Grundlagen der Künstlichen Intelligenz

10. Constraint-Satisfaction-Probleme: Algorithmen

Malte Helmert

Universität Basel

12. April 2013

Constraint-Satisfaction-Probleme: Überblick

Kapitelüberblick Constraint-Satisfaction-Probleme:

- Einführung ($\rightsquigarrow$ Kapitel 9)
- Algorithmen ($\rightsquigarrow$ dieses Kapitel)
- Problemstruktur ($\rightsquigarrow$ Kapitel 11)

CSP-Algorithmen

In diesem Kapitel betrachten wir Lösungsalgorithmen
für Constraint-Netze.

Grundkonzepte:

- Suche: systematisches Ausprobieren von partiellen Belegungen
- Backtracking: Verwerfen inkonsistenter partieller Belegungen
- Inferenz: Herleiten schärferer äquivalenter Constraints,
  um Suchraum zu verkleinern (Backtracking früher möglich)

Suche vs. Inferenz

Trade-off Suche vs. Inferenz

Je komplexer die Inferenz,

- desto weniger Suchknoten müssen durchsucht werden und
- desto mehr Zeitaufwand wird pro Suchknoten benötigt

Wir beginnen mit dem Extremfall ohne Inferenz:
naives Backtracking

Naives Backtracking

Naives Backtracking (= ohne Inferenz)

function NaiveBacktracking($\mathcal{C}, \alpha$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

select some variable v for which α is not defined

for each $d \in \text{dom}(v)$ in some order:
    $\alpha' := \alpha \cup \{v \mapsto d\}$
    $\alpha'' :=$ NaiveBacktracking($\mathcal{C}, \alpha'$)
    if $\alpha'' \neq$ **inconsistent**:
        return α''
return inconsistent

Eingabe: Constraint-Netz $\mathcal{C}$ und partielle Belegung α von $\mathcal{C}$
(erster Aufruf: die leere Belegung $\alpha = \emptyset$)

Ergebnis: Lösung von $\mathcal{C}$ oder **inconsistent**

Ist das ein neuer Algorithmus?

Wir haben diesen Algorithmus schon gesehen:
Backtracking entspricht Tiefensuche (vgl. Kapitel 4)
mit folgendem Zustandsraum:

- Zustände: konsistente partielle Belegungen
- Anfangszustand: leere Belegung $\emptyset$
- Zielzustände: konsistente totale Belegungen
- Aktionen: $assign_{v,d}$ weist Variable v Wert $d \in \mathrm{dom}(v)$ zu
- Kosten: alle 0 (alle Lösungen gleich gut)
- Transitionen:
  - für jede nicht-totale Belegung α wähle Variable $v = \mathrm{selected}(\alpha)$, die in α unbelegt ist
  - Transition $\alpha \xrightarrow{assign_{v,d}} \alpha \cup \{v \mapsto d\}$ für alle $d \in \mathrm{dom}(v)$

Warum Tiefensuche?

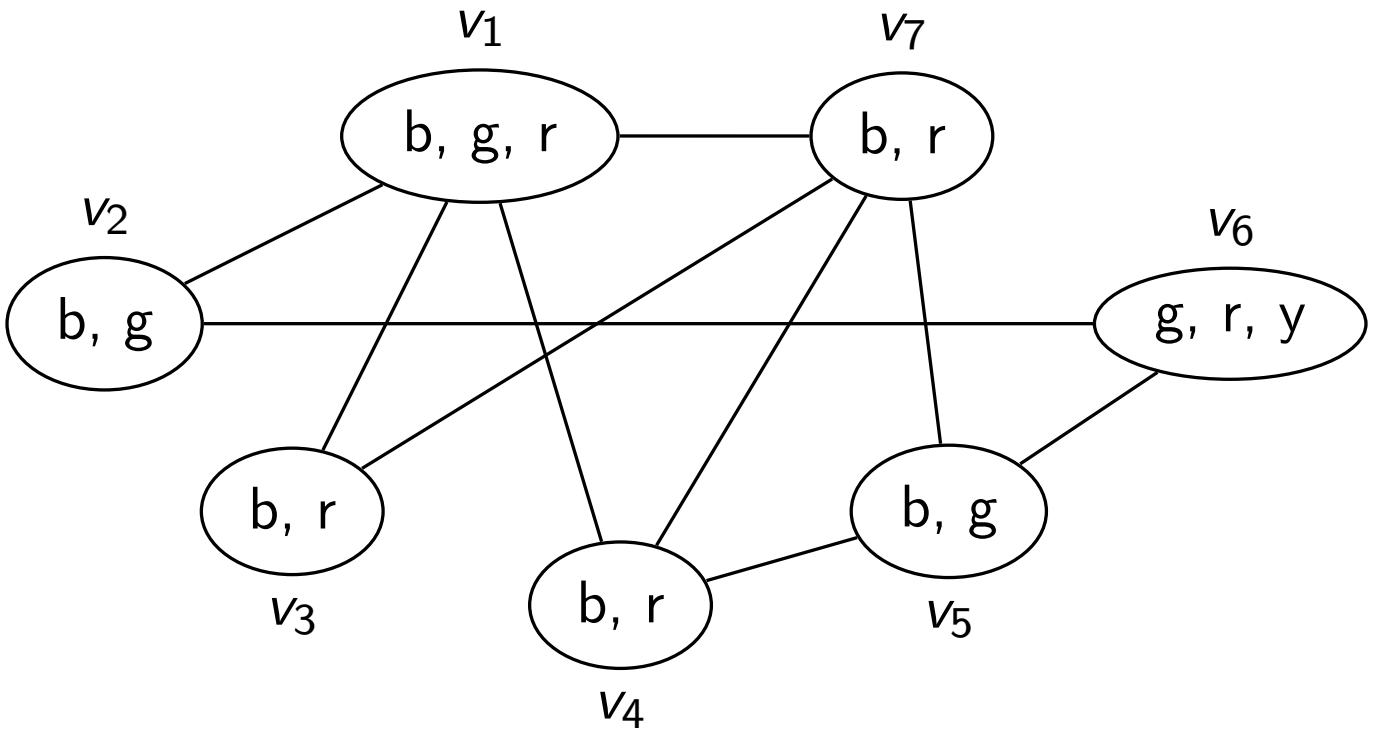
Tiefensuche ist für CSPs besonders geeignet:

- Pfadlänge beschränkt (durch Anzahl Variablen)
- Alle Lösungen in derselben Tiefe (in unterster Suchebene)
- Zustandsraum gerichteter Baum,
  Anfangszustand ist Wurzel $\rightsquigarrow$ keine Duplikate (Warum?)

Somit tritt keiner der für Tiefensuche problematischen Fälle auf.

Naives Backtracking: Beispiel

Betrachte das Constraint-Netz für folgendes Graphfärbungsproblem:

