Úvod do programování
s omezujícími podmínkami (pokračování)

Polynomiální a NP-úplné problémy

- **Polynomiální problémy**
  - existuje algoritmus polynomiální složitosti pro řešení problému
- **NP-úplné problémy**
  - řešitelné nedeterministickým polynomiálním algoritmem
  - potenciální řešení lze ověřit v polynomiálním čase
  - v nejhorším případě exponenciální složitost (pokud neplatí P=NP)

Složitost: polynomiální problémy

- **Lineární rovnice nad reálnými čísly**
  - proměnné nad doménami z $\mathbb{R}$, omezení: lineární rovnice
  - Gaussova eliminace
  - polynomiální složitost

Složitost: polynomiální problémy

- **Lineární rovnice nad reálnými čísly**
  - proměnné nad doménami z $\mathbb{R}$, omezení: lineární rovnice
  - Gaussova eliminace
  - polynomiální složitost
- **Lineární nerovnice nad reálnými čísly**
  - lineární programování, simplexová metoda
  - často stačí polynomiální složitost

Složitost: NP-úplné problémy

- **Boolean omezení**
  - 0/1 proměnné
  - omezení ≡ Boolean formule (konjunkce, disjunkce, implikace, . . .)
    - příklad: proměnné A, B, C, omezení $(A \vee B), (C \Rightarrow A)$, CSP problém $(A \vee B) \wedge (C \Rightarrow A)$

Složitost: NP-úplné problémy

- **Boolean omezení**
  - 0/1 proměnné
  - omezení $\equiv$ Boolean formule (konjunkce, disjunkce, implikace, ...)
    - příklad: proměnné A, B, C, omezení $(A \vee B), (C \Rightarrow A)$, CSP problém $(A \vee B) \wedge (C \Rightarrow A)$
  - problém splnitelnosti Boolean formule (SAT problém): NP-úplný
  - n proměnných:

Složitost: NP-úplné problémy

- **Boolean omezení**
  - 0/1 proměnné
  - omezení $\equiv$ Boolean formule (konjunkce, disjunkce, implikace, ...)
    - příklad: proměnné A, B, C, omezení $(A \vee B), (C \Rightarrow A)$, CSP problém $(A \vee B) \wedge (C \Rightarrow A)$
  - problém splnitelnosti Boolean formule (SAT problém): NP-úplný
  - n proměnných: 2^n možností

Složitost: NP-úplné problémy

- **Boolean omezení**
  - 0/1 proměnné
  - omezení $\equiv$ Boolean formule (konjunkce, disjunkce, implikace, ...)
    - příklad: proměnné A, B, C, omezení $(A \vee B), (C \Rightarrow A)$, CSP problém $(A \vee B) \wedge (C \Rightarrow A)$
  - problém splnitelnosti Boolean formule (SAT problém): NP-úplný
  - n proměnných: 2^n možností
- **Omezení nad konečnými doménami**
  - obecný CSP problém
  - problém splnitelnosti nad obecnými relacemi
  - NP-úplný problém
  - n proměnných, d maximální velikost domény:

Složitost: NP-úplné problémy

- **Boolean omezení**
  - 0/1 proměnné
  - omezení $\equiv$ Boolean formule (konjunkce, disjunkce, implikace, ...)
    - příklad: proměnné A, B, C, omezení $(A \vee B), (C \Rightarrow A)$, CSP problém $(A \vee B) \wedge (C \Rightarrow A)$
  - problém splnitelnosti Boolean formule (SAT problém): NP-úplný
  - n proměnných: 2^n možností
- **Omezení nad konečnými doménami**
  - obecný CSP problém
  - problém splnitelnosti nad obecnými relacemi
  - NP-úplný problém
  - n proměnných, d maximální velikost domény: d^n možností

Složitost a úplnost

Úplné vs. neúplné algoritmy

- úplný algoritmus prozkoumává množinu všech řešení
- neúplný algoritmus: neprozkoumává celou množinu řešení
  - **nevím** jako možná odpověd', ziskem může být efektivita
- příklad: neúplný polynomiální algoritmus pro NP-úplný problém

Složitost a úplnost

Úplné vs. neúplné algoritmy

- úplný algoritmus prozkoumává množinu všech řešení
- neúplný algoritmus: neprozkoumává celou množinu řešení
  - **nevím** jako možná odpověd', ziskem může být efektivita
- příklad: neúplný polynomiální algoritmus pro NP-úplný problém

Složitost řešiče

- Gaussova eliminace (P), SAT řešiče (NP), obecný CSP řešič (NP)

