Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

12. Mai 2014

Constraint-Netzwerke

Definition

Ein Constraint-Netzwerk $\mathfrak{N} = (X, D, C)$ besteht aus einer endlichen Menge $X = \{x_1, \ldots, x_n\}$ von Variablen mit Domänen $D = \{D_1, \ldots, D_n\}$ und einer Menge von Constraints $C = \{C_1, \ldots, C_k\}$. Die Menge aller Scopes von Constraints heißt das *Schema* des Constraint-Netzwerks.

Ein Constraint beschreibt, welche Werte für die Variablen aus seinem Scope gleichzeitig zulässig sind.

Sind alle Domänen endlich bezeichnet man ein Constraint-Netzwerk auch als Constraint Satisfaction Problem (CSP)

Abkürzungen

Wir schreiben:

- $\vec{a}_n = \langle x_1 = a_1, \ldots, x_n = a_n \rangle$
- $\langle \vec{a}_n, x_k = a_k \rangle = \langle \vec{a}_n, a_k \rangle = \langle x_1 = a_1, \ldots, x_n = a_n, x_k = a_k \rangle$
- $X = \{x_1, \ldots, x_n\}$
- $D = \{D_1, \ldots, D_n\}$
- $C \subseteq \{C_{ij} \mid 1 \leq i < j \leq n\}$ (d.h. es gibt höchstens einen binären Constraint mit Scope $\{x_i, x_j\}$ und keine weiteren Constraints)
- $C_{ij} = C_{ji}$ (d.h. die Reihenfolge der Indizes spielt keine Rolle)
- $D' = \{D'_1, \ldots, D'_n\}$
- $D'_i \leftarrow_C D_i$ für die Zuweisung einer Kopie
- $D' \leftarrow_C D$ für die Zuweisung von Kopien aller Elemente von D, jedes D'_i ist also eine Kopie des entsprechenden D_i

Lösung von CSPs: Backtracking

function BACKTRACKING-SUCHE(X, D, C)
  $i \leftarrow 1,\ D'_1 \leftarrow_C D_1$
  while $1 \leq i \leq n$ **do**
    $\langle a_i, D'_i \rangle \leftarrow$ WÄHLE-WERT($\vec{a}_{i-1}, x_i, X, D', C$)
    if $a_i = null$ **then** *[Kein Wert für x_i]*
      $i \leftarrow i - 1$ *[Backtracking]*
    else
      $i \leftarrow i + 1,\ D'_i \leftarrow_C D_i$
    end if
  end while
  if $i = 0$ **then**
    return inkonsistent
  else
    return $\vec{a}_n$
  end if
end function

Lösung von CSPs: Auswahl der Werte

function WÄHLE-WERT($\vec{a}_{i-1}, x_i, X, D', C$)
    while $D'_i \neq \emptyset$ **do**
        Wähle $a_i \in D'_i$
        $D'_i \leftarrow D'_i \setminus \{a_i\}$
        if KONSISTENT?($\vec{a}_i, C$) **then**
            return $\langle a_i, D'_i \rangle$
        end if
    end while
    return *null*
end function

Komplexität des Backtrackings

Seien

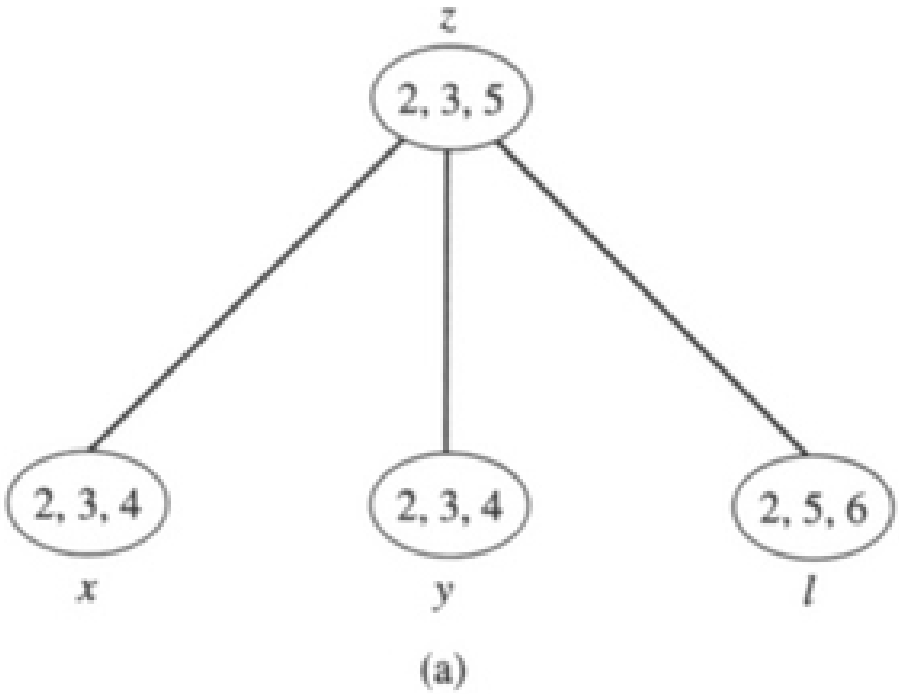
- e die Anzahl der Constraints, $e = |C|$
- k die maximale Größe einer Domäne, $k = max_{1 \leq i \leq n} |D_i|$
- r die maximale Stelligkeit eines Constraints
- t die maximale Anzahl der Tupel in einem Constraint; $t \leq k^r$

Die Constraints können so gespeichert werden, dass das Finden eines Tupels in Zeit $\log t \leq r \log k \leq n \log k$ möglich ist. Dann

- Eine Variable kann in e Constraints vorkommen, also ist $\textsc{Konsistent?} \in O(e \log t) \leq O(er \log k)$
- $\textsc{Wähle-Wert}$ kann $\textsc{Konsistent?}$ bis zu k mal aufrufen, also ist $\textsc{Wähle-Wert} \in O(ek \log t) \leq O(ekr \log k)$
- Für binäre Constraint-Netzwerke ist $\textsc{Konsistent?} \in O(n)$, $\textsc{Wähle-Wert} \in O(nk)$

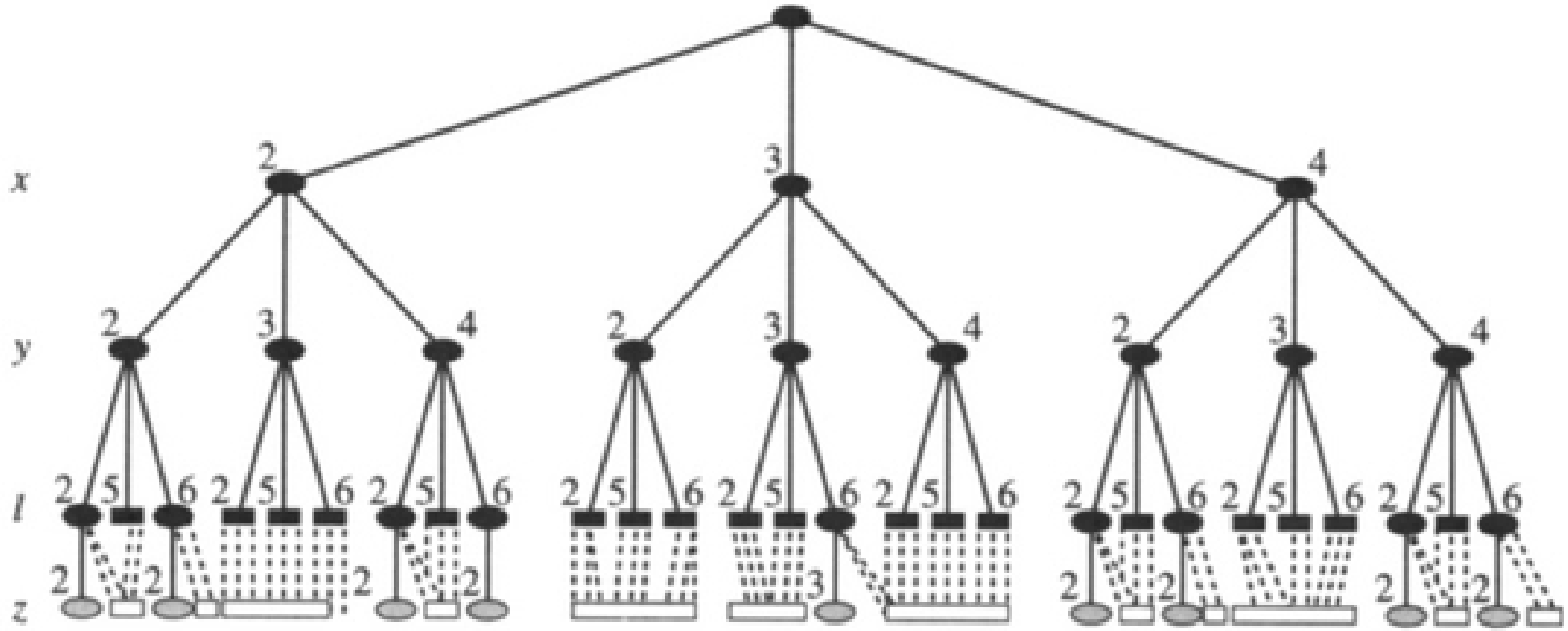
Backtracking: Einfluss der Variablenordnung

$X = \{x, y, l, z\}$
$D_x = \{2, 3, 4\}, D_y = \{2, 3, 4\}, D_l = \{2, 5, 6\}, D_z = \{2, 3, 5\}$
$C_{xz} = (x = 0 \mod z), C_{yz} = (y = 0 \mod z),$
$C_{lz} = (l = 0 \mod z)$



Backtracking: Einfluss der Variablenordnung

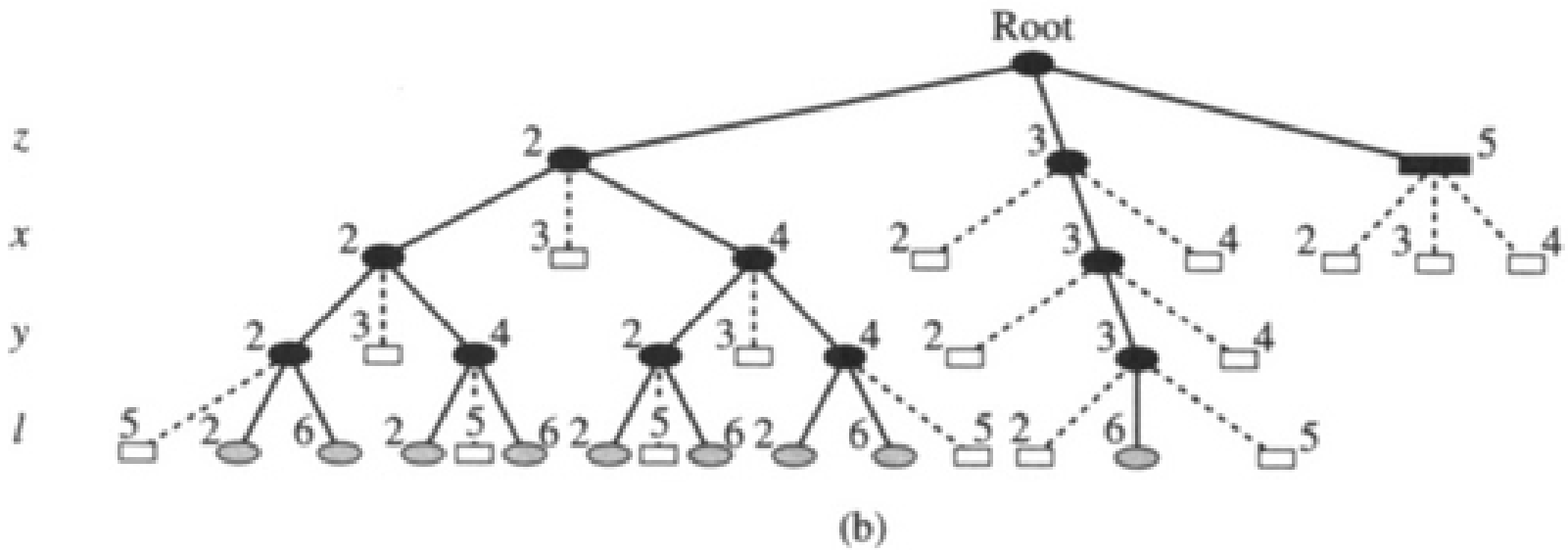
Ordnung der Variablen: x, y, l, z: 48 Suchzustände, 18 Sackgassen



(c)

Backtracking: Einfluss der Variablenordnung

Ordnung der Variablen: z, x, y, l: 20 Suchzustände, 1 Sackgasse



Heuristik: Ordnung der Variablen, Werte

- Fail first: Wähle die Variable, die den Suchraum am meisten einschränkt (z.B. dynamisch die Variable mit der kleinsten Domäne, *min-domain*, *minimum remaining values (MRV)*)
- Wähle die Variable, die in den meisten Constraints vorkommt (*degree heuristic*)
- Suche für diese Variable den Wert aus, der die zukünftige Auswahl am wenigsten beschränkt (*least constraining value*)
- Randomisiertes Backtracking: Wähle Variablen zufällig aus; wenn die Suche nach einer gewissen Zeit kein Ergebnis liefert breche ab und starte mit einer anderen (zufälligen) Variablenordnung neu

Verbesserung des Backtrackings

- Geschickte Auswahl der Variablenordnung
- Reduktion des Verzweigungsfaktors: Propagation als Vorbereitungsschritt
- Vermeiden von „thrashing,“ dem wiederholten Durchsuchen der gleichen partiellen Lösungen
  - Look-Ahead Schemata
  - Look-Back Schemata
- Look-Ahead: Stelle (durch begrenzte Propagation) fest, wie aktuelle Entscheidungen die zukünftige Suche beeinflussen
  - Welche Variable als nächstes instanziiert werden soll
  - Welcher Wert dieser Variablen zugewiesen werden soll
- Look-Back: Steuer das Backtracking-Verhalten:
  - Bis zu welcher Stelle soll beim Backtracking zurückgesprungen werden
  - Lerne Constraints, die vermeiden, dass der aktuelle Konflikt in zukünftigen Suchpfaden wieder auftritt

Propagation

Propagation oder „lokale Konsistenz“ bezeichnet das Entfernen von Werten, die mit den Constraints im Widerspruch stehen, aus den Domänen der Variablen.

