Constraint Satisfaction

Mario Mohr

GWV-Tutorium 2013/14

31. Januar 2014

Constraint Satisfaction Problem - I

Ein **Constraint Satisfaction Problem** wird durch

- eine Menge von Variablen mit ihren jeweiligen Domänen und
- eine Menge von Einschränkungen (Constraints) über Mengen von Variablen

definiert.
Eine Lösung eines CSPs ist eine Variablenbelegung, die alle Constraints erfüllt.

Constraint Satisfaction Problem - II: Beispiel Sudoku

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

Variablen: $Feld_{1,1}, Feld_{1,2}, ..., Feld_{4,4}$
Domänen: $D(Feld_{1,1}) = D(Feld_{1,2}) = ... = D(Feld_{4,4}) = \{1, 2, 3, 4\}$
Constraints:

- keine Duplikate in Zeilen, z.B.
  $Feld_{1,1} \cup Feld_{2,1} \cup Feld_{3,1}, Feld_{4,1} = \{1,2,3,4\}$
- keine Duplikate in Spalten, z.B.
  $Feld_{1,1} \cup Feld_{1,2} \cup Feld_{1,3}, Feld_{1,4} = \{1,2,3,4\}$
- keine Duplikate in Zellen, z.B.
  $Feld_{1,1} \cup Feld_{1,2} \cup Feld_{2,1}, Feld_{2,2} = \{1,2,3,4\}$
- Startbelegungen von Feldern, z.B.
  $Feld_{2,1} = 4$

Generate-and-Test (Bruteforce)

Beim **Generate-and-Test** - Ansatz

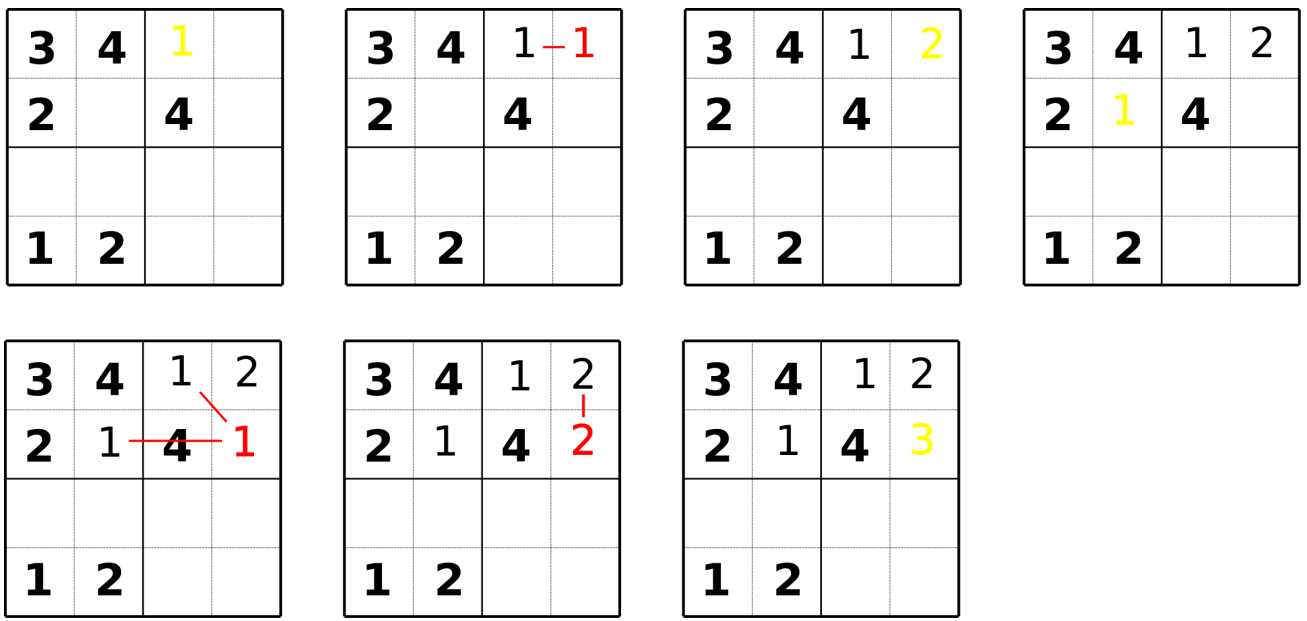
- werden systematisch alle Kombinationen von Variablenbelegungen ausprobiert,
- bis eine gefunden wird, die alle constraints erfüllt; diese wird als Lösung ausgegeben.

Dieser Ansatz ist auch unter dem Namen **Bruteforce** bekannt.

Backtracking

Beim **Backtracking** - Ansatz

- werden, ähnlich wie bei Generate-and-Test, systematisch Kombinationen von Variablenbelegungen ausprobiert;
- allerdings wird beim ersten Konflikt die Zusammenstellung einer Kombination abgebrochen und bei der letzten Konfliktfreien Teilkombination fortgesetzt.

