Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Tanya Braun (Übungen)

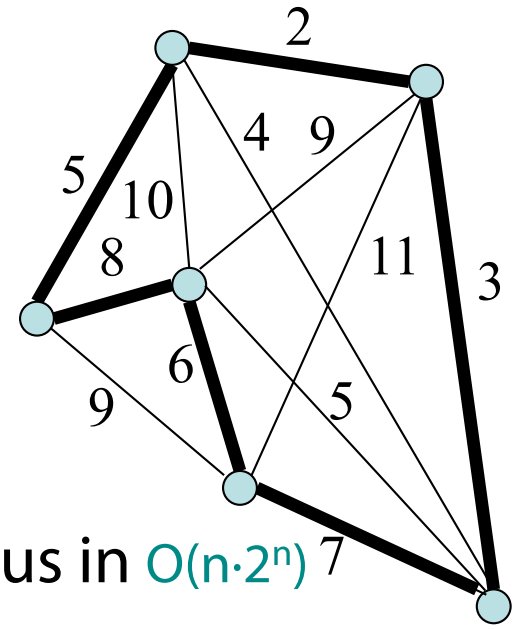
sowie viele Tutoren

Bisher betrachtet...

- Algorithmen für verschiedene Probleme
  - Sortierung, LCS, Ereignisplanung, Minimaler Spannbaum, Kürzester Weg, usw.
  - Laufzeiten $O(n^2)$, $O(n \log n)$, $O(n)$, $O(nm)$, etc.,
    also polynomiell (polynomial) zur Größe der Eingabe
  - Manchmal $O(n+m)$ also sogar noch linear zur Größe der Eingabe
  - Die genannten Probleme heißen traktabel
- Können wir alle (oder die meisten) Probleme in polynomieller Zeit lösen?
- Nein. Es gibt intraktable oder schwierige Probleme
- Es gibt „schwierige Probleme“, die sich bisher nicht in Polynomialzeit lösen lassen.
- Gibt es „schwierige Probleme“, die sich sicher nicht in Polynomialzeit lösen lassen? Ja.

TSP: Beispiel für ein schwieriges Problem

- Problem des Handlungsreisenden (Traveling Salesman Problem, TSP)
  - Eingabe: Ungerichteter Graph mit Längen als Beschriftungen für Kanten
  - Ausgabe: Kürzeste Tour, auf der alle Knoten genau einmal vorkommen



- Entscheidungsproblem (k-TSP): Gibt es Tour mit Kantenkosten ≤k?
- Bester bekannter vollständiger Algorithmus in $O(n \cdot 2^n)$
- Das Problem muss z.Zt. als intraktabel klassifiziert werden
- Interessant: Geg. Lösung (Tour ≤k) polynomiell verifzierbar!
- Also: Raten und polynomiell verifizieren
- Von k-TSP zu TSP?

Kantengewichte = 1: Hamilton-Kreis

Probleme und deren Lösung

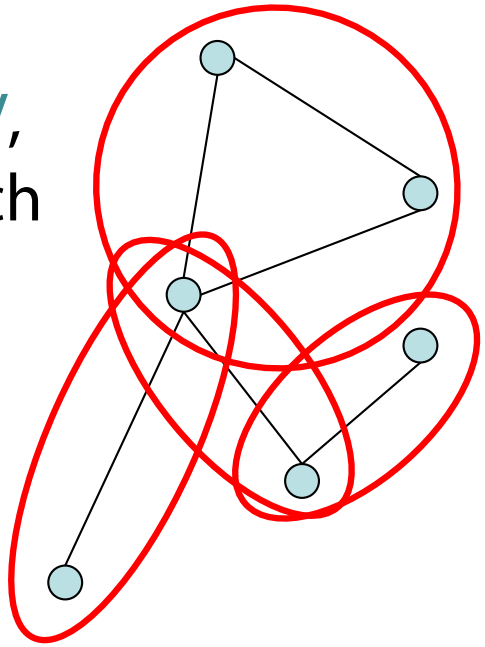
- Ein Problem ist eine Menge von Tupeln jeweils mit der Struktur (*Eingabe*, *Ausgabe*)
  - Beispiele:
    - Add = { ((1, 1), 2), ... ((11, 2), 13) ... }
    - 42-TSP = { (G_1, ja), (G_2, nein), ... (G_{100011}, ja), ... }
  - Wir können also Probleme (oder deren Spezifikation) in Mengen zu Problemklassen zusammenfassen (Menge von Menge von Tupeln)
  - Ein Problem kann auf verschiedene Weise (endlich) beschrieben/spezifiziert sein
- Lösung zu einem Problem: Funktion
- Problem zu lösen: endlich beschriebene Funktion berechnen
- Berechnen der Funktion übernimmt ein Algorithmus

Nichtdeterminismus? Wozu dient das?

- Betrachtet: Entscheidungsprobleme
  - Antwort: Ja oder Nein
- Sei **P** die Menge der Probleme,
  die in polynomieller Zeit berechenbar sind
- **NP** (nichtdeterministisch polynomielle Zeit) ist die Menge von Problemen, die durch einen nichtdeterministischen Computer gelöst werden kann
  - Vorstellung: Wenn eine Lösung in einem Suchraum existiert, kann sie geraten werden
  - Validierung einer vorgeschlagenen Lösung polynomiell
    (NP= Menge der Probleme, bei denen Lösungsvorschläge in polynomieller Zeit verifiziert werden können)

Ein weiteres schwieriges Problem

- (Maximales-)Cliquen-Problem (Clique)
  - Eingabe: Ungerichteter Graph G=(V,E)
  - Ausgabe: Größte Untermengen C von V, so dass jedes Paar von Knoten in C durch eine Kante aus E verbunden ist
  - Eine solche Menge heißt Clique
  - Bester bekannter Algorithmus für dieses Optimierungsproblem ist in $O(n \cdot 2^n)$



- Entscheidungsproblem (k-Clique): Gibt es Clique $C \subseteq V$ mit $|C| \leq k$?

P und NP

- **P** = Probleme, die in polynomieller Zeit gelöst werden
- **NP** = Probleme für die Lösungen in polynomieller Zeit verifiziert werden können

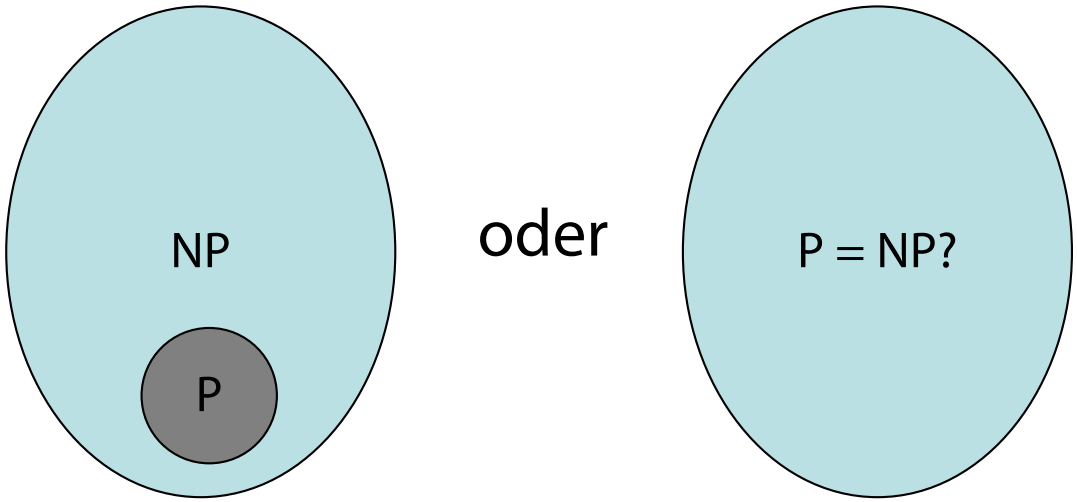
- K-Clique-Problem ist in NP (und damit Clique):
- K-TSP-Problem ist in NP (und damit TSP)
- Ist Sortierung in NP?
  - Kein Entscheidungsproblem
- Also vielleicht: Sortiertheit in NP?
  - Ja, leicht zu verifizieren, man braucht gar nicht zu raten
  - Sortiertheit ist sogar in P
- **P $\subseteq$ NP**

Schwierige bzw. intraktable Probleme

- Besseren Algorithmus suchen?
  - Haben viele schlaue Leute schon 50 Jahre lang versucht
- Beweisen, dass es keinen polynomiellen Algorithmus geben kann
  - Haben viele schlaue Leute schon 50 Jahre lang versucht
- Zeigen, dass alle schwierigen Probleme in gewisser Weise äquivalent sind, d.h., kann man eins in $O(n^k)$ (k fix) lösen, kann man alle anderen auch in $O(n^k)$ lösen
  - Hat schon jemand erreicht (Karp 1972)
  - Funktioniert für mindestens 10.000 schwierige Probleme

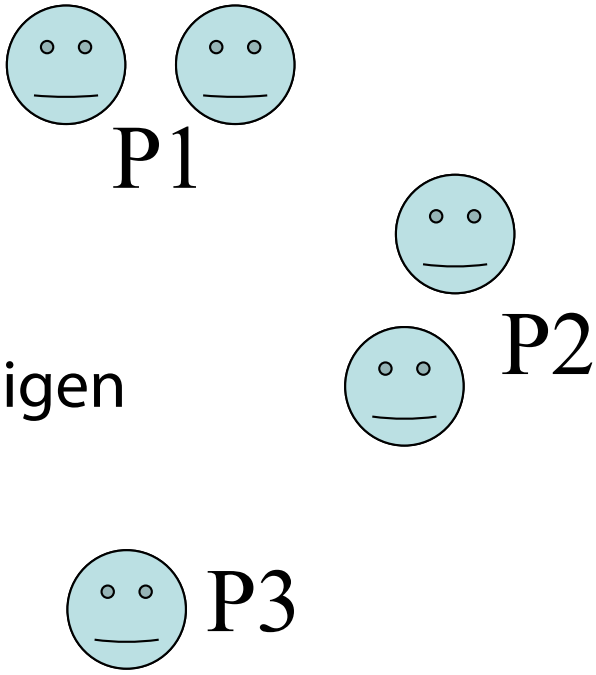
P und NP

- Ist **P = NP?**
  - Eines der großen offenen Probleme in der Informatik
  - Es wird angenommen, dass das nicht gilt
  - Das Clay Mathematics Institute [claymath.org] bietet $1 Million für den ersten Beweis



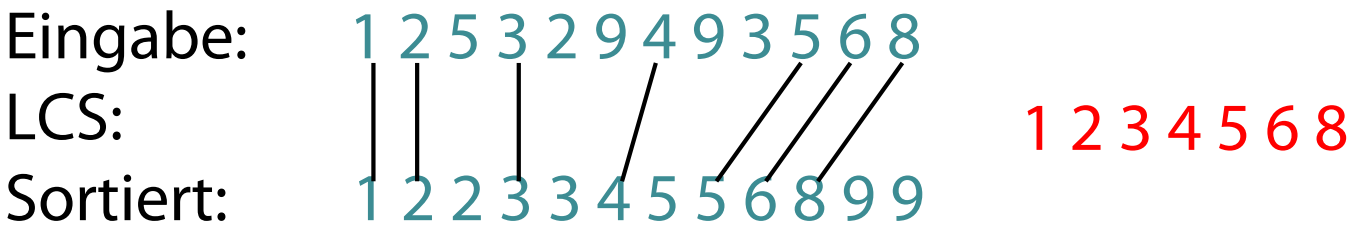
Vorteile dieser Äquivalenz

- Ausnutzung von Forschungsergebnissen
- Falls für ein Problem eine polynomielle Lösung entwickelt wird, dann ist sie für alle einsetzbar
- Realistischer: Sobald eine exponentielle untere Schranke gezeigt wird, gilt sie für alle äquivalenten Probleme
- Wenn wir für Problem ∏ Äquivalenz zu 10.000 schwierigen Problemen zeigen können, sind das starke Indizien, dass ∏ schwierig ist
- Problemklassen und Äquivalenzbegriff definieren



Longest-Increasing-Subsequence-Problem

- Gegeben sei eine Liste von Zahlen
  1 2 5 3 2 9 4 9 3 5 6 8
- Finde ***längste*** Teilsequenz, in der keine Zahl kleiner ist als die vorige
  - Beispiel: 1 2 5 9
  - Teilsequenz der originalen Liste
  - Die Lösung ist Teilsequenz der sortierten Liste



Reduktion von Problemen aufeinander

- Beispiel: Reduktion von LIS auf LCS in $O(n \log n)$
- Reduktion von Π' auf Π in $O(n^k)$, wobei k fix ist
  - Beispiel: Löse Quadrieren durch Multiplizieren
- Π kann nicht einfacher sein als Π'
  - Multiplizieren kann nicht einfacher sein als Quadrieren
- Übertragbar auch für intraktable Probleme (Probleme in NP)

NP-Schwere

- Wenn gezeigt werden kann, dass alle Probleme Π' in NP auf ein Problem Π reduziert werden kann, heißt Π NP-schwer (Π kann natürlich noch schwerer sein)

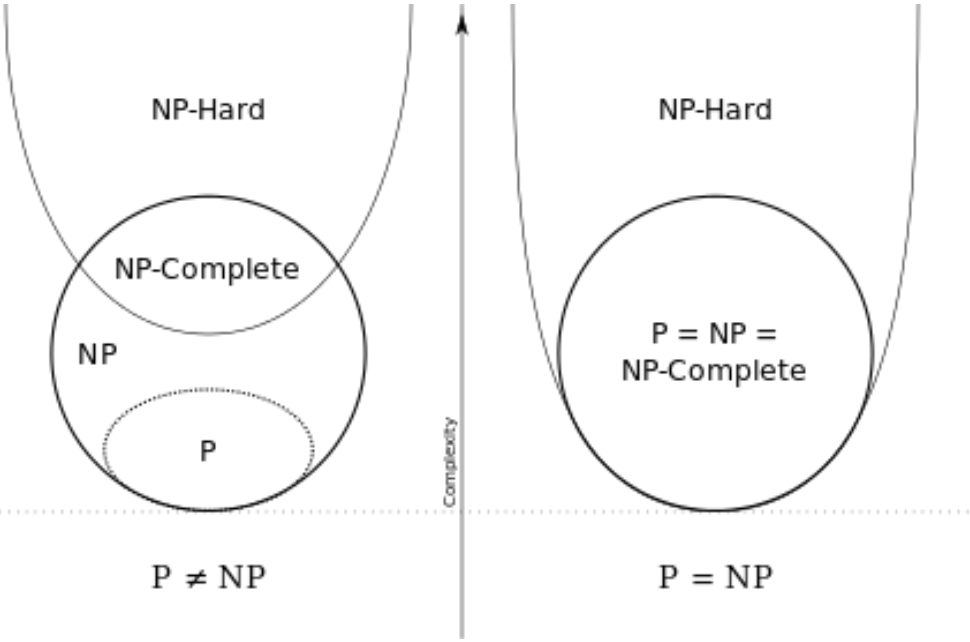
- Die Probleme Π sind interessant

- Wenn Π in NP, dann ist Π gewissermaßen eines der schwersten Probleme in NP

Gibt es schwerste NP-Probleme in NP?