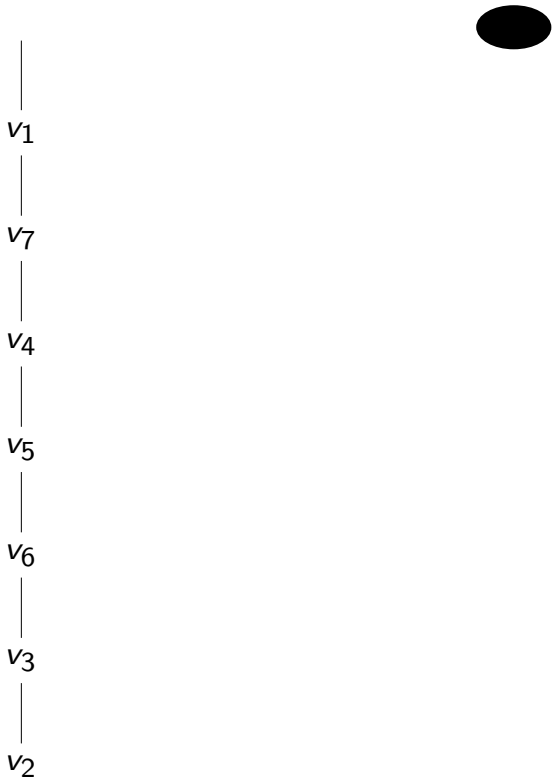


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

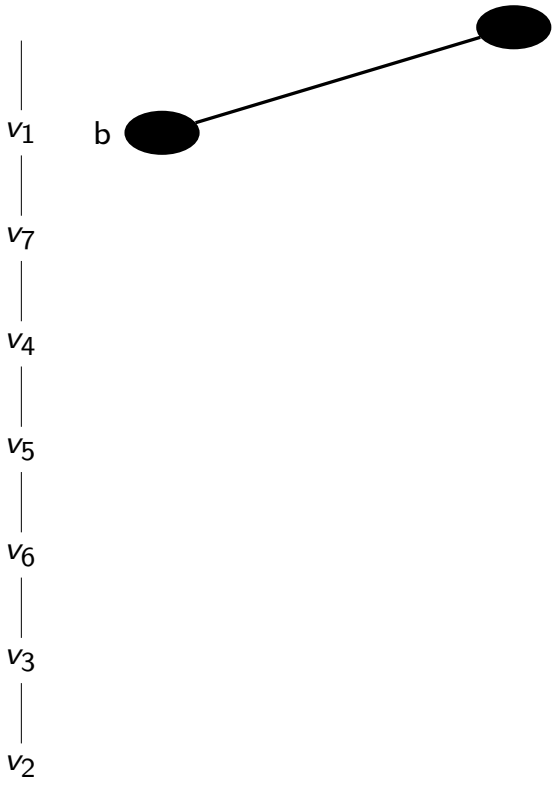


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

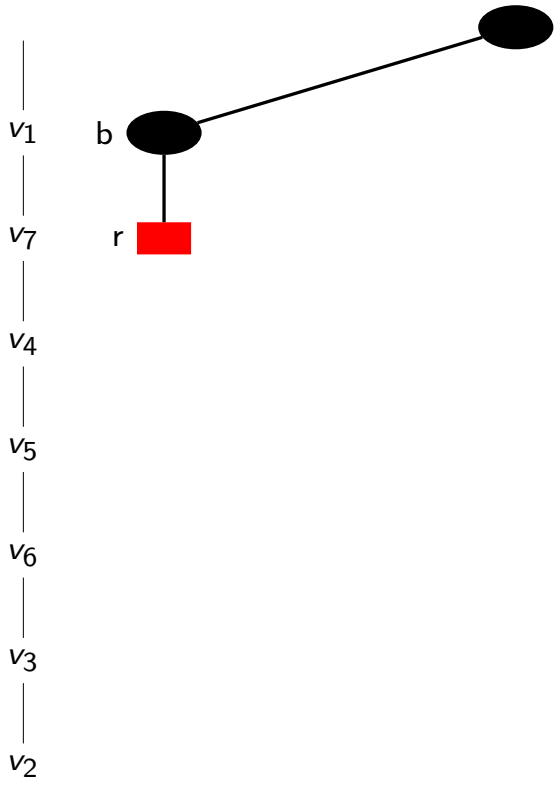


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

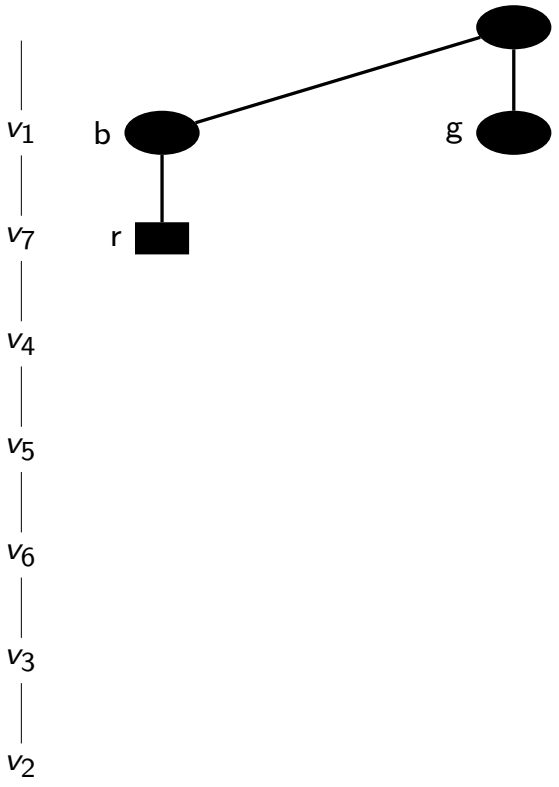


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

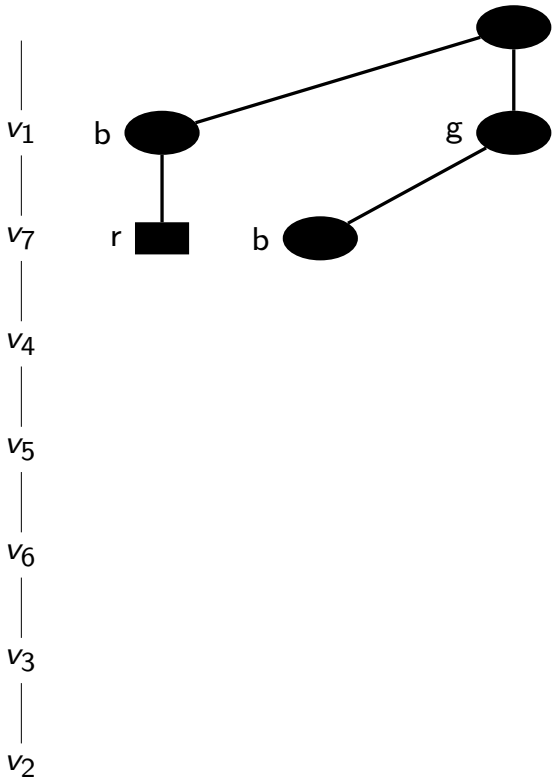


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

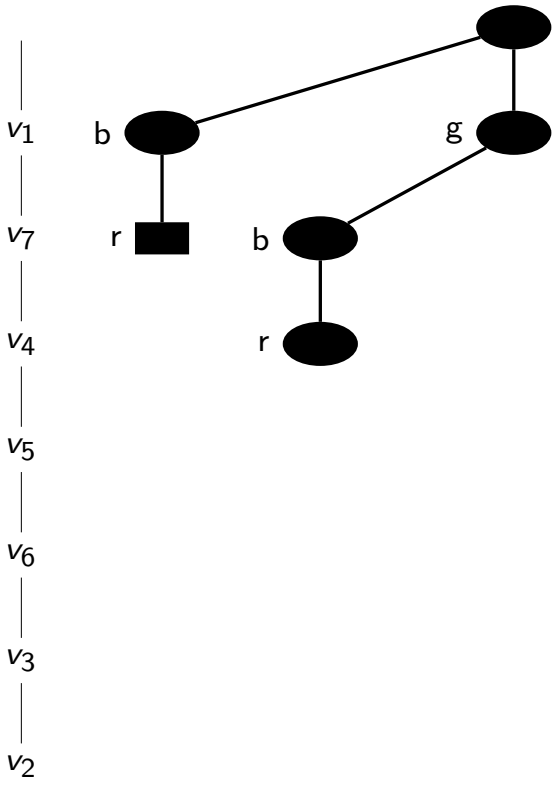


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

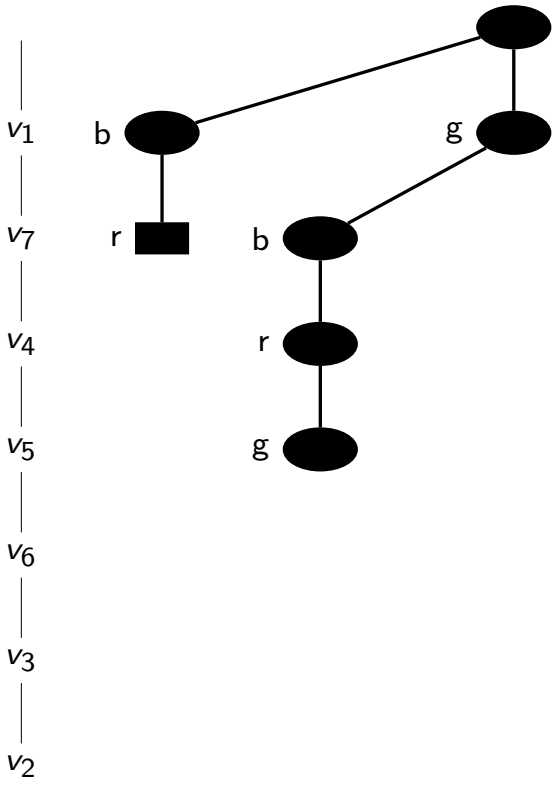


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

