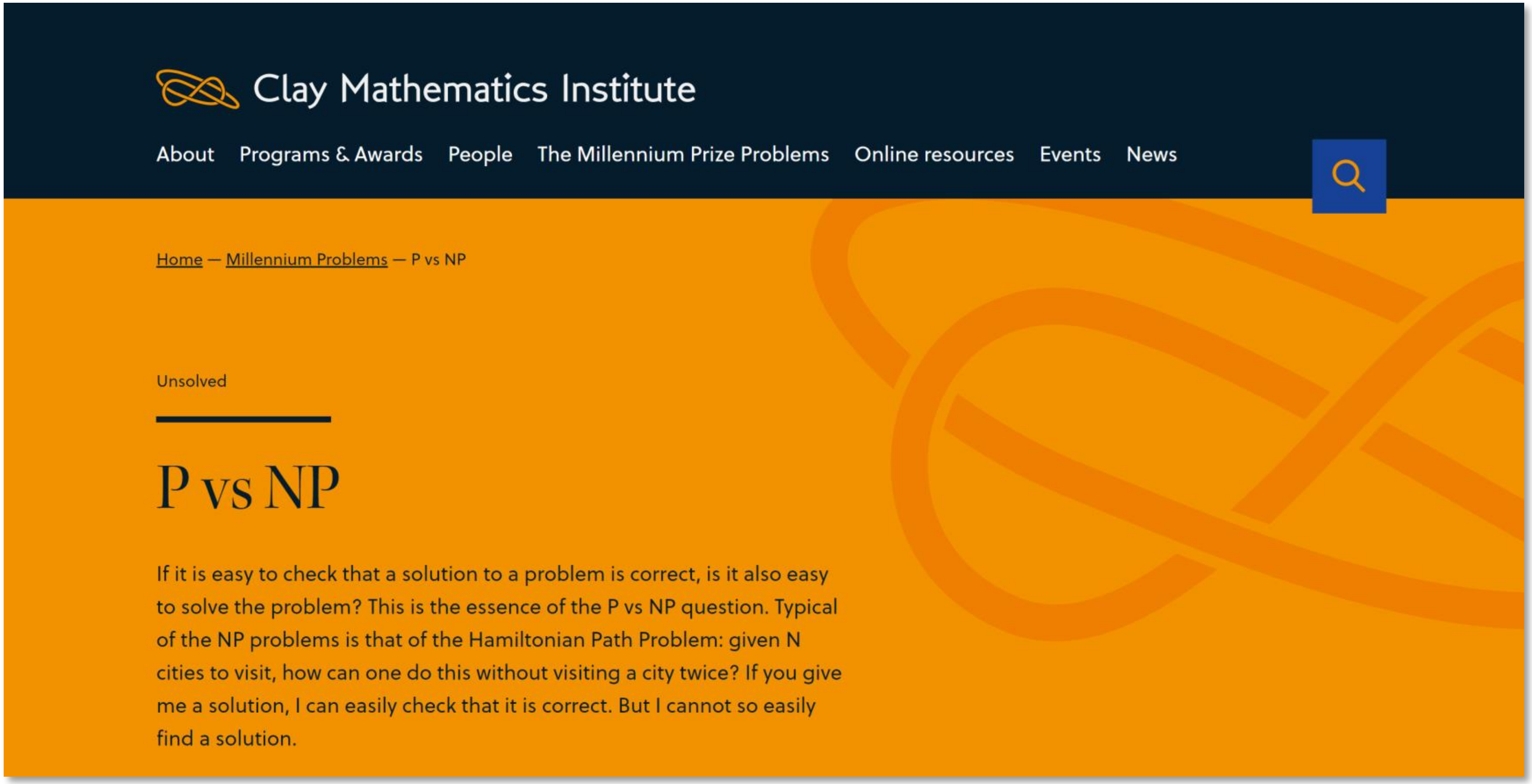
Komplexitätsklasse NP:
Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse **NP**, wenn es einen Polynomialzeit-Algorithmus A_{L_E} mit Ausgabe 0/1 gibt, der **bei Eingabe eines Zeugen S_P für Eingabe $P \in L_E$ bzw. für jede Eingabe S_P für Eingabe $P \notin L_E$** stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also
$P \in L_E \Leftrightarrow \exists S_P\colon A_{L_E}(P, S_P) = 1$ für alle P gilt.

Jedes Problem in **P** ist auch in **NP**: Algorithmus A_{L_E} entscheidet ohne Hilfe

Also: **P** ⊆ **NP** , aber bis heute offen, ob auch **NP** ⊆ **P**

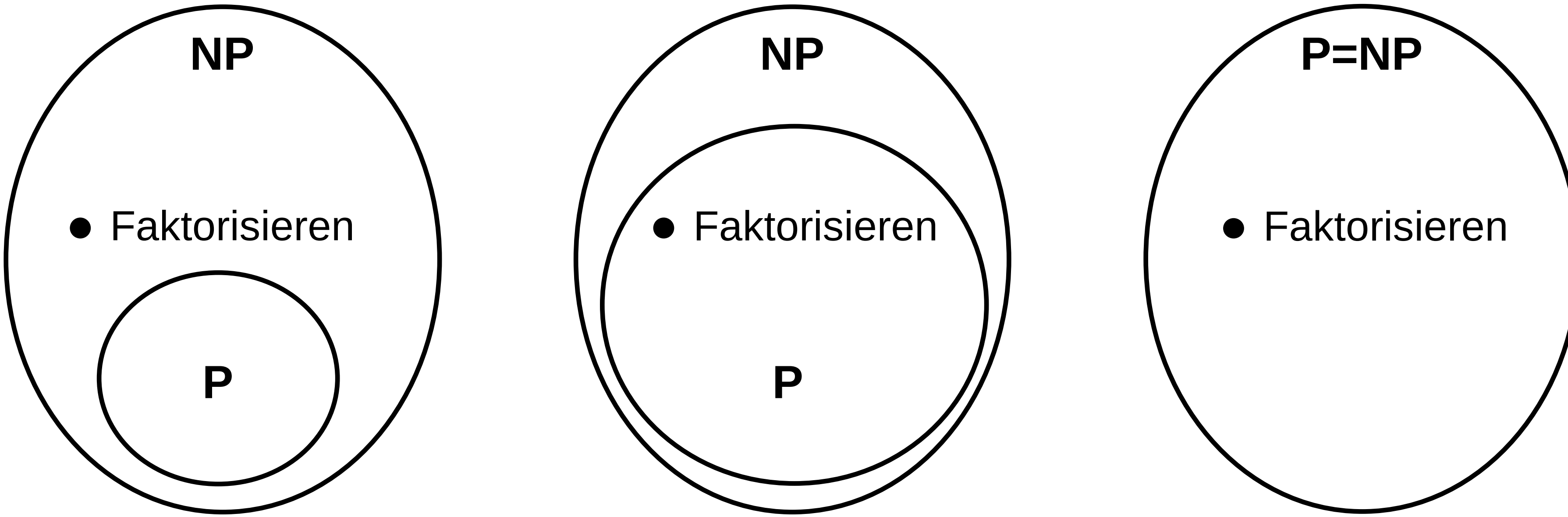
P vs. NP (II)

Eines der sechs verbleibenden (von ursprünglich sieben) ungelösten großen mathematischen Probleme:



P vs. NP (III)

Mögliche Welten:



P≠NP
Faktorisieren schwierig

gilt im Augenblick
als wahrscheinlichste Welt
(Quantum-Computer
verfeinern Bild)

P≠NP
Faktorisieren leicht

P=NP
alle (entscheidbaren)
Probleme leicht

Die Klasse **co-P** besteht aus allen Entscheidungsproblemen L_E, für die es einen Polynomialzeit-Algorithmus A_{L_E} gibt, so dass $P \notin L_E \iff A_{L_E}(P) = 1$ für alle P gilt.

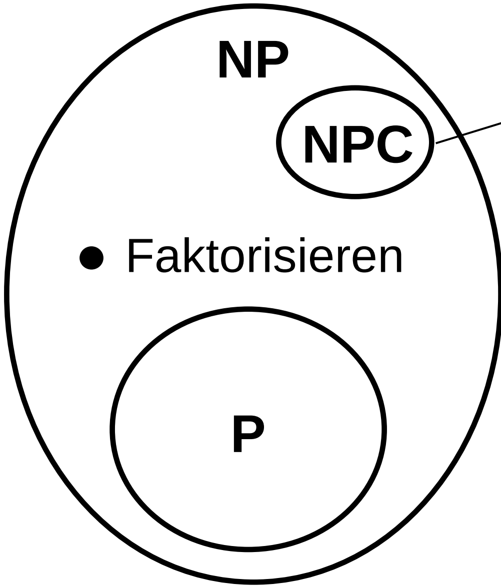
(A_{L_E} signalisiert stets korrekt, wenn P nicht in der Menge.)

Überlegen Sie sich, dass **P**=**co-P** ist.

Überlegen Sie sich, dass ein **NP**-Problem, bei dem der Zeuge maximal 2 Bits lang ist, in **P** sein muss.

NP-Vollständigkeit

Ziel: Identifiziere schwierigsten Probleme in NP



NPC = Klasse der **NP**-vollständigen Probleme („**NP**-**C**omplete“)

Eigenschaften:

(a) **NPC** $\subseteq$ **NP**

(b) Wenn **P** $\neq$ **NP**, dann definitiv **NPC** $\not\subseteq$ **P**

Reduktionen (I)

Reduktion=„Problemtransformation“

Zur Erinnerung:

Berechnungsproblem:
Gegeben: Problem **P**
Gesucht: Lösung **S**

$\Rightarrow$

Entscheidungsproblem:
Gegeben: Problem **P**, String **s**
Gesucht: Ist **s** Präfix der Binärdarstellung einer Lösung **S**?

…dann auch Berechnungsproblem einfach $\Longleftarrow$ Wenn Entscheidungsproblem einfach…

Entscheidungsproblem mindestens so schwierig wie Berechnungsproblem

Reduktionen (II)

Übertrage auf NP-Entscheidungsprobleme:

NP-Problem L_A:
Gegeben: Problem `P`
Gesucht: Entscheidung

NP-Problem L_B:
Gegeben: Problem `Q`
Gesucht: Entscheidung

Reduktion von L_A auf L_B ist Polynomialzeit-Algorithmus R, so dass gilt: $P \in L_A \Leftrightarrow R(P) \in L_B$ für alle P. Schreibweise $L_A \leq L_B$.