- Interessanterweise ja!
- Die Probleme heißen NP-vollständige Probleme
- Wenn eines der schwersten Probleme in NP in Polynomialzeit gelöst werden kann, dann jedes in NP (P = NP)
- Beispiel: Wenn TSP NP-vollständig ist, löse TSP in $O(n^{100})$ und du hast gezeigt, dass P=NP

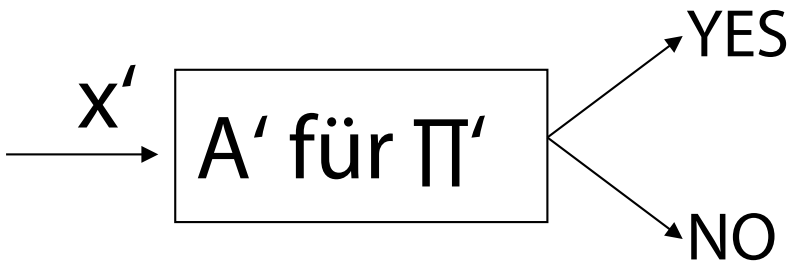
➜ Setz dich reich und berühmt zur Ruhe



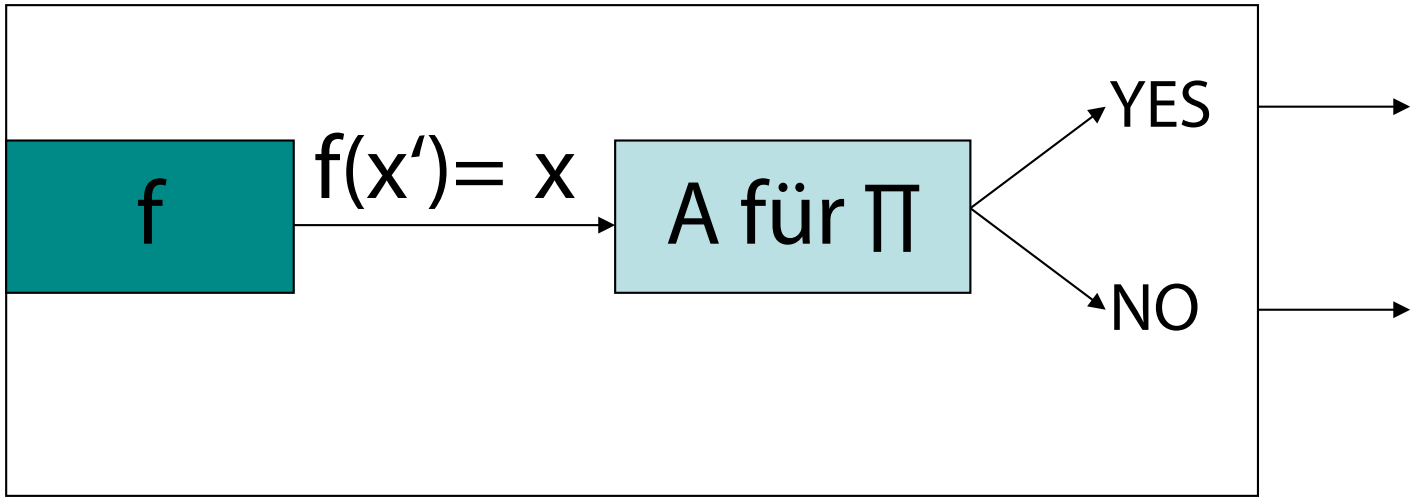
Reduktion von Problemen

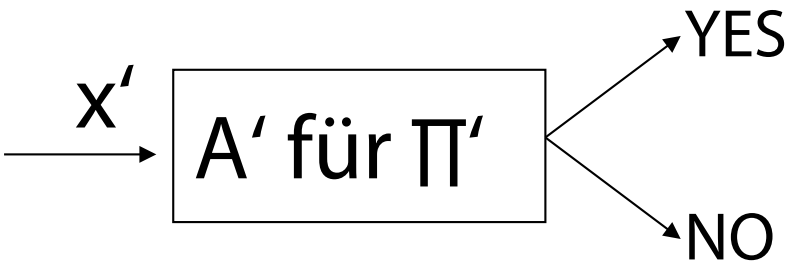
- Ein Problem Π‘ kann auf ein Problem Π reduziert werden, ...
- wenn jede Instanz $I_{\Pi'}$ von Π‘ „einfach“ auf eine Instanz von Π zurückgeführt werden kann, ...
- in dem Sinne, dass eine Lösung für I_{Π} die Lösung für $I_{\Pi'}$ bestimmt

Reduktionen: Π’ nach Π



Reduktionen: Π’ to Π





Reduktion: Ein Beispiel

- Π': Gegeben eine Sequenz von booleschen Werten x_i, ist einer davon wahr (also TRUE)?
- Π: Gegeben eine Menge von Ganzzahlen (Integer) y_i, ist ihre Summe ≥ 0?
- Transformation: $(x_1, x_2, \ldots, x_n) \rightarrow (y_1, y_2, \ldots, y_n)$, wobei $y_i = 1$ falls $x_i = \text{TRUE}$, $y_i = 0$ falls $x_i = \text{FALSE}$

Transformationsfunktion f in O(n)

Reduktion: Beispiel 2

- Gegeben: Aussagenlogische Formel wie z.B.

$$\underbrace{(q \vee \neg r \vee s)}_{\text{Klausel}} \wedge \underbrace{(\neg q)}_{\text{Klausel}} \wedge \underbrace{(\neg r \vee \neg s)}_{\text{Klausel}}$$

- Transformation auf lineares Gleichungssystem mit Variablen aus dem Bereich Integer
  - Integer-Variablen x_p für alle booleschen Variablen p
  - Gleichungsystem:

    $0 \leq x_p \leq 1$ für alle booleschen Variablen p

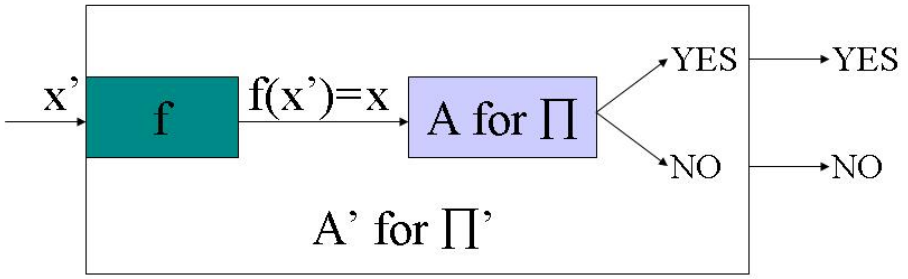
    $x_q + (1 - x_r) + x_s \geq 1$

    $(1 - x_q) \geq 1$

    $(1 - x_r) + (1 - x_s) \geq 1$

Transformations-funktion f in O(n)

Reduktionen



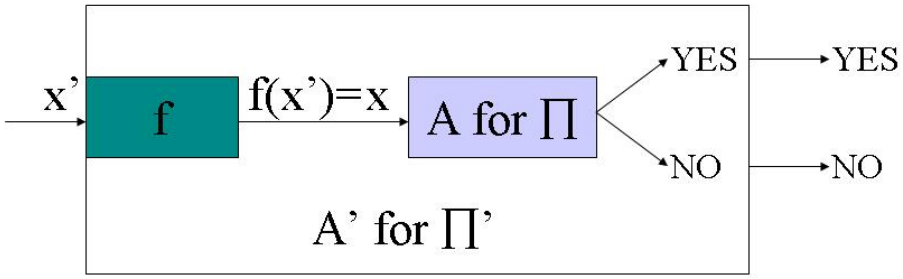
- Π‘ ist Polynomialzeit-reduzierbar auf Π ($\Pi' \leq_p \Pi$) genau dann, wenn es eine polynomielle Funktion f zur Abbildung von Eingaben x‘ für Π‘ in Eingaben x für Π, so dass für jedes x‘ gilt, dass

$$\Pi'(x') = \Pi(f(x'))$$

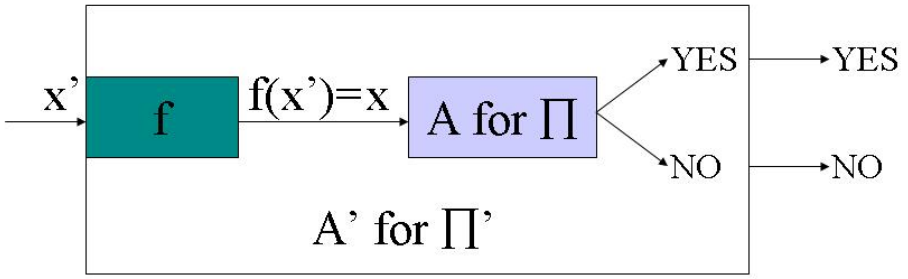
Reduktionen

- Π' ist Polynomialzeit-reduzierbar auf Π ($\Pi' \leq_p \Pi$) genau dann, wenn es eine polynomielle Funktion f zur Abbildung von Eingaben x‘ für Π' in Eingaben x für Π, so dass für jedes x‘ gilt, dass

$$\Pi' (x') = \Pi(f(x'))$$



Reduktionen



- Π‘ ist Polynomialzeit-reduzierbar auf Π ($\Pi' \leq_p \Pi$) genau dann, wenn es eine polynomielle Funktion f zur Abbildung von Eingaben x‘ für Π‘ in Eingaben x für Π, so dass für jedes x‘ gilt, dass

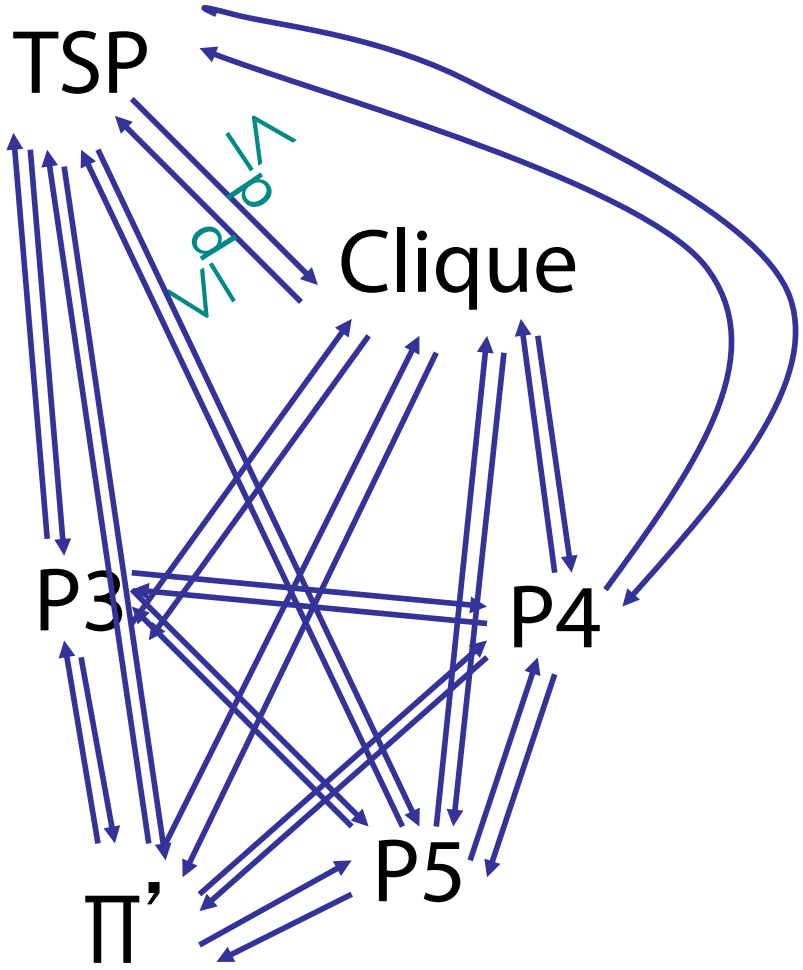
$$\Pi'(x') = \Pi(f(x'))$$

- Falls $\Pi \in P$ und $\Pi' \leq_p \Pi$ dann $\Pi' \in P$
- Falls $\Pi \in NP$ und $\Pi' \leq_p \Pi$ dann $\Pi' \in NP$
- Falls $\Pi'' \leq_p \Pi'$ and $\Pi' \leq_p \Pi$ then $\Pi'' \leq_p \Pi$ (Transitivität)

Äquivalenz zwischen schwierigen Problemen

Optionen

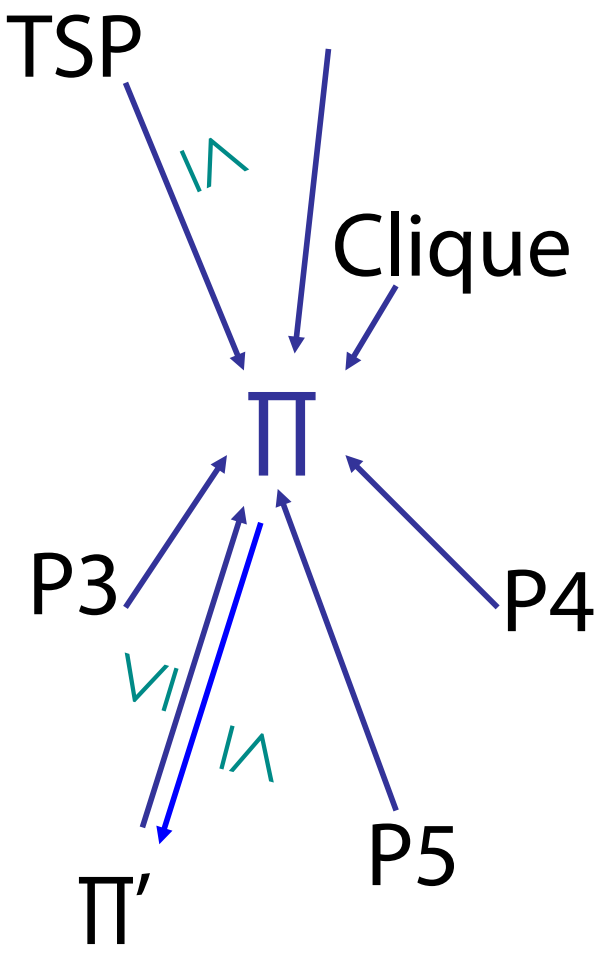
- Zeige Reduktionen zwischen allen Paaren von Problemen
- Reduziere die Anzahl von Reduktionen durch Verwendung der Transitivität von $\leq_p$



Äquivalenz zwischen schwierigen Problemen

Optionen

- Zeige Reduktionen zwischen allen Paaren von Problemen
- Reduziere die Anzahl von Reduktionen durch Verwendung der Transitivität von $\leq_p$
- Zeige Reduzierbarkeit aller Probleme in NP auf festes Π in NP
- Um zu zeigen, dass ein Problem Π' in NP ist, zeigen wir $\Pi \leq_p \Pi'$

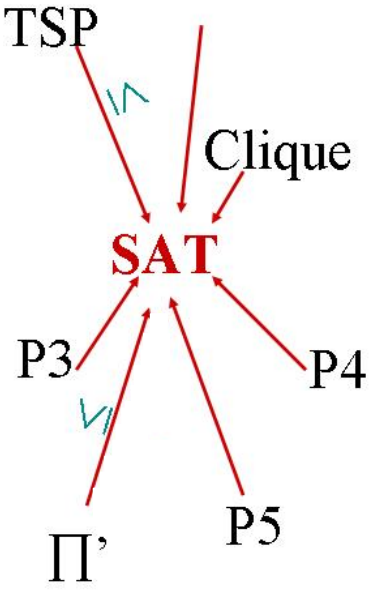


Karp, Richard M. "Reducibility Among Combinatorial Problems". In Raymond E. Miller and James W. Thatcher (editors). Complexity of Computer Computations, New York: Plenum. pp. 85–103, **1972**