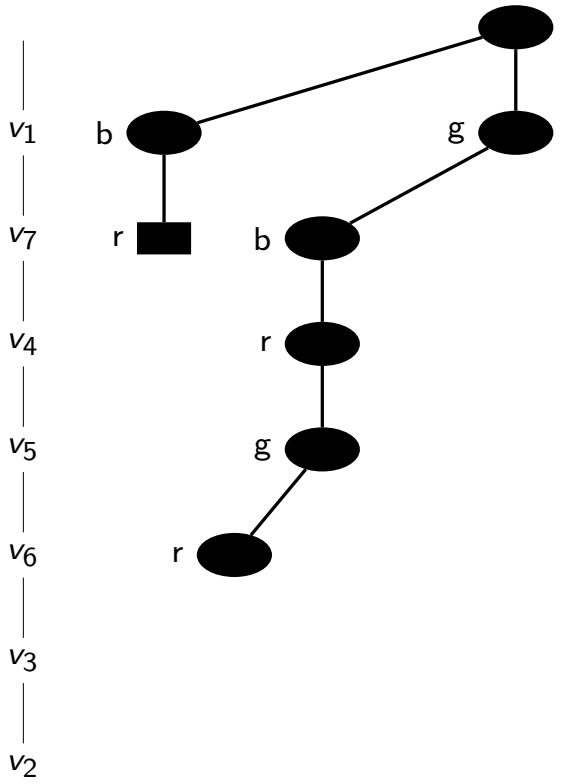


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

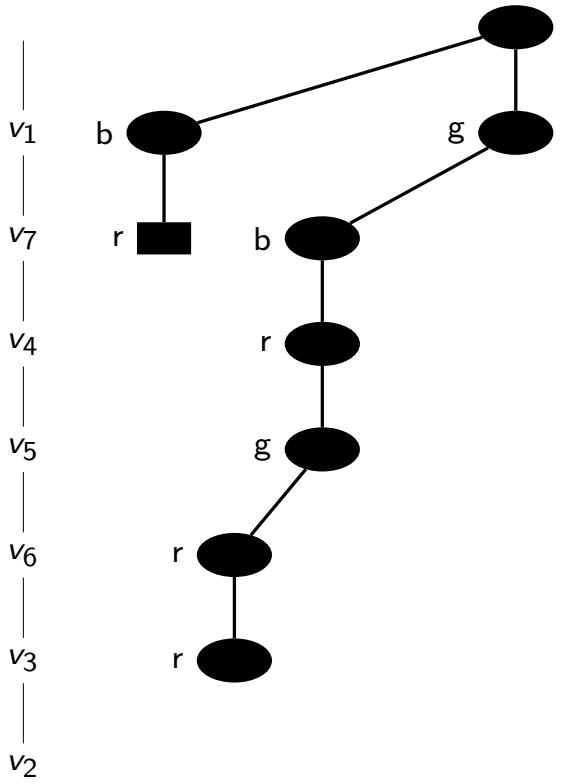


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

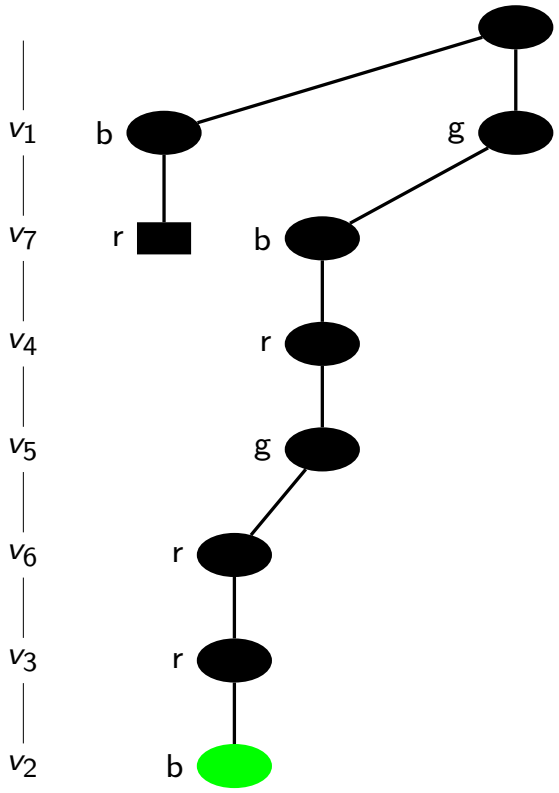


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

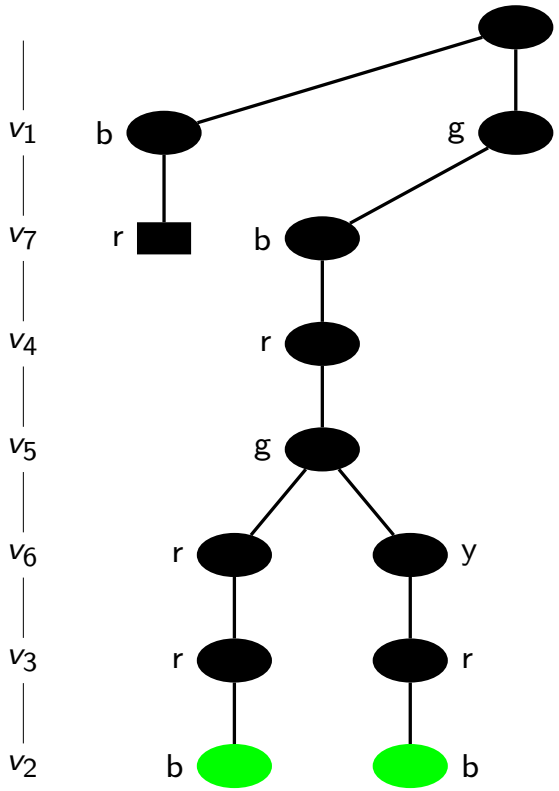


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

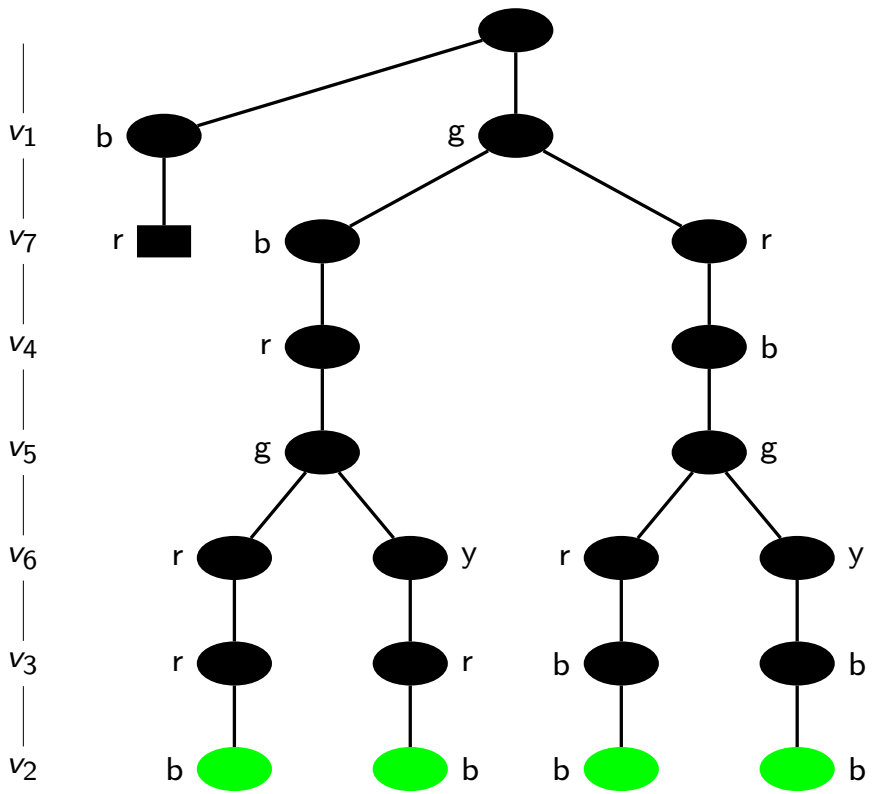


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)

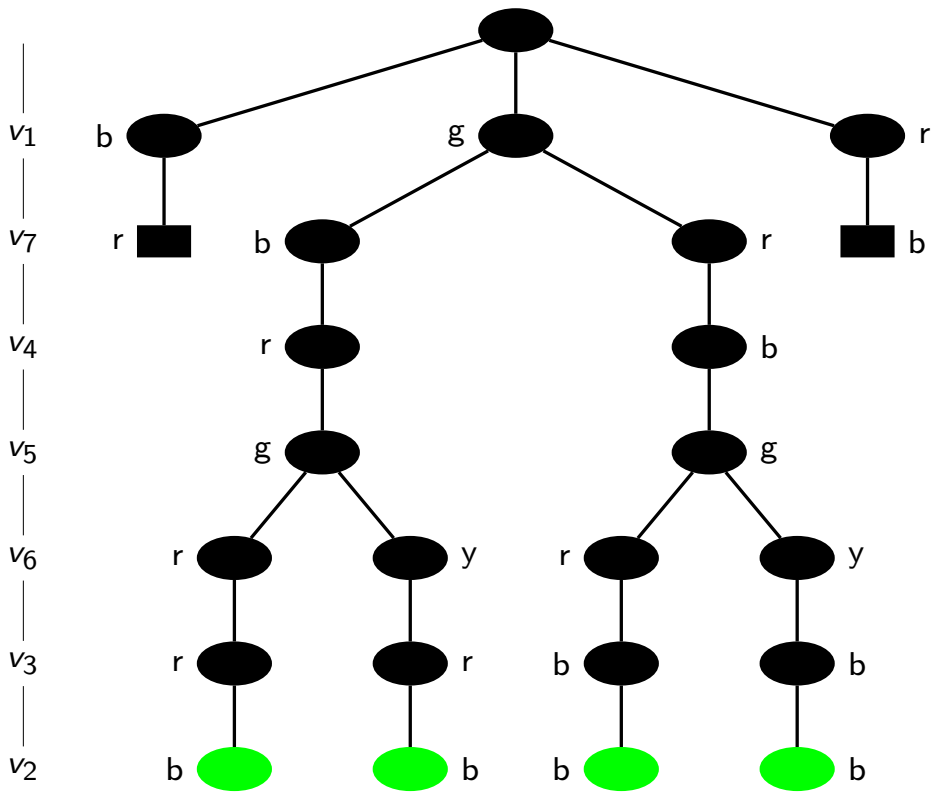


Naives Backtracking: Beispiel

Suchbaum für naives Backtracking mit

- fester Variablenreihenfolge $v_1, v_7, v_4, v_5, v_6, v_3, v_2$
- alphabetischer Reihenfolge der Werte