Es gibt verschiedene Formen von lokaler Konsistenz:

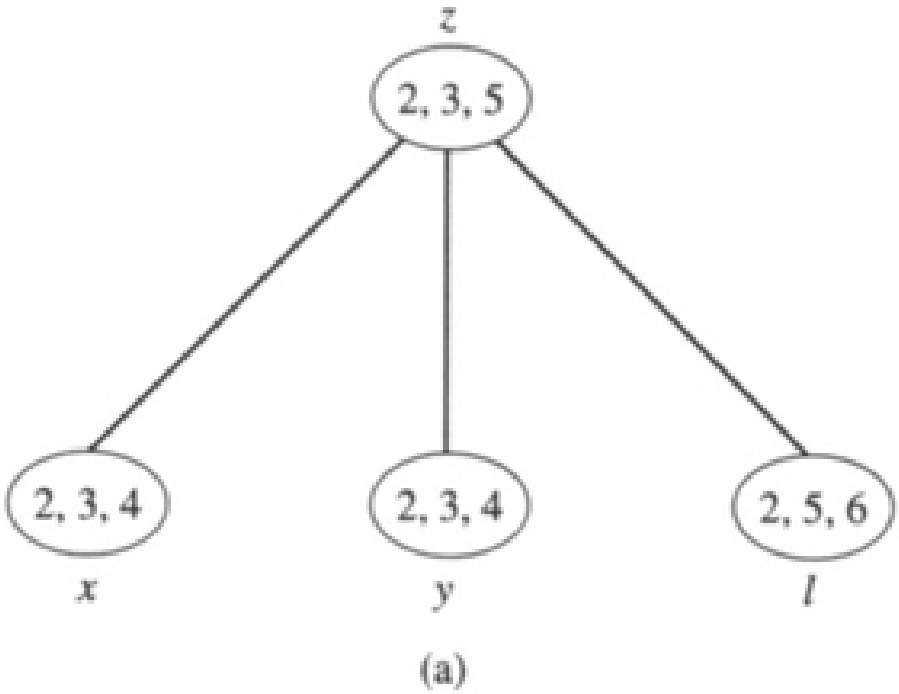
- Knotenkonsistenz, 1-Konsistenz (node consistency): nur Werte, die mit den unären Constraints vereinbar sind werden in den Domains behalten
- Bogenkonsistenz, 2-Konsistenz (arc consistency): Sei x_i eine Variable und x_j eine benachbarte Variable (d.h., es gibt einen Constraint mit Scope $\{x_i, x_j\}$). Zu jedem Wert von x_i gibt es einen Wert im Bereich von x_j, der konsistent ist.
- Pfadkonsistenz, 3-Konsistenz (path consistency): Zu jeder konsistenten Instanziierung von x_i, x_j und jedem zu x_i, x_j benachbarten Knotens x_k gibt es eine konsistente Instanziierung von x_i, x_j, x_k.

(Starke) k-Konsistenz

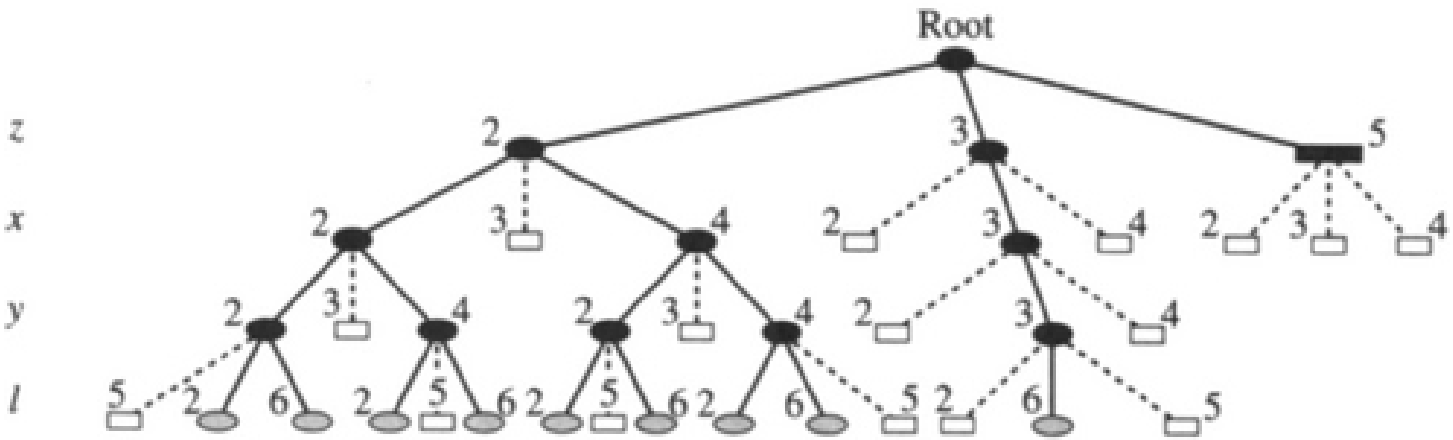
- Ein CSP ist k-konsistent, wenn sich jede konsistente Instanziierung von $k-1$ Variablen zu jeder anderen Variablen fortsetzen lässt.
- Für binäre Netzwerke entspricht 2-Konsistenz der Bogenkonsistenz, 3-Konsistenz der Pfadkonsistenz.
- Ein CSP ist stark k-konsistent, wenn es $1, 2, \ldots, k$-konsistent ist.

Backtracking: Einfluss von Propagation

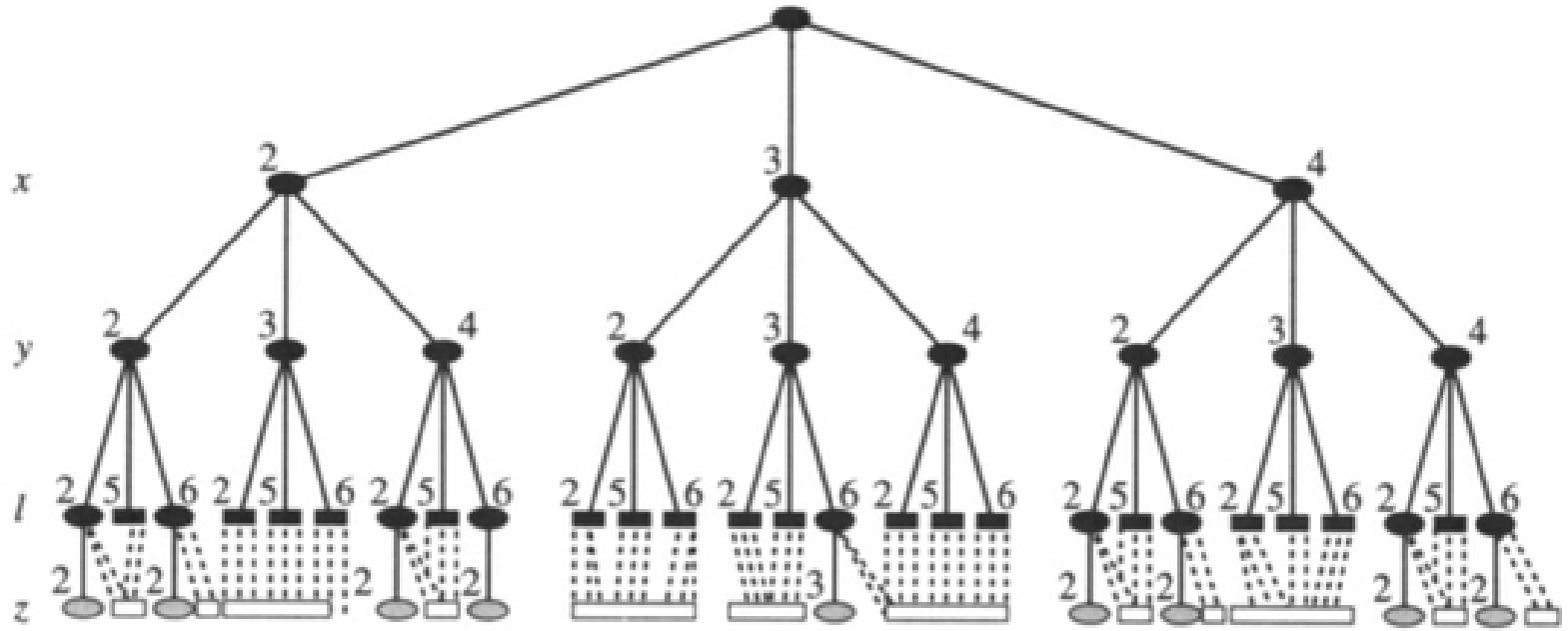
$X = \{x, y, l, z\}$
$D_x = \{2, 3, 4\}, D_y = \{2, 3, 4\}, D_l = \{2, 5, 6\}, D_z = \{2, 3, 5\}$
$C_{xz} = (x = 0 \mod z), C_{yz} = (y = 0 \mod z),$
$C_{lz} = (l = 0 \mod z)$



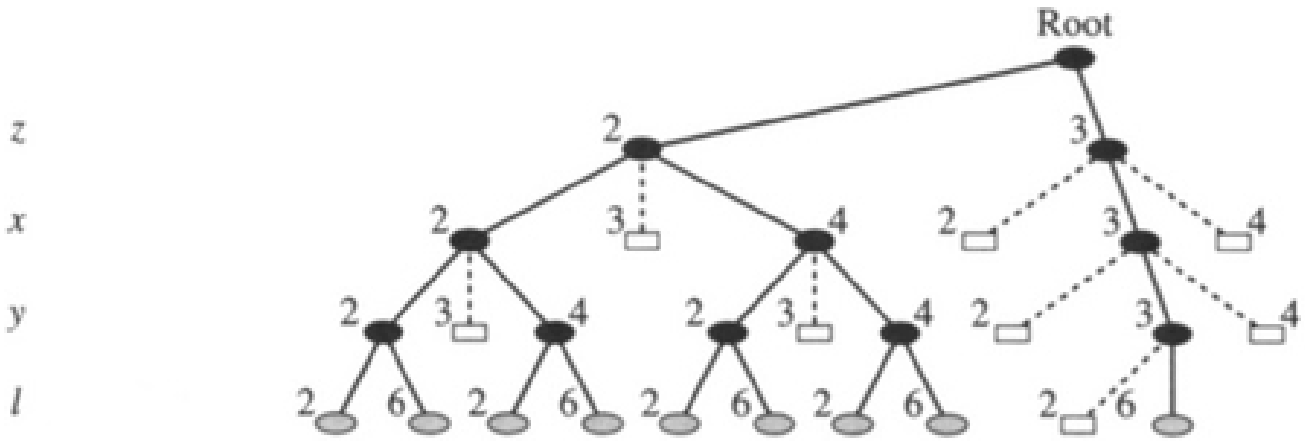
Backtracking: Einfluss von Propagation



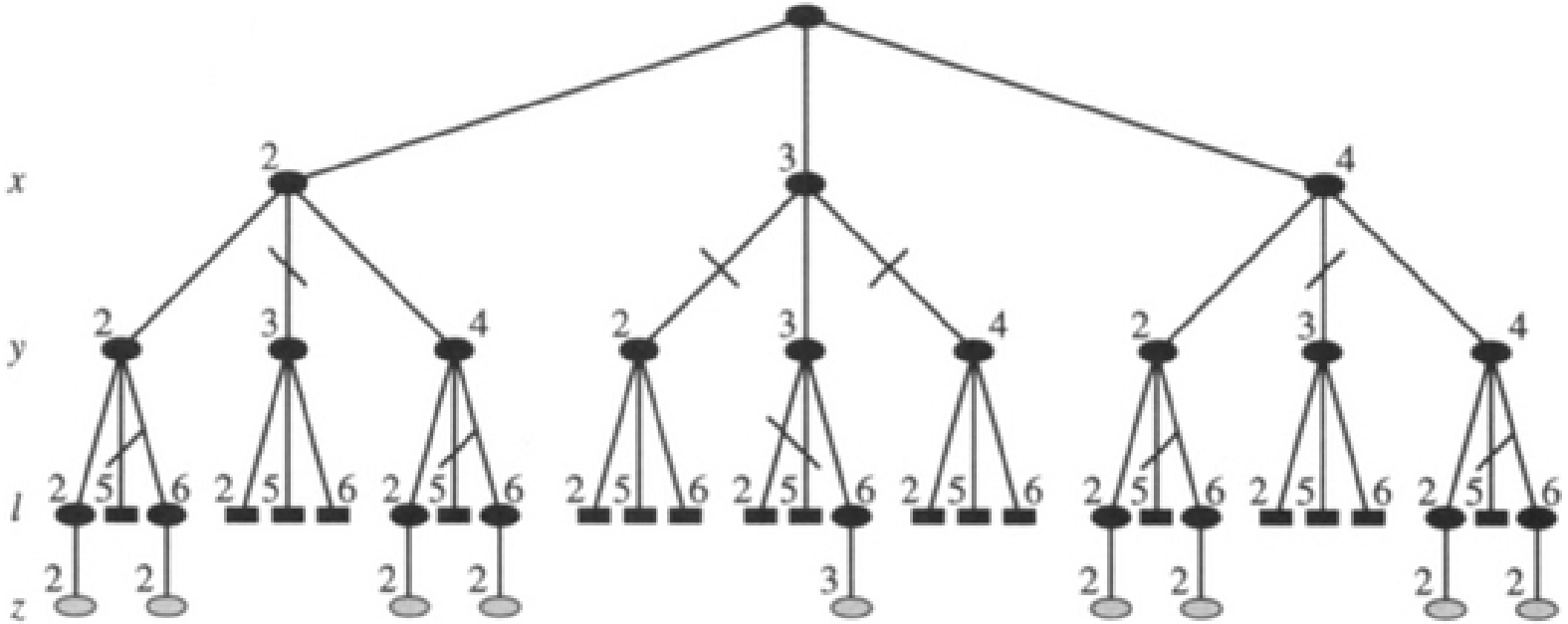
(b)



Backtracking: Einfluss von Propagation



(a)



Backtrack-Freie Netzwerke

Das obere Netzwerk auf der vorhergehenden Folie enthält keine Sackgassen mehr; eine Lösung kann daher immer ohne Backtracking gefunden werden.