Reduktion transformiert Problem P in Problem $Q = R(P)$, so dass korrekte Entscheidung für Q automatisch korrekte Entscheidung für P liefert

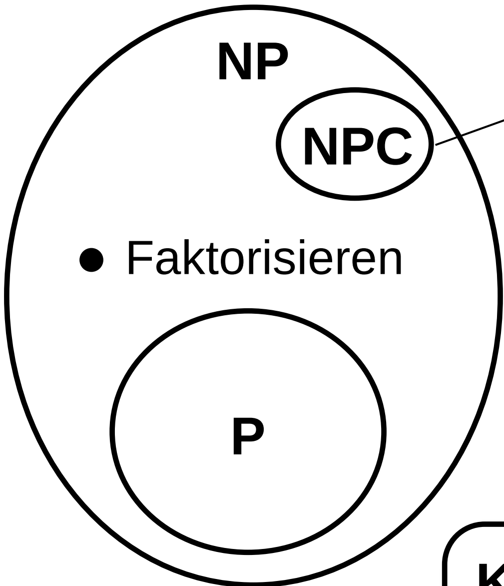
```
decideA(P)

Q=R(P);
return decideB(Q);
```

```
decideB(Q)

…
return d; //0/1
```

NP-vollständige Probleme

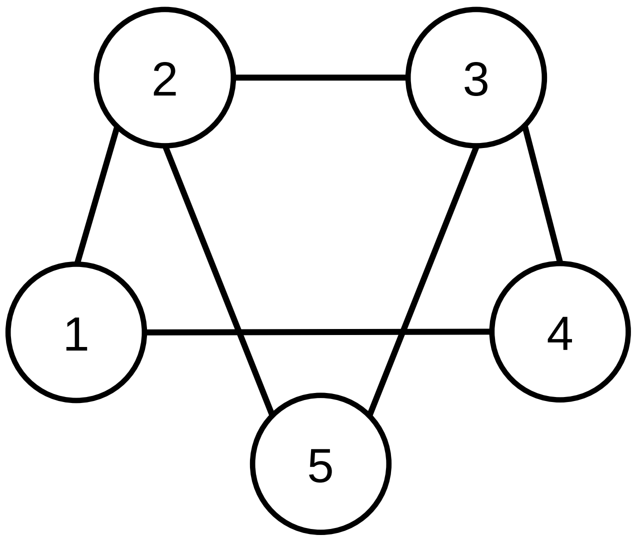


Alle Entscheidungsprobleme L_C aus **NP**, die mindestens so schwierig wie jedes andere Problem L_A aus **NP**: $L_A \leq L_C$ für alle $L_A \in$ **NP**.

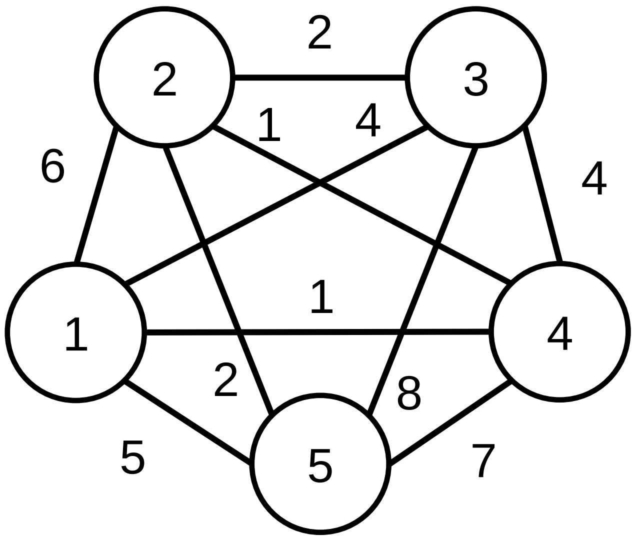
Komplexitätsklasse NPC (**NP**-vollständige Probleme) besteht aus allen Problemen $L_C \in$ **NP**, so dass $L_A \leq L_C$ für alle $L_A \in$ **NP**.

Zwei Bedingungen an L_C:
(1) Problem L_C ist selbst in **NP**
(2) jedes **NP**-Problem darauf reduzierbar („L_C ist **NP**-hart")

Reduktion: Hamiltonscher Zyklus ≤ TSP (I)



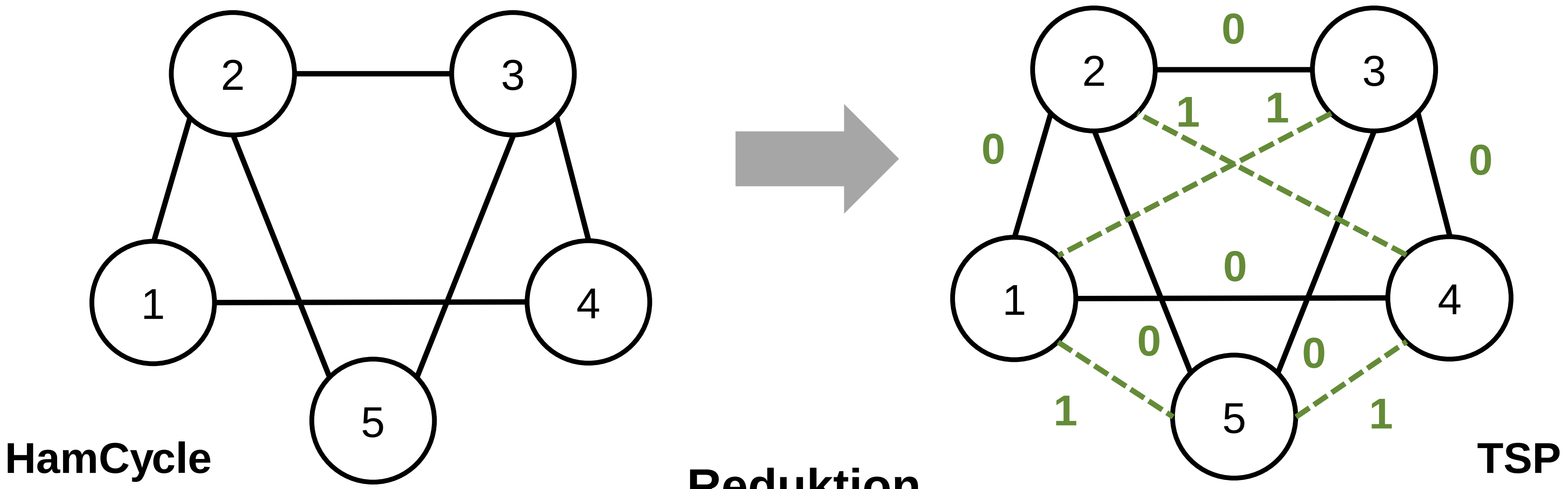
HamCycle für G
Gibt es Tour im Graphen G?
(jeden Knoten einmal besuchen und zu Startknoten zurück)



TSP für (G, B)
Gibt es Tour im Graphen G mit Gewicht maximal B?

(beide Probleme in **NP**)

Reduktion: Hamiltonscher Zyklus ≤ TSP (II)

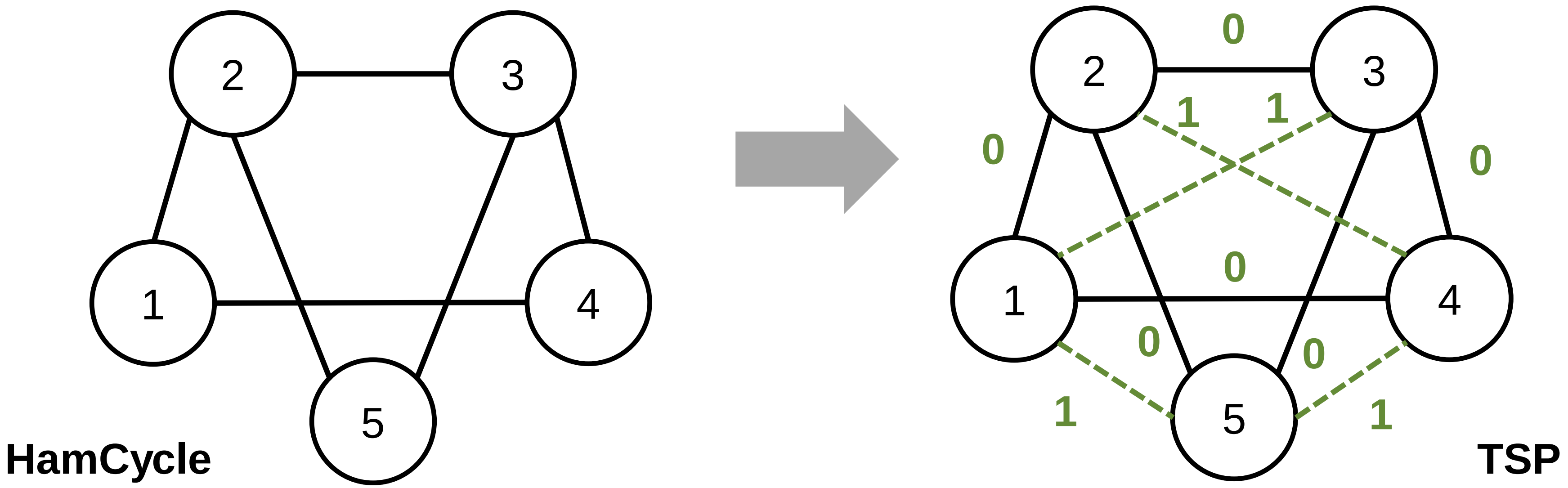


Reduktion

existierende Kanten bekommen Gewicht 0;
vervollständige Graphen mit Kanten und Gewichten 1;
setze $B = 0$

zu zeigen: $G \in$ **HamCycle** $\iff R(G) = (G^*, B) \in$ **TSP**

Reduktion: Hamiltonscher Zyklus ≤ TSP (III)



Wenn Hamiltonscher Zyklus in G $\Longrightarrow$ dann ist diese Tour entlang 0-Kanten in G^* und erfüllt Schranke $B = 0$

also ist diese Tour Hamiltonscher Zyklus in G $\Longleftarrow$ Wenn TSP-Tour für Schranke $B = 0$, dann nur entlang 0-Kanten in G^*,