Složitost a úplnost

Úplné vs. neúplné algoritmy

- úplný algoritmus prozkoumává množinu všech řešení
- neúplný algoritmus: neprozkoumává celou množinu řešení
  - **nevím** jako možná odpověď, ziskem může být efektivita
- příklad: neúplný polynomiální algoritmus pro NP-úplný problém

Složitost řešiče

- Gaussova eliminace (P), SAT řešiče (NP), obecný CSP řešič (NP)

Složitost algoritmů propagace omezení

- většinou polynomiální neúplné algoritmy

Složitost a úplnost

- **Úplné vs. neúplné algoritmy**
  - úplný algoritmus prozkoumává množinu všech řešení
  - neúplný algoritmus: neprozkoumává celou množinu řešení
    - **nevím** jako možná odpověd', ziskem může být efektivita
  - příklad: neúplný polynomiální algoritmus pro NP-úplný problém
- **Složitost řešiče**
  - Gaussova eliminace (P), SAT řešiče (NP), obecný CSP řešič (NP)
- **Složitost algoritmů propagace omezení**
  - většinou polynomiální neúplné algoritmy
- **Složitost prohledávacích algoritmů**
  - úplné algoritmy, příklady: backtracking, generuj & testuj
  - neúplné algoritmy, neprohledávají celý prostor řešení, příklad: omezení času prohledávání

Grafová reprezentace CSP

- **Reprezentace podmínek**
  - intenzionální (matematická/logická formule)
  - extenzionální (výčet k-tic kompatibilních hodnot, 0-1 matice)

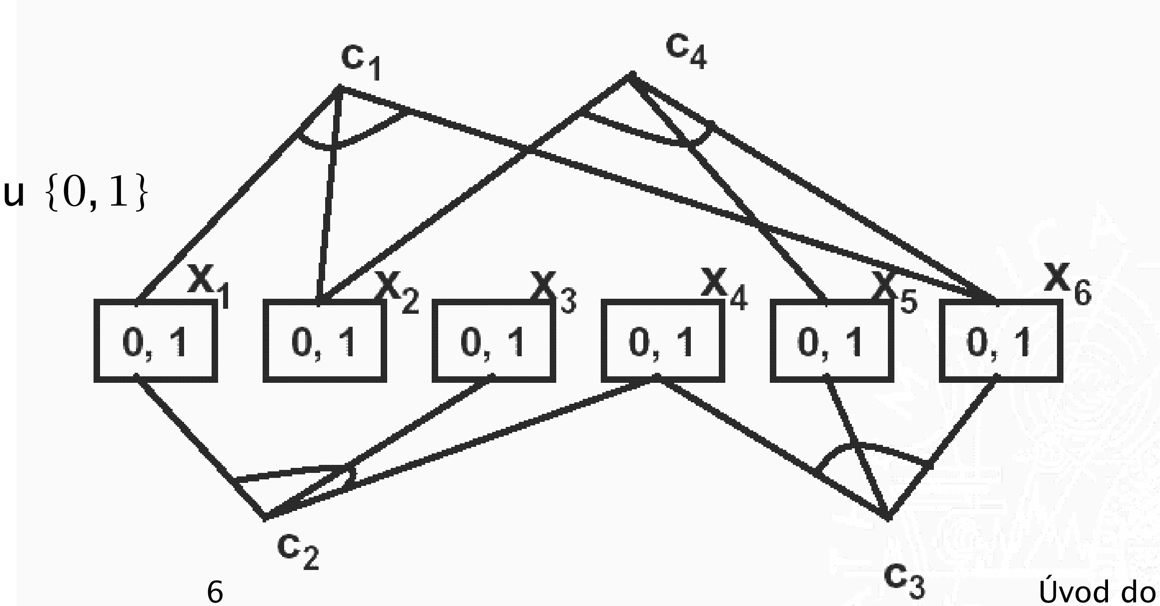
Grafová reprezentace CSP

- **Reprezentace podmínek**
  - intenzionální (matematická/logická formule)
  - extenzionální (výčet k-tic kompatibilních hodnot, 0-1 matice)
- **Graf:** vrcholy, hrany (hrana spojuje dva vrcholy)
- **Hypergraf:** vrcholy, hrany (hrana spojuje množinu vrcholů)
- Reprezentace CSP pomocí **hypergrafu podmínek**
  - vrchol = proměnná, hyperhrana = podmínka