Definition

Ein Netzwerk R heißt *backtrack-frei* entlang einer Variablenordnung d wenn jedes Blatt im Backtracking-Suchbaum eine Lösung von R ist.

Bogenkonsistenz

Definition

Sei x_i eine Variable und x_j eine benachbarte Variable (d.h., es gibt einen Constraint C_{ij} mit Scope $\{x_i, x_j\}$). x_i ist bogenkonsistent zu x_j, wenn es zu jedem Wert von x_i einen Wert im Bereich von x_j gibt, der C_{ij} erfüllt.
x_i und x_j sind bogenkonsistent, wenn x_i bogenkonsistent zu x_j und x_j bogenkonsistent zu x_i ist.
Ein binäres Constraint-Netzwerk ist bogenkonsistent, wenn jedes Paar von Variablen bogenkonsistent ist.

Ein binäres Constraint-Netzwerk, das nicht bogenkonsistent ist, kann durch Änderung der Domänen in ein äquivalentes bogenkonsistentes Netzwerk überführt werden.

Bogenkonsistenz: Revision von Domänen

function REVISE(x_i, x_j, D, C)
  $change \leftarrow$ falsch
  for all $a_i \in D_i$ **do**
    if Es gibt kein $a_j \in D_j$ mit $\langle a_i, a_j \rangle \in C_{ij}$ **then**
      $D_i \leftarrow D_i \setminus \{a_i\}$
      $change \leftarrow$ wahr
    end if
  end for
  return *change*
end function

Revise berechnet $D_i \leftarrow D_i \cap (C_{ij} \bowtie D_j)$ und gibt wahr zurück, wenn D_i verändert wurde.

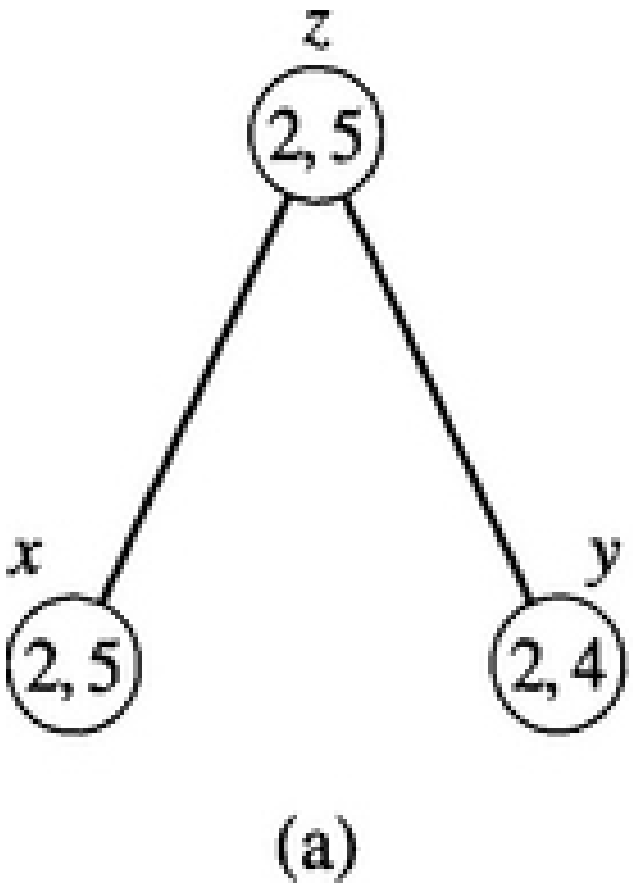
Sei k eine Obergrenze für die Größe der D_i. Die Komplexität von REVISE ist $O(k^2)$.

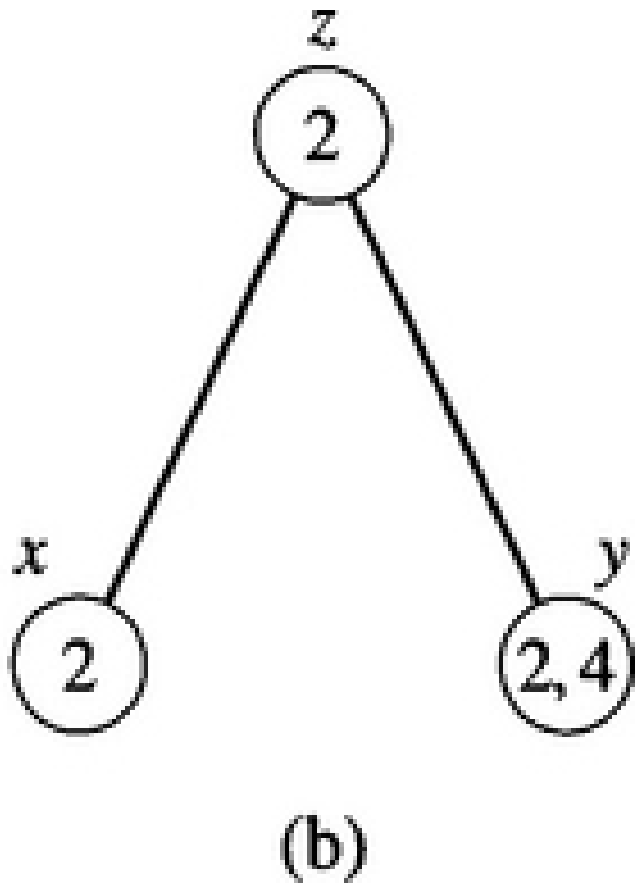
Bogenkonsistenz: AC-1

function AC-1(X, D, C)
  repeat
    dg $\leftarrow$ falsch
    for $\langle x_i, x_j \rangle$ für die C_{ij} existiert **do**
      dg $\leftarrow$ dg $\vee$ Revise(x_i, x_j, D, C)
      dg $\leftarrow$ dg $\vee$ Revise(x_j, x_i, D, C)
    end for
  until dg $=$ falsch
end function

Bogenkonsistenz: Beispiel

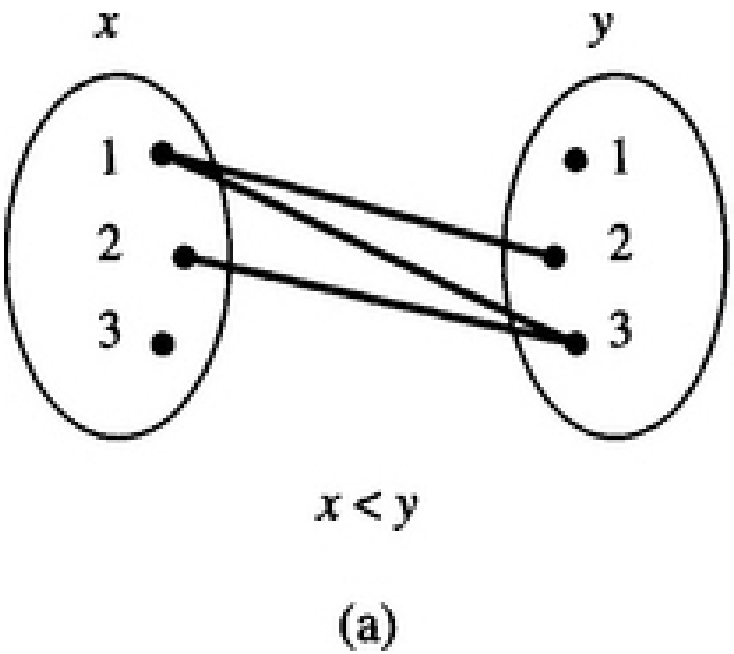
$C_{xz} = (x = 0 \mod z), C_{yz} = (y = 0 \mod z)$

