

Grundlagen der Künstlichen Intelligenz

25. Constraint-Satisfaction-Probleme: Kantenkonsistenz

Malte Helmert

Universität Basel

17. April 2015

Constraint-Satisfaction-Probleme: Überblick

Kapitelüberblick Constraint-Satisfaction-Probleme:

- 22.–23. Einführung
- 24.–26. Kernalgorithmen
 - 24. Backtracking
 - 25. Kantenkonsistenz
 - 26. Pfadkonsistenz
- 27.–28. Problemstruktur

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** = **verschärfter Wertebereich** für v_2)
- $R_{v_1 v_2} = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\}$ kann verschärft werden zu $\{\langle 1, 2 \rangle\}$ (**verschärfter** binärer Constraint)
- $v_1 < v_3$ („neuer“ **binärer Constraint** = trivialer Constraint **verschärft**)

Trade-off Suche vs. Inferenz

Inferenz formal

Inferenz ist Ersetzen des gegebenen Constraint-Netzes durch ein **schärferes äquivalentes** Netz.

Trade-off:

- je **komplexer** die Inferenz ist und
- je **häufiger** sie angewendet wird,
- desto **weniger Suchknoten** müssen durchsucht werden, aber
- desto **mehr Zeitaufwand** wird **pro Suchknoten** benötigt

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 $\text{dom}(v) = \{d\}$ 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  $\alpha$  is inconsistent with  $\mathcal{C}$ :  
    return inconsistent
```

```
if  $\alpha$  is a total assignment:  
    return  $\alpha$ 
```

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

```
if  $\text{dom}'(v) \neq \emptyset$  for all variables  $v$ :  
    select some variable  $v$  for which  $\alpha$  is not defined  
    for each  $d \in \text{copy of } \text{dom}'(v)$  in some order:  
         $\alpha' := \alpha \cup \{v \mapsto d\}$   
         $\text{dom}'(v) := \{d\}$   
         $\alpha'' := \text{BacktrackingWithInference}(\mathcal{C}', \alpha')$   
        if  $\alpha'' \neq \text{inconsistent}$ :  
            return  $\alpha''$ 
```

```
return inconsistent
```

Backtracking mit Inferenz

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

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

```
if  $\alpha$  is a total assignment:  
    return  $\alpha$ 
```

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

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

```
    select some variable  $v$  for which  $\alpha$  is not defined
```

```
    for each  $d \in \text{copy of } \text{dom}'(v)$  in some order:
```

```
         $\alpha' := \alpha \cup \{v \mapsto d\}$ 
```

```
         $\text{dom}'(v) := \{d\}$ 
```

```
         $\alpha'' := \text{BacktrackingWithInference}(\mathcal{C}', \alpha')$ 
```

```
        if  $\alpha'' \neq \text{inconsistent}$ :
```

```
            return  $\alpha''$ 
```

```
return inconsistent
```

Backtracking mit Inferenz: Diskussion

- **inference** ist ein Platzhalter:
verschiedene Inferenzmethoden können eingesetzt 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

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 Kapitel

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

~> selten eine gute Idee, nicht mindestens
Forward Checking zu verwenden

Im Folgenden betrachten wir mächtigere Inferenzmethoden.

Kantenkonsistenz

Kantenkonsistenz: Definition

Definition (kantenkonsistent)

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

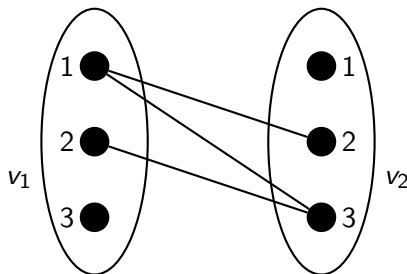
- (a) Eine Variable $v \in V$ ist **kantenkonsistent** in Bezug auf eine andere Variable $v' \in V$, wenn für jeden Wert $d \in \text{dom}(v)$ ein Wert $d' \in \text{dom}(v')$ mit $\langle d, d' \rangle \in R_{vv'}$ existiert.
- (b) Das Constraint-Netz $\mathcal{C}$ ist **kantenkonsistent**, wenn jede 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 $\text{dom}(v_1) = \text{dom}(v_2) = \{1, 2, 3\}$ 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 $\text{dom}(v)$, 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( $\mathcal{C}, v, v'$ ):
```

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

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

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

```
      remove  $d$  from  $\text{dom}(v)$ 
```

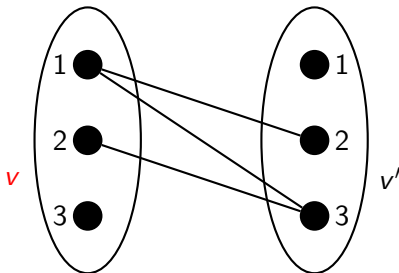
Eingabe: Constraint-Netz $\mathcal{C}$ und zwei Variablen v, v' von $\mathcal{C}$

Effekt: Macht v kantenkonsistent in Bezug auf v' .