Das erste Problem ∏ heißt SAT

- Boolesche Erfüllbarkeit für aussagenlogische Formeln (SAT):
  - Eingabe: Eine Formel φ mit m Klauseln über n Variablen
    Ausgabe: Ja, falls es eine Zuweisung TRUE/FALSE auf die Variablen gibt, so dass die Formel φ erfüllt ist
- Wiederholung:
  - Jede Formel φ kann in konjunktive Normalform (KNF) transformiert werden, ohne dass sich der Wahrheitswert ändert
  - Eine Konjunktion von Klauseln,
    wobei eine Klausel eine Disjunktion von Literalen ist
    - Ein Literal ist eine boolesche Variable oder ihre Negation
  - Z.B. $(A \vee B) \wedge (\neg B \vee C \vee \neg D)$ ist in KNF
    $(A \vee B) \vee (\neg A \wedge C \vee D)$ ist nicht in KNF

SAT



- SAT $\in$ NP (leicht zu sehen)
- Theorem [Cook'71]: Für jedes $\Pi' \in$ NP , gilt $\Pi' \leq_p$ SAT (hier ohne Beweis)
- Ein Problem Π, auf das alle $\Pi' \in$ NP reduziert werden können ($\Pi' \leq_p \Pi$), heißt NP-schwer
  - Nicht unbedingt ein Entscheidungsproblem
- Ein NP-schweres Problem, das in NP liegt, heißt NP-vollständig
- SAT ist NP-vollständig

Cook, Stephen. The complexity of theorem proving procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158. **1971**

Karps 21 NP-vollständige Probleme (1972)

SATISFIABILITY: das Erfüllbarkeitsproblem der Aussagenlogik für Formeln in Konjunktiver Normalform

CLIQUE: Cliquenproblem

SET PACKING: Mengenpackungsproblem

VERTEX COVER: Knotenüberdeckungsproblem

- SET COVERING: Mengenüberdeckungsproblem
- FEEDBACK ARC SET: Feedback Arc Set
- FEEDBACK NODE SET: Feedback Vertex Set
- DIRECTED HAMILTONIAN CIRCUIT: siehe Hamiltonkreisproblem
  - UNDIRECTED HAMILTONIAN CIRCUIT: siehe Hamiltonkreisproblem

0-1 INTEGER PROGRAMMING: siehe Integer Linear Programming

3-SAT: siehe 3-SAT

CHROMATIC NUMBER: graph coloring problem

- CLIQUE COVER: Cliquenproblem
- EXACT COVER: Problem der exakten Überdeckung
  - 3-dimensional MATCHING: 3-dimensional matching (Stable Marriage mit drei Geschlechtern)
  - STEINER TREE: Steinerbaumproblem
  - HITTING SET: Hitting-Set-Problem
  - KNAPSACK: 0-1-Rucksackproblem
    - JOB SEQUENCING: Job sequencing
    - PARTITION: Partitionsproblem

MAX-CUT: Maximaler Schnitt

Seit 1972 wurden mehr als 10000 Probleme als NP-vollständig charakteristiert

Maximaler, nicht minimaler Schnitt

SAT

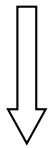
- Referenzproblem
- Algorithmische Lösungen für spezielle Eingaben von SAT sind auch für die Lösung anderer Probleme sehr interessant
- Neue Entwurfsmuster für Algorithmen erweitern Ihren Erfahrungsschatz

Davis, Martin; Putnam, Hilary. A Computing Procedure for Quantification Theory. Journal of the ACM 7 (3): 201–215, **1960**

Davis, Martin; Logemann, George; Loveland, Donald. A Machine Program for Theorem Proving. Communications of the ACM 5 (7): 394–397, **1962**

DPLL-Prozedur: Hauptidee

$$\{\ p \vee q, \neg w \vee \neg q, q \vee \neg r \vee s,\ \neg p \vee \neg q,\ \neg p \vee \neg r \vee \neg s,\ p, \neg w \vee \neg s\ \}$$



Pure Literal?

$$\{\ p \vee q,\ q \vee \neg r \vee s,\ \neg p \vee \neg q,\ \neg p \vee \neg r \vee \neg s,\ p\ \}$$

Entwurfsmuster:

- Identifiziere „leichte“ Teile der Eingabe
- Löse die leichten Teile zur Verkleinerung der Eingabe

DPLL-Prozedur: Hauptidee

$$\{\ p \vee q,\ q \vee \neg r \vee s,\ \neg p \vee \neg q,\ \neg p \vee \neg r \vee \neg s,\ p\ \}$$

Propagierung ⇩ **Zuweisung**: $p = \mathbf{T}$

$$\{\ q \vee \neg r \vee s,\ \neg q,\ \neg r \vee \neg s\ \}$$

Entwurfsmuster:

- Identifiziere Teile der Eingabe ohne Wahlpunkte
- Propagiere Folgerungen
  zur Verkleinerung der Eingabe

DPLL-Prozedur: Hauptidee

$$\{\ p \vee q,\ q \vee \neg r \vee s,\ \neg p \vee \neg q,\ \neg p \vee \neg r \vee \neg s,\ p\ \}$$

Propagierung ⇩ **Zuweisung**: $p = \mathbf{T}$

$$\{\ q \vee \neg r \vee s,\ \neg q,\ \neg r \vee \neg s\ \}$$

⇩ **Zuweisung**: $q = \mathbf{F}$

$$\{\ \neg r \vee s,\ \neg r \vee \neg s\ \}$$

DPLL-Prozedur: Hauptidee

$$\{\ p \vee q,\ q \vee \neg r \vee s,\ \neg p \vee \neg q,\ \neg p \vee \neg r \vee \neg s,\ p\ \}$$

Propagierung ⇓ **Zuweisung**: $p = \mathbf{T}$

$$\{\ q \vee \neg r \vee s,\ \neg q,\ \neg r \vee \neg s\ \}$$

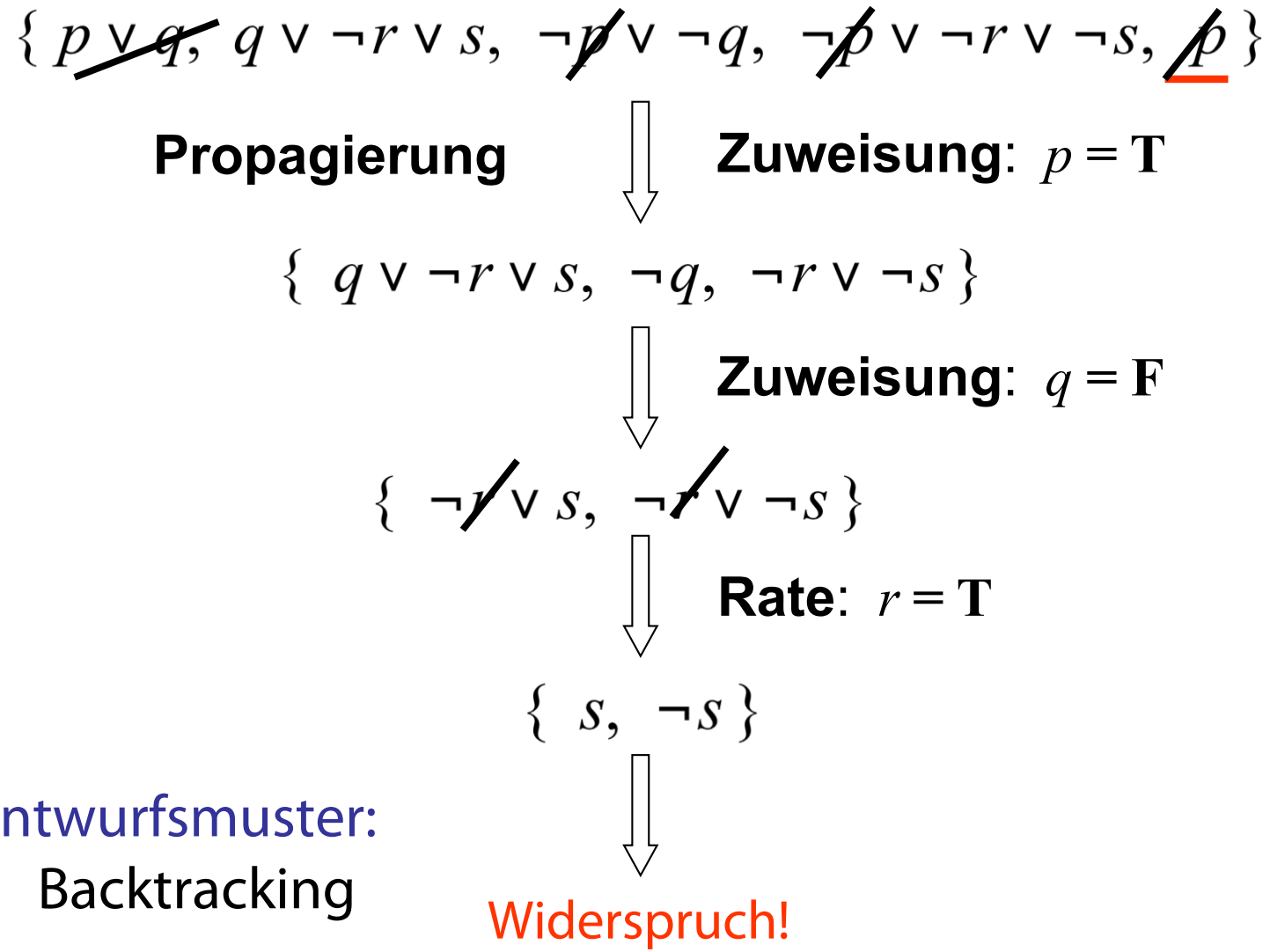
⇓ **Zuweisung**: $q = \mathbf{F}$

$$\{\ \neg r \vee s,\ \neg r \vee \neg s\ \}$$

⇓ **Rate**: $r = \mathbf{T}$

$$\{\ s,\ \neg s\ \}$$

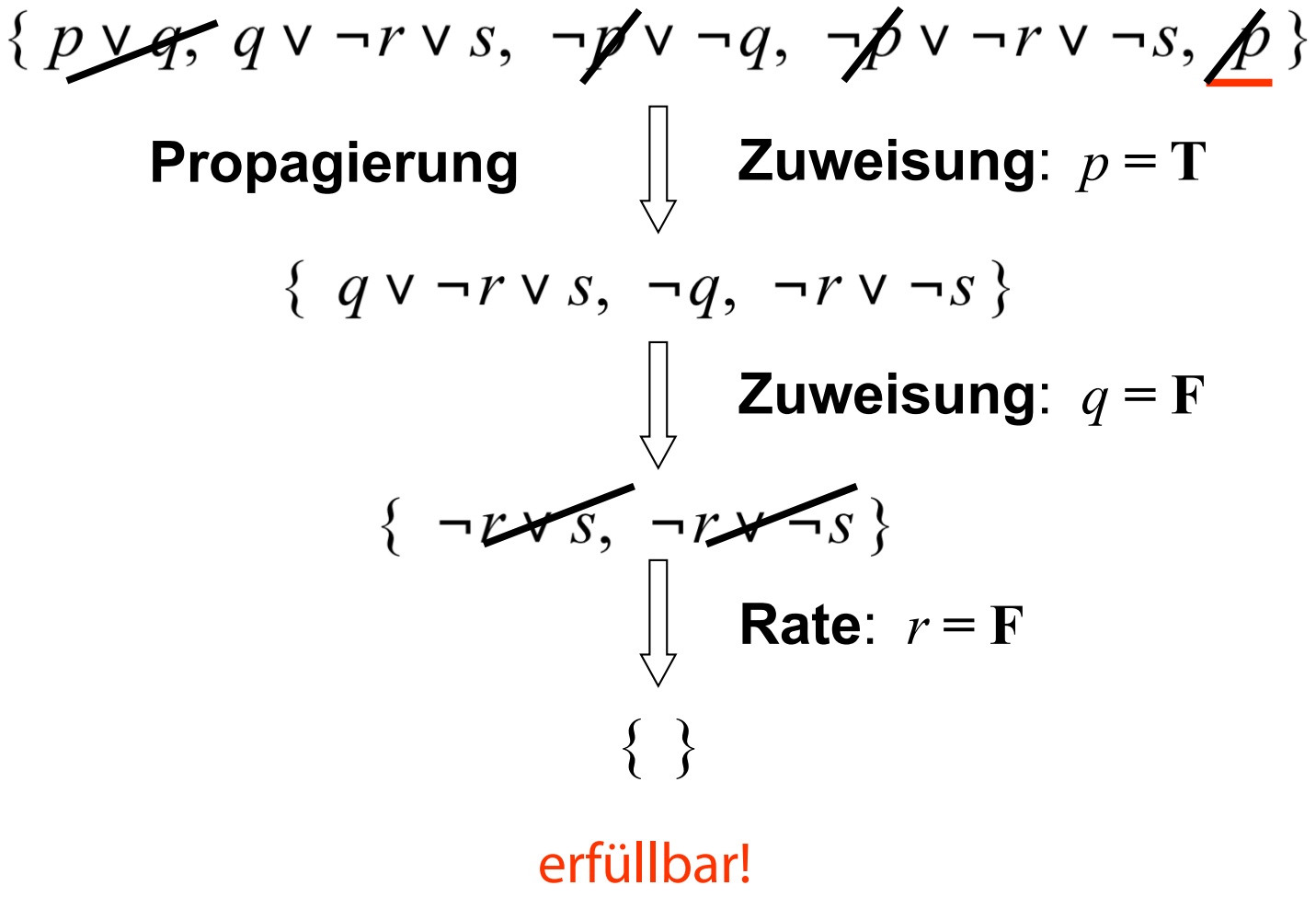
DPLL-Prozedur: Hauptidee



Entwurfsmuster:

- Backtracking

DPLL-Prozedur: Hauptidee



Danksagung

- Nachfolgende Präsentationen zum SAT-Solving sind von Daniel Le Berre

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (\ a\ \vee b) \wedge (\neg b \vee\ \ c\ \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee\ \ f\ ) \ldots$$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = \mathit{False}$ and $f = \mathit{False}$
  - Assign $a = \mathit{False}$ and imply assignments

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = \mathit{False}$ and $f = \mathit{False}$
  - Assign $a = \mathit{False}$ and imply assignments

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = \mathit{False}$ and $f = \mathit{False}$
  - Assign $a = \mathit{False}$ and imply assignments
  - A conflict is reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = \mathit{False}$ and $f = \mathit{False}$
  - Assign $a = \mathit{False}$ and imply assignments
  - A conflict is reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $\varphi \wedge \neg a \wedge \neg c \wedge \neg f \Rightarrow \bot$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = \mathit{False}$ and $f = \mathit{False}$
  - Assign $a = \mathit{False}$ and imply assignments
  - A conflict is reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $\varphi \wedge \neg a \wedge \neg c \wedge \neg f \Rightarrow \bot$
  - $\varphi \Rightarrow a \vee c \vee f$