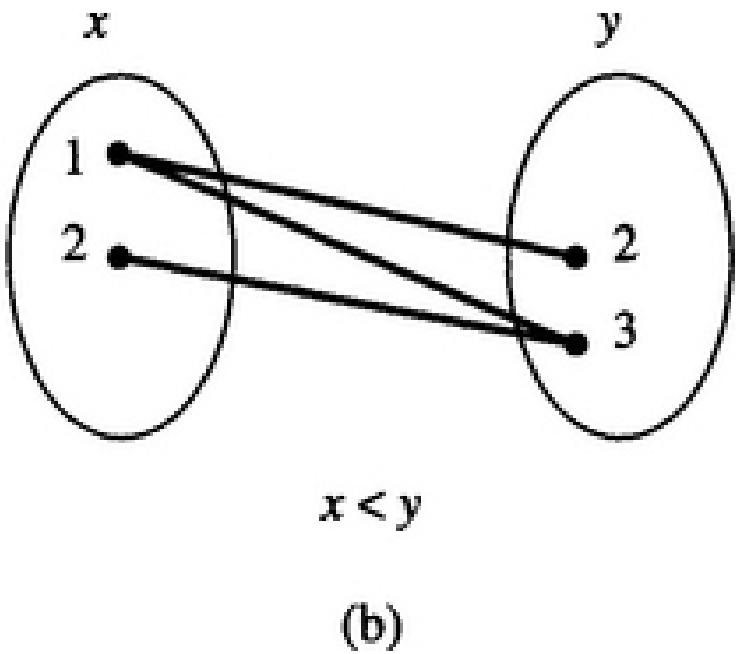




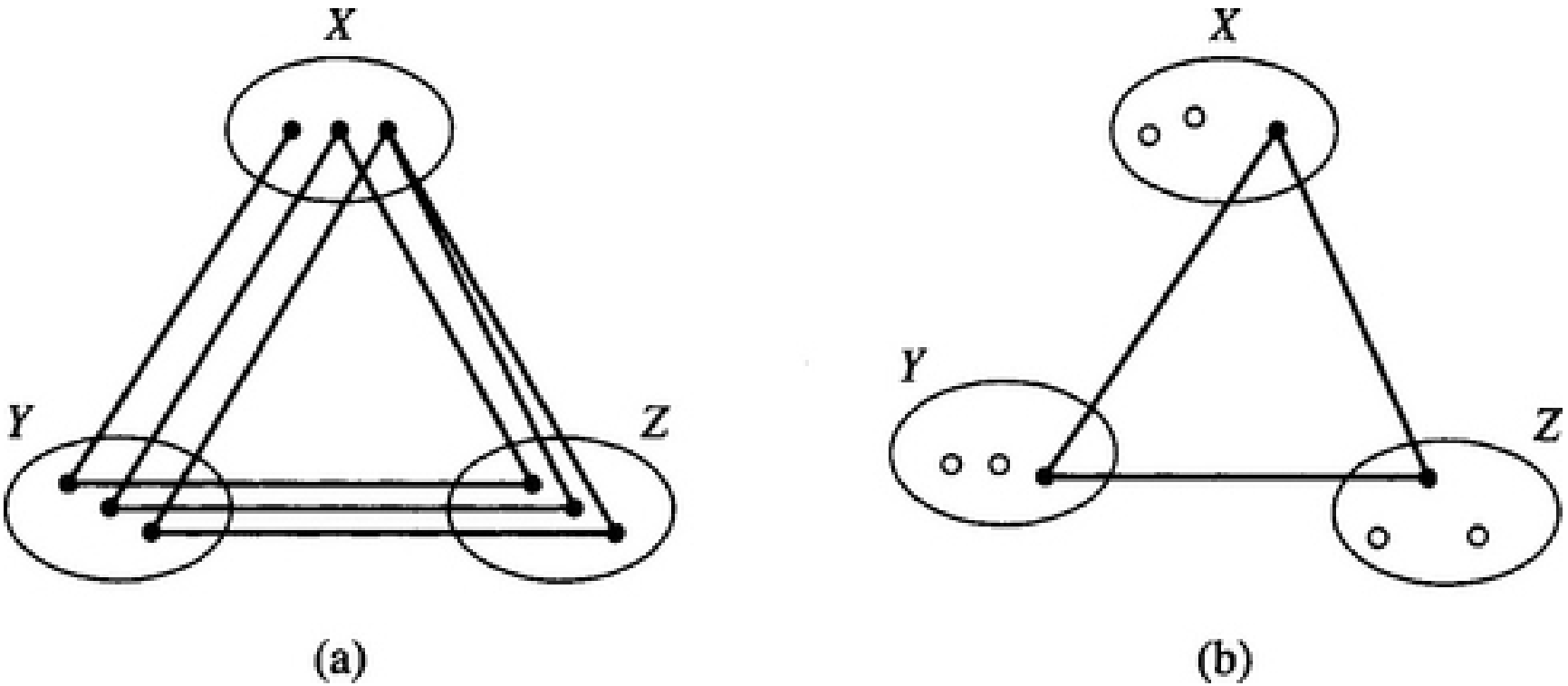
Bogenkonsistenz: Matching-Diagramm

$D_x = D_y = \{1, 2, 3\}, C_{xy} = (x < y)$



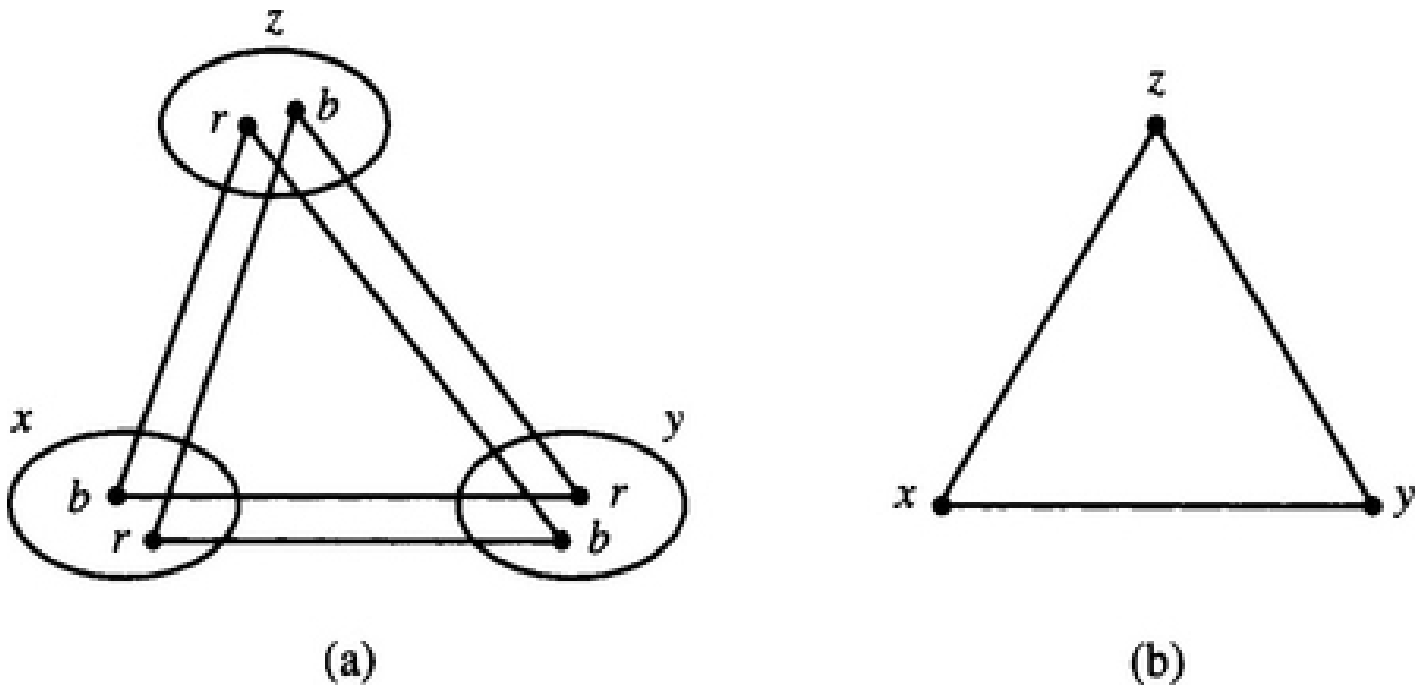


Bogenkonsistenz: Beispiel



Bogenkonsistenz: Beispiel

$D_x = D_y = D_z = \{r, b\}$,
$C_{xy} = (x \neq y)$, $C_{xz} = (x \neq z)$, $C_{yz} = (y \neq z)$,



In diesem Beispiel kann durch Bogenkonsistenz keine Verbesserung erreicht werden, da jeweils 2 Variablen konsistent belegt werden können, und nur die Kombination aus 3 Variablen inkonsistent wird.

AC-3

function $\text{AC-3}(X, D, C)$
  for $\langle x_i, x_j \rangle$ für die C_{ij} existiert **do**
    $queue \leftarrow queue \cup \{(x_i, x_j), (x_j, x_i)\}$
  end for
  while $queue \neq \emptyset$ **do**
    Wähle (x_i, x_j) aus queue
    $queue \leftarrow queue \setminus \{(x_i, x_j)\}$
    if $\text{Revise}(x_i, x_j, C)$ **then**
      $queue \leftarrow queue \cup \{(x_k, x_i) \mid k \neq i, k \neq j\}$
    end if
  end while
end function

Gerichtete Bogenkonsistenz

Definition

Seien $R = (X, D, C)$ ein Constraint-Netzwerk und d eine Ordnung auf X. R heißt gerichtet bogenkonsistent (*directed arc consistent, DAC)*, wenn jedes x_i bogenkonsistent mit jedem $x_j, j > i$ ist.

Ein zyklenfreies, gerichtet bogenkonsistentes Constraint-Netzwerk ist backtrack-frei.

Look-Ahead

Propagation kann während der Backtracking-Suche eingesetzt werden

- Dadurch wird der Suchraum verkleinert, aber der Aufwand für jede Variableninstanziierung steigt
- Der Aufwand für Propagation sollte daher beschränkt werden; stärkere Konsistenz als Bogenkonsistenz wird während der Suche nur in wenigen Fällen verwendet
- Starke Propagation (Pfadkonsistenz, i-Konsistenz) vor der Suche macht Look-Ahead (manchmal) überflüssig, hat aber selber hohe Komplexität
- Look-Ahead kann dynamisch existierende Abhängigkeiten in Betracht ziehen

Die verschiedenen Look-Ahead-Strategien unterscheiden sich durch die verwendete Wähle-Wert Funktion

Verallgemeinertes Look-Ahead

function LOOK-AHEAD(X, D, C, WÄHLE-WERT)
  $D' \leftarrow_C D$, $i \leftarrow 1$
  while $1 \leq i \leq n$ **do**
    $D^i \leftarrow_C D'$ *[Werte von D' vor Instanziierung von x_i]*
    $a_i \leftarrow$ WÄHLE-WERT$(\vec{a}_{i-1}, x_i, X, D', C)$
    if $a_i = null$ **then**
      $i \leftarrow i - 1$, $D'_k \leftarrow D^i_k$ für alle $k > i$
    else
      $i \leftarrow i + 1$
    end if
  end while
  if $i = 0$ **then**
    return inkonsistent
  else
    return $\vec{a}_n$
  end if
end function

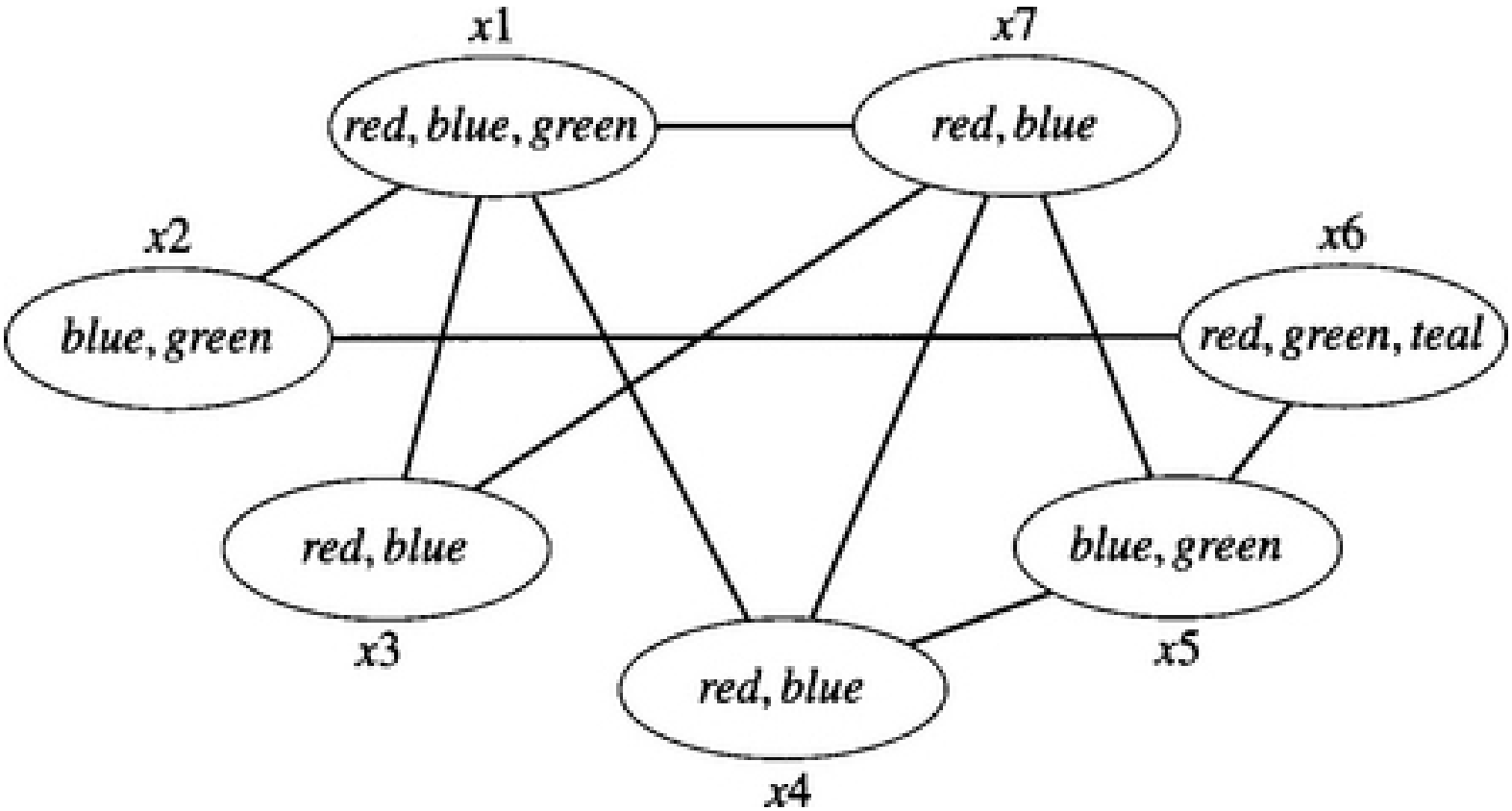
Forward Checking

- Forward Checking ist eine Variante des Look-Aheads, die nur relativ wenig Propagation durchführt
- Die Konsistenz jeder uninstanziierten Variable wird als gesondertes Problem betrachtet: Bei der Instanziierung von x_i werden alle Constraints, deren Scope Variablen aus $\vec{x_i}$ und ein x_k mit $k > i$ enthält, gesondert betrachtet
- Constraints zwischen x_j, x_k mit $j, k > i$ werden nicht propagiert
- Die Propagation wird nur einmal durchgeführt, nicht bis zu einem Fixpunkt

Forward Checking

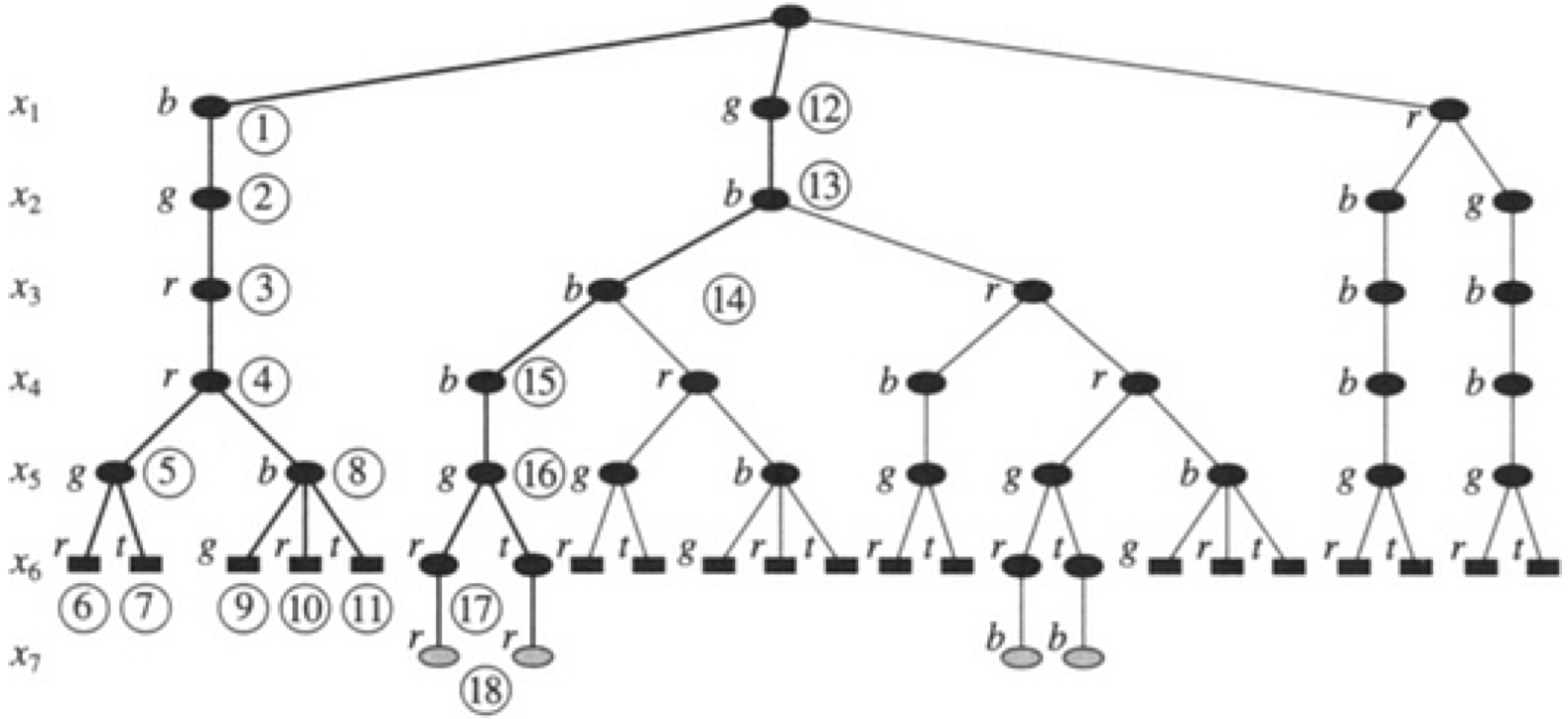
function WÄHLE-WERT-FC($\vec{a}_{i-1}, x_i, X, D', C$)
  $D'' \leftarrow_C D'$
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    for all $k, i < k \leq n$ **do**
      for all $a_k \in D'_k$ **do**
        if $\neg$KONSISTENT?$(\langle \vec{a}_i, x_k = a_k \rangle, C)$ **then**
          $D'_k \leftarrow D'_k \setminus \{a_k\}$
        end if
      end for
      if $D'_k = \emptyset$ **then**
        $D'_k \leftarrow D''_k$ für alle $k > i$
      else
        return a_i
      end if
    end for
  end while
  return *null*
end function