(ohne inkonsistente Knoten; bei Zielknoten fortgesetzt)



Naives Backtracking: Diskussion

- Naives Backtracking muss oft ähnliche Suchpfade (partielle Belegungen gleich bis auf wenige Variablen) erschöpfend durchsuchen.
- „Kritische“ Variablen nicht erkannt, daher (zu) spät belegt
- Entscheidungen, die später zwangsläufig zu Constraint-Verletzungen führen, werden erst erkannt, wenn alle beteiligten Variablen belegt wurden

$\rightsquigarrow$ mehr Intelligenz durch Fokus auf kritischen Entscheidungen und Inferenz von Konsequenzen der bisherigen Entscheidungen

Variablen- und Wertordnungen

Naives Backtracking

function NaiveBacktracking($\mathcal{C}, \alpha$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

select some variable v for which α is not defined

for each $d \in \text{dom}(v)$ in some order:
    $\alpha' := \alpha \cup \{v \mapsto d\}$
    $\alpha'' :=$ NaiveBacktracking($\mathcal{C}, \alpha'$)
    if $\alpha'' \neq$ **inconsistent**:
        return α''
return inconsistent

Variablen- und Wertordnungen

Variablenordnung:

- Backtracking lässt offen, in welcher Reihenfolge Variablen belegt werden
- beeinflusst oft dramatisch die Grösse des Suchraums und damit die Performance der Suche
  $\rightsquigarrow$ Beispiel: Übungsaufgaben

Wertordnung:

- Backtracking lässt ebenfalls offen, in welcher Reihenfolge die Werte der ausgewählten Variable v betrachtet werden
- nicht ganz so wichtig, da in Teilbäumen ohne Lösung nicht von Belang (Warum nicht?)
- wenn Lösung im Teilbaum existiert, sollte nach Möglichkeit zunächst Wert ausgewählt werden, der zur Lösung führt (Warum?)

Statische vs. dynamische Ordnungen

Wir unterscheiden:

- statische Ordnungen (im Voraus festgelegt)
- dynamische Ordnungen (ausgewählte Variable/ ausgewählte Wertordnung hängt vom Suchzustand ab)

Vergleich:

- dynamische Ordnungen offensichtlich mächtiger
- statische Ordnungen verursachen dafür keinen Overhead während der Suche

Die folgenden Ordnungen können statisch vorgenommen werden, sind aber effektiver, wenn man sie mit Inferenz ($\rightsquigarrow$ später) kombiniert und dynamisch auswertet.

Variablenordnungen

Zwei häufige Kriterien zur Variablenordnung:

- Minimum Remaining Values: wähle zuerst Variablen aus, deren Wertebereich möglichst klein ist
  - Intuition: wenige Teilbäume $\rightsquigarrow$ kleiner Baum
  - Extremfall: nur ein Wert $\rightsquigarrow$ erzwungene Belegung
- Most Constraining Variable: wähle zuerst Variablen aus, die an möglichst vielen nichttrivialen Constraints beteiligt sind
  - Intuition: Constraints möglichst früh testen $\rightsquigarrow$ früh Inkonsistenzen erkennen $\rightsquigarrow$ kleiner Baum

Kombination: verwende Minimum-Remaining-Values-Kriterium, dann Most-Constraining-Variable-Kriterium zum Tie-Breaking

Wertordnungen

Definition (Konflikt)

Sei $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ ein Constraint-Netz.
Für Variablen $v \neq v'$ und Werte $d \in \mathrm{dom}(v)$, $d' \in \mathrm{dom}(v')$ steht $v \mapsto d$ im Konflikt mit $v' \mapsto d'$, falls $\langle d, d' \rangle \notin R_{vv'}$.

Kriterium zur Wertordnung für partielle Belegung α
und ausgewählte Variable v:

- Minimum Conflicts: Bevorzuge Werte $d \in \mathrm{dom}(v)$, für die $v \mapsto d$ an möglichst wenigen Konflikten mit in α unbelegten Variablen beteiligt ist.

Inferenz

Inferenz

Inferenz

Herleiten zusätzlicher Constraints (hier: unär oder binär),
die aus den bekannten Constraints logisch folgen,
d. h. in allen Lösungen erfüllt sind.

Beispiel: Constraint-Netz mit Variablen v_1, v_2, v_3
mit Wertebereich $\{1, 2, 3\}$ und Constraints $v_1 < v_2$ und $v_2 < v_3$.

Wir können beispielsweise folgern:

- v_2 kann nicht 3 sein (neuer unärer Constraint
  = Einschränkung des Wertebereichs von v_2)
- $v_1 < v_3$ (neuer binärer Constraint
  = trivialer Constraint verschärft)
- $R_{v_1 v_2} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$ kann verschärft werden
  zu $\{\langle 1, 2\rangle\}$ (verschärfter binärer Constraint)

Nutzen von Inferenz

- Formal ist Inferenz Ersetzen des gegebenen Constraint-Netzes durch ein schärferes äquivalentes Netz.
- Nutzen: kleinerer Suchbaum
- dem Nutzen steht der Berechnungsaufwand gegenüber

Wo Inferenz verwenden?

Unterschiedliche Verwendungsmöglichkeiten für Inferenz:

- einmalig als Vorverarbeitung vor der Suche
- mit Suche kombiniert: bei jedem rekursiven Aufruf der Backtracking-Prozedur
  - bereits belegte Variablen $v \mapsto d$ können wie $\mathrm{dom}(v) = \{d\}$ verstanden werden $\rightsquigarrow$ mehr Schlussfolgerungen möglich
  - bei Backtracking müssen Verschärfungen durch Inferenz zurückgenommen werden, da sie die gegebene Belegung als Voraussetzung hatten

$\rightsquigarrow$ mächtig, aber eventuell teuer

Backtracking mit Inferenz

function BacktrackingWithInference($\mathcal{C}, \alpha$):

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

$\mathcal{C}' := \langle V, \text{dom}', (R'_{uv}) \rangle :=$ copy of $\mathcal{C}$
apply inference to $\mathcal{C}'$

if $\text{dom}'(v) \neq \emptyset$ for all variables v:

    select some variable v for which α is not defined

    for each $d \in$ copy of $\text{dom}'(v)$ in some order:
        $\alpha' := \alpha \cup \{v \mapsto d\}$
        $\text{dom}'(v) := \{d\}$
        $\alpha'' :=$ BacktrackingWithInference($\mathcal{C}', \alpha'$)
        if $\alpha'' \neq$ **inconsistent**:
            return α''

return inconsistent

Backtracking mit Inferenz

function BacktrackingWithInference($\mathcal{C}, \alpha$):

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

$\mathcal{C}' := \langle V, \text{dom}', (R'_{uv}) \rangle :=$ copy of $\mathcal{C}$
apply inference to $\mathcal{C}'$

if $\text{dom}'(v) \neq \emptyset$ for all variables v:

    select some variable v for which α is not defined

    for each $d \in$ copy of $\text{dom}'(v)$ in some order:
        $\alpha' := \alpha \cup \{v \mapsto d\}$
        $\text{dom}'(v) := \{d\}$
        $\alpha'' :=$ BacktrackingWithInference$(\mathcal{C}', \alpha')$
        if $\alpha'' \neq$ **inconsistent**:
            return α''

return inconsistent

Backtracking mit Inferenz: Diskussion

- inference ist ein Platzhalter:
  verschiedene Inferenzmethoden können eingesetzt werden
- bei vielen Inferenzmethoden wird der anfängliche Test
  auf Inkonsistenz von α überflüssig
  - kein inkonsistentes α kann erreicht werden,
    da konfligierende Werte durch Inferenz
    aus den Wertebereichen gestrichen werden
- Inferenzmethode kann Unlösbarkeit (gegeben α) erkennen
  und durch Leeren eines Wertebereichs signalisieren
- effizient implementierte Inferenz oft inkrementell:
  zuletzt belegtes Paar $v \mapsto d$ wird mitgeteilt und verwendet,
  um die Berechnung zu beschleunigen

Forward Checking

Wir beginnen mit einer sehr einfachen Inferenz-Methode:

Forward Checking

Inferenz: Entferne alle Variablen-/Werte-Paare aus dom', die mit bereits belegten Paaren im Konflikt stehen.

$\rightsquigarrow$ Definition von Konflikt im vorigen Abschnitt

Inkrementelle Berechnung:

- Immer, wenn $v \mapsto d$ zur Belegung hinzugefügt wird, entferne alle mit $v \mapsto d$ im Konflikt stehenden Paare.