SAT: Die Mutter aller NP-vollständigen Probleme

SAT

Gegeben: Boolesche Formel ϕ aus $\wedge, \vee, \neg$ in n Variablen $x_1, x_2, \ldots, x_n$
(ϕ polynomielle Komplexität in n)

Gesucht: Entscheide, ob ϕ erfüllende Belegung hat oder nicht

Beispiel: $\phi(x_1, x_2, x_3, x_4) = [\neg x_2 \vee (x_3 \wedge \neg x_4)] \wedge x_2 \wedge \neg[(x_1 \vee \neg x_2) \wedge x_4]$

hat erfüllende Belegung
$x_1 \leftarrow false,\ x_2 \leftarrow true,\ x_3 \leftarrow true,\ x_4 \leftarrow false$

Offensichtlich: **SAT∈NP** (gegeben Belegung als Zeuge, werte Formel aus)

Idee: SAT ist NP-hart (I)

Gegeben: beliebiges Problem $L_A \in$ **NP** mit poly-Algorithmus **`verifyA(P,S)`**

Eingabe **`P`**	Eingabe **`S`**	interner Speicher	Zustand (Zeile)	Ausgabe **`d`**
01010101000	1010011010100	0000000000000	000000001	0

```
verifyA(P,S) // check alleged solution

1  …
2  i=i+1
3  …
4  return d;
```

01010101000	1010011010100	0111001010000	000000100	**`d`**

Idee: SAT ist NP-hart (II)

Kodiere legitime Anfangszustand als Boolesche Formel: „Eingabe enthält `P` und Zeilennr.= `1`“

Kodiere legitime Endzustand als Boolesche Formel: „Wenn in Zeile `4`, dann ist Ausgabe `d=1`“

Eingabe `P`	Eingabe `S`	interner Speicher	Zustand (Zeile)	Ausgabe `d`
01010101000	1010011010100	0000000000000	000000001	0

```
verifyA(P,S) // check alleged solution

1  …
2  i=i+1
3  …
4  return d;
```

Kodiere legitime Rechenschritte als Boolesche Formel „Wenn in Zeile `2` und `i=0`, dann erlaubte Nachfolge Zeile `3` und `i=1`“ usw.

01010101000	1010011010100	0111001010000	000000100	d

Idee: SAT ist NP-hart (III)

Kodiere legitime Anfangszustand
als Boolesche Formel:
„Eingabe enthält **`P`** und Zeilennr.= **`1`**“

Kodiere legitime Endzustand als
Boolesche Formel:
„Wenn in Zeile **`4`**, dann ist Ausgabe **`d=1`**“

ϕ_P(alle Eingabebits)

=

gültiger Anfangszustand für **`P`** ∧ gültige Übergänge ∧ Endzustand mit **`d=1`**

Größe der Formel ϕ_P bleibt polynomiell, da Laufzeit von **`verifyA`** polynomiell

Kodiere legitime Rechenschritte
als Boolesche Formel
„Wenn in Zeile **`2`** und **`i=0`**, dann erlaubte
Nachfolge Zeile **`3`** und **`i=1`**“ usw.

Idee: SAT ist NP-hart (IV)

Reduktion $R(\mathtt{P})$ von L_A auf **SAT** berechnet:

ϕ_P(alle Eingabebits)
=
gültiger Anfangszustand für `P` ∧ gültige Übergänge ∧ Endzustand mit `d=1`

Wenn `P` in L_A, gibt es Lösung `S`, die `verifyA` akzeptiert mit `d=1`,
dann gibt es aber auch erfüllende Belegung für „Rechenschritte“ ϕ_P

Wenn `P` nicht in L_A, gibt es keine Lösung `S`, die `verifyA` akzeptiert,
dann gibt es aber auch keine erfüllende Belegung für „Rechenschritte“ ϕ_P

SAT ≤ 3SAT (I)

Boolesche Formeln in konjunktiver Normalform (KNF) mit jeweils 3 Literalen:

$$\phi(x_1, x_2, x_3, x_4) = (\neg x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_3 \vee x_4)$$

KNF = Und-Verknüpfung von Klauseln

Klausel besteht aus 3 Literalen $X_j \in \{x_j, \neg x_j\}$

Klausel = Oder-Verknüpfung

transformiere, falls weniger Literale in Klausel:
$(X_j) = (X_j \vee X_j \vee X_j)$,
$(X_j \vee X_k) = (X_j \vee X_k \vee X_k)$

3SAT

Gegeben: Boolesche **3KNF-**Formel ϕ in n Variablen $x_1, x_2, \dots, x_n$
(ϕ polynomielle Komplexität in n)

Gesucht: Entscheide, ob ϕ erfüllende Belegung hat oder nicht

SAT ≤ 3SAT (II)

Boolesche Formel σ aus $\wedge, \vee, \neg$ in n Variablen $y_1, y_2, \ldots, y_n$
(σ polynomielle Komplexität in n)

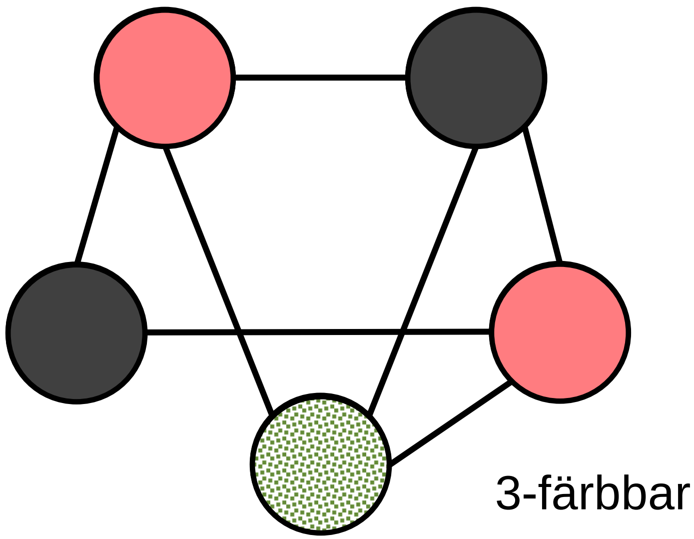
Polynomialzeit $\Downarrow$ (ohne Beweis)

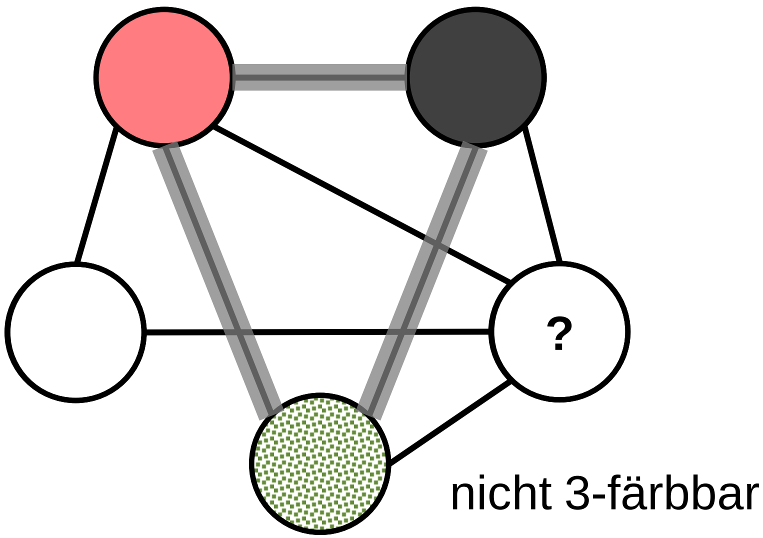
3KNF-Formel ϕ in $\mathrm{poly}(n)$ Variablen $x_1, x_2, \ldots, x_{poly(n)}$
(ϕ polynomielle Komplexität in n)

so dass σ genau dann erfüllbar ist, wenn ϕ erfüllbar ist

Folglich:
SAT ≤ 3SAT

3-Färbbarkeit von Graphen





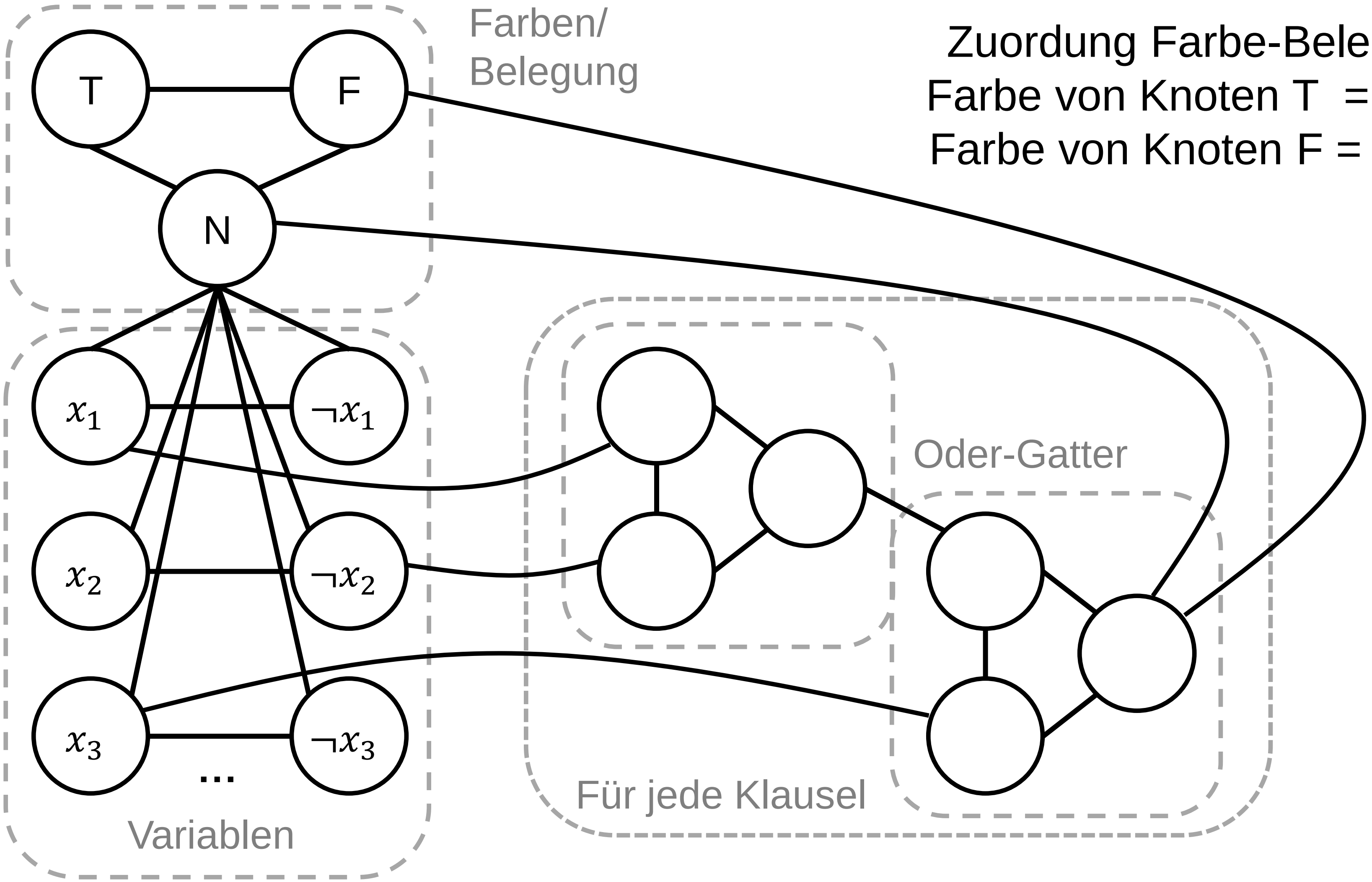
3COLORING für *G*
Gibt es Knotenfärbung
im Graphen *G*
mit 3 Farben, so dass
benachbarte Knoten
nie gleiche Farbe haben?