Beispiel


x1
red, blue, green
x7
red, blue
x2
blue, green
x6
red, green, teal
red, blue
x3
blue, green
x5
red, blue
x4

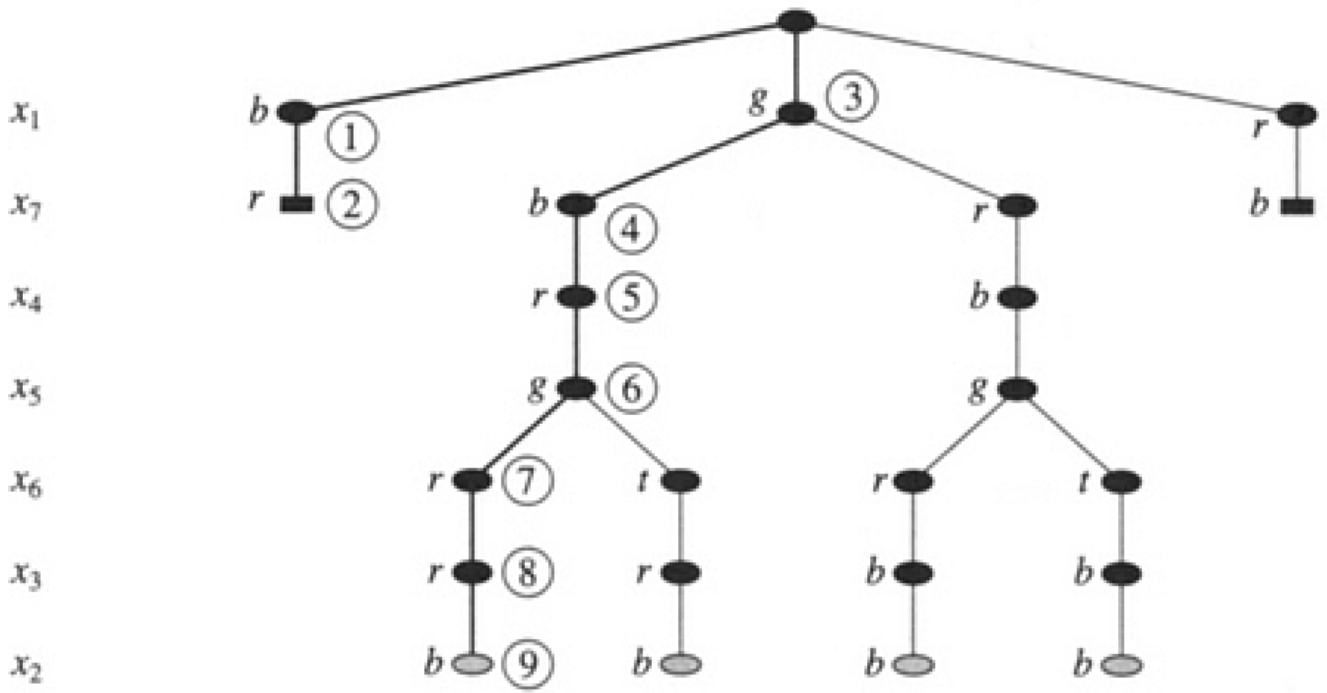
Beispiel

Suche mit Variablenordnung $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$

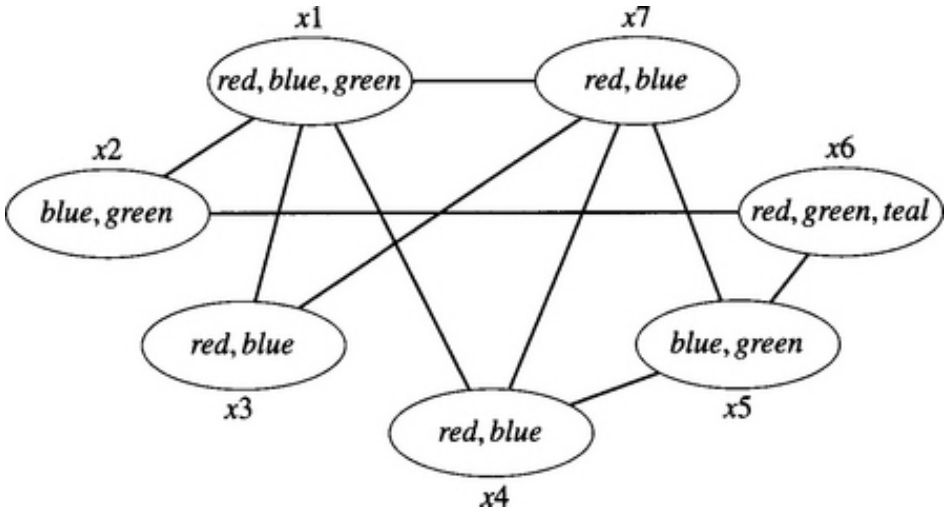


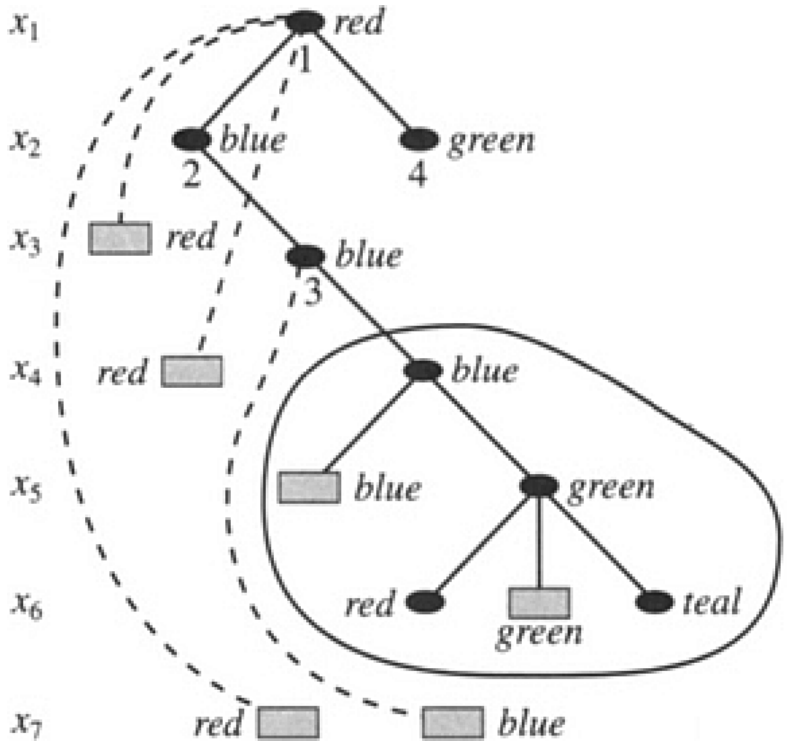
Beispiel

Suche mit Variablenordnung $\{x_1, x_7, x_4, x_5, x_6, x_3, x_2\}$



Beispiel


x1
red, blue, green
x7
red, blue
x2
blue, green
x6
red, green, teal
red, blue
x3
red, blue
x4
blue, green
x5


x_1
red
1
x_2
blue
2
green
4
x_3
red
blue
3
x_4
red
blue
x_5
blue
green
x_6
red
green
teal
x_7
red
blue

AC-Suche

function WÄHLE-WERT-AC($\vec{a}_{i-1}, x_i, X, D', C$)
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    repeat
      wert-entfernt? $\leftarrow$ falsch
      for all j, k mit $i < j \leq n, i < k \leq n, k \neq j$ **do**
        for all $a_j \in D'_j$ **do**
          if $\neg\exists a_k \in D'_k$ mit KONSISTENT?$(\langle \vec{a}_i, a_j, a_k \rangle, C)$ **then**
            $D'_j \leftarrow D'_j \setminus \{a_j\}$
            wert-entfernt? $\leftarrow$ wahr
          end if
        end for
      end for
    until wert-entfernt? $=$ falsch
    if irgendein $D'_k = \emptyset$ **then** $D'_k \leftarrow D''_k$ für alle $k > i$
    else return a_i
    end if
  end while
  return *null*
end function

Full Look-Ahead

function WÄHLE-WERT-FLA($\vec{a}_{i-1}, x_i, X, D', C$)
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    for all j, k mit $i < j \leq n, i < k \leq n, k \neq j$ **do**
      for all $a_j \in D'_j$ **do**
        if $\neg\exists a_k \in D'_k$ mit KONSISTENT?$(\langle \vec{a}_i, a_j, a_k \rangle, C)$ **then**
          $D'_j \leftarrow D'_j \setminus \{a_j\}$
        end if
      end for
    end for
    if irgendein $D'_k = \emptyset$ **then** $D'_k \leftarrow D''_k$ für alle $k > i$
    else return a_i
    end if
  end while
  return *null*
end function

Partial Look-Ahead

function WÄHLE-WERT-FLA($\vec{a}_{i-1}, x_i, X, D', C$)
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    for all j, k mit $i < j \leq n, j < k \leq n$ **do**
      for all $a_j \in D'_j$ **do**
        if $\neg\exists a_k \in D'_k$ mit KONSISTENT?$(\langle \vec{a}_i, a_j, a_k \rangle, C)$ **then**
          $D'_j \leftarrow D'_j \setminus \{a_j\}$
        end if
      end for
    end for
    if irgendein $D'_k = \emptyset$ **then** $D'_k \leftarrow D''_k$ für alle $k > i$
    else return a_i
    end if
  end while
  return *null*
end function

Look Ahead: Variablenordnung

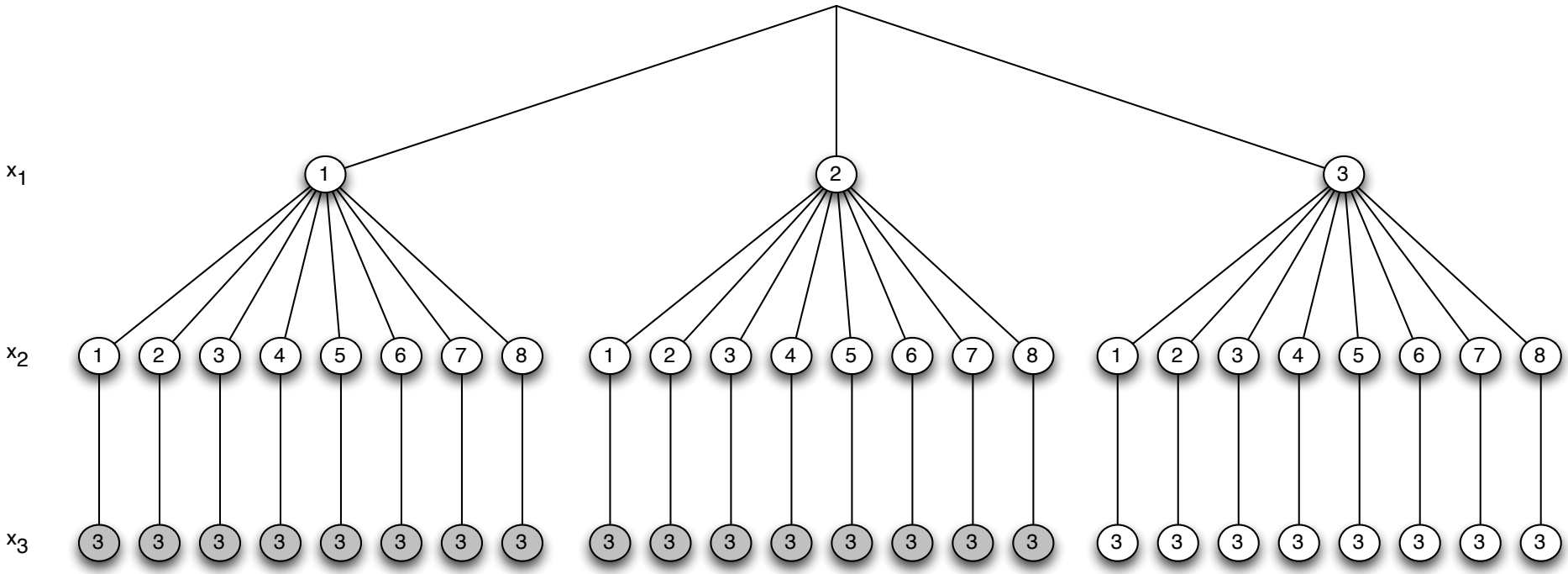
Die durch Look Ahead gewonnene Information kann zur dynamischen Auswahl von Variablenordnungen oder zur Verbesserung der Werteauswahl-Heuristiken verwendet werden: z.B. Dynamic Variable Forward Checking: Ordne die uninstanziierten Variablen nach jedem Look-Ahead-Schritt so, dass die Variable mit der kleinsten Domäne als nächstes instanziiert wird.