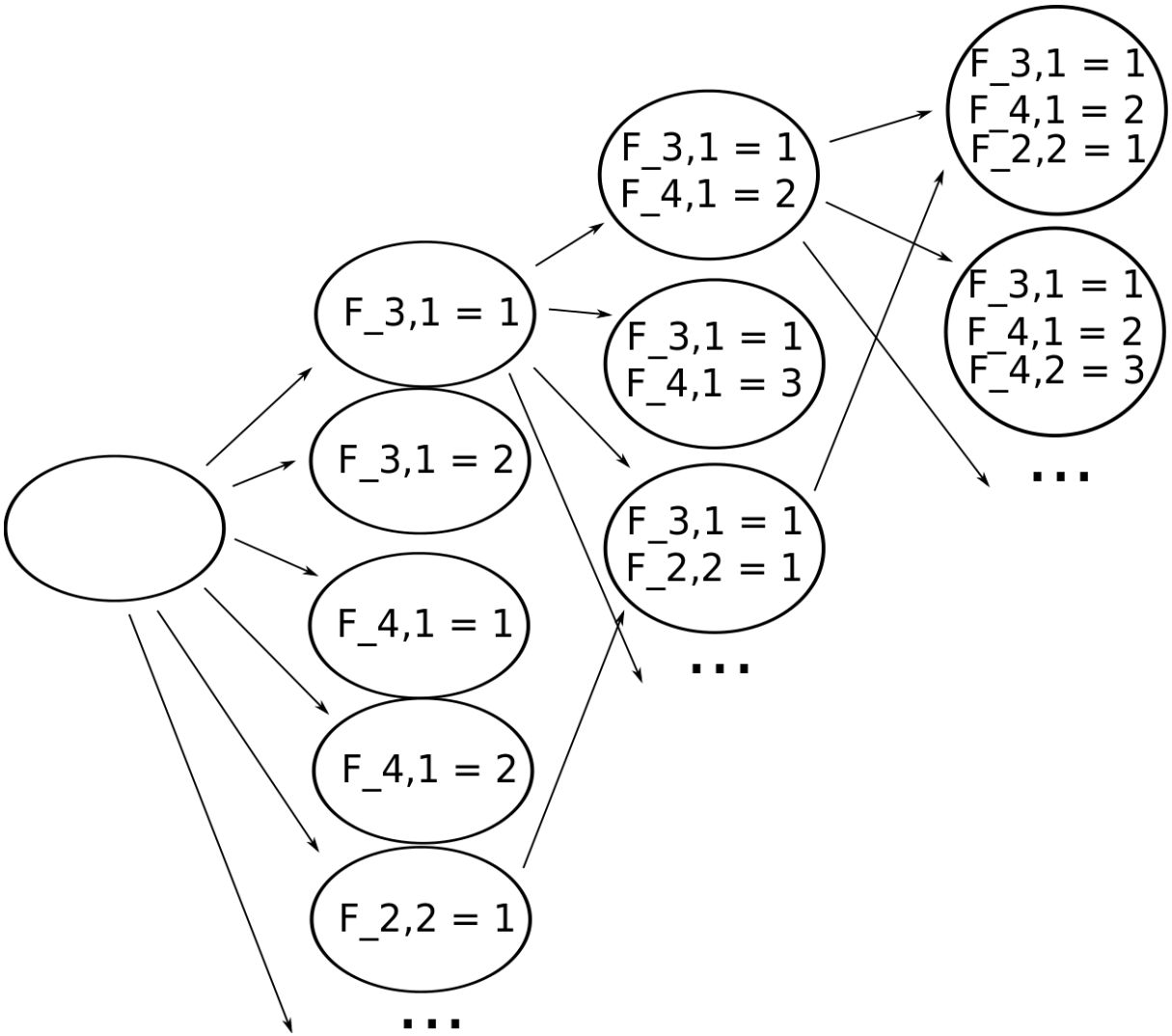


Generische Suche - I

Ein CSP kann auch durch einen beliebigen **Suchalgorithmus** gelöst werden. Dabei

- sind die **Knoten** des Graphen (teilweise) Belegungen der Variablen
- die **Kanten** des Graphen Konfliktfreie Belegungen jeweils einer weiteren Variablen,
- der Startknoten der ohne Belegung einer Variablen und
- ein Zielknoten ein Knoten mit vollständiger Variablenbelegung.

Generische Suche - II



Arc-Consistency-Algorithmen

Bei **Arc-Consistency**-Algorithmen

- werden die Variablen, ihre Domänen und die constraints in einem **Constraint Network** zusammengefasst;
- anschließend werden durch Propagation von Informationen im Netzwerk die Variablendomänen so weit wie möglich eingeschränkt.