Forward Checking: Diskussion

Eigenschaften von Forward Checking:

- korrekte Inferenzmethode (erhält Äquivalenz)
- beeinflusst Wertebereiche (= unäre Constraints),
  aber nicht die binären Constraints
- macht Konsistenztest am Anfang der Backtracking-Prozedur
  überflüssig (Warum?)
- billige, aber dennoch oft nützliche Inferenz-Methode

$\rightsquigarrow$ selten eine gute Idee, nicht mindestens
Forward Checking zu verwenden

Im Folgenden betrachten wir mächtigere Inferenzmethoden.

Kantenkonsistenz

Kantenkonsistenz: Definition

Definition (kantenkonsistent)

Sei $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ ein Constraint-Netz.

(a) Eine Variable $v \in V$ ist kantenkonsistent in Bezug auf eine andere Variable $v' \in V$, wenn für jeden Wert $d \in \mathrm{dom}(v)$ ein Wert $d' \in \mathrm{dom}(v')$ mit $\langle d, d' \rangle \in R_{vv'}$ existiert.

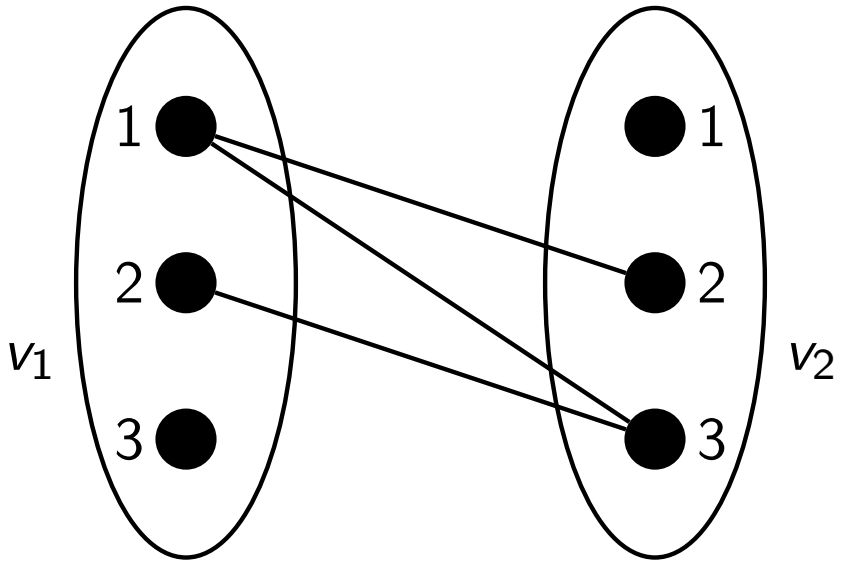
(b) Das Constraint-Netz $\mathcal{C}$ ist kantenkonsistent, wenn jede Variable $v \in V$ kantenkonsistent in Bezug auf jede andere Variable $v' \in V$ ist.

Anmerkungen:

- Definition für Variablenpaare ist asymmetrisch
- v immer kantenkonsistent in Bezug auf v', wenn Constraint zwischen v und v' trivial ist

Kantenkonsistenz: Beispiel

Betrachte ein Constraint-Netz mit Variablen v_1 und v_2, Wertebereichen $\mathrm{dom}(v_1) = \mathrm{dom}(v_2) = \{1, 2, 3\}$ und dem durch $v_1 < v_2$ beschriebenen Constraint.



Kantenkonsistenz von v_1 in Bezug auf v_2 und von v_2 in Bezug auf v_1 ist verletzt.

Herstellen von Kantenkonsistenz

- Herstellen von Kantenkonsistenz, d. h. Entfernen von Werten aus $\mathrm{dom}(v)$, die die Kantenkonsistenz von v in Bezug auf v' verletzen, ist eine korrekte Inferenzmethode. (Warum?)
- mächtiger als Forward Checking (Warum?)
- Im folgenden betrachten wir Algorithmen zum Herstellen von Kantenkonsistenz.

Verarbeitung eines einzelnen Variablenpaars: revise

function revise$(\mathcal{C}, v, v')$:

$\langle V, \mathrm{dom}, (R_{uv})\rangle := \mathcal{C}$

for each $d \in \mathrm{dom}(v)$:

  if there is no $d' \in \mathrm{dom}(v')$ with $\langle d, d'\rangle \in R_{vv'}$:

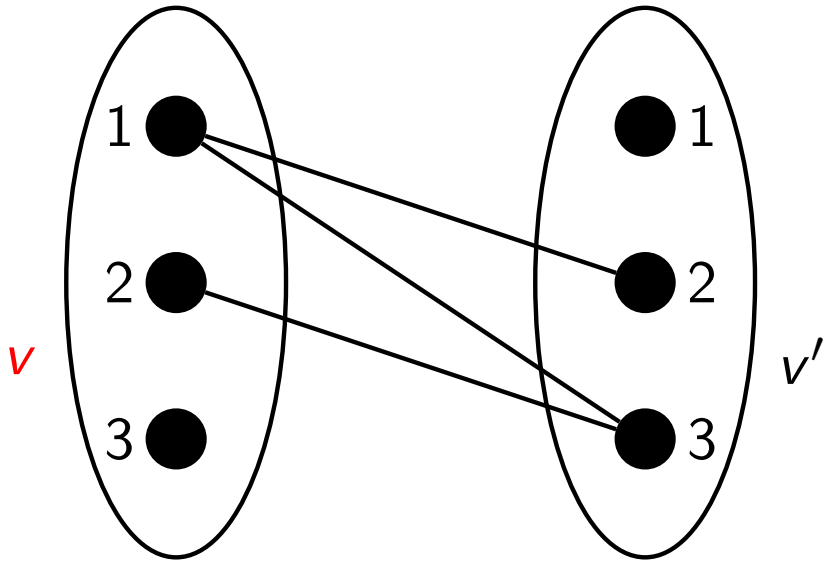
    remove d from $\mathrm{dom}(v)$

Eingabe: Constraint-Netz $\mathcal{C}$ und zwei Variablen v, v' von $\mathcal{C}$

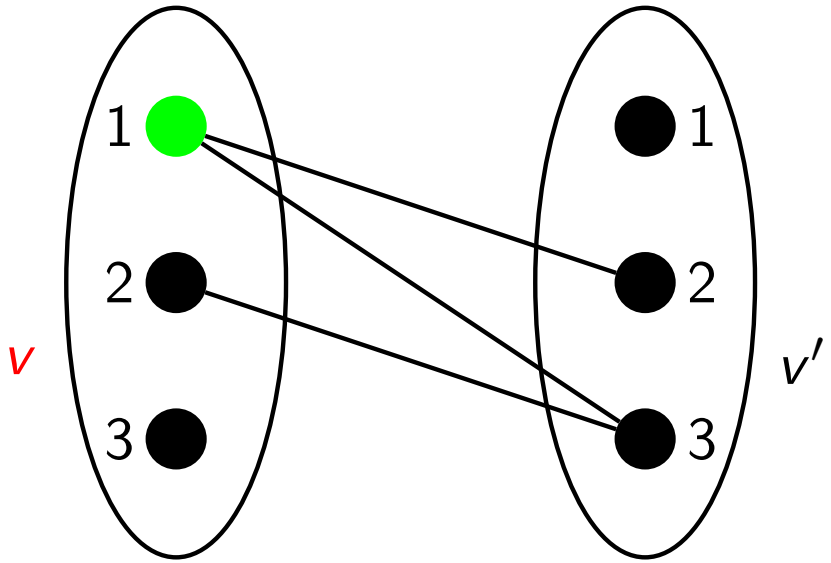
Effekt: Macht v kantenkonsistent in Bezug auf v'.
Alle verletzenden Werte werden aus $\mathrm{dom}(v)$ entfernt.