Backjumping

- Der Backtracking-Algorithmus geht in der Rückwärtsrichtung immer einen Schritt zurück
- Das ist oft nicht der Schritt in dem das Problem aufgetreten ist
- Idee: Finde den zuletzt zugewiesenen Wert einer Variable, der für die Inkonsistenz verantwortlich ist und springe direkt zu dieser Stelle zurück.

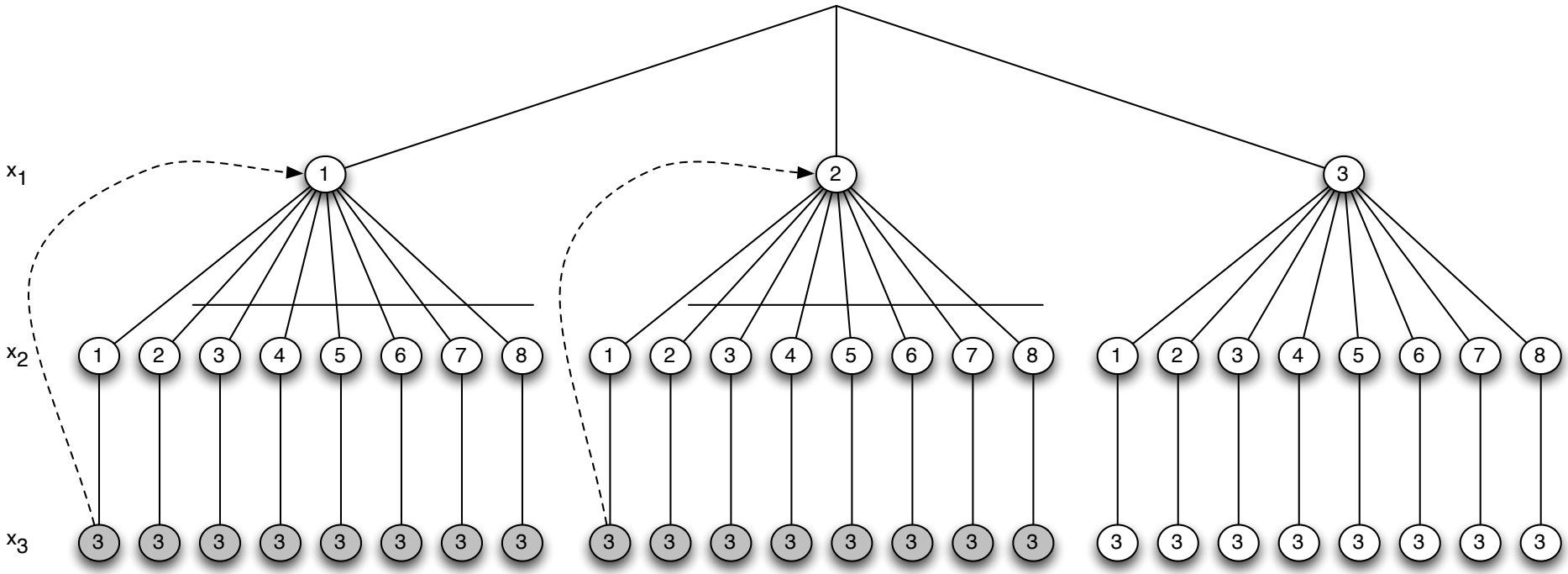
Beispiel

$X = \{x_1, x_2, x_3\}$
$D_1 = \{1, 2, 3\}, D_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}, D_3 = \{3\}$
$C_{13} : x_1 = x_3$



Backjumping

$X = \{x_1, x_2, x_3\}$
$D_1 = \{1, 2, 3\}, D_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}, D_3 = \{3\}$
$C_{13} : x_1 = x_3$



Backjumping

Eine Instanziierung $\vec{a}_i$ heißt *Sackgasse* (*dead end state*) (auf Ebene i), wenn sie mit jedem möglichen Wert von x_{i+1} inkonsistent ist. x_{i+1} heißt dann *dead-end Variable.*

Sei $\vec{a} = a_{i_1}, \ldots, a_{i_k}$ eine konsistente Instanziierung, x eine uninstanziierte Variable. Wenn es keinen Wert $b \in D_x$ gibt, der mit $\vec{a}$ konsistent ist, dann heißt $\vec{a}$ eine *Konfliktmenge* (conflict set) zu x; man sagt $\vec{a}$ steht im Konflikt mit x.

Steht kein Subtupel von $\vec{a}$ im Konflikt mit x so heißt $\vec{a}$ eine *minimale Konfliktmenge* zu x.

No-Good

Definition

Sei (X, D, C) ein Constraint-Netzwerk. Eine partielle Instanziierung $\vec{a}$, die in keiner Lösung von (X, D, C) vorkommt heißt *No-Good* oder *Nogood*.

Eine Konfliktmenge ist immer ein Nogood, aber es gibt auch Nogoods, die keine Konfliktmengen sind.

Backjumping

Backjumping versucht immer, wenn bei der Suche eine Sackgasse auftritt so weit wie möglich zurückzuspringen ohne dabei Lösungen zu verlieren.

Definition

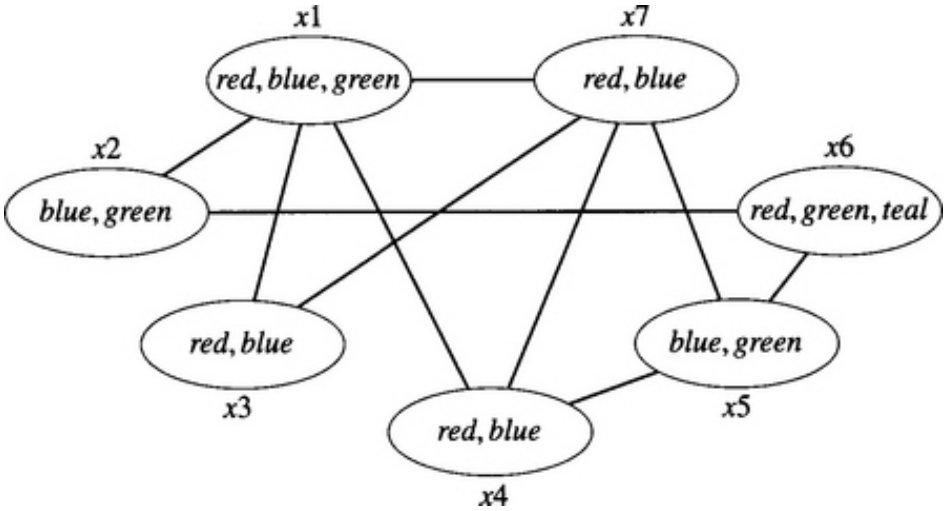
Sei $\vec{a}_i$ ein Blatt des Suchbaumes, das eine Sackgasse ist. a_j mit $j \leq i$ heißt *sicher* (*safe*), wenn $\vec{a}_j$ ein Nogood ist, d.h., wenn $\vec{a}_j$ nicht zu einer Lösung erweitert werden kann.

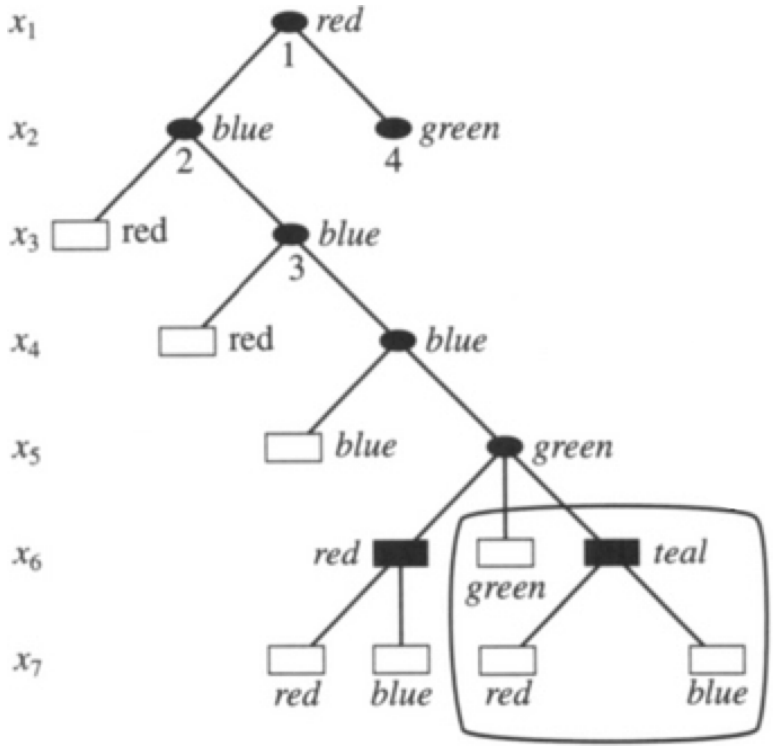
Was „so weit wie möglich“ bedeutet hängt von der Information ab, die der Algorithmus mitführt

Arten des Backjumpings

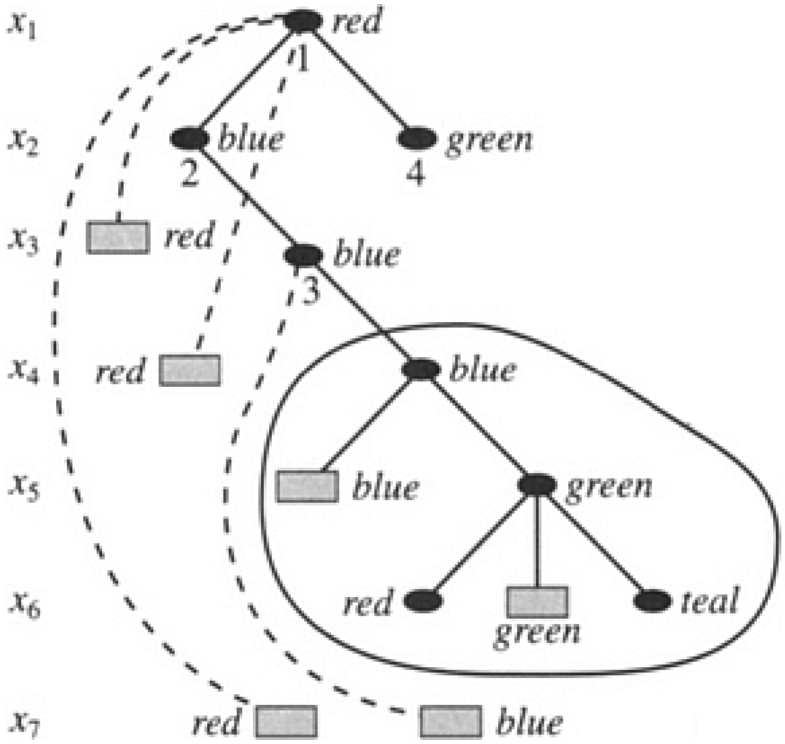
- Gaschnigs Backjumping: Verwendet Information aus der aktuellen Belegung um zurückzuspringen, springt aber nur an Blättern des Suchbaumes
- Graphenbasiertes Backjumping: Verwendet die Struktur des Constraintgraphen, um auch an inneren Sackgassen zurückzuspringen, beachtet aber die aktuelle Belegung der Variablen nicht
- Konfliktgesteuertes Backjumping: Verbindet Gaschnigs und graphenbasiertes Backjumping; springt an Blättern und inneren Knoten zurück und verwendet die aktuelle Belegung der Variablen um Konflikte zu identifizieren