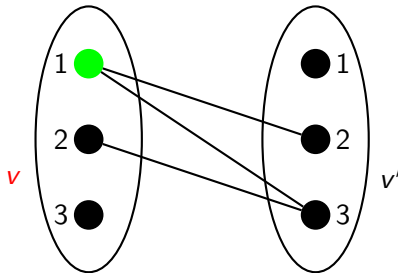
Alle verletzenden Werte werden aus $\text{dom}(v)$ entfernt.

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

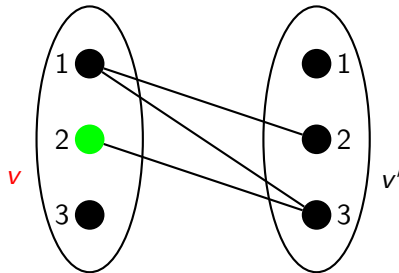
Beispiel: revise



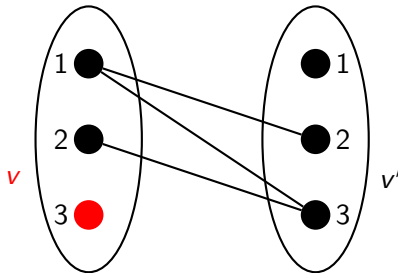
Beispiel: revise



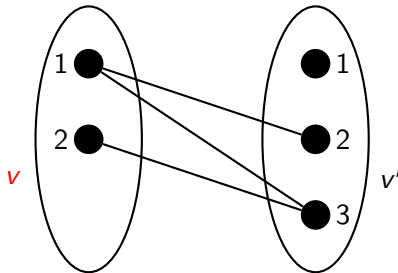
Beispiel: revise



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.

~> effizienterer Algorithmus: AC-3

Herstellen von Kantenkonsistenz: AC-3

Idee: merke **potenziell inkonsistente** Variablenpaare in Queue

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

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

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 an arbitrary 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)
- ~> effizient z. B. ein Stack
- AC-3 hat denselben Effekt wie AC-1: es stellt Kantenkonsistenz her
- **Beweisidee:** Invariante der **while**-Schleife:
Wenn $\langle u, v \rangle \notin queue$, 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 ein Paar $\langle u, v \rangle$, für das ein nichttrivialer Constraint R_{uv} oder R_{vu} existiert. (Es gibt maximal $2e$ viele solche Paare.)

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte ein Paar $\langle u, v \rangle$, für das ein nichttrivialer Constraint R_{uv} oder R_{vu} existiert. (Es gibt maximal $2e$ viele solche Paare.)

Jedes Mal, wenn das Paar in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich der zweiten beteiligten Variable reduziert.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte ein Paar $\langle u, v \rangle$, für das ein nichttrivialer Constraint R_{uv} oder R_{vu} existiert. (Es gibt maximal $2e$ viele solche Paare.)

Jedes Mal, wenn das Paar in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich der zweiten beteiligten Variable reduziert.

Das kann höchstens k mal passieren.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte ein Paar $\langle u, v \rangle$, für das ein nichttrivialer Constraint R_{uv} oder R_{vu} existiert. (Es gibt maximal $2e$ viele solche Paare.)

Jedes Mal, wenn das Paar in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich der zweiten beteiligten Variable reduziert.

Das kann höchstens k mal passieren.

Damit wird jedes Paar $\langle u, v \rangle$ höchstens $k + 1$ mal in die Queue eingefügt $\rightsquigarrow$ insgesamt höchstens $O(ek)$ Einfügeoperationen.

AC-3: Zeitaufwand (Beweis)

Beweis.

Betrachte ein Paar $\langle u, v \rangle$, für das ein nichttrivialer Constraint R_{uv} oder R_{vu} existiert. (Es gibt maximal $2e$ viele solche Paare.)

Jedes Mal, wenn das Paar in die Queue eingefügt wird (ausser beim ersten Mal), wurde zuvor der Wertebereich der zweiten beteiligten Variable reduziert.

Das kann höchstens k mal passieren.

Damit wird jedes Paar $\langle u, v \rangle$ höchstens $k + 1$ mal in die Queue eingefügt $\rightsquigarrow$ insgesamt höchstens $O(ek)$ Einfügeoperationen.

Dies begrenzt die Zahl der **while**-Iterationen auf $O(ek)$, so dass die revise-Aufrufe höchstens Zeit $O(ek) \cdot O(k^2) = O(ek^3)$ benötigen. □

Zusammenfassung

Zusammenfassung: Inferenz

- **Inferenz:** Herleiten zusätzlicher Constraints, die aus den bekannten Constraints logisch folgen
- ~> **schärferes äquivalentes** Constraint-Netz
- **Trade-off** Suche vs. Inferenz
- Inferenz als **Vorverarbeitung** oder **Integration** in Backtracking

Zusammenfassung: Forward Checking, Kantenkonsistenz

- billige und einfache Inferenz: **Forward Checking**
 - entferne Werte, die zu belegten Werten in Konflikt stehen
- teurer und mächtiger: **Kantenkonsistenz**
 - entferne wiederholt Werte, die keinen passenden „Partner“ bei einer anderen Variable haben, bis Fixpunkt erreicht
 - effiziente Implementierung über AC-3: $O(ek^3)$
mit e : #nichttriviale Constraints, k : Wertebereichsgrösse