Clause Learning

- During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \ldots$$

  - Assume decisions $c = \mathit{False}$ and $f = \mathit{False}$
  - Assign $a = \mathit{False}$ and imply assignments
  - A conflict is reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $\varphi \wedge \neg a \wedge \neg c \wedge \neg f \Rightarrow \bot$
  - $\varphi \Rightarrow a \vee c \vee f$
  - Learn new clause $(a \vee c \vee f)$

Entwurfsmuster:

- Gewonnene Erkenntnisse explizit machen und wiederverwenden

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \quad = \quad & (a \vee b) \wedge (\neg b \vee \ c \ \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee \ f \ ) \wedge \\ & (a \vee \ c \ \vee \ f \ ) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \quad = \quad & (a \vee b) \wedge (\neg b \vee \ c \ \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee \ f \ ) \wedge \\ & (a \vee \ c \ \vee \ f \ ) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\begin{aligned}\varphi \quad = \quad & (a \vee b) \wedge (\neg b \vee \ c \ \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee \ f \ ) \wedge \\ & (a \vee \ c \ \vee \ f \ ) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$
  - Assignment $a = \mathit{False}$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$
  - Assignment $a = \mathit{False}$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$
  - Assignment $a = \mathit{False}$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$
  - Assignment $a = \mathit{False}$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a
  - A conflict is again reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$
  - Assignment $a = \mathit{False}$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a
  - A conflict is again reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $\varphi \wedge \neg c \wedge \neg f \Rightarrow \bot$

Non-Chronological Backtracking

- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = \mathit{False}$, $f = \mathit{False}$, $h = \mathit{False}$ and $i = \mathit{False}$
  - Assignment $a = \mathit{False}$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a
  - A conflict is again reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $\varphi \wedge \neg c \wedge \neg f \Rightarrow \bot$
  - $\varphi \Rightarrow c \vee f$

Non-Chronological Backtracking

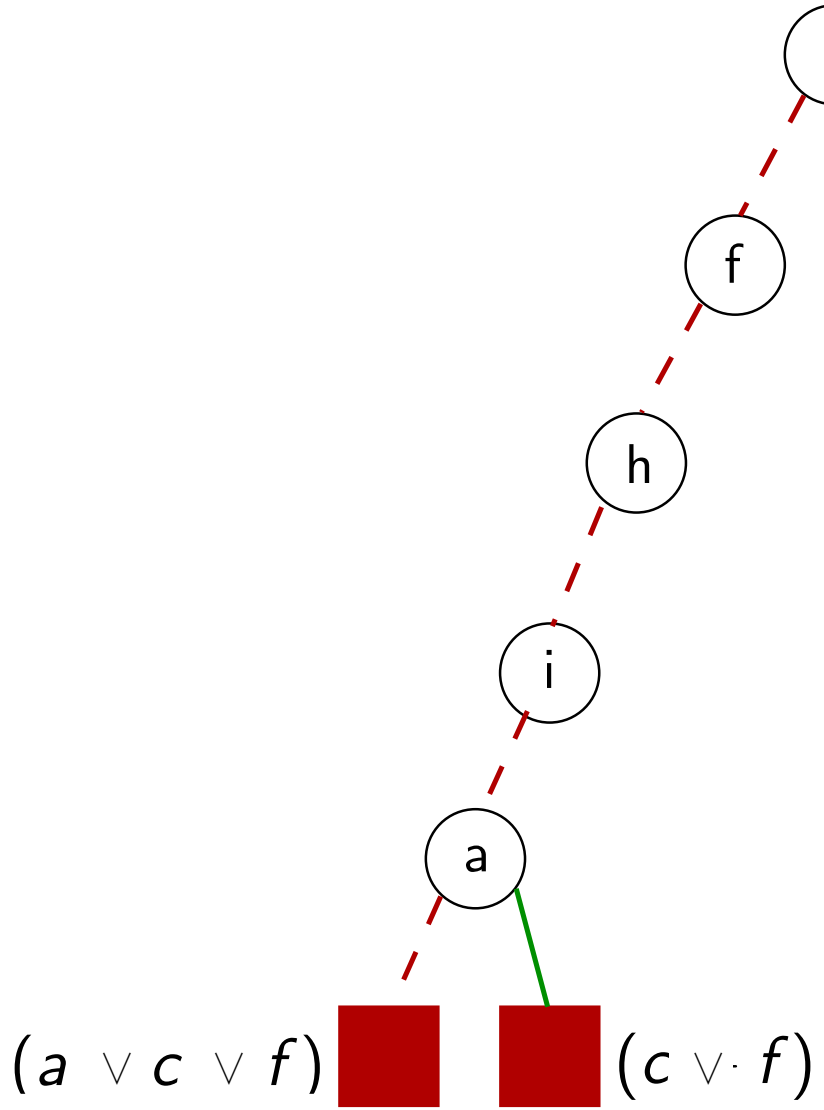
- During backtrack search, for each conflict backtrack to one of the causes of the conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

  - Assume decisions $c = False$, $f = False$, $h = False$ and $i = False$
  - Assignment $a = False$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies a
  - A conflict is again reached : $(\neg d \vee \neg e \vee f)$ is unsatisfied
  - $\varphi \wedge \neg c \wedge \neg f \Rightarrow \bot$
  - $\varphi \Rightarrow c \vee f$

  - Learn new clause $(c \vee f)$

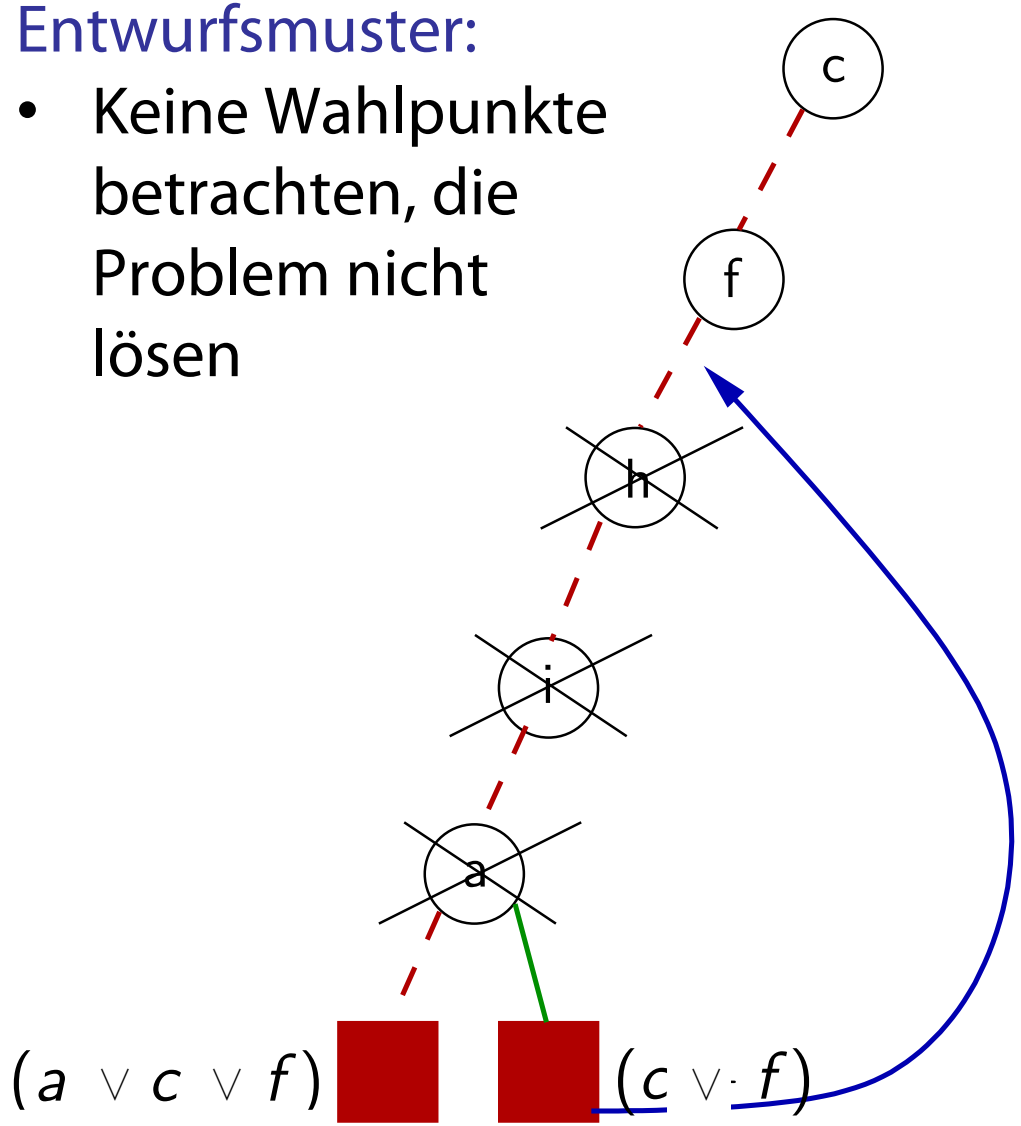
Non-Chronological Backtracking



Non-Chronological Backtracking

Entwurfsmuster:

- Keine Wahlpunkte betrachten, die Problem nicht lösen



- Learnt clause : $(c \vee f)$
- Need to backtrack, given new clause
- Backtrack to most recent decision : $f = False$
- Clause learning and non-chronological backtracking are hallmarks of modern SAT solvers

Danksagung

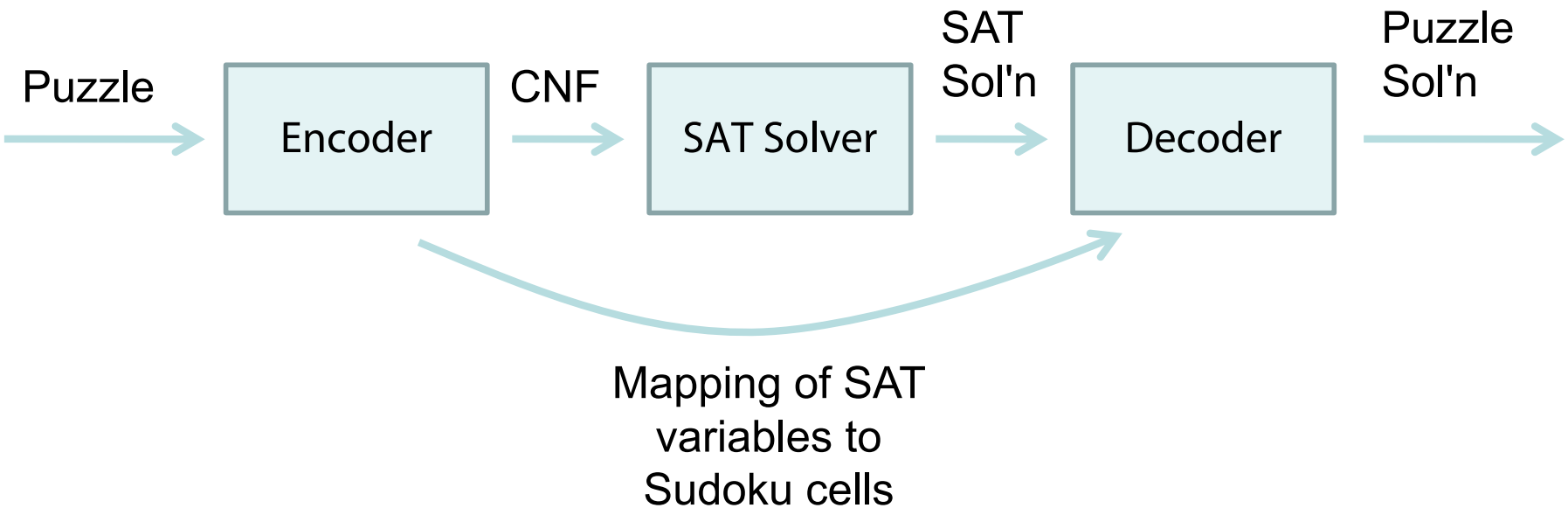
- Präsentationen zur Soduku-Kodierung sind von Will Klieber und Gi-Hwon Kwon

Sudoku

		6	1		2	5		
	3	9				1	4	
				4				
9		2		3		4		1
	8						7	
1		3		6		8		9
				1				
	5	4				9	1	
		7	5		3	2		

- Played on a $n \times n$ board.
- A single number from 1 to n must be put in each cell; some cells are pre-filled.
- Board is subdivided into $\sqrt{n} \times \sqrt{n}$ blocks.
- Each number must appear exactly once in each row, column, and block.
- Designed to have a unique solution.

Puzzle-Solving Process



Naive Encoding (1a)

- Use n^3 variables, labelled "$x_{0,0,0}$" to "$x_{n,n,n}$".
- Variable $x_{r,c,d}$ represents whether the number d is in the cell at row r, column c.

Example of Variable Encoding

3	2	1	4
4	1	2	3
1	4	3	2
2	3	4	1

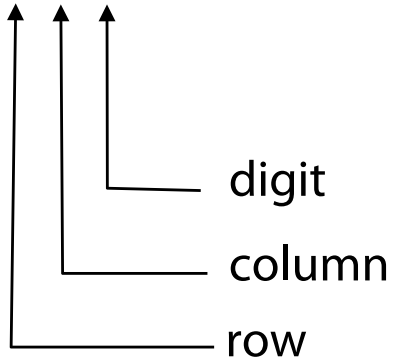
Variable $x_{r,c,d}$ represents whether the digit *d* is in the cell at row *r*, column *c*.

$$x_{1,1,3} = \texttt{true},\ x_{1,2,2} = \texttt{true},\ x_{1,3,1} = \texttt{true},\ x_{1,4,4} = \texttt{true}$$

$$x_{2,1,4} = \texttt{true},\ x_{2,2,1} = \texttt{true},\ x_{2,3,2} = \texttt{true},\ x_{2,4,3} = \texttt{true}$$

$$x_{3,1,1} = \texttt{true},\ x_{3,2,4} = \texttt{true},\ x_{3,3,3} = \texttt{true},\ x_{3,4,2} = \texttt{true}$$

$$x_{4,1,2} = \texttt{true},\ x_{4,2,3} = \texttt{true},\ x_{4,3,4} = \texttt{true},\ x_{4,4,1} = \texttt{true}$$



All others are false.

Naive Encoding (1b)

- Use n^3 variables, labelled "$x_{0,0,0}$" to "$x_{n,n,n}$".
- Variable $x_{r,c,d}$ represents whether the number d is in the cell at row r, column c.
- "Number d must occur exactly once in column c"
  $\Rightarrow$ "Exactly one of $\{x_{1,c,d}, x_{2,c,d}, \ldots, x_{n,c,d}\}$ is true".
- How do we encode the constraint that **exactly one** variable in a set is true?

Naive Encoding (2)

- How do we encode the constraint that ***exactly one*** variable in a set is true?
- We can encode "**exactly one**" as the conjunction of "**at least one**" and "**at most one**".
  - Encoding "**at least one**" is easy: simply take the logical OR of all the propositional variables.
  - Encoding "**at most one**" is harder in CNF. Standard method: "no two variables are both true".
  - I.e., enumerate every possible pair of variables and require that one variable in the pair is false. This takes $O(n^2)$ clauses.
  - [Example on next slide]

Naive Encoding (3)

- Example for 3 variables (x_1, x_2, x_3).
- "At least one is true":

$$x_1 \vee x_2 \vee x_3.$$

- "At most one is true":

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

- "Exactly one is true":

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

Naive Encoding (4)

The following constraints are encoded:

- Exactly one digit appears in each cell.
- Each digit appears exactly once in each row.
- Each digit appears exactly once in each column.
- Each digit appears exactly once in each block.