3COLORING $\in$ NP:
Gegeben Färbung,
durchlaufe Knoten und prüfe
jeweils Farbe der Nachbarknoten

3SAT ≤ 3COLORING (I)

$$\phi(x_1, \dots, x_n) = \cdots \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge \cdots$$

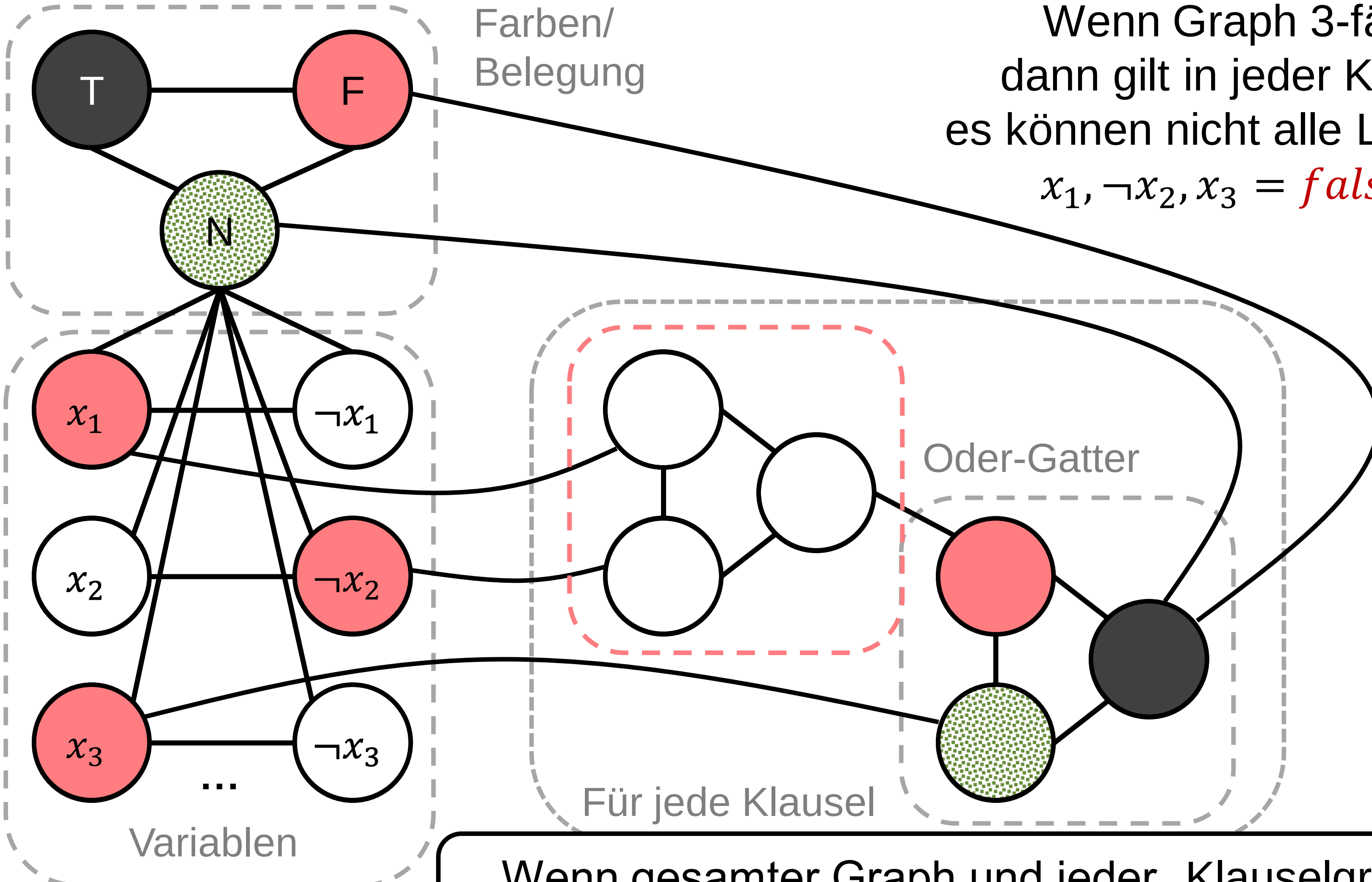
Zuordung Farbe-Belegung:
Farbe von Knoten T = $true$
Farbe von Knoten F = $false$



3SAT ≤ 3COLORING (II)

$\phi(x_1, \ldots, x_n) = \cdots \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge \cdots$

Wenn Graph 3-färbbar,
dann gilt in jeder Klausel:
es können nicht alle Literale
$x_1, \neg x_2, x_3 = false$ sein

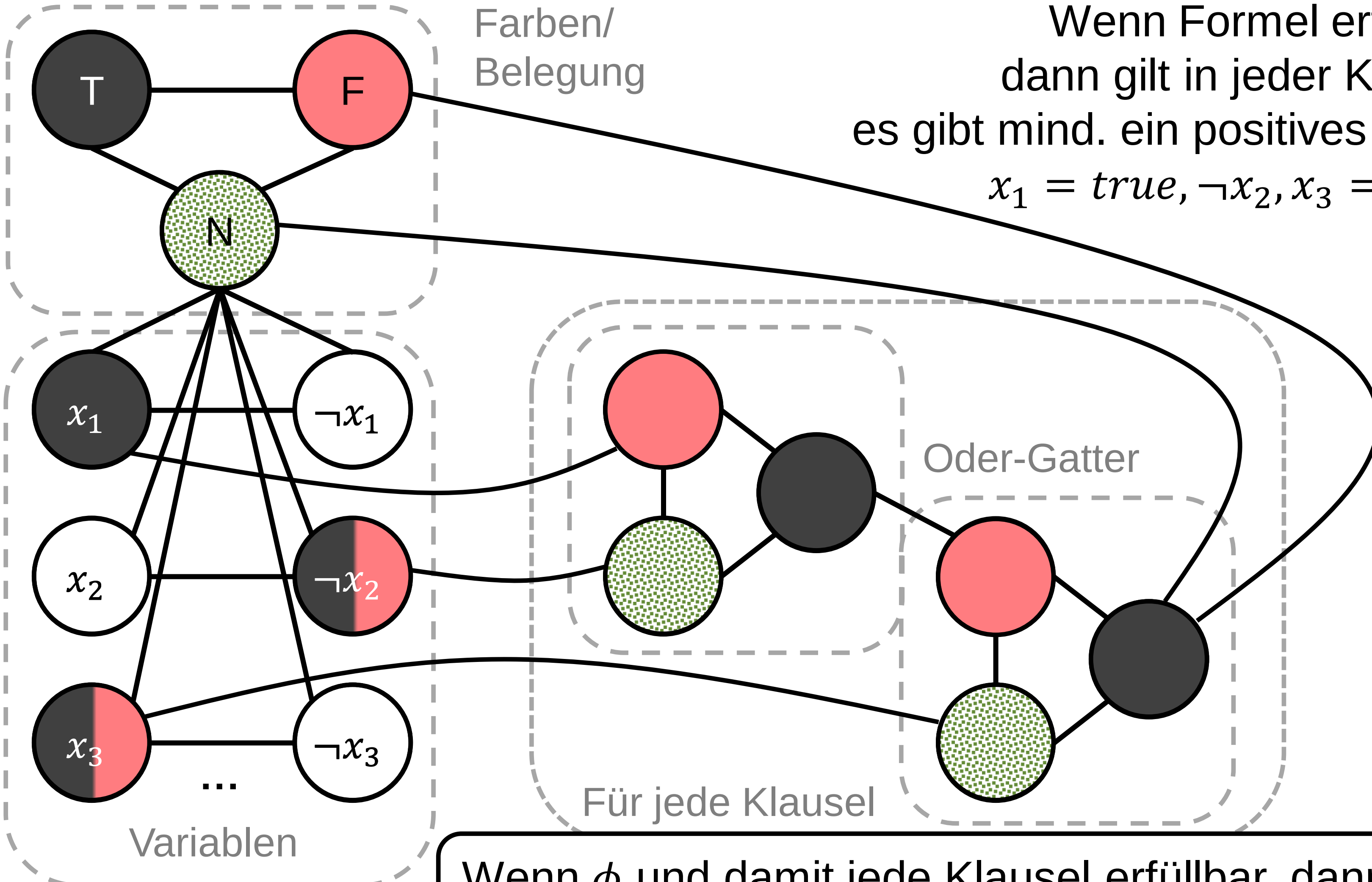


Wenn gesamter Graph und jeder „Klauselgraph“ 3-färbbar, dann auch jede Klausel und ϕ erfüllbar

3SAT ≤ 3COLORING (III)

$\phi(x_1, \dots, x_n) = \cdots \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge \cdots$

Wenn Formel erfüllbar, dann gilt in jeder Klausel: es gibt mind. ein positives Literal $x_1 = true, \neg x_2, x_3 = false$