Zeitaufwand: $O(k^2)$, wenn k maximale Wertebereichsgrösse
(geeignete Kodierung von (R_{uv}) und dom vorausgesetzt)

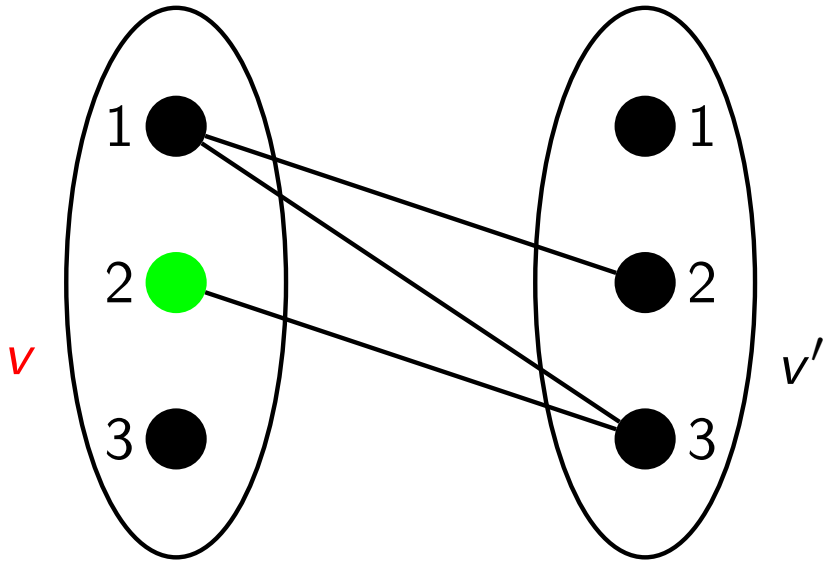
Beispiel: revise



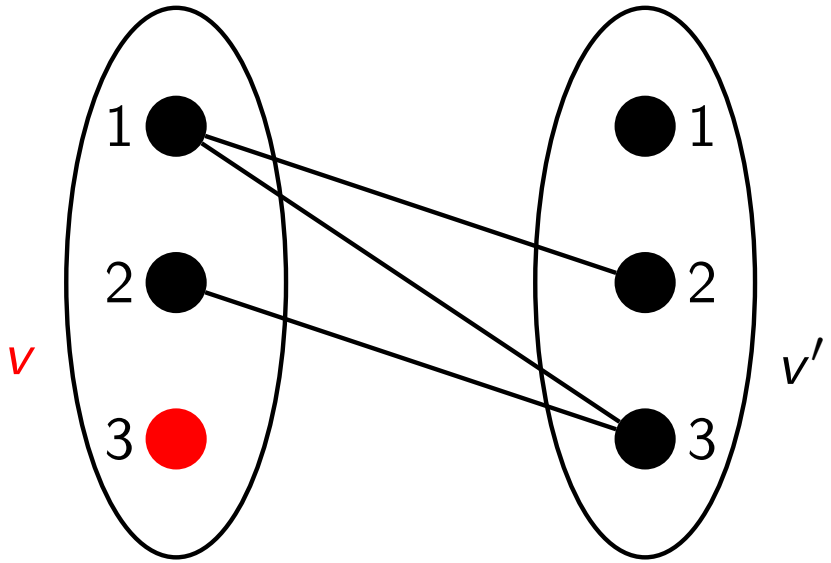
Beispiel: revise



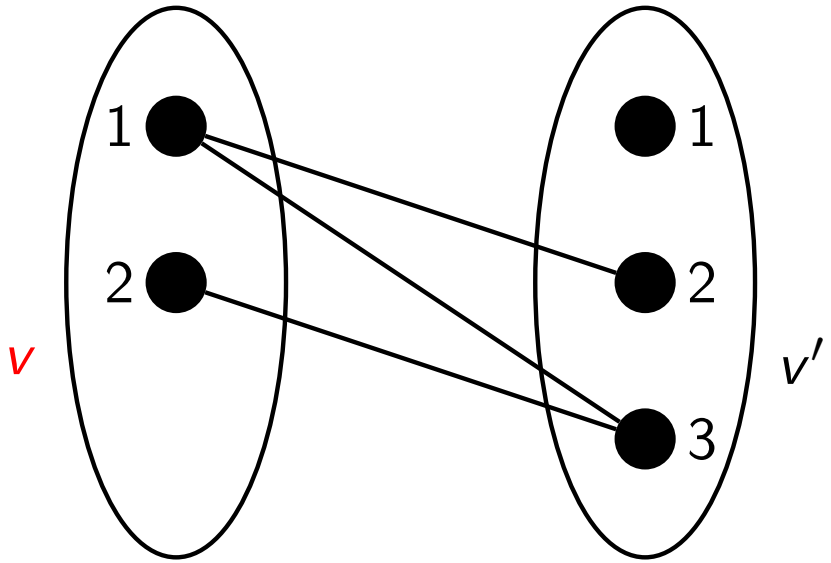
Beispiel: revise



Beispiel: revise



Beispiel: revise



Herstellen von Kantenkonsistenz: AC-1

function AC-1($\mathcal{C}$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

repeat

  for each nontrivial constraint R_{uv}:

    revise($\mathcal{C}, u, v$)

    revise($\mathcal{C}, v, u$)

until no domain has changed in this iteration

Eingabe: Constraint-Netz $\mathcal{C}$

Effekt: transformiert $\mathcal{C}$ in äquivalentes kantenkonsistentes Netz

Zeitaufwand: ?

Herstellen von Kantenkonsistenz: AC-1

function AC-1($\mathcal{C}$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

repeat

  for each nontrivial constraint R_{uv}:

    revise($\mathcal{C}, u, v$)

    revise($\mathcal{C}, v, u$)

until no domain has changed in this iteration

Eingabe: Constraint-Netz $\mathcal{C}$

Effekt: transformiert $\mathcal{C}$ in äquivalentes kantenkonsistentes Netz

Zeitaufwand: $O(n \cdot e \cdot k^3)$, wenn n Variablen, e nichttriviale Constraints und k maximale Wertebereichsgrösse

AC-1: Diskussion

- AC-1 erfüllt seine Aufgabe, ist aber ineffizient.
- Oft werden Variablenpaare wieder und wieder überprüft, deren Wertebereiche sich nicht geändert haben.
- Diese Überprüfungen können eingespart werden.

$\rightsquigarrow$ effizienterer Algorithmus: AC-3

Herstellen von Kantenkonsistenz: AC-3

Idee: merke potenziell inkonsistente Variablenpaare in Queue

function AC-3($\mathcal{C}$):

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

$queue := \emptyset$

for each nontrivial constraint R_{uv}:

    insert $\langle u, v \rangle$ into *queue*

    insert $\langle v, u \rangle$ into *queue*

while $queue \neq \emptyset$:

    remove any element $\langle u, v \rangle$ from *queue*

    revise($\mathcal{C}, u, v$)

    if dom(u) changed in the call to revise:

        for each $w \in V \setminus \{u, v\}$ where R_{wu} is nontrivial:

            insert $\langle w, u \rangle$ into *queue*

AC-3: Diskussion

- *queue* kann eine beliebige Datenstruktur sein,
  die Einfügen und Entfernen erlaubt
  (Reihenfolge des Entfernens ist für Ergebnis egal)

$\rightsquigarrow$ effizient z. B. ein Stack

- AC-3 hat denselben Effekt wie AC-1:
  es stellt Kantenkonsistenz her
- Beweisidee: Invariante der **while**-Schleife:
  Wenn $\langle u, v \rangle \notin \mathit{queue}$, dann u kantenkonsistent in Bezug auf v

AC-3: Zeitaufwand

Satz (Zeitaufwand von AC-3)

Sei $\mathcal{C}$ ein Constraint-Netz mit e nichttrivialen Constraints und maximaler Wertebereichsgrösse k.
Dann läuft AC-3 *in Zeit* $O(e \cdot k^3)$.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte einen einzelnen nichttrivialen Constraint.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte einen einzelnen nichttrivialen Constraint.
Jedes Mal, wenn er in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich einer der beteiligten Variablen reduziert.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte einen einzelnen nichttrivialen Constraint.
Jedes Mal, wenn er in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich einer der beteiligten Variablen reduziert.
Das kann höchstens $2k$ mal passieren.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte einen einzelnen nichttrivialen Constraint.
Jedes Mal, wenn er in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich einer der beteiligten Variablen reduziert.
Das kann höchstens $2k$ mal passieren.
Damit wird jeder Constraint höchstens $2k + 1 = O(k)$ mal in die Queue eingefügt und es gibt insgesamt höchstens $O(e \cdot k)$ Einfügeoperationen.

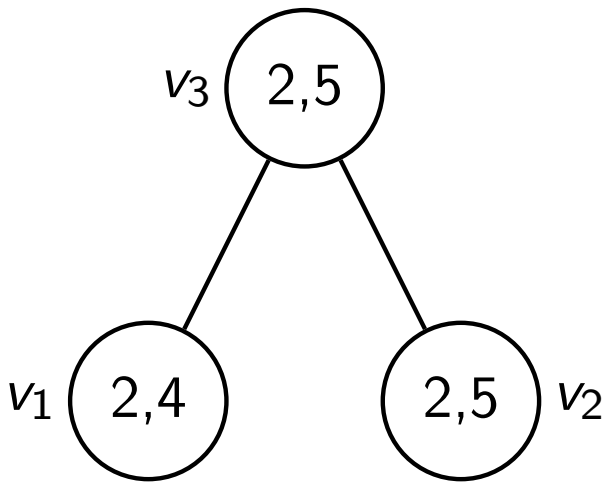
AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte einen einzelnen nichttrivialen Constraint.
Jedes Mal, wenn er in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich einer der beteiligten Variablen reduziert.
Das kann höchstens $2k$ mal passieren.
Damit wird jeder Constraint höchstens $2k + 1 = O(k)$ mal in die Queue eingefügt und es gibt insgesamt höchstens $O(e \cdot k)$ Einfügeoperationen.
Dies begrenzt die Zahl der Iterationen der **while**-Schleife auf $O(ek)$, weswegen die revise-Aufrufe höchstens Zeit $O(ek) \cdot O(k^2) = O(ek^3)$ benötigen. □