Each application of the above constraints has the form:
"exactly one of a set variables is true".

All of the above constraints are
independent of the prefilled cells.

Problem with Naive Encoding

- We need n^3 total variables.
  (n rows, n cols, n digits)
- And $O(n^4)$ total clauses.
  - To require that the digit "1" appear exactly once in the first row, we need $O(n^2)$ clauses.
  - Repeat for each digit and each row.
- For some projects, the naive encoding might be OK.
- For large n, it might be a problem

Simple Idea: Variable Elimination

- Simple idea: Don't emit variables that are made true or false by prefilled cells.
  - Larger grids have larger percentage prefilled.
- Also, don't emit clauses that are made true by the prefilled cells.
- This makes encoding and decoding more complicated.

Simple Idea: Variable Elimination

Example: Consider the CNF formula

$$(a \vee d) \wedge (a \vee b \vee c) \wedge (c \vee \neg b \vee e).$$

- Suppose the variable *b* is preset to **`true`**.
- Then the clause $(a \vee b \vee c)$ is automatically true, so we skip the clause.
- Also, the literal $\neg b$ is false, so we leave it out from the 3rd clause.
- Final result: $(a \vee d) \wedge (c \vee e)$.

Results

PUZZLE 100x100	NumVars	NumClauses	Sat Time
Var Elim Only	36,415	712,117	1.04 sec

PUZZLE 144x144	NumVars	NumClauses	Sat Time
Var Elim Only	38,521	596,940	0.91 sec

3-SAT

Jedes SAT-Problem kann auf 3-SAT (max. 3 Literale in Klausel) reduziert werden (Übungsaufgabe)

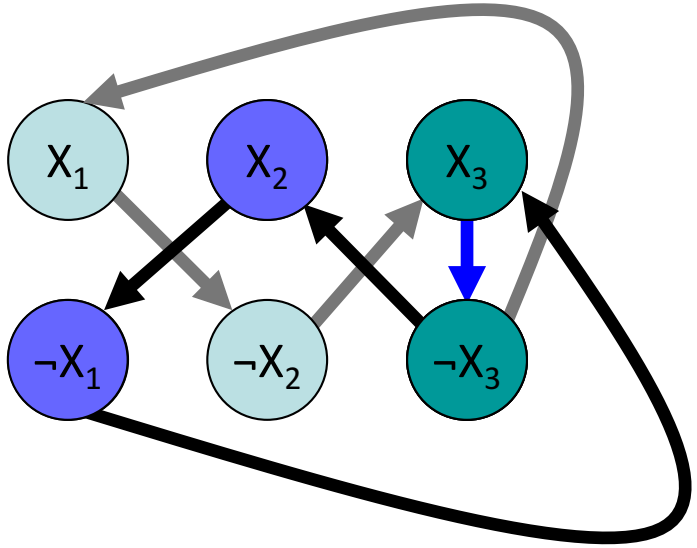
Was ist mit 2-SAT?

- Nicht jede Formel liegt im 2-SAT-Fragment
- Ist 2-SAT schwieriger oder leichter als 3-SAT?
- Finde polynomiellen Algorithmus für 2-SAT

2-SAT Algorithmus

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

- $(a) = (a \vee a)$
- $(a \vee b) = (\neg a \Rightarrow b)$
  $(a \vee b) = (\neg b \Rightarrow a)$



2-SAT Algorithmus

Eine 2-KNF-Formel ist unerfüllbar

gdw.

im Graphen G_F existiert ein Zyklus der Form $x_i \quad \ldots \quad \neg x_i \quad \ldots \quad x_i$

Zyklen mit x und ¬x finden

- Als All-Pair-Shortest-Paths– Problem
  - mit unendlichen Kosten für nichtvorhandene Kanten
  - für alle x überprüfen:
    (x,¬x) und (¬x,x) < ∞?
  - Laufzeit polynomiell
- mit Tiefensuche
  - in Zusammenhangskomponenten zerlegen
  - in O(nm)
    n=Anzahl der Variablen
    m=Anzahl der Klauseln

Bedeutung für Anwendungen

- Formalisierung als 2-SAT versuchen
- Gelungen für:
  - Konfliktfreie Plazierung geometrischer Objekte
  - Clusterung von Daten
  - Anordnungsprobleme (Scheduling)
  - Viele weitere…

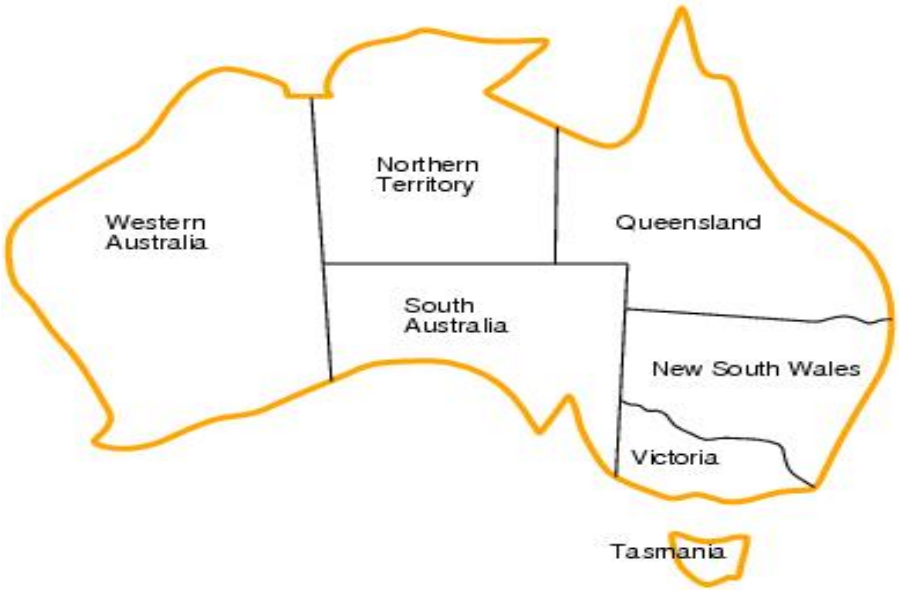
- Nicht immer bietet sich SAT für die Formalisierung von Anwendungsproblemen an

Danksagung

In den nachfolgenden Präsentationen wurden Bilder entnommen aus:

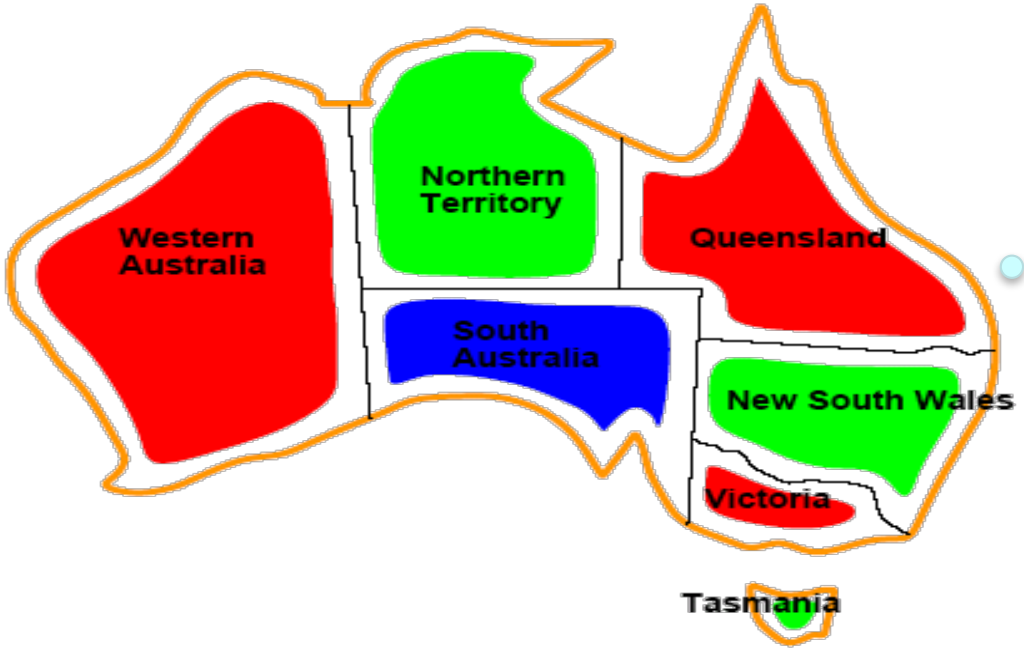
- „Constraint Satisfaction Problems“
  CS 271: Fall 2007, Instructor: Padhraic Smyth

Beispiel: Einfärbung



- Variablen: *WA, NT, Q, NSW, V, SA, T*
- Wertebereich für Variablen: *{rot, grün, blau}*
- Beschränkungen:
  - Benachbarte Regionen müssen verschiedene Farben haben
    - Z.B.: $WA \neq NT$

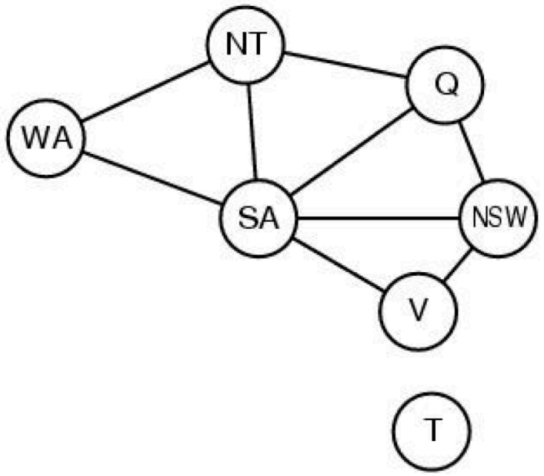
Beispiel: Einfärbung



Darstellbar als planarer Graph

- Lösungen sind Variablenbelegungen, die alle Einschränkungen erfüllen, z.B.:
  - *{WA=rot, NT=grün, Q=rot, NSW=grün, V=rot, SA=blau, T=grün}*

Constraint-Satisfaction-Problem



Kante steht für "ungleich" und stellt Constraint dar

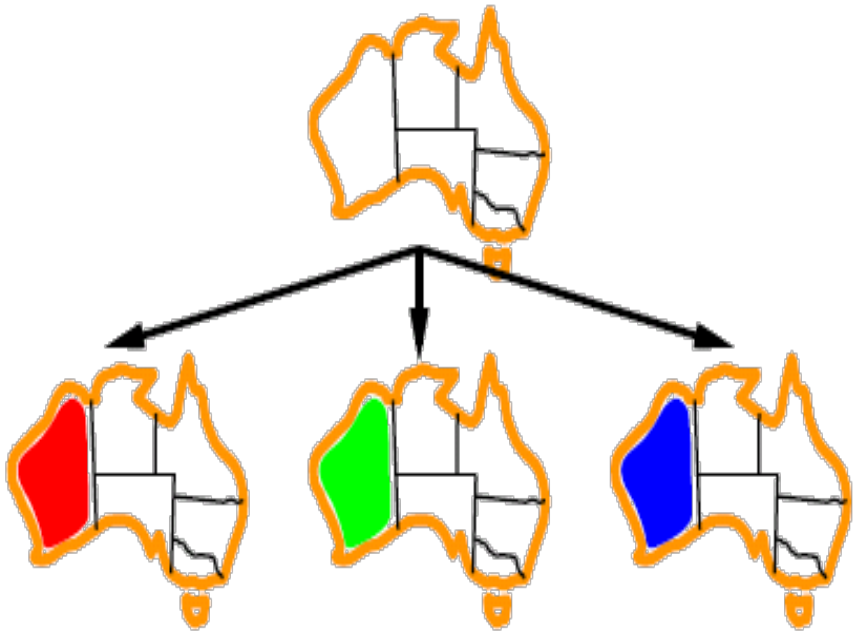
- Lösungen sind Variablenbelegungen, die alle Einschränkungen erfüllen, z.B.:
  - *{WA=rot, NT=grün, Q=rot, NSW=grün, V=rot, SA=blau, T=grün}*

Färbung von Graphen

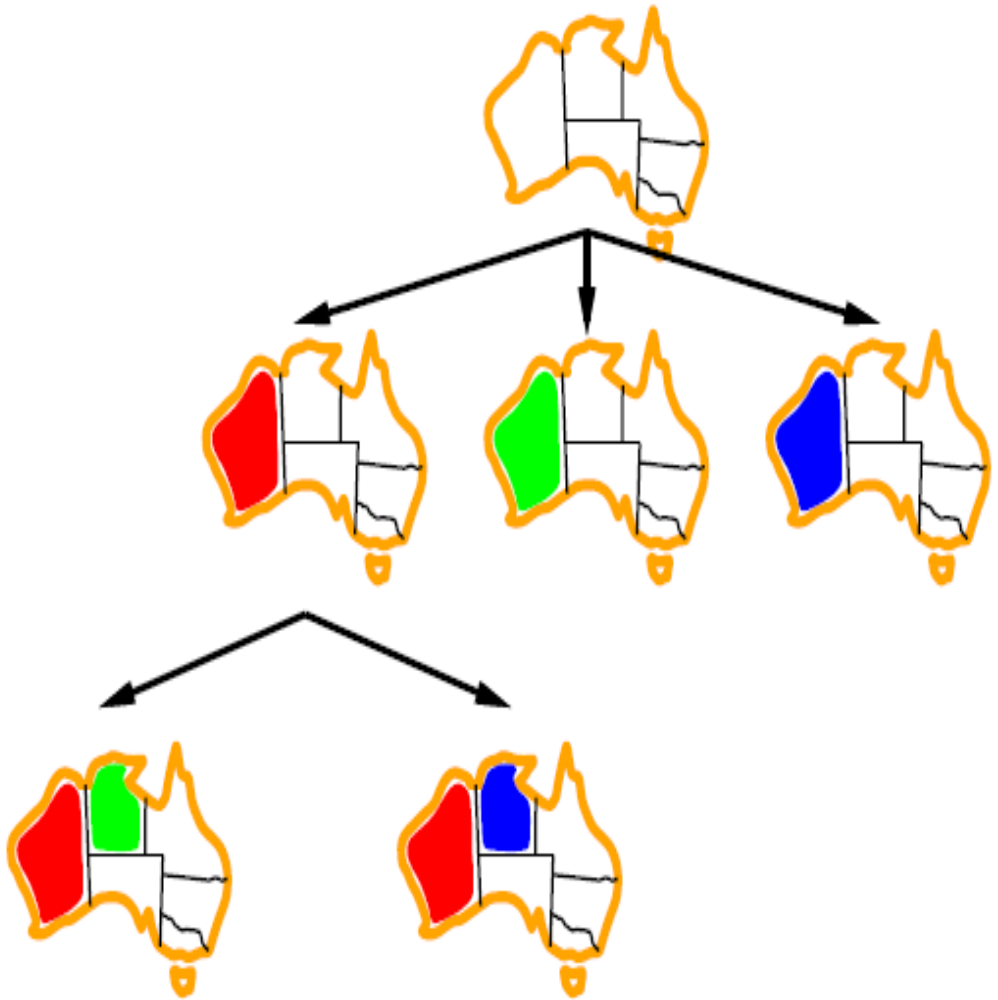
- Gegeben ein planarer Graph
  (keine Kantenüberscheidungen in 2D-Ebene)
- Guthries Vermutung (1852):
  Jeder planare Graph kann mit 4 oder weniger Farben eingefärbt werden (Vier-Farben-Satz)
  - Gezeigt (durch Computer) im Jahre 1977
    (Appel und Haken)
  - Erstes große mathematische Problem,
    das mit Hilfe von Computern gelöst wurde
- Minimale Anzahl = chromatische Zahl
- Bestimmung der chromatischen Zahl ist NP-schwer