Wenn ϕ und damit jede Klausel erfüllbar, dann auch jeder „Klauselgraph“ und gesamter Graph 3-färbbar

Einer für alle, alle für einen

Theorem: Wenn Problem L_B **NP**-vollständig und $L_B \leq L_C$ für $L_C \in$ NP gilt, dann ist auch L_C **NP**-vollständig

Beachte: $L_A \leq L_B$ und $L_B \leq L_C$

per Reduktion R_{AB} per Reduktion R_{BC}

per Reduktion $R_{AC} = R_{BC} \circ R_{AB}$

(Hintereinanderausführung)

Also folgt aus **3SAT** ≤ **3COLORING** und **3COLORING** ∈ **NP** auch, dass **3COLORING NP**-vollständig

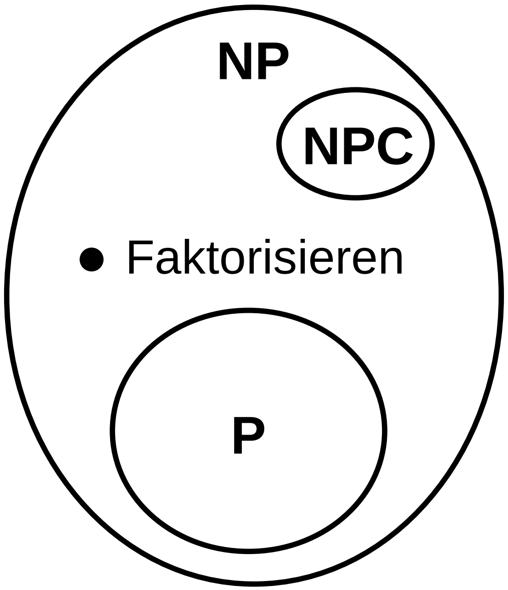
NPC – eine Auswahl

SAT - Formel ϕ erfüllbar?
3SAT - Formel ϕ in 3KNF erfüllbar?
3COLORING - Graph mit drei Farben kantenkonsistent färbbar?
HamCycle - Gibt es Tour im Graphen?
TSP - Gibt es Tour im Graphen, mit Gesamtgewicht $\leq B$?

VertexCover - Gibt es im Graphen Knotenmenge der Größe $\leq B$, so dass jede Kante an einem der Knoten hängt?
IndependentSet - Gibt es im Graphen Knotenmenge der Größe $\geq B$, so dass kein Knotenpaar durch Kante verbunden?
Knapsack - Für Gegenstände mit Wert und Volumen, gibt es Auswahl mit Gesamtwert $\geq W$, aber Gesamtvolumen $\leq V$
…

TECHNISCHE UNIVERSITÄT DARMSTADT
Cryptoplexity
Cryptography & Complexity Theory
Technische Universität Darmstadt
www.cryptoplexity.de

P vs. NP vs. NPC (I)



Theorem: Für jedes **NP**-vollständige Problem L_C gilt: $L_C \in$ **P** $\Leftrightarrow$ **P=NP**.

„⇒“ Betrachte beliebiges $L_A \in$ **NP**

Da L_C **NP**-vollständig, gibt es poly-Reduktion R mit $L_A \leq L_C$

Wegen $L_C \in$ **P** gibt es poly-Algorithmus für L_C

$poly(n)$

```
decideA(P)

Q=R(P);
return decideC(Q);
```

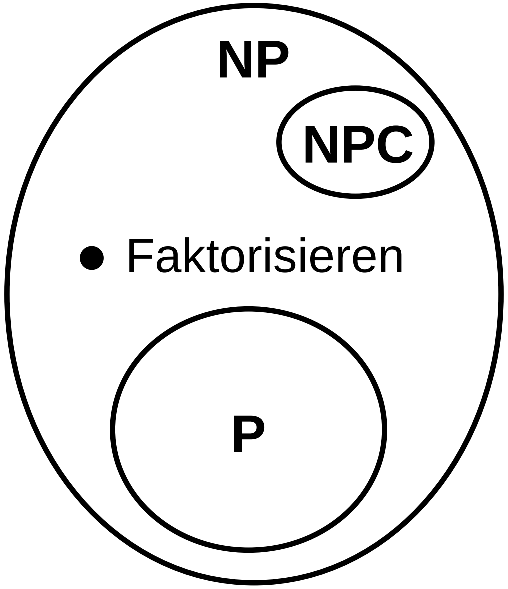
$poly(n)$

```
decideC(Q)

…
return d; //0/1
```

Dann aber auch $L_A \in$ **P**

P vs. NP vs. NPC (II)



Theorem: Für jedes **NP**-vollständige Problem L_C gilt: $L_C \in$ **P** $\Leftrightarrow$ **P**=**NP**.

„$\Leftarrow$“ Da L_C **NP**-vollständig, gilt $L_C \in$ **NP**

und wegen **P**=**NP** somit $L_C \in$ **P**

Wenn also Polynomialzeit-Algorithmus für **TSP** oder **3SAT** oder…
dann bereits Polynomialzeit-Algorithmen für alle Probleme in **NP**

Approximation? (I)

NPC-Probleme vermutlich nicht effizient lösbar,
aber evtl. leicht approximierbar:

```
3SAT-Approx(ϕ,n)

1  A[]=ALLOC(n); //assigment for variables
2  FOR i=1 TO n DO
3     A[i]=true resp. A[i]=false with probability 1/2
4  return A;
```

Behauptung: Algorithmus erfüllt im Erwartungswert mind. 1/2 aller Klauseln

K_i 0-1-Zufallsvariable, die angibt, ob i-te Klausel unter Belegung **A** erfüllt

$$E[K_i] = Prob[K_i = 1] = \begin{cases} 1/2 & wenn\ (X_i) \\ 3/4 & wenn\ (X_i \vee X_j), i \neq j \\ 7/8 & wenn\ (X_i \vee X_j \vee X_k), i \neq j, k\ und\ j \neq k \end{cases}$$

Approximation? (II)

NPC-Probleme vermutlich nicht effizient lösbar,
aber evtl. leicht approximierbar:

```
3SAT-Approx(ϕ,n)

A[]=ALLOC(n); //assigment for variables
FOR i=1 TO n DO
   A[i]=true resp. A[i]=false with probability 1/2
return A;
```

Behauptung: Algorithmus erfüllt im Erwartungswert mind. 1/2 aller Klauseln

K_i 0-1-Zufallsvariable, die angibt, ob i-te Klausel unter Belegung **A** erfüllt

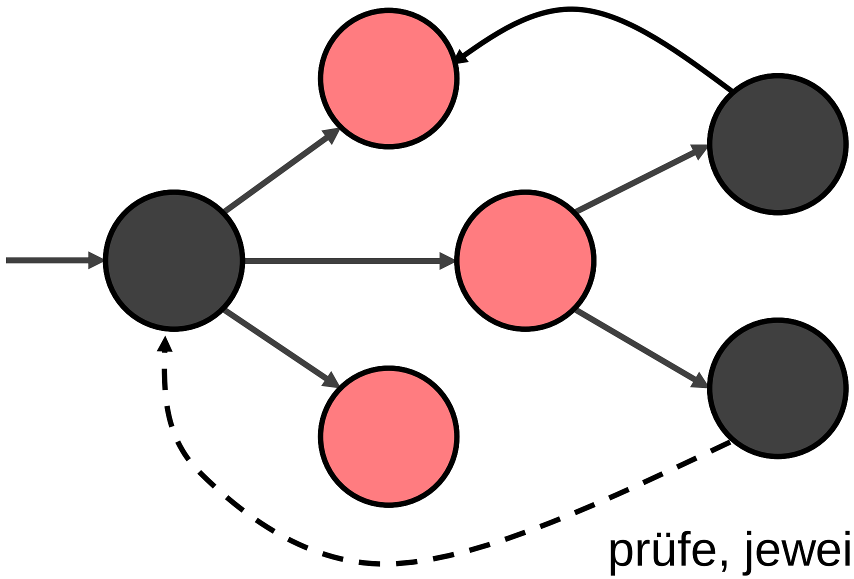
Bei m Klauseln folgt aus Linearität des Erwartungswertes:

$$E[\text{erfüllte Klauseln}] = E[\sum_{i=1}^{m} K_i] = \sum_{i=1}^{m} E[K_i] \geq m/2$$

2-Färbbarkeit und 2SAT in P

2-Färbbarkeit von Graphen ist (relativ) einfach (I)

Idee: Farbe eines Knoten bestimmt eindeutig Farben der Nachbarknoten



prüfe, jeweils ob Färbung
Widerspruch erzeugt

Ansatz:
Beginn mit einem Knoten und beliebiger Farbe
Durchlaufe Graph per BFS, färbe Knoten und identifiziere evtl. Widersprüche

2-Färbbarkeit von Graphen ist (relativ) einfach (II)

```
2ColoringSub(G,s,col) //G=(V,E), s node

s.color=col; newQueue(Q); enqueue(Q,s);
WHILE !isEmpty(Q) DO
   u=dequeue(Q);
   IF u.color==BLACK THEN nextcol=RED
   ELSE nextcol=BLACK;
   FOREACH v in adj(G,u) DO
      IF v.color==u.color THEN return 0;
      If v.color==WHITE THEN
         v.color=nextcol;
         enqueue(Q,v);
return 1; //no contradiction
```

wechsele Farbe

prüfe auf Widersprüche

nimm nur noch nicht gefärbte Knoten auf

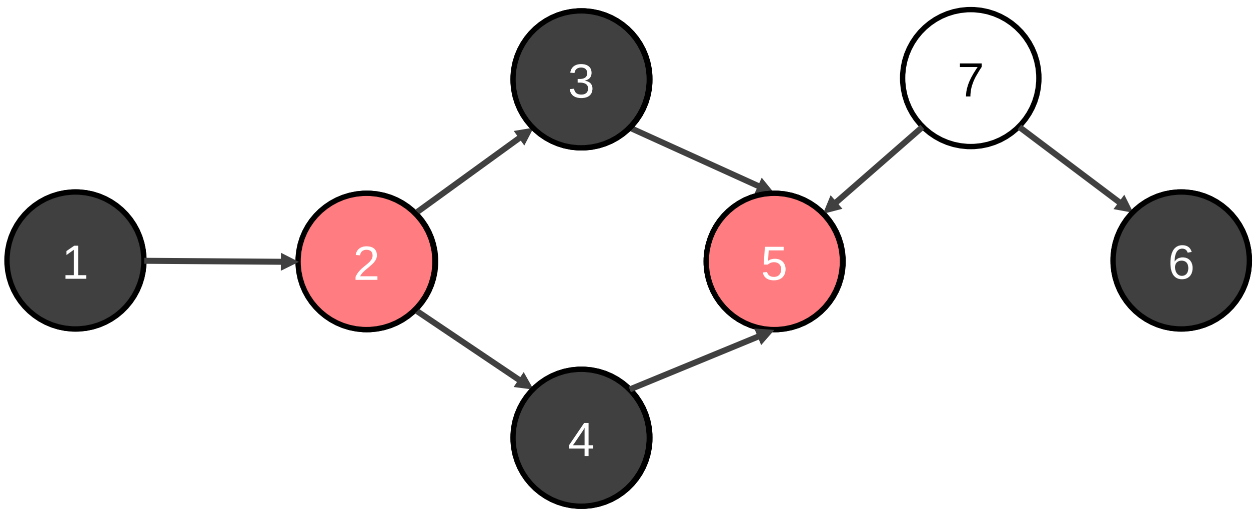
(zunächst nur für zusammenhängenden Graphen,
mit vorgegebenem Startknoten und vorgegebener Startfarbe)

2-Färbbarkeit von Graphen ist (relativ) einfach (III)

Achtung: Wir müssen evtl. mit anderen Startkonten nochmal starten

Wie Startfarbe jeweils wählen?