AC-3: Beispiel

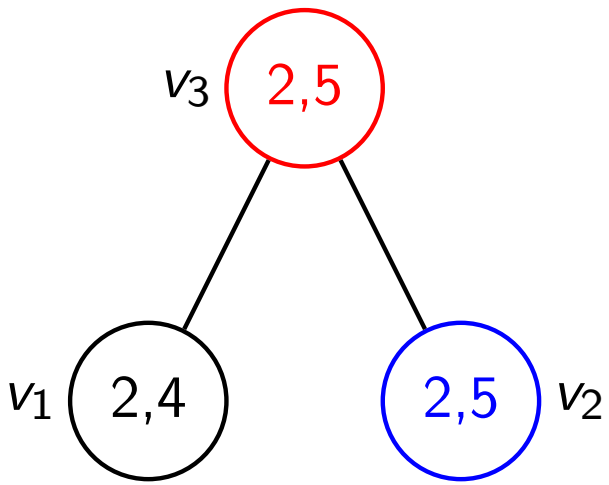
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)
(v_3, v_1)
(v_2, v_3)
(v_3, v_2)

AC-3: Beispiel

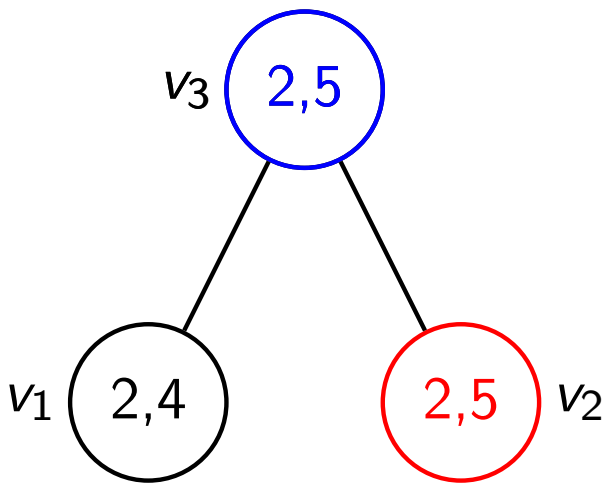
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)
(v_3, v_1)
(v_2, v_3)
(v_3, v_2)

AC-3: Beispiel

Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").

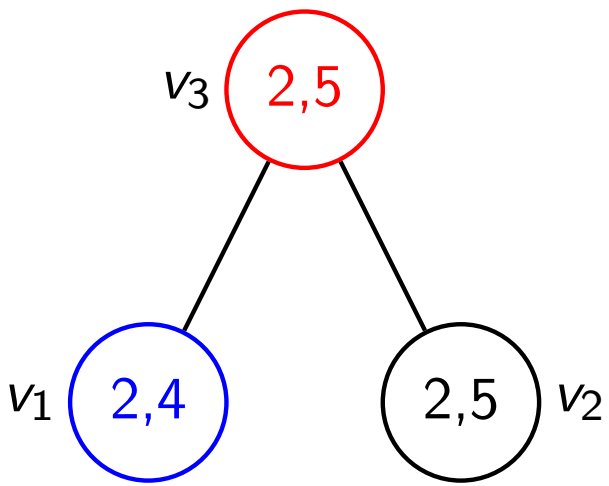


Queue

(v_1, v_3)
(v_3, v_1)
(v_2, v_3)

AC-3: Beispiel

Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").

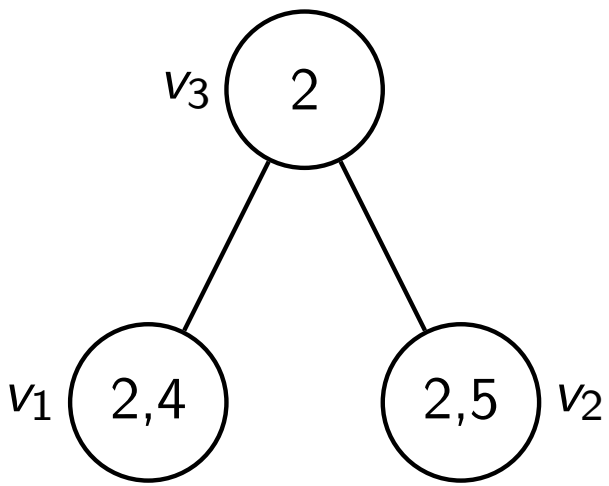


Queue

(v_1, v_3)
(v_3, v_1)

AC-3: Beispiel

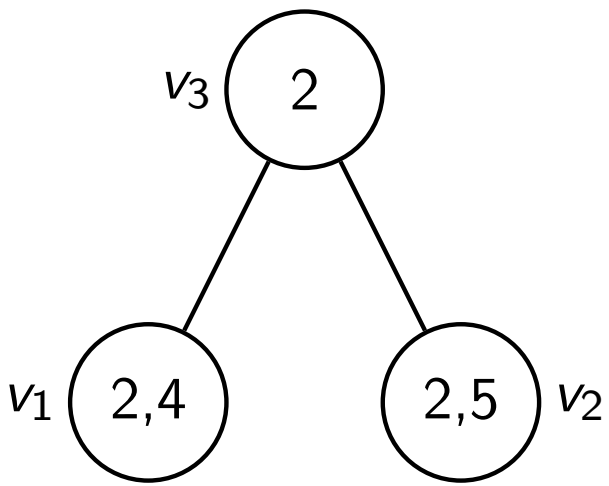
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)

AC-3: Beispiel

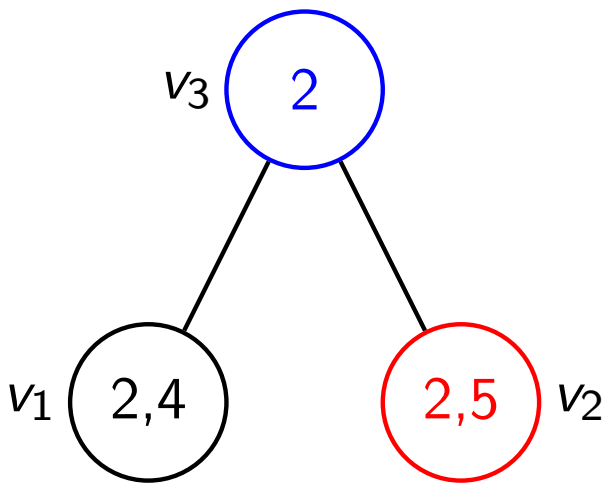
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\text{dom}(v_1) = \{2, 4\}$ und $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)
(v_2, v_3)

AC-3: Beispiel

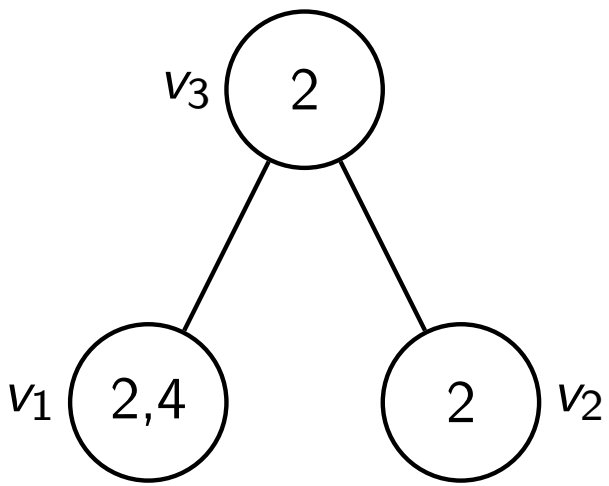
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)
(v_2, v_3)

AC-3: Beispiel

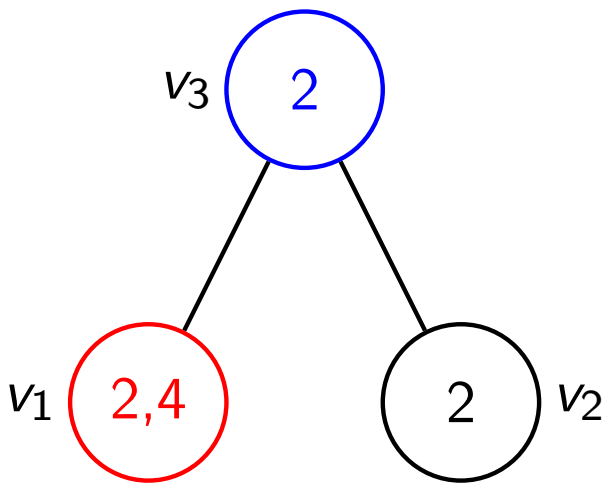
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)

AC-3: Beispiel

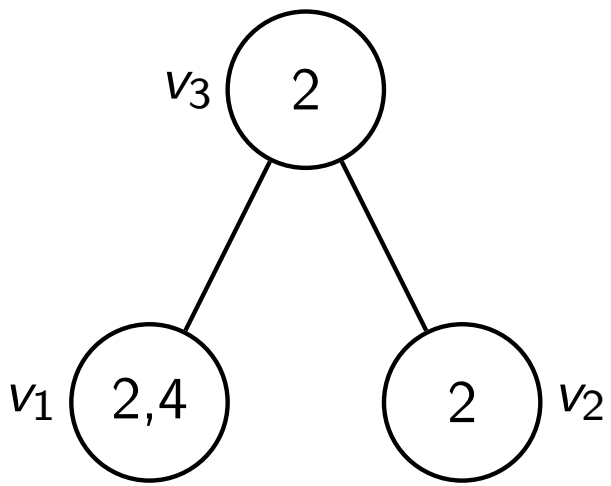
Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\mathrm{dom}(v_1) = \{2, 4\}$ und $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue
(v_1, v_3)

AC-3: Beispiel

Betrachte Constraint-Netz mit drei Variablen v_1, v_2, v_3 mit $\text{dom}(v_1) = \{2, 4\}$ und $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ sowie den Constraints $v_3|v_1$ und $v_3|v_2$ ("teilt").