Kenneth Appel, Wolfgang Haken (unter Mithilfe von J. Koch),
Every Planar Map is Four-Colorable. In: *Contemporary Mathematics*.
Band 98. American Mathematical Society, Providence, RI **1989**

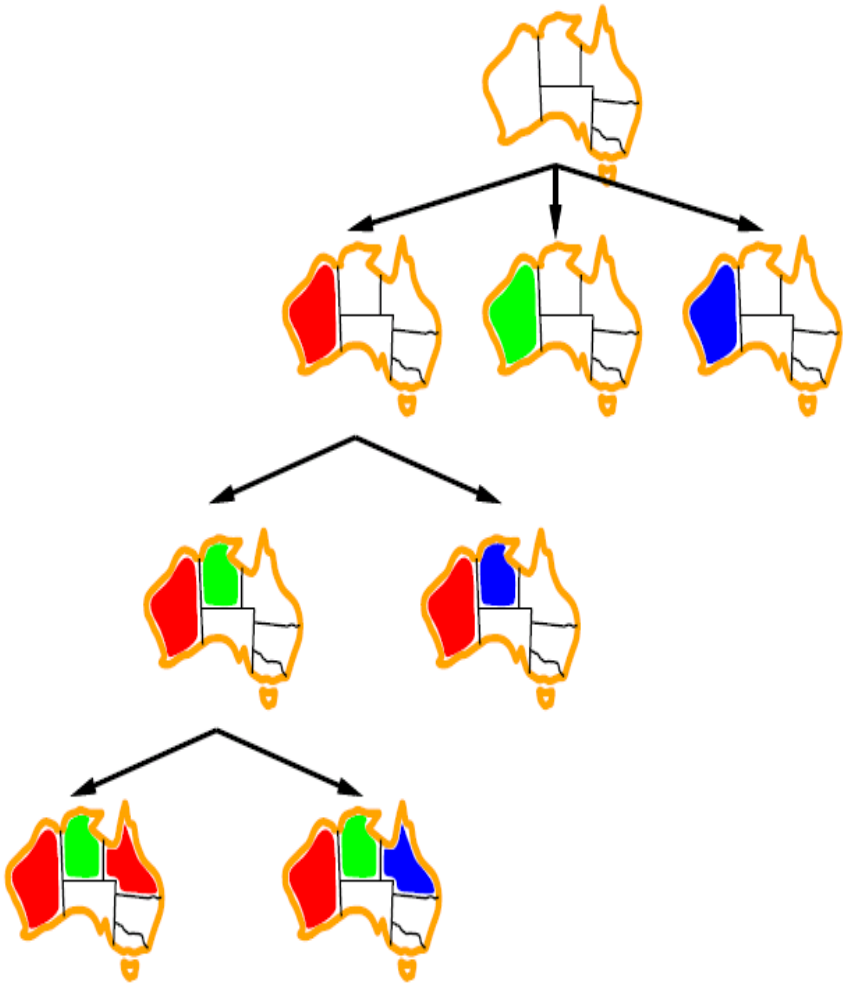
Backtracking um 3 Farben zu probieren



Backtracking?



Backtracking?



Backtracking

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure

    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure

    if *assignment* is complete **then return** *assignment*

    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

    for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**

        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**

            add *{var=value}* to assignment

            result ← RRECURSIVE-BACTRACKING(*assignment, csp*)

            if *result* ≠ *failure* **then return** *result*

            remove *{var=value}* from *assignment*

    return *failure*

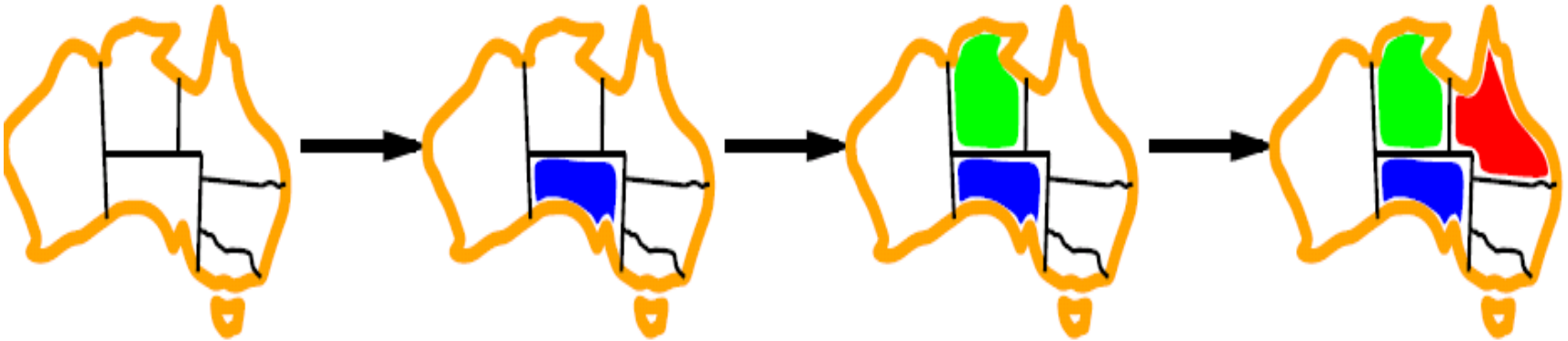
Welche Variable auswählen?

Minimum Remaining Values (MRV)

var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

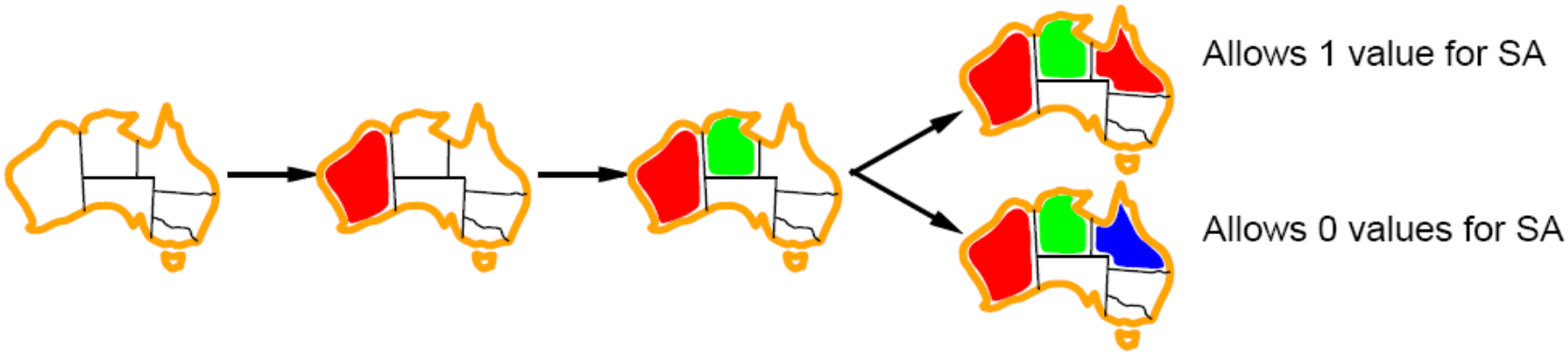
- Auch genannt: Most-Constrained-Variable-Heuristik
- Heuristik: *Wähle Variable mit kleinster Menge möglicher Werte*
  - Frühe Erkennung von Sackgassen

Degree-Heuristik



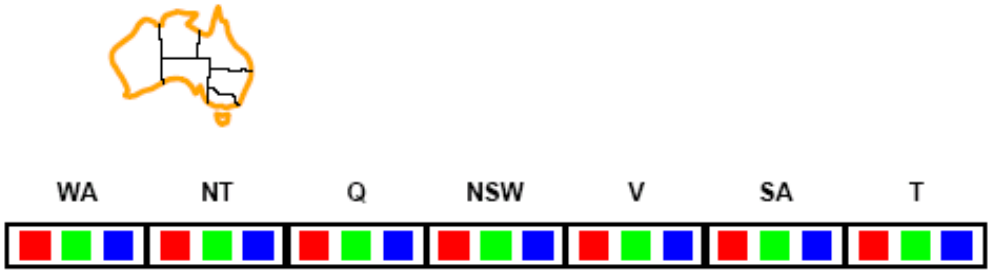
- *Degree-Heuristik*: Wähle Variable, die an der größten Zahl von Beschränkungen bzgl. anderer nicht belegter Variablen beteiligt ist
- Degree-Heuristik gut als Tie-Braker z.B. für erste Variable
- *In welcher Reichenfolge sollen die möglichen Werte der gewählten Variable getestet werden?*

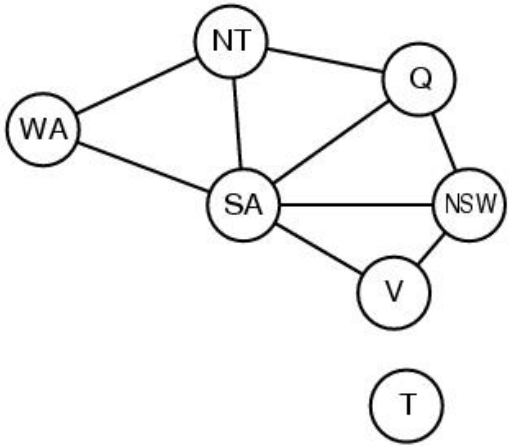
Least-Constraining-Value für Werteordnung



- Least-Constraining-Value-Heuristik
- Heuristik: Wähle Variable, die die wenigsten Beschränkungen für andere Variablen generiert
  - Maximale Flexibilität für weitere Variablenzuweisungen

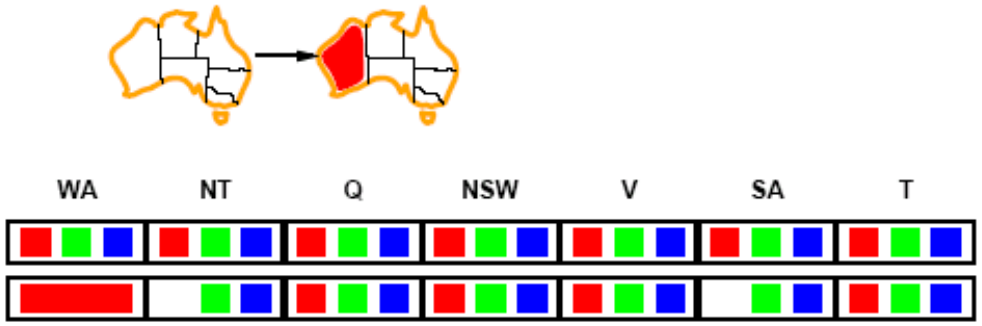
Forward-Checking (Propagierung)

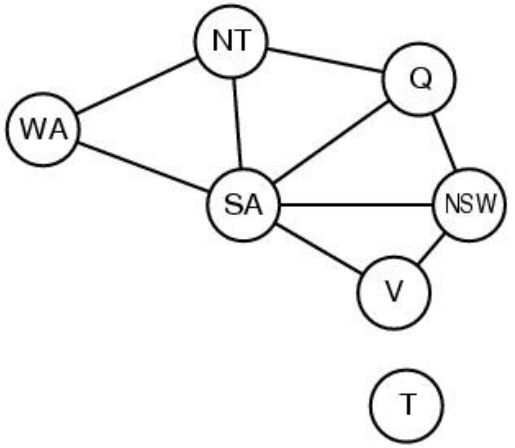




- Können wir Sackgassen früh erkennen?
  - *… und später vermeiden?*
- *Forward-Checking: Führe Verzeichnis von möglichen Werten der Variablen*
- Beende Suchzweig, wenn für eine Variable kein Wert übrigbleibt

Forward-Checking

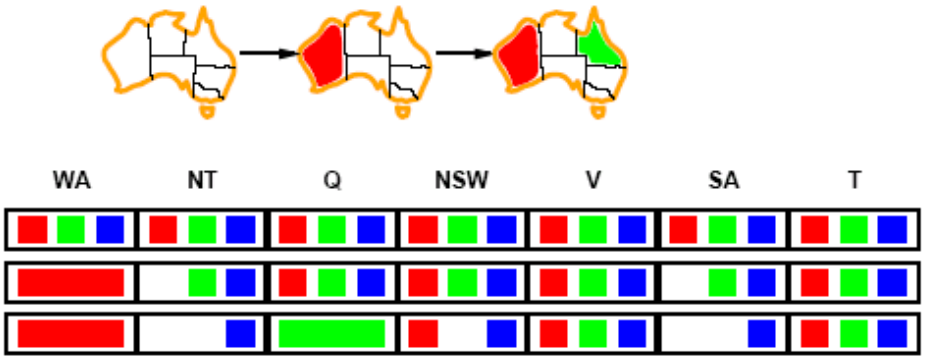


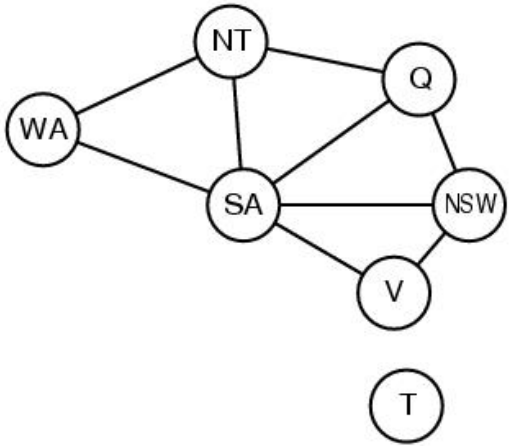


- Wähle *{WA=red}*

- Effekte auf andere Variablen, die mit WA verbunden sind
  - *NT kann nicht mehr rot sein*
  - *SA kann nicht mehr rot sein*