Knoten 7 wäre nicht mehr färbbar, obwohl Graph 2-färbbar ist

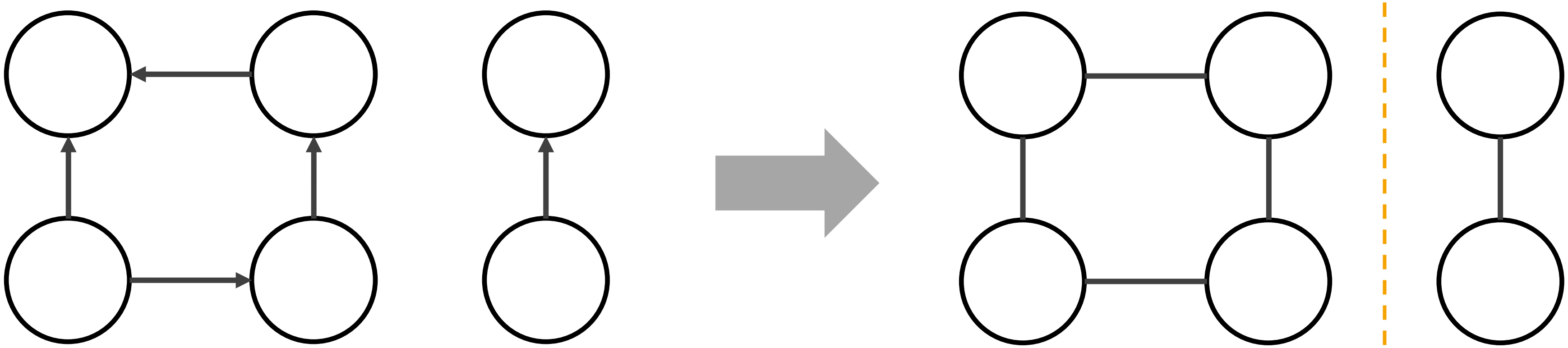


Starte mit 1 und Farbe schwarz

Starte erneut mit 6 und Farbe schwarz

Von gerichtet zu ungerichtet

Lösung: Betrachte ungerichtete Variante des Graphen



Ändert Lösungsmenge nicht, da verbundene Knoten in beiden Fällen unterschiedliche Farben haben müssen

Bei Neustart **keine** Kante zwischen Zusammenhangskomponenten:

Jede individuelle 2-Färbung der Zusammenhangskomponenten kann zu 2-Färbung des Graphen kombiniert werden

2-Färbbarkeit von Graphen

Laufzeit: $\Theta(|V| + |E|)$

```
2Coloring(G) // G=(V,E) undirected graph

FOREACH u in V Do u.color=WHITE;
FOREACH u in V DO
    IF u.color==WHITE THEN
        IF 2ColoringSub(G,u,BLACK)==0 THEN return 0;
return 1;
```

Algorithmus findet (ohne zusätzlichen Aufwand) auch Färbung

2-SAT auch so einfach?

O.b.d.A. stets zwei Literale pro Klausel,
sonst schreibe $X_1 = (X_1 \vee X_1)$

$$\phi(x_1, x_2, x_3, x_4)$$
$$=$$
$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

Wenn $x_1 \leftarrow false$ gesetzt wird,
dann x_2 eindeutig festgelegt für erfüllende Belegung
(hier: $x_2 \leftarrow false$)

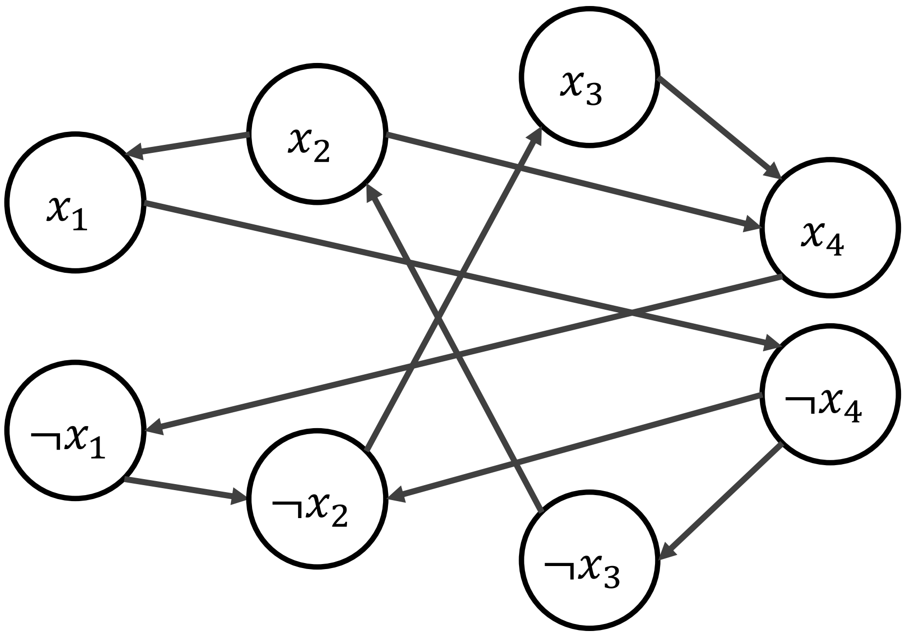
Aber: Wenn $x_1 \leftarrow true$ gesetzt wird, dann
erstmal noch zwei Möglichkeiten für x_2

keine „Symmetrie“ zwischen Belegungen wie bei Farben bei 2-Färbbarkeit

2-SAT → Implikationsgraph

Konstruiere aus Formel ϕ (gerichteten) Implikationsgraphen $G = (V, E)$:

1. Knotenmenge V besteht aus Literalen $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$,
2. Für jede Klausel $(X_j \vee X_k)$ nimm Kanten $(\neg X_j, X_k)$ und $(\neg X_k, X_j)$ auf



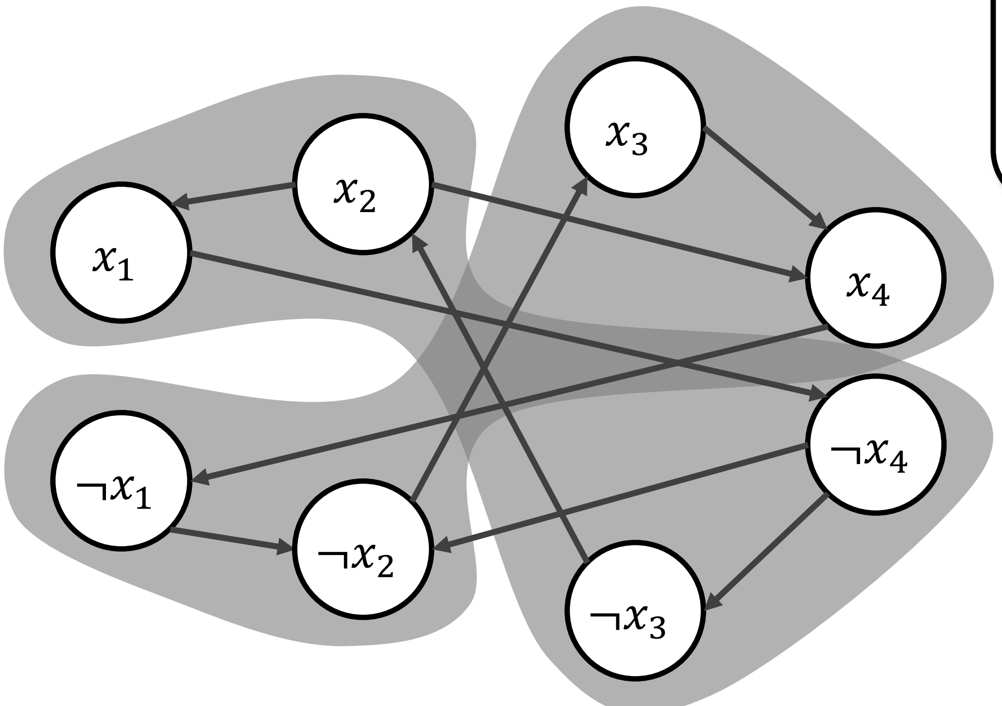
Intuition:
$X_j \vee X_k \ = \ \neg X_j \Rightarrow X_k$
$X_j \vee X_k \ = \ \neg X_k \Rightarrow X_j$