Queue

Pfadkonsistenz

Jenseits von Kantenkonsistenz: Pfadkonsistenz

Grundidee der Kantenkonsistenz:

- zu jeder Belegung einer Variable u muss es eine passende Belegung jeder anderen Variable v geben
- ansonsten werden Werte von u, die nicht auf v erweiterbar sind, verboten

$\rightsquigarrow$ neuer unärer Constraint auf u

Idee lässt sich auf drei Variablen erweitern (Pfadkonsistenz):

- zu jeder gemeinsamen Belegung von Variablen u, v muss es eine passende Belegung jeder anderen Variablen w geben
- ansonsten werden Wertepaare für u und v, die nicht auf w erweiterbar sind, verboten

$\rightsquigarrow$ neuer binärer Constraint auf u und v

Jenseits von Kantenkonsistenz: i-Konsistenz

Generelles Konzept der i-Konsistenz für $i \geq 2$:

- zu jeder gemeinsamen Belegung von $v_1, \ldots, v_{i-1}$ muss es eine passende Belegung jeder anderen Variable v_i geben
- ansonsten werden Wertetupel für $v_1, \ldots, v_{i-1}$, die nicht auf v_i erweiterbar sind, verboten

$\rightsquigarrow$ neuer $(i-1)$-stelliger Constraint auf $v_1, \ldots, v_{i-1}$

- 2-Konsistenz = Kantenkonsistenz
- 3-Konsistenz = Pfadkonsistenz (*)

Wir betrachten allgemeine i-Konsistenz nicht näher, zumal höhere Werte als $i = 3$ selten verwendet werden und wir uns hier auf höchstens binäre Constraints beschränken.

(*) übliche Definitionen von 3-Konsistenz vs. Pfadkonsistenz unterscheiden sich, wenn ternäre (dreistellige) Constraints erlaubt sind

Pfadkonsistenz: Definition

Definition (pfadkonsistent)

Sei $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ ein Constraint-Netz.

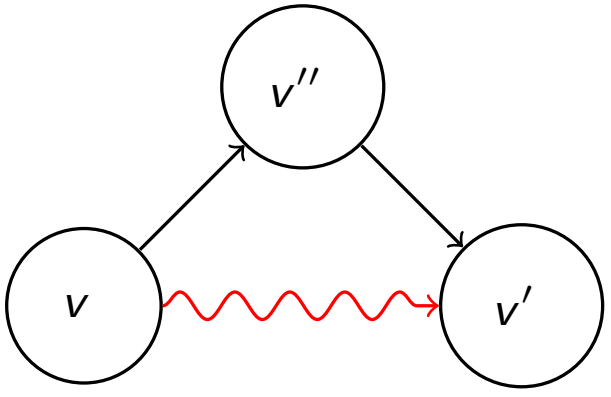
(a) Zwei verschiedene Variablen $v, v' \in V$ sind pfadkonsistent in Bezug auf eine dritte Variable $v'' \in V$, wenn für beliebige Werte $d \in \mathrm{dom}(v), d' \in \mathrm{dom}(v')$ mit $\langle d, d' \rangle \in R_{vv'}$ immer ein Wert $d'' \in \mathrm{dom}(v'')$ mit $\langle d, d'' \rangle \in R_{vv''}$ und $\langle d', d'' \rangle \in R_{v'v''}$ existiert.

(b) Das Constraint-Netz $\mathcal{C}$ ist pfadkonsistent, wenn für drei verschiedene Variablen v, v', v'' immer gilt, dass v und v' pfadkonsistent in Bezug auf v'' sind.

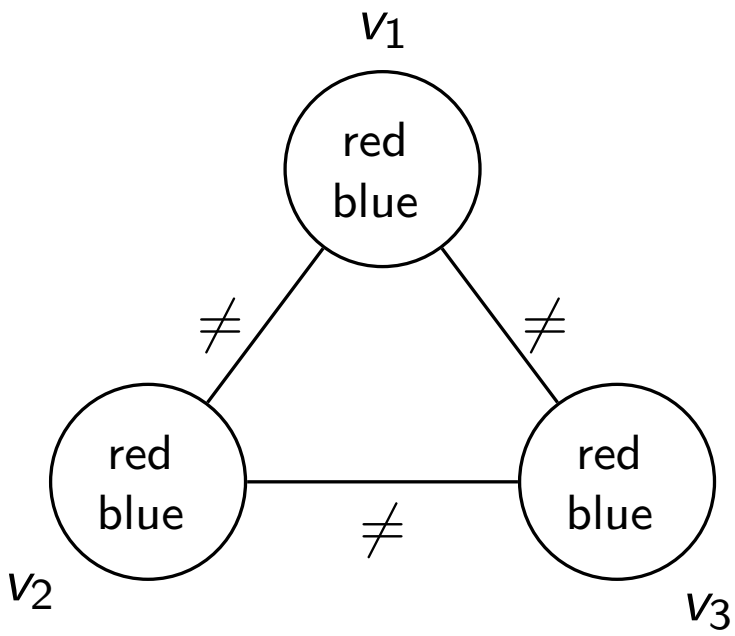
Pfadkonsistenz: Anmerkungen

Anmerkungen:

- Selbst wenn Constraint $R_{vv'}$ trivial ist, kann Pfadkonsistenz nichttriviale Constraints zwischen v und v' inferieren.
- Wenn das Netz kantenkonsistent ist,
  kann Pfadkonsistenz nur dann zu neuer Information führen,
  wenn sowohl $R_{vv''}$ als auch $R_{v'v''}$ nichttrivial sind.
- Name „Pfadkonsistenz“:
  Pfad $v \to v'' \to v'$ führt zu neuer Information über $v \to v'$



Pfadkonsistenz: Beispiel



kantenkonsistent, aber nicht pfadkonsistent

Verarbeitung eines Variablentripels: revise-3

Analog zu revise für Kantenkonsistenz:

function revise-3($\mathcal{C}, v, v', v''$):

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

for each $\langle d, d' \rangle \in R_{vv'}$:
  if there is no $d'' \in \mathrm{dom}(v'')$ with $\langle d, d'' \rangle \in R_{vv''}$
  and $\langle d', d'' \rangle \in R_{v'v''}$:
    remove $\langle d, d' \rangle$ from $R_{vv'}$

Eingabe: Constraint-Netz $\mathcal{C}$ und drei Variablen v, v', v'' von $\mathcal{C}$

Effekt: Macht v, v' pfadkonsistent in Bezug auf v''.
Alle verletzenden Paare werden aus $R_{vv'}$ entfernt.

Zeitaufwand: $O(k^3)$, wenn k maximale Wertebereichsgrösse

Herstellen von Pfadkonsistenz: PC-2

Analog zu AC-3 für Kantenkonsistenz:

function PC-2($\mathcal{C}$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

$queue := \emptyset$

for each set of two variables $\{u, v\}$:

    for each $w \in V \setminus \{u, v\}$:

        insert $\langle u, v, w \rangle$ into $queue$

while $queue \neq \emptyset$:

    remove any element $\langle u, v, w \rangle$ from $queue$

    revise($\mathcal{C}, u, v, w$)

    if R_{uv} changed in the call to revise:

        for each $w' \in V \setminus \{u, v\}$:

            insert $\langle w', u, v \rangle$ into $queue$

            insert $\langle w', v, u \rangle$ into $queue$

PC-2: Diskussion

Die Aussagen zu AC-3 gelten analog.

- PC-2 stellt Pfadkonsistenz her
- Beweisidee: Invariante der **while**-Schleife:
  Wenn $\langle u, v, w \rangle \notin \mathit{queue}$, dann u, v pfadkonsistent in Bezug auf w
- mögliche Optimierung: $\langle u, v, w \rangle$ immer nur dann in *queue* einfügen, wenn R_{uw} und R_{vw} nichttrivial
- Laufzeit $O(n^3 k^5)$ für n Variablen und maximale Wertebereichsgrösse k (Warum?)