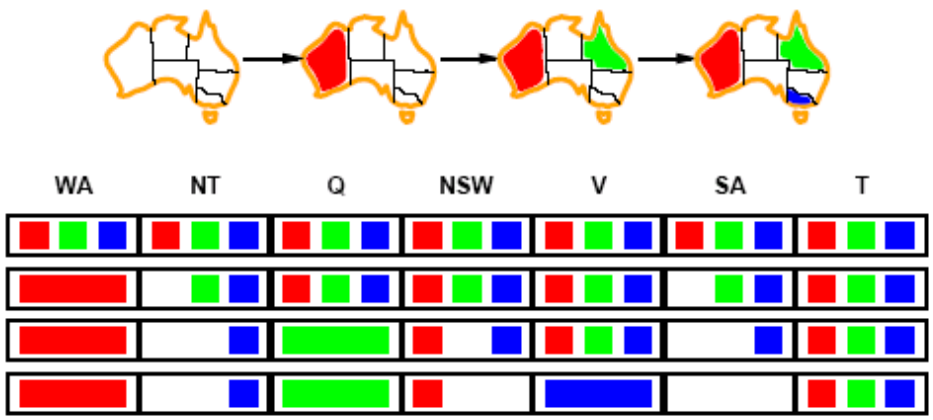
Forward-Checking

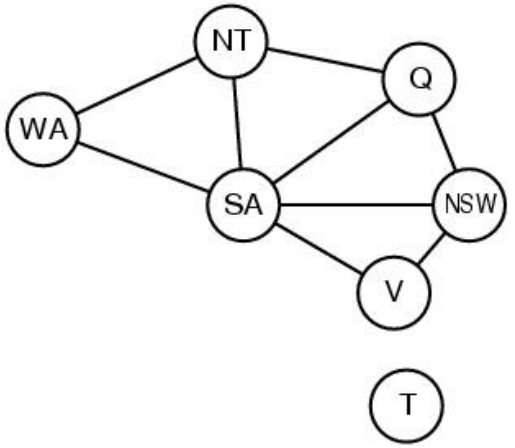




- Wähle *{Q=green}*
- Effekte auf andere Variablen, die mit WA verbunden sind
  - *NT kann nicht mehr grün sein*
  - *NSW kann nicht mehr grün sein*
  - *SA kann nicht mehr grün sein*
- *MRV-Heuristik würde als nächstes NT oder SA wählen*

Forward-Checking



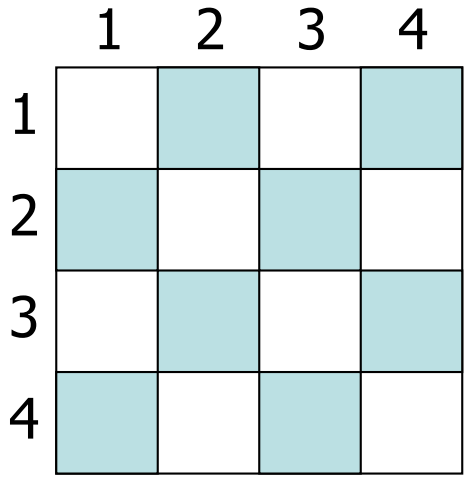


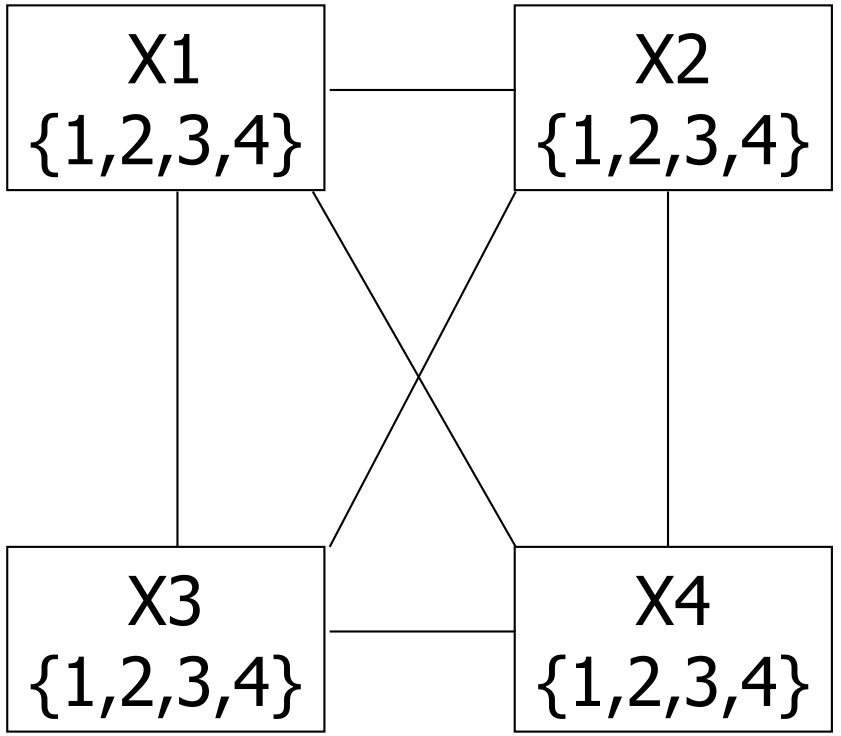
- Fall für *V* blau gewählt würde

- Effekte auf andere Variablen, die mit WA verbunden sind
  - *NSW kann nicht mehr blau sein*
  - *SA hat keinen möglichen Wert mehr*

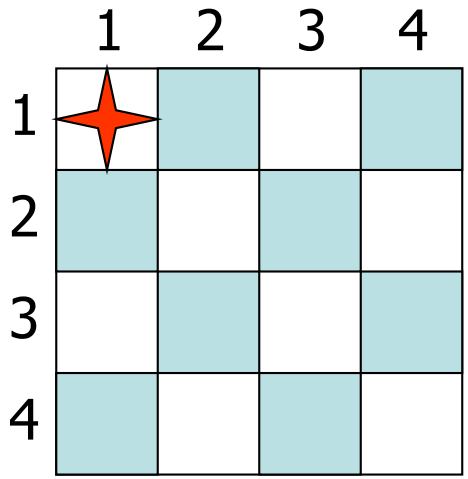
- FC hat eine Belegung entdeckt, die inkonsistent mit den Einschränkungen ist, so dass Backtracking einsetzten kann