Wenn $X_j = false$
bzw. $\neg X_j = true$,
dann muss
$X_k = true$ sein

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

Graph → Starke Zusammenhangskomponenten

Suche starke Zusammenhangskomponenten im Implikationsgraph



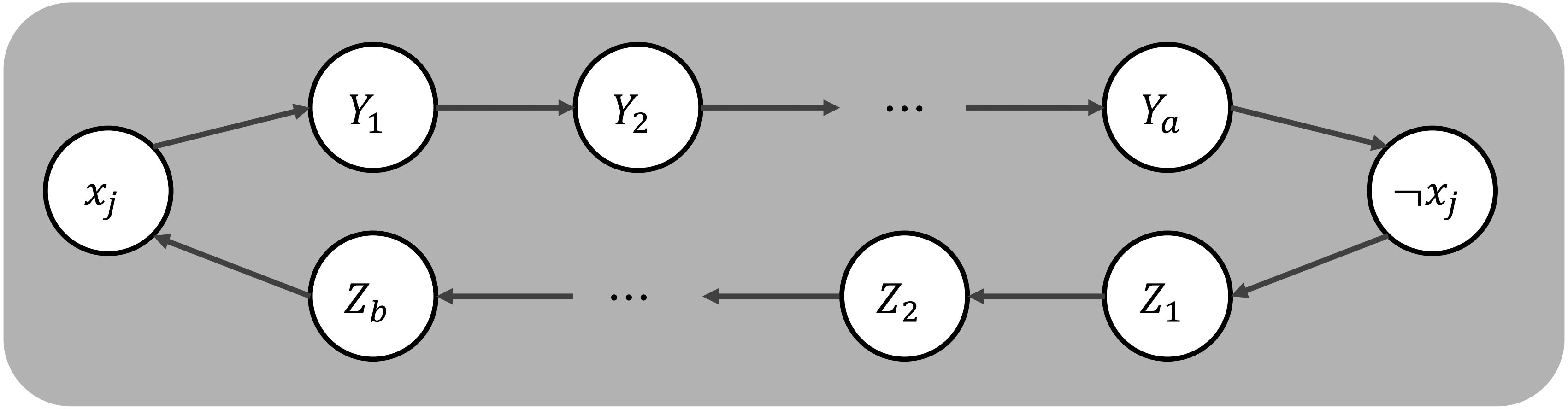
Formel ist genau dann erfüllbar, wenn in keiner Zusammenhangskomponenten x_j und $\neg x_j$ für ein j liegt

(Ist hier der Fall, also Formel erfüllbar)

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

Beide Literale in SCC ⇒ Nicht Erfüllbar (I)

Dann gibt es Kantenweg von x_j nach $\neg x_j$ und auch zurück in der SCC:



Kanten müssen durch entsprechende Klauseln in Formel entstanden sein:

$$\phi = \cdots \wedge \left(\neg x_j \vee Y_1\right) \wedge (\neg Y_1 \vee Y_2) \wedge \cdots \wedge (\neg Y_{a-1} \vee Y_a) \wedge (\neg Y_a \vee \neg x_j)$$
$$\wedge \left(\ x_j \vee \neg Z_b\right) \wedge (Z_b \vee \neg Z_{b-1}) \wedge \cdots \wedge (Z_2 \vee \neg Z_1) \wedge (Z_1 \vee\ \ x_j)$$

Beide Literale in SCC $\Rightarrow$ Nicht Erfüllbar (II)

Wenn $x_j = true$, dann **nicht erfüllbar**

müsste $true$ sein, um erfüllbar zu sein; ist aber $false$

muss $true$ sein, um erfüllbar zu sein

muss $true$ sein, um erfüllbar zu sein

muss $true$ sein, um erfüllbar zu sein

wird $false$

wird $false$

wird $false$

wird $false$

$$\phi = \cdots \wedge \left(\neg x_j \vee Y_1\right) \wedge (\neg Y_1 \vee Y_2) \wedge \cdots \wedge (\neg Y_{a-1} \vee Y_a) \wedge (\neg Y_a \vee \neg x_j)$$
$$\wedge \left(\ x_j \vee \neg Z_b\right) \wedge (Z_b \vee \neg Z_{b-1}) \wedge \cdots \wedge (Z_2 \vee \neg Z_1) \wedge (Z_1 \vee \ x_j)$$

Beide Literale in SCC $\Rightarrow$ Nicht Erfüllbar (III)

Wenn $x_j = false$, dann **nicht erfüllbar**

müsste $true$ sein, um erfüllbar zu sein; ist aber $false$

wird $false$

muss $true$ sein, um erfüllbar zu sein

wird $false$

muss $true$ sein, um erfüllbar zu sein

wird $false$

muss $true$ sein, um erfüllbar zu sein

wird $false$

$$\phi = \cdots \wedge (\neg x_j \vee Y_1) \wedge (\neg Y_1 \vee Y_2) \wedge \cdots \wedge (\neg Y_{a-1} \vee Y_a) \wedge (\neg Y_a \vee \neg x_j)$$

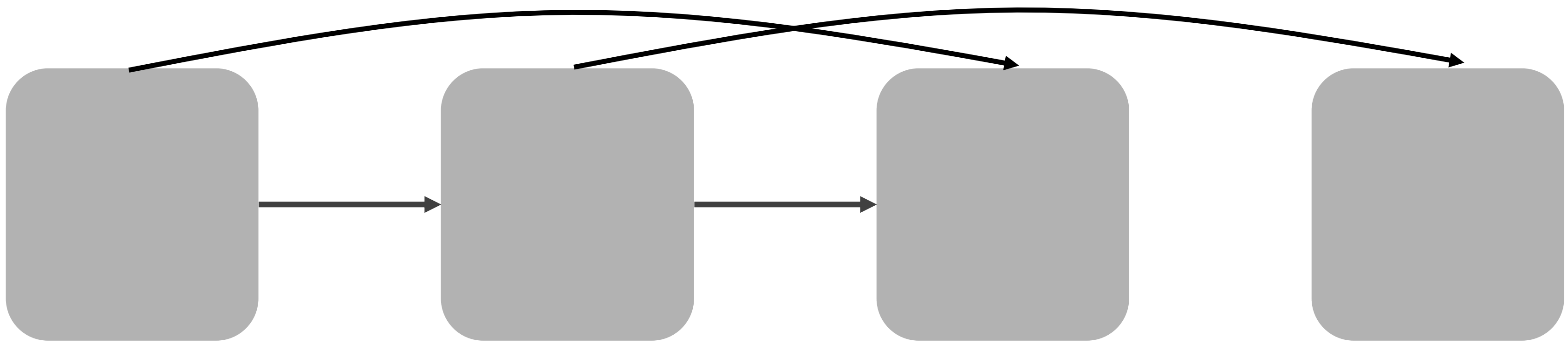
$$\wedge (\ x_j \vee \neg Z_b) \wedge (Z_b \vee \neg Z_{b-1}) \wedge \cdots \wedge (Z_2 \vee \neg Z_1) \wedge (Z_1 \vee \ x_j)$$

Erfüllende Belegung berechnen (I)

Annahme: kein x_j und $\neg x_j$ in gleicher SCC

Sortiere „SCC-dag“ topologisch!

„SCC-dag“: Graph mit Superknoten aus allen Knoten einer SCC; Kante zwischen SCCs, wenn Kante für zwei Knoten aus SCCs



Erfüllende Belegung: $x_j \leftarrow true$, wenn x_j in SCC nach SCC mit $\neg x_j$;
$x_j \leftarrow false$, wenn $\neg x_j$ in SCC nach SCC mit x_j

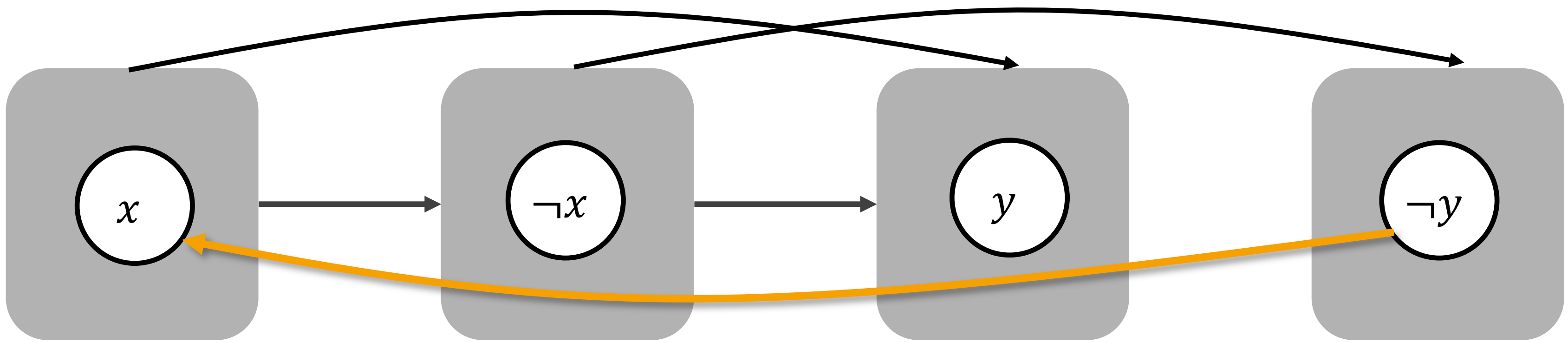
(wohldefiniert, da kein x_j und $\neg x_j$ in gleicher SCC)

Erfüllende Belegung berechnen (II)

Annahme: kein x_j und $\neg x_j$ in gleicher SCC

Betrachte Klausel $(x \vee y)$ – würde nicht erfüllt, wenn $x = y = false$

– führte zum Widerspruch!