Gaschnigs Backjumping

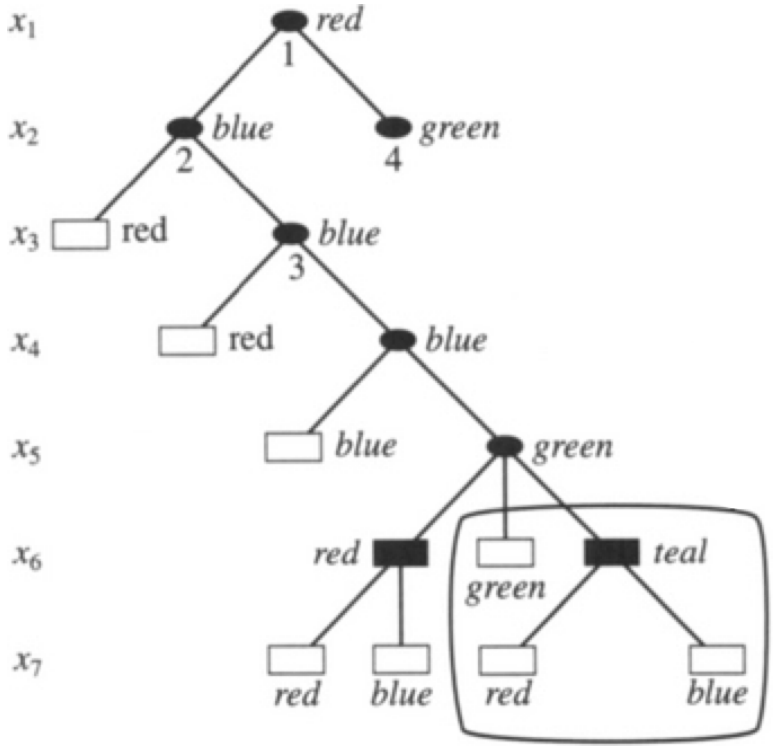




FC versus Gaschnigs Backjumping



Forward Checking



Gaschnigs Backjumping

Software-Entwicklung mit Constraints

Einsatz von Constraints

- Constraint-Solver als eigenständiges Programm
- Einbettung eines Constraint-Solvers als Bibliothek
- Constraint-Programmiersprachen
  - Constraint-Logische Programmiersprachen (CLP)
  - Constraint-Funktionale Programmiersprachen

MiniZinc

- Sprache zur Beschreibung von (reinen) Constraintproblemen
- Kann mit vielen Constraint-Solvern verwendet werden
- Damit sowohl Einbettung als Bibliothek als auch Integration in CLP-Systeme wie Eclipse möglich

Beispiel: Karte von Australien

Beispiel: Karte von Australien

- Färben der Karte durch drei Farben: R, G, B
- Benachbarte Regionen dürfen nicht die gleiche Farbe bekommen

Beispiel: Karte von Australien

- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{R, G, B\}$
- $C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW,$
  $SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$

Beispiel: Karte von Australien

- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{1, 2, 3\}$
- $C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW,$
  $SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$

Beispiel: Karte von Australien

$$X = \{WA, NT, Q, NSW, V, SA, T\}$$
$$D_i = \{1, 2, 3\}$$

```
% Number of Colors
int: nc = 3;

% Constraint Variables
var 1..nc: wa;   var 1..nc: nt;   var 1..nc: q;
var 1..nc: nsw;  var 1..nc: v;    var 1..nc: sa;
var 1..nc: t;
```

Beispiel: Karte von Australien

$$C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW, \\ SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

```
constraint wa  != nt;
constraint wa  != sa;
constraint sa  != v;
constraint sa  != nsw;
constraint sa  != q;
constraint sa  != nt;
constraint nt  != q;
constraint q   != nsw;
constraint nsw != v;
```

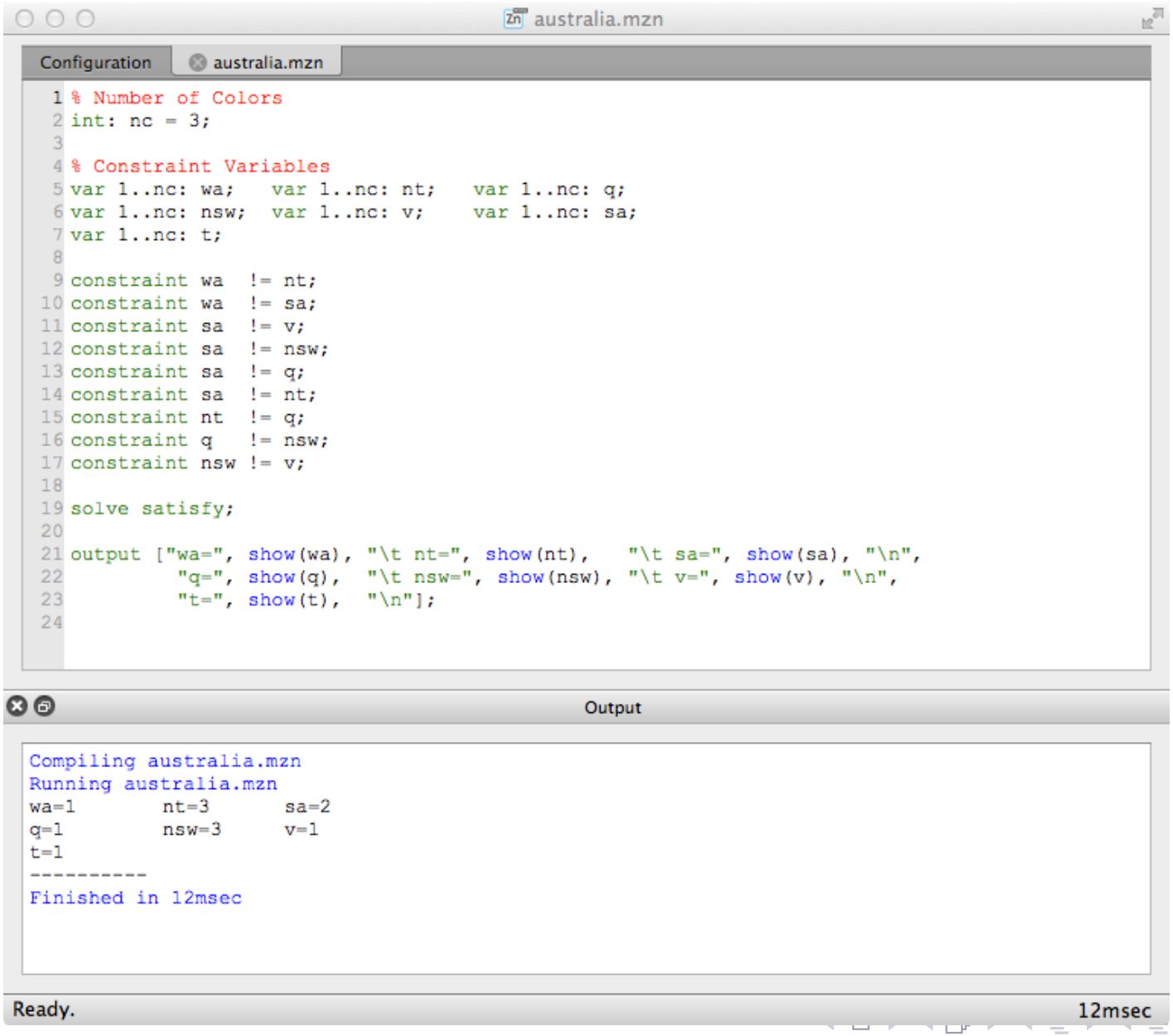
Beispiel: Karte von Australien

```
solve satisfy;

output ["wa=", show(wa), "\t nt=", show(nt),
        "\t sa=", show(sa), "\n",
        "q=", show(q),  "\t nsw=", show(nsw),
        "\t v=", show(v), "\n",
        "t=", show(t),  "\n"];
```

MiniZinc

```
Silverbird:Code$ mzn-g12fd australia.mzn
wa=1      nt=3      sa=2
q=1       nsw=3     v=1
t=1
----------
```

MiniZinc


australia.mzn
Configuration
australia.mzn
1 % Number of Colors
2 int: nc = 3;
3
4 % Constraint Variables
5 var 1..nc: wa; var 1..nc: nt; var 1..nc: q;
6 var 1..nc: nsw; var 1..nc: v; var 1..nc: sa;
7 var 1..nc: t;
8
9 constraint wa != nt;
10 constraint wa != sa;
11 constraint sa != v;
12 constraint sa != nsw;
13 constraint sa != q;
14 constraint sa != nt;
15 constraint nt != q;
16 constraint q != nsw;
17 constraint nsw != v;
18
19 solve satisfy;
20
21 output ["wa=", show(wa), "\t nt=", show(nt), "\t sa=", show(sa), "\n",
22 "q=", show(q), "\t nsw=", show(nsw), "\t v=", show(v), "\n",
23 "t=", show(t), "\n"];
24
Output
Compiling australia.mzn
Running australia.mzn
wa=1 nt=3 sa=2
q=1 nsw=3 v=1
t=1

Finished in 12msec
Ready.
12msec

Beispiel: Optimierung

- Optimales Ausnutzen von Resourcen beim Backen von Kuchen
- Zwei Rezepte stehen zur Auswahl
  - Bananenkuchen (250g Mehl, 2 Bananen, 75g Zucker, 100g Butter), Preis 4 Euro
  - Schokoladenkuchen (200g Mehl, 75g Kakao, 150g Zucker, 150g Butter), Preis 4,50 Euro
- Vorhandene Rohstoffe:
  - 4kg Mehl
  - 2kg Zucker
  - 500g Butter
  - 6 Bananen
  - 500g Kakao
- Ziel: Maximieren des Gewinns

Beispiel: Optimierung

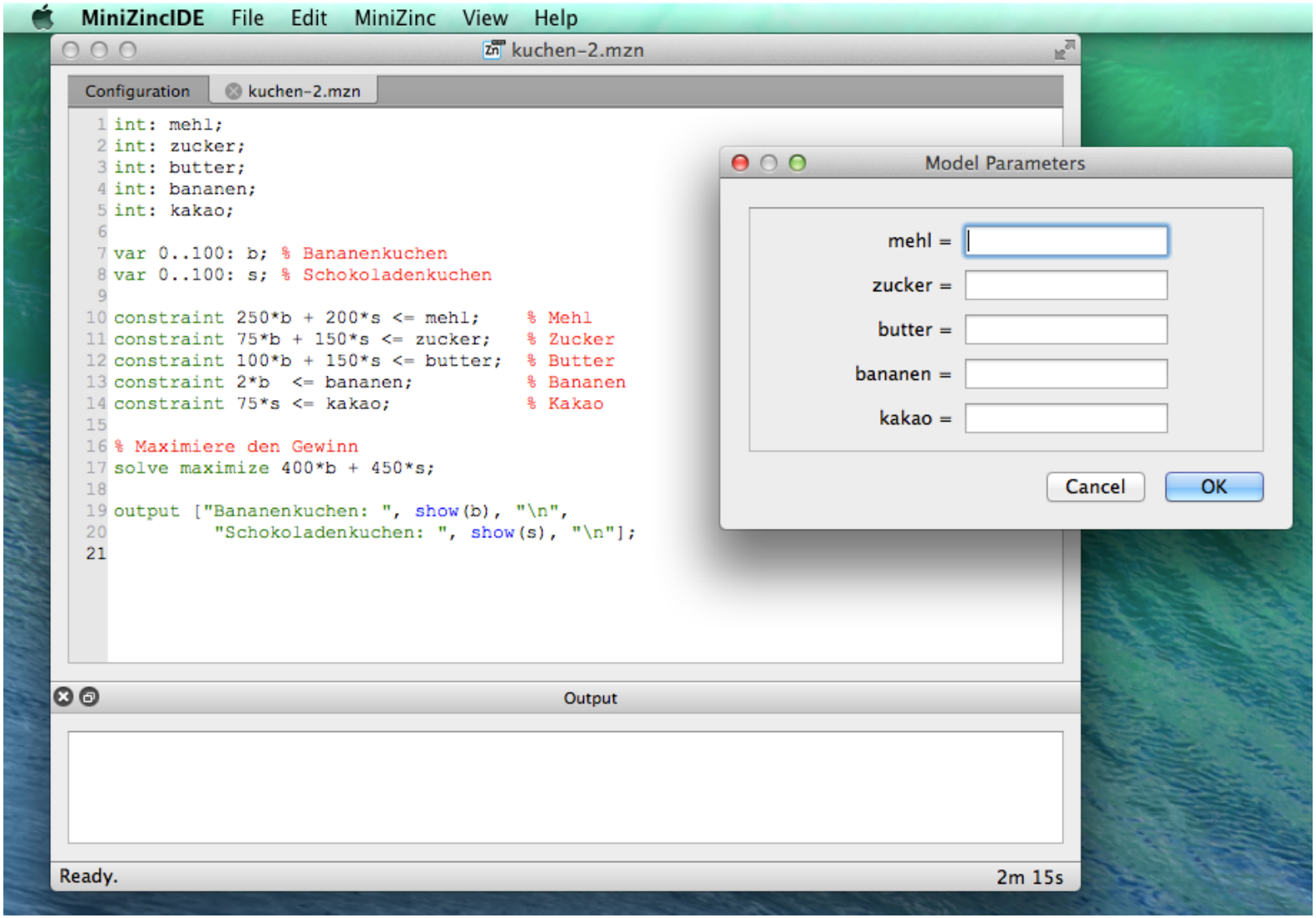
Seien b die Anzahl der Bananenkuchen und s die Anzahl der Schokoladenkuchen:

$$250b + 200s \leq 4000 \quad \text{(Mehl)}$$
$$75b + 150s \leq 2000 \quad \text{(Zucker)}$$
$$100b + 150s \leq 500 \quad \text{(Butter)}$$
$$2b \leq 6 \quad \text{(Bananen)}$$
$$75s \leq 500 \quad \text{(Kakao)}$$

Beispiel: Optimierung

```
var 0..100: b; % Bananenkuchen
var 0..100: s; % Schokoladenkuchen

constraint 250*b + 200*s <= 4000;    % Mehl
constraint 75*b + 150*s <= 2000;     % Zucker
constraint 100*b + 150*s <= 500;     % Butter
constraint 2*b  <= 6;                % Bananen
constraint 75*s <= 500;              % Kakao

% Maximiere den Gewinn
solve maximize 400*b + 450*s;

output ["Bananenkuchen: ", show(b), "\n",
        "Schokoladenkuchen: ", show(s), "\n"];
```