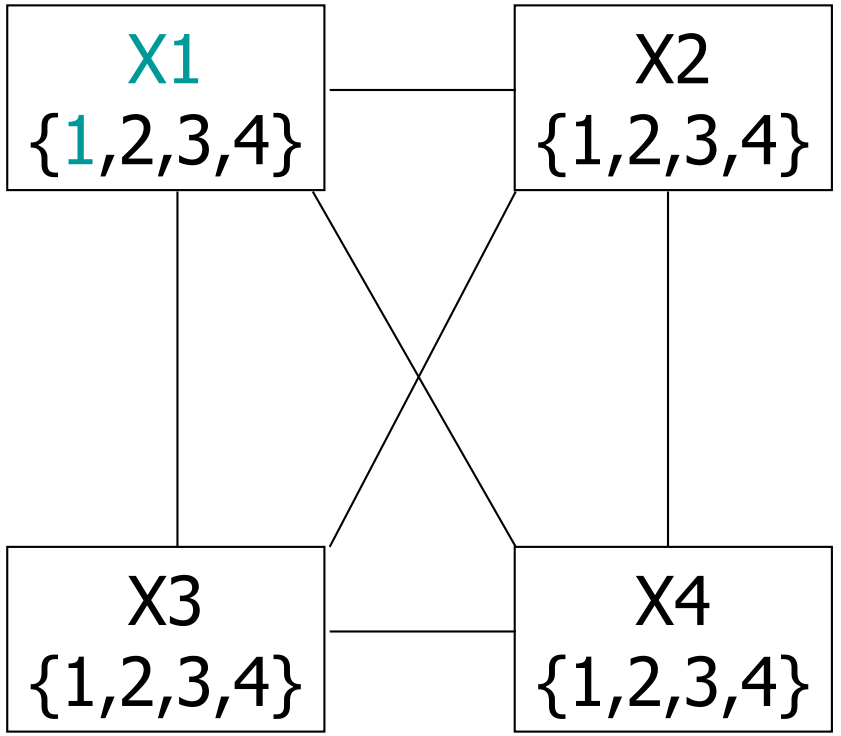
Beispiel: 4-Damen-Problem



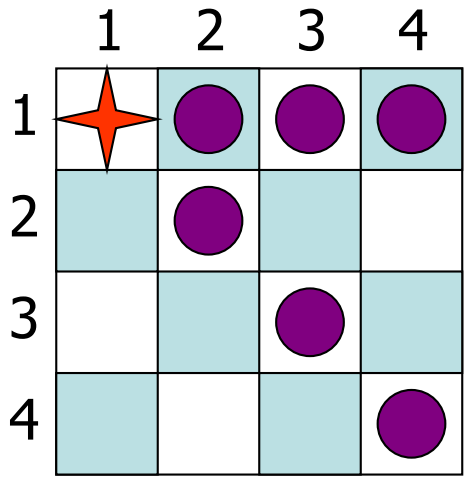


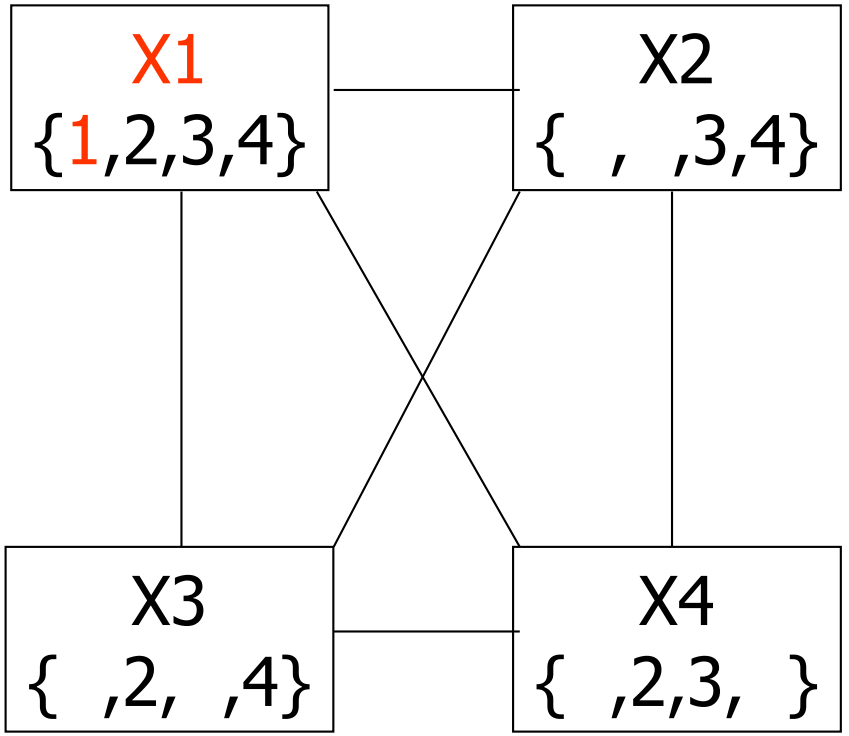
Beispiel: 4-Damen-Problem



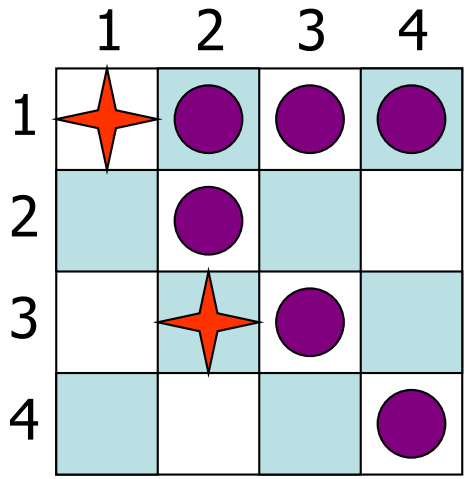


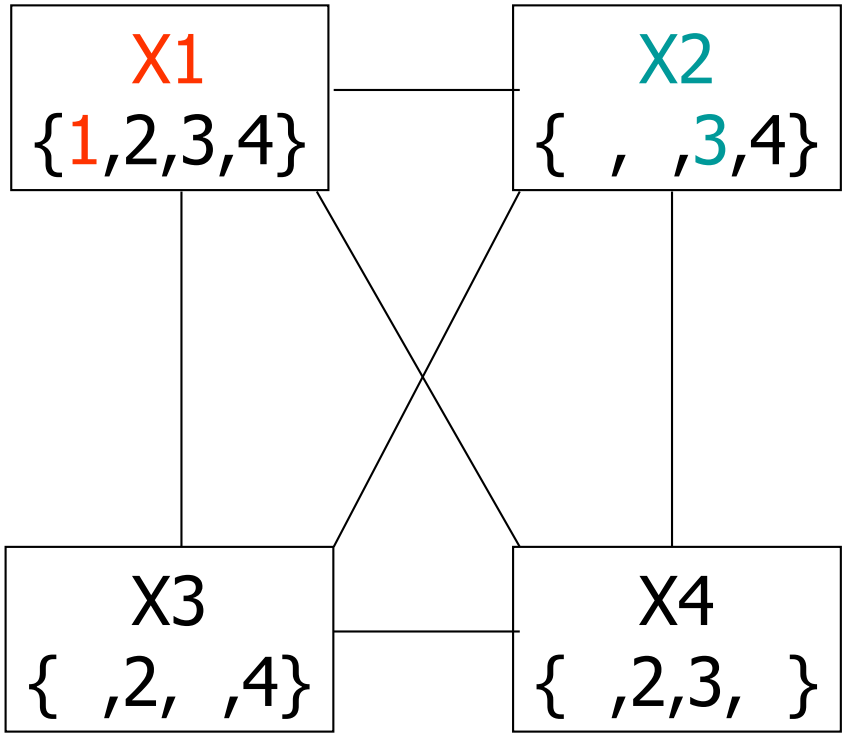
Beispiel: 4-Damen-Problem



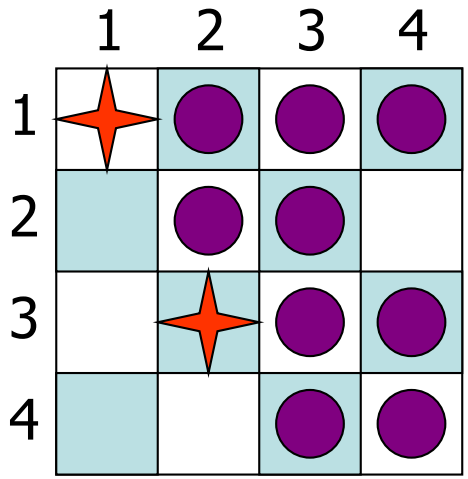


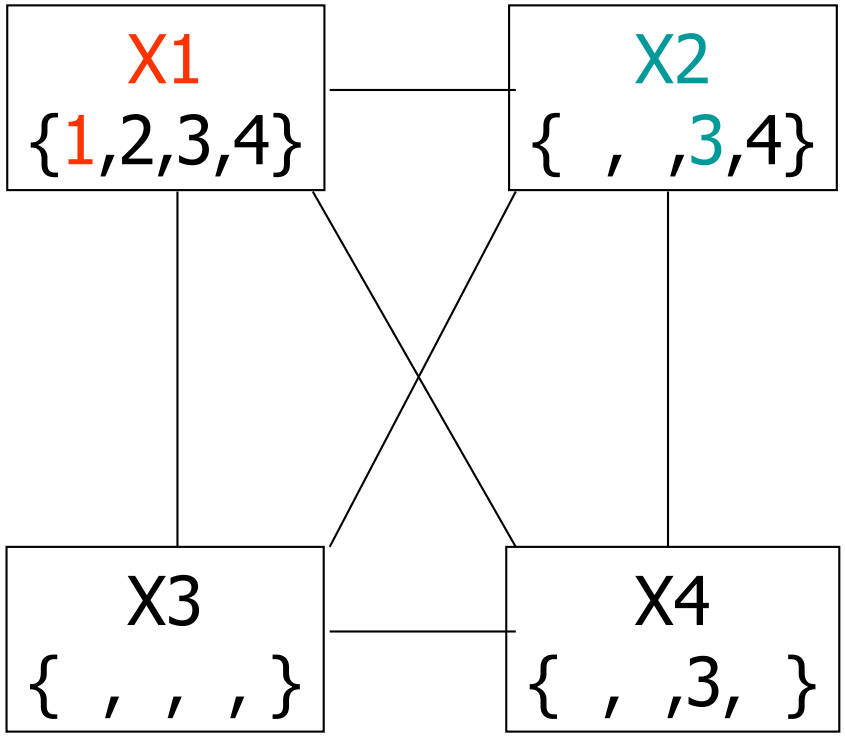
Beispiel: 4-Damen-Problem



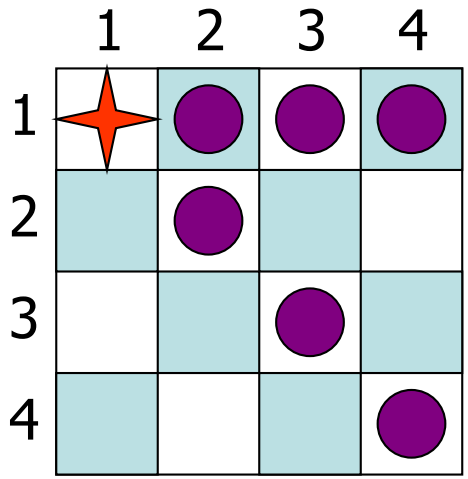


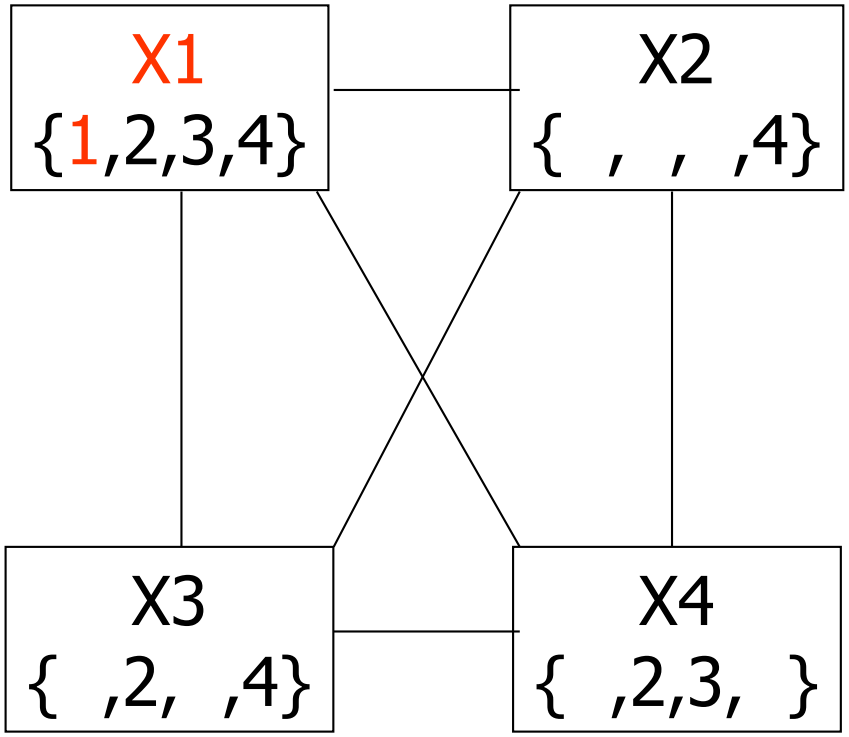
Beispiel: 4-Damen-Problem



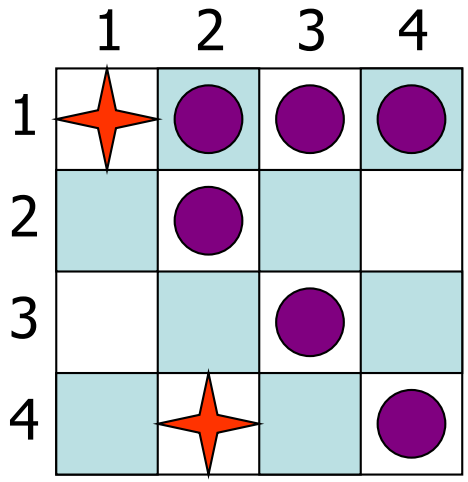


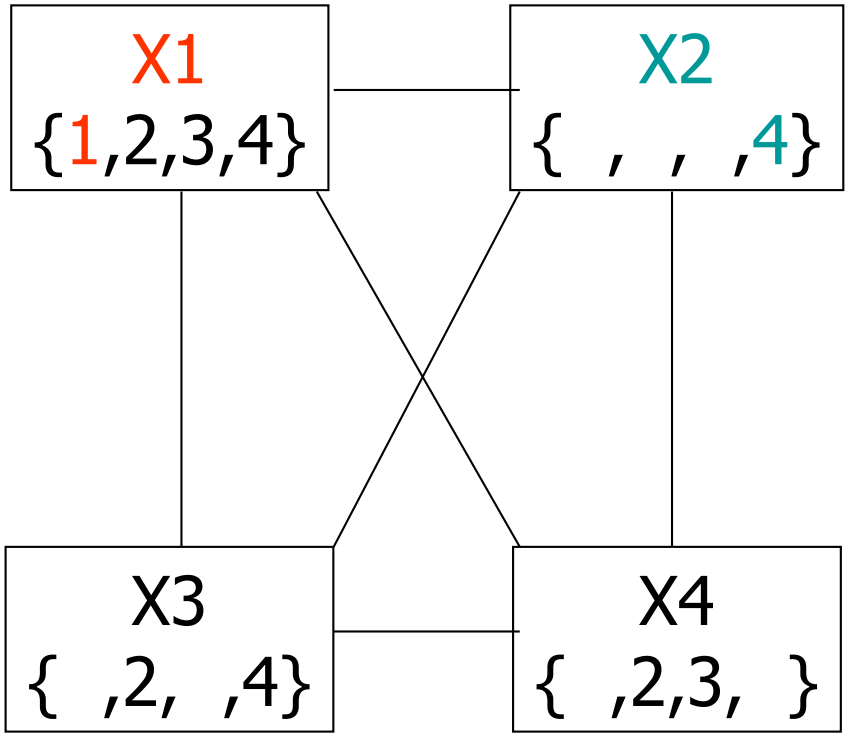
Beispiel: 4-Damen-Problem



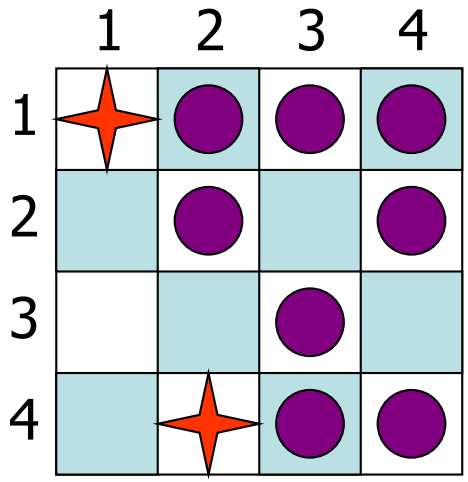


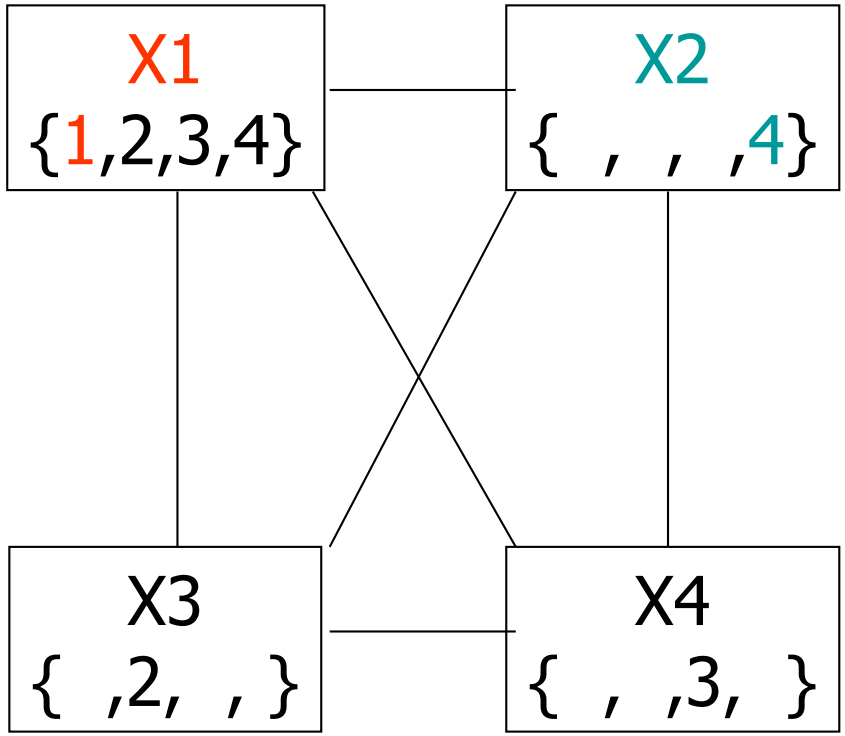
Beispiel: 4-Damen-Problem



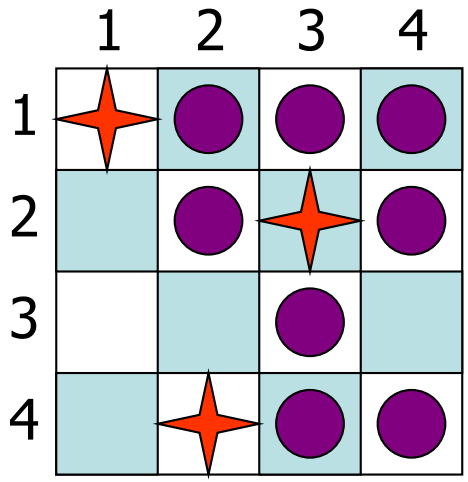


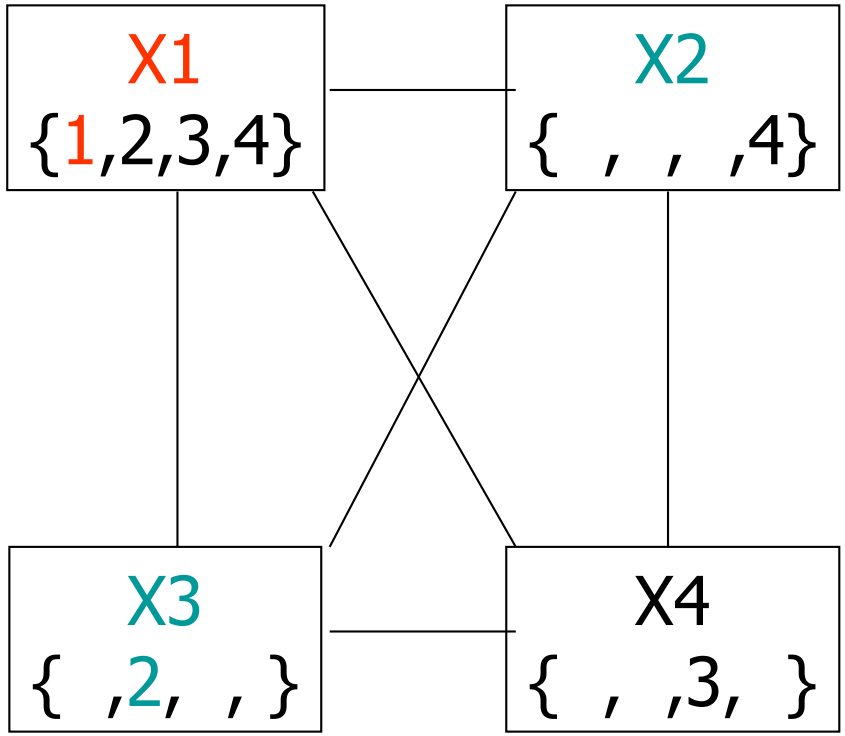
Beispiel: 4-Damen-Problem



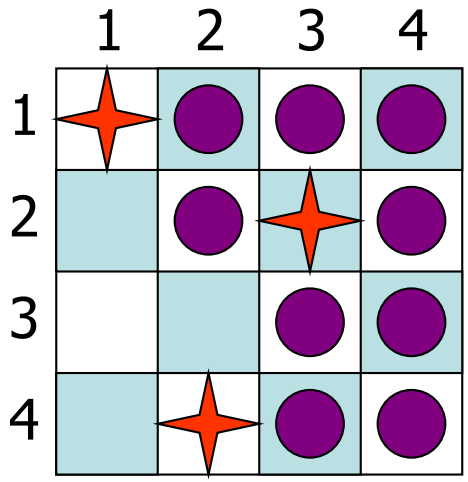


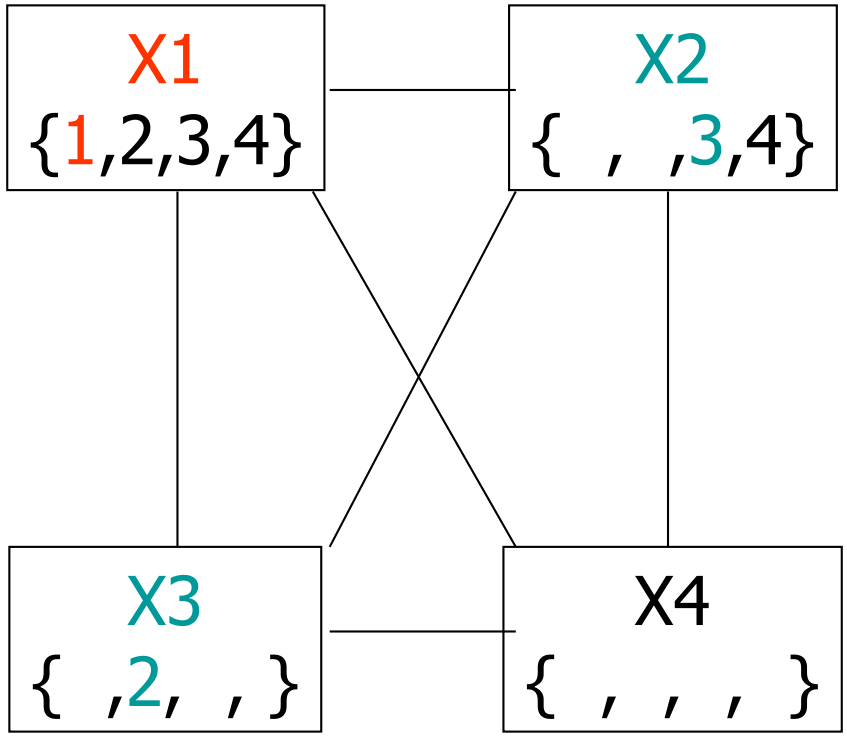
Beispiel: 4-Damen-Problem





Beispiel: 4-Damen-Problem





Zusammenfasssung

- Offenes Problem: P=NP?
- Clique, TSP, chromatische Zahl, n-Queens ...
- NP-schwer, NP-vollständig, Reduktion von Problemen
- Referenzprobleme SAT / chromatische Zahl
  - Entwurfsmuster zur algorithmischen Lösung schwieriger Probleme
- Lösen eines kombinatorischen Anwendungsproblems
  - Entwurfsmuster: Reduktion auf bekanntes, wohluntersuchtes Problem
- Schwierig heißt formal (mindestens) NP-schwer