„y nach $\neg x$" in topologischer Sortierung (evtl. in gleicher SCC)

Einerseits: Klausel erzeugt Kanten $(\neg x, y)$ und $(\neg y, x)$

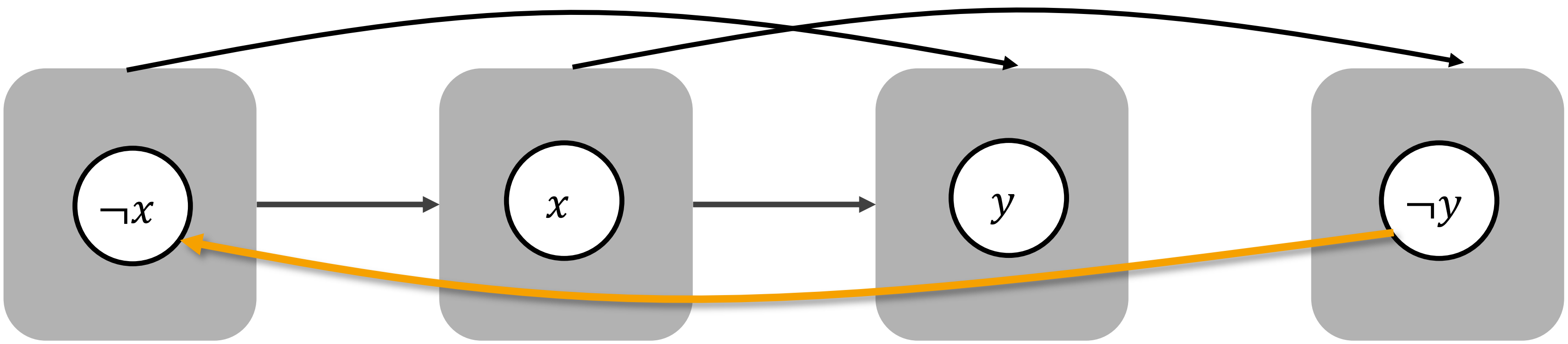
Rückkante zwischen SCCs

Andererseits: Belegung $x = false$ bedingt, dass „$\neg x$ (echt) nach x"
Belegung $y = false$ bedingt, dass „$\neg y$ (echt) nach y"

Erfüllende Belegung berechnen (III)

Annahme: kein x_j und $\neg x_j$ in gleicher SCC

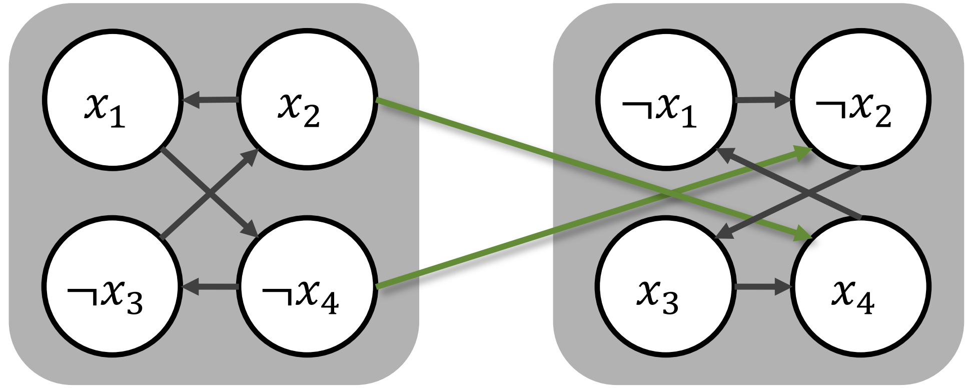
Andere Klausel-Kombinationen $(\neg x \vee y)$, $(x \vee \neg y)$, $(\neg x \vee \neg y)$ führen analog zum Widerspruch



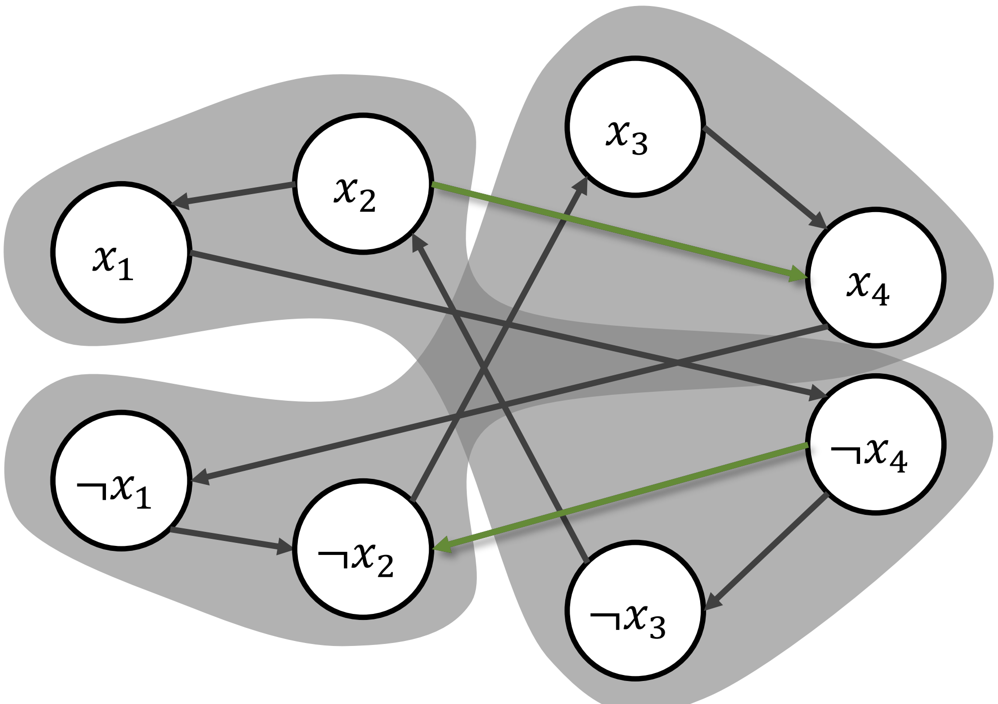
Beispiel $(\neg x \vee y)$

Also werden durch die Belegung alle Klauseln und somit Formel erfüllt

Zurück zum Beispiel



Topologisch Sortierung



Belegung:

$x_1 \leftarrow false$
$x_2 \leftarrow false$
$x_3 \leftarrow true$
$x_4 \leftarrow true$

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

Überlegen Sie sich, dass das Verfahren für 2-SAT mit den Zusammenhangskomponenten sogar gilt, wenn man „triviale“ Klauseln $x_j \vee \neg x_j$ in der Formel hat.

Beim gleichen Verfahren: Wann ist, wenn einer der Wege trivial ist, also ohne Y bzw. Z-Zwischenknoten auskommt? Wie sieht die Formel aus, geht das Verfahren damit durch?

Fügen Sie ein Klausel zur Beispielformel ϕ hinzu, so dass sie nicht mehr erfüllbar ist. Vergewissern Sie sich, dass Sie dann ein Paar $x_j, \neg x_j$ in der gleichen SCC haben.

Kleine Änderung, große Wirkung

MAX-2SAT-Problem:

Gegeben: 2SAT-Formel ϕ, Zahl k

Gesucht: Gibt es Belegung, die mind. k Klauseln erfüllt?

$\in$ **NPC** (ist **NP**-vollständig)!

Offensichtlich: **MAX-2SAT** $\in$ **NP**

Gegeben Belegung als Zeuge,
prüfe, ob mindestens k Klauseln erfüllt werden

Zeige zusätzlich: **3SAT** $\leq$ **MAX-2SAT**

3SAT ≤ MAX-2SAT (I)

$$\sigma(x_1, \dots, x_n) = \cdots \land (X_i \lor X_j \lor X_k) \land \dots$$ (m Klauseln)

eine neue Variable
und 10 Klauseln in ϕ
pro Klausel in σ

$$\phi(x_1, \dots, x_n, w_1, \dots, w_m)$$
$$= \cdots$$
$$\land (X_i) \land (X_j) \land (X_k) \land (w_h)$$
$$\land \left(\neg X_i \lor \neg X_j\right) \land (\neg X_i \lor \neg X_k) \land \left(\neg X_j \lor \neg X_k\right)$$
$$\land (X_i \lor \neg w_h) \land (X_j \lor \neg w_h) \land (X_k \lor \neg w_h) \land \dots$$

3SAT ≤ MAX-2SAT (II)

$$\sigma(x_1, \dots, x_n) = \cdots \wedge (X_i \vee X_j \vee X_k) \wedge \dots$$ (m Klauseln)

eine neue Variable und 10 Klauseln in ϕ pro Klausel in σ

$$\phi(x_1, \dots, x_n, w_1, \dots, w_m)$$
$$= \cdots$$
$$\wedge (X_i) \wedge (X_j) \wedge (X_k) \wedge (w_h)$$
$$\wedge (\neg X_i \vee \neg X_j) \wedge (\neg X_i \vee \neg X_k) \wedge (\neg X_j \vee \neg X_k)$$
$$\wedge (X_i \vee \neg w_h) \wedge (X_j \vee \neg w_h) \wedge (X_k \vee \neg w_h) \wedge \dots$$

Wenn Klausel in σ nicht erfüllbar ($X_i = X_j = X_k = false$),
dann maximal 6 der 10 Klauseln in ϕ erfüllbar (für $w_h = false$)

3SAT ≤ MAX-2SAT (III)

$$\sigma(x_1, \dots, x_n) = \cdots \wedge (X_i \vee X_j \vee X_k) \wedge \dots$$ (m Klauseln)

eine neue Variable
und 10 Klauseln in ϕ
pro Klausel in σ

$$\phi(x_1, \dots, x_n, w_1, \dots, w_m)$$
$$= \cdots$$
$$\wedge (X_i) \wedge (X_j) \wedge (X_k) \wedge (w_h)$$
$$\wedge (\neg X_i \vee \neg X_j) \wedge (\neg X_i \vee \neg X_k) \wedge (\neg X_j \vee \neg X_k)$$
$$\wedge (X_i \vee \neg w_h) \wedge (X_j \vee \neg w_h) \wedge (X_k \vee \neg w_h) \wedge \dots$$

Wenn Klausel in σ erfüllbar, dann nie mehr als 7 der 10 Klauseln in ϕ erfüllbar,
und für geeignetes w_h auch wirklich 7 der 10 Klauseln in ϕ erfüllbar
($w_h = false$ falls nur ein Literal X_i, X_j, X_k $true$, sonst $w_h = true$)

3SAT ≤ MAX-2SAT (IV)

$$\sigma(x_1, \dots, x_n) = \cdots \wedge (X_i \vee X_j \vee X_k) \wedge \dots$$ (m Klauseln)

eine neue Variable
und 10 Klauseln in ϕ
pro Klausel in σ

$$\phi(x_1, \dots, x_n, w_1, \dots, w_m)$$
$$= \cdots$$
$$\wedge (X_i) \wedge (X_j) \wedge (X_k) \wedge (w_h)$$
$$\wedge \left(\neg X_i \vee \neg X_j\right) \wedge (\neg X_i \vee \neg X_k) \wedge \left(\neg X_j \vee \neg X_k\right)$$
$$\wedge (X_i \vee \neg w_h) \wedge (X_j \vee \neg w_h) \wedge (X_k \vee \neg w_h) \wedge \dots$$

Sind mindestens $k = 7m$ Klauseln erfüllbar?

Wenn σ erfüllbar, dann mindestens $k = 7m$ Klauseln in ϕ erfüllbar;
Wenn σ nicht erfüllbar, dann weniger als $k = 7m$ Klauseln in ϕ erfüllbar

ENDE