Beispiel: Optimierung



Beispiel: Optimierung

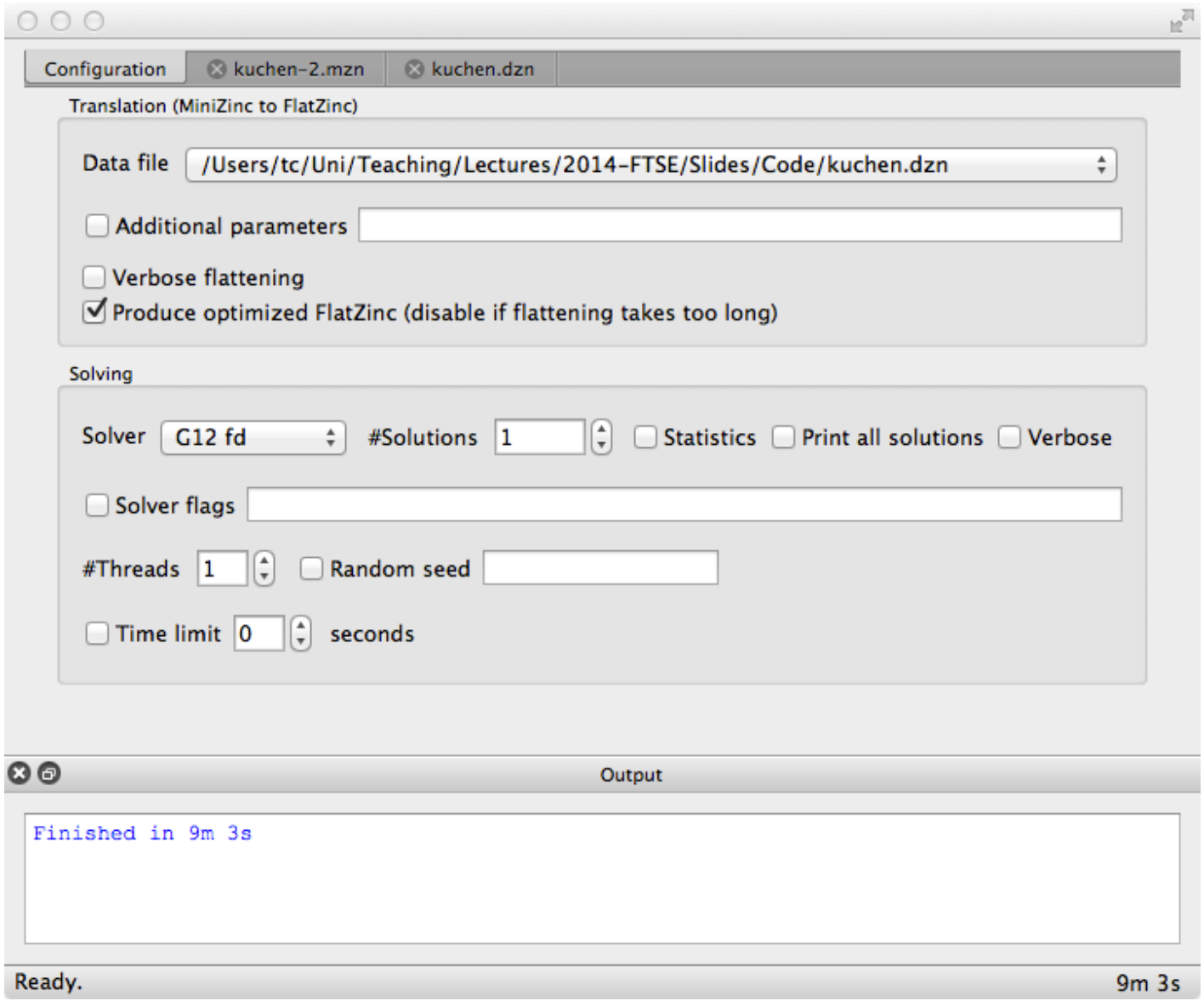
```
int: mehl;
int: zucker;
int: butter;
int: bananen;
int: kakao;

var 0..100: b; % Bananenkuchen
var 0..100: s; % Schokoladenkuchen

constraint 250*b + 200*s <= mehl;     % Mehl
constraint 75*b + 150*s <= zucker;    % Zucker
constraint 100*b + 150*s <= butter;   % Butter
constraint 2*b  <= bananen;           % Bananen
constraint 75*s <= kakao;             % Kakao

...
```

Beispiel: Optimierung

Datei `kuchen.dzn`

```
mehl    = 4000;
zucker  = 2000;
butter  = 500;
bananen = 6;
kakao   = 500;
```

Beispiel: Optimierung



Beispiel: Optimierung

```
int: mehl;
int: zucker;
...

var 0..100: b; % Bananenkuchen
var 0..100: s; % Schokoladenkuchen

constraint assert(mehl >= 0,
  "Menge Mehl muss positiv sein.");
constraint assert(zucker >= 0,
  "Menge Zucker muss positiv sein.");
constraint assert(butter >= 0,
  "Menge Butter muss positiv sein.");

...
```

MiniZinc Programme

- `include` Anweisungen
- Deklarationen von Parametern und Variablen: *typeinst* : *var* [=*expr*]
- Zuweisungen: *var* = *expr*
  (Zuweisungen an Domainvariablen entsprechen Constraints)
- Constraints: `constraint` *boolexpr*
- Beschreibung der gesuchten Lösung:
  - `solve satisfy`
  - `solve maximize` *arithexpr*
  - `solve minimize` *arithexpr*
- Ausgabeanweisung: `output`[*stringexpr*, . . .]

Typen

- Basistypen: `float`, `int`, `bool`
- Operationen: `+`, `-`, `*`, `/`, `abs`, `sqrt`, `ln`, `log2`; `+`, `-`, `*`, `div`, `mod`, ...
- Für Parameter auch: `string`, `ann` (Annotationen)
- Arrays: `array [`*index*`, ...] of var float:` *array*
- Operationen: `[]`, `++`
- Mengen: `set of float:` *myset*
- Operationen: `in`, `subset`, `superset`, `union`, `inter`, `diff`, `symdiff`, `card`

Finite Elemente Modellierung

- Modellierung des Temperaturgradienten in einer rechteckigen Fläche
- Laplace'sche Bedingung: Im Gleichgewichtszustand ist die Temperatur jedes Elements der Durchschnitt aus den orthogonalen Nachbarn

Finite Elemente Modellierung

```
int: w = 4;
int: h = 4;

array[0..w,0..h] of var float: t; % Temperatur im Punkt (i,j)
var float: left;   % linke Kantentemperatur
var float: right;  % rechte Kantentemperatur
var float: top;    % obere Kantentemperatur
var float: bottom; % untere Kantentemperatur

% Laplace-Gleichung
constraint forall(i in 1..w-1, j in 1..h-1)(
  4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
```

Finite Elemente Modellierung

```
% Kanten-Constraints
constraint forall(i in 1..w-1)(t[i,0] = left);
constraint forall(i in 1..w-1)(t[i,h] = right);
constraint forall(j in 1..h-1)(t[0,j] = top);
constraint forall(j in 1..h-1)(t[w,j] = bottom);

% Eck-Constraints
constraint t[0,0]=0.0 /\ t[0,h]=0.0 /\ t[w,0]=0.0 /\ t[w,h]=0.

left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;
output [...];
```

Komprehensionen und Aggregatfunktionen

- Array-Komprehension: [*expr* | *gens*] konstruiert Elemente entsprechend *expr* aus den Generatoren *gens*.
- Beispiel: `[i * i | i in 1..9]` ist das Array mit den Quadraten $1^2, \ldots 9^2$
- Mengen-Komprehensionen: {*expr* | *gens*}
- Aggregatfunktionen berechnen einen Wert aus Array/Menge
- Arithmetisch: `sum`, `product`, `min`, `max`
- Boole'sch: `forall`, `exists`, `xorall`, `iffall`
- Statt *agg-func*([*expr* | *gens*]) kann man auch schreiben *agg-func*(*gens*)(*expr*)

Beispiel: Allgemeine Resourcenoptimierung

```
% Number of different products
int: nproducts;
set of int: Products = 1..nproducts;

% Profit per unit for each product
array[Products] of int: profit;
array[Products] of string: pname;

% Number of resources
int: nresources;
set of int: Resources = 1..nresources;

% Amount of each resource available
array[Resources] of int: capacity;
array[Resources] of string: rname;
```

Beispiel: Allgemeine Resourcenoptimierung

```
% Units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;
constraint assert(forall (r in Resources, p in Products)
          (consumption[p,r] >= 0), "Error: negative consumpti

% Bound on number of Products
int: mproducts = max (p in Products)
                    (min (r in Resources where consumption[p,
                         (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;
```

Beispiel: Allgemeine Resourcenoptimierung

```
% Production cannot use more than the available Resources:
constraint forall (r in Resources) (
      used[r] = sum (p in Products)(consumption[p, r] * produc
      /\ used[r] <= capacity[r]
);

% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);

output [ show(pname[p]) ++ " = " ++ show(produce[p]) ++ ";\n"
         p in Products ] ++
       [ show(rname[r]) ++ " = " ++ show(used[r]) ++ ";\n" |
         r in Resources ];
```

Beispiel: Allgemeine Resourcenoptimierung

```
% Data file for simple production planning model

nproducts = 2;        %banana cakes and chocolate cakes
profit = [400, 450]; %in cents
pname = ["banana-cake", "chocolate-cake"];

nresources = 5;       %flour, banana, sugar, butter, cocoa
capacity = [4000, 6, 2000, 500, 500];
rname = ["flour","banana","sugar","butter","cocoa"];

consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```

Beispiel: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Sudoku

- 81 Quadrate: $A1$–$I9$
- Jedes soll mit Ziffern 1–9 belegt werden
- Einige Quadrate schon ausgefüllt
- Keine Ziffer kommt zweimal in einer Zeile, Spalte oder 3×3 Box vor

Sudoku Constraints

$$\text{AllDifferent}(A1, A2, \ldots, A9)$$
$$\text{AllDifferent}(B1, B2, \ldots, B9)$$
$$\ldots$$
$$\text{AllDifferent}(A1, B1, \ldots, I1)$$
$$\text{AllDifferent}(A2, B2, \ldots, I2)$$
$$\ldots$$
$$\text{AllDifferent}(A1, A2, A3, B1, B2, B3, C1, C2, C3)$$
$$\text{AllDifferent}(A4, A5, A6, B4, B5, B6, C4, C5, C6)$$
$$\ldots$$

Sudoku (1)

```
include "alldifferent.mzn";

int: S = 3;
int: N = S * S;

set of int: SubSquareRange = 1..S;
set of int: BoardRange = 1..N;

array[1..N,1..N] of 0..N: initial;
array[1..N,1..N] of var BoardRange: sudoku;

% Constrain initial values
constraint forall (i,j in BoardRange)(
  if initial[i,j] > 0
  then sudoku[i,j] = initial[i,j] else true endif);
```

Sudoku (2)

```
% All rows are different
constraint forall (i in BoardRange)(
  alldifferent([sudoku[i,j] | j in BoardRange]));

% All columns are different
constraint forall (j in BoardRange) (
  alldifferent([sudoku[i,j] | i in BoardRange]));

% All sub-squares are different
constraint forall (sr, sc in SubSquareRange)(
  alldifferent([sudoku[(sr-1)*S + r, (sc-1)*S + c]
                       | r,c in SubSquareRange]));

solve satisfy;
```

Sudoku (Daten)

```
initial=[|
0, 0, 3, 0, 2, 0, 6, 0, 0|
9, 0, 0, 3, 0, 5, 0, 0, 1|
0, 0, 1, 8, 0, 6, 4, 0, 0|
0, 0, 8, 1, 0, 2, 9, 0, 0|
7, 0, 0, 0, 0, 0, 0, 0, 8|
0, 0, 6, 7, 0, 8, 2, 0, 0|
0, 0, 2, 6, 0, 9, 5, 0, 0|
8, 0, 0, 2, 0, 3, 0, 0, 9|
0, 0, 5, 0, 1, 0, 3, 0, 0|];
```

MiniZinc

```
Silverbird:Code$ mzn-g12fd sudoku-1.mzn sudoku-1.dzn
4 8 3  9 2 1  6 5 7
9 6 7  3 4 5  8 2 1
2 5 1  8 7 6  4 9 3

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

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

Beispiel: Job-Shop Scheduling

```
int: jobs;                                          % no of jobs
int: tasks;                                         % no of tasks per job
array [1..jobs,1..tasks] of int: d;                 % task durations
int: total = sum(i in 1..jobs, j in 1..tasks)
               (d[i,j]);                            % total duration
int: digs = ceil(log(10.0,int2float(total)));       % digits for output
array [1..jobs,1..tasks] of var 0..total: s;        % start times
var 0..total: end;                                  % total end time

constraint %% ensure the tasks occur in sequence
    forall(i in 1..jobs) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );
```

Beispiel: Job-Shop Scheduling

```
constraint %% ensure no overlap of tasks
    forall(j in 1..tasks) (
        forall(i,k in 1..jobs where i < k) (
    s[i,j] + d[i,j] <= s[k,j] \/
            s[k,j] + d[k,j] <= s[i,j]
        )
    );

solve minimize end;

output ["end = ", show(end), "\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in 1..jobs, j in 1..tasks ];
```