Optimization Problems - hard & soft constraints

- Die in einem CSP auftretenden constraints werden auch als **hard constraints** bezeichnet; sie **müssen** von jeder gültigen Lösung erfüllt werden.
- Darüber hinaus gibt es sogenannte **soft constraints**; dabei handelt es sich um Kostenfunktionen, die die Qualität einer gültigen Lösung bestimmen (je geringer die Kosten, desto besser).
- Probleme, die ausschließlich durch soft constrains eingeschränkt sind, nennt man **Optimierungsprobleme** (optimization problems).
- Probleme mit sowohl hard als auch soft constraints nennt man textbfbeschränkte Optimierungsprobleme (constrained optimization problems).

CSPs / Optimierungsprobleme als Graph

Eine alternative Möglichkeit, CSPs und Optimierungsprobleme als Graphen darzustellen:

- Jeder **Knoten** steht für eine vollständige Variablenbelegung
- **Kanten** verbinden Knoten, die sich in der Belegung genau einer Variablen unterscheiden

Greedy Descent

Beim **Greedy Descent**

- wird in jeder Iteration die Variablenbelegunsänderung vorgenommen, die die größte Qualitätsverbesserung bewirkt;
- die KI *steigt also auf dem Kostengraphen immer weiter ab*.

Komplexe Kostenfunktionen

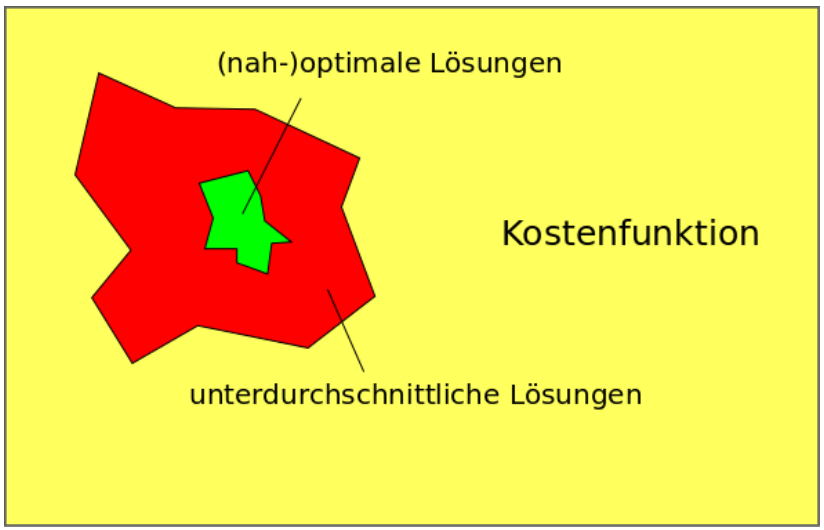
Ein reiner Greedy Descent - Algorithmus liefert nur selten garantiert die beste Lösung, denn

- ein lokales Minimum der Kostenfunktion ist nicht zwangsläufig auch ein globales
- der Weg von der Start- zur *optimalen Belegung*

Komplexe Kostenfunktionen

Ein reiner Greedy Descent - Algorithmus liefert nur selten garantiert die beste Lösung, denn

- ein lokales Minimum der Kostenfunktion ist nicht zwangsläufig auch ein globales
- der Weg von der Start- zur *optimalen Belegung* kann über unterdurchschnittlich gute Lösungen, also 'Kämme' ('ridges') in der Kostenfunktion führen:



Stochastic Local Search

Um die Beschränkung auf ein lokales Optimum zu verhindern, können wir das Verhalten der KI vom Zufall beeinflussen lassen. Bei der **Stochastic Local Search** wird jede Iteration in Abhängigkeit einer Wahrscheinlichkeitsverteilung zufällig einer der folgenden Schritte durchgeführt:

- **Greedy Descent**
- **Random Restart**: Wähle für den nächsten Schritt eine zufällige (noch nicht probierte) Belegung der Variablen, also einen zufälligen Knoten des Graphen
- **Random Walk**: Bewege dich im nächsten Schritt zu einem (semi-)zufällig gewählten Nachbarknoten

Beim **Simulated Annealing** ('Simuliertes Abkühlen) werden jede Iteration folgende Schritte durchgeführt:

1. berechne die Qualität eines zufällig gewählten Nachbarknotens
2. IF (die gewählte Variablenbelegung ist besser als die aktuelle)
   - bewege dich zum gewählten Nachbarknoten
3. ELSE
   - mit einer Wahrscheinlichkeit positiv abhängig vom Qualitätsunterschied und der 'Temperatur':
     - reduziere die 'Temperatur' und bewege dich zum ausgewählten Knoten ODER
     - GOTO 1.

Die KI wählt also mit sinkender Temperatur (also steigender Iterationszahl) immer weniger zufällig den Nachfolgerknoten aus.

Bei der **Parallel Search** werden k Suchen gleichzeitig ausgeführt, d.h. der Graph wird an k Stellen gleichzeitig durchsucht;

bei der **Beam Search** wird zusätzlich jede Iteration global nur die n besten Nachbarn erkundet.