Grafová reprezentace CSP

- **Reprezentace podmínek**
  - intenzionální (matematická/logická formule)
  - extenzionální (výčet k-tic kompatibilních hodnot, 0-1 matice)
- **Graf:** vrcholy, hrany (hrana spojuje dva vrcholy)
- **Hypergraf:** vrcholy, hrany (hrana spojuje množinu vrcholů)
- Reprezentace CSP pomocí **hypergrafu podmínek**
  - vrchol = proměnná, hyperhrana = podmínka
- Příklad
  - proměnné $x_1,\ldots,x_6$ s doménou $\{0,1\}$
  - omezení $c_1 : x_1 + x_2 + x_6 = 1$

$$c_2 : x_1 - x_3 + x_4 = 1$$
$$c_3 : x_4 + x_5 - x_6 > 0$$
$$c_4 : x_2 + x_5 - x_6 = 0$$



Binární CSP

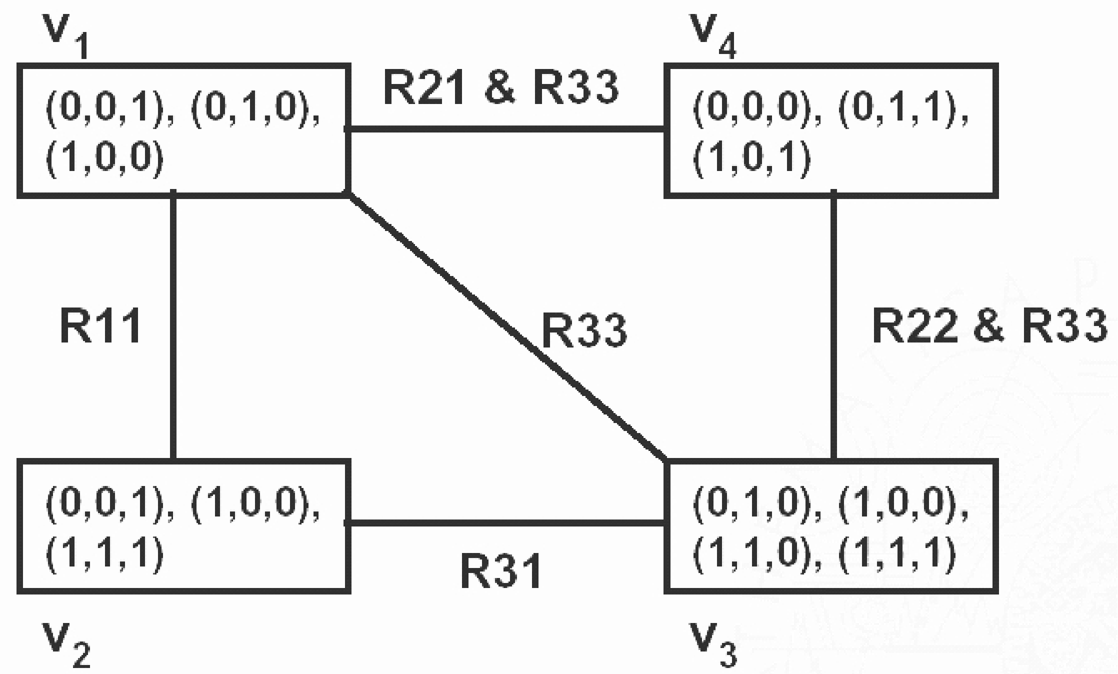
- **Binární CSP**
  - CSP, ve kterém jsou pouze binární podmínky
  - unární podmínky zakódovány do domény proměnné
- **Graf podmínek** pro binární CSP
  - není nutné uvažovat hypergraf, stačí graf (podmínka spojuje pouze dva vrcholy)
- **CSP lze transformovat na binární CSP**
- **Ekvivalence CSP**
  - dva CSP problémy jsou ekvivalentní, pokud mají stejnou množinu řešení
- **Rozšířená ekvivalence CSP**
  - řešení problémů lze mezi sebou „syntakticky” převést
  - např: obecný CSP převedeme na binární CSP a porovnáme tyto binární CSP

Duální problém

- **Duální problém**: původním omezením odpovídají nové duální proměnné
  - **proměnné:** k-ární podmínku c_i převedeme na duální proměnnou v_i s doménou obsahující konzistentní k-tice
  - **omezení:** pro každou dvojici podmínek c_i a c_j sdílející proměnné zavedeme binární podmínku R_{ij} mezi v_i a v_j omezující duální proměnné na k-tice, ve kterých mají sdílené proměnné stejnou hodnotu

- Příklad
  - proměnné $x_1, \ldots, x_6$ s doménou $\{0, 1\}$
  - omezení $c_1 : x_1 + x_2 + x_6 = 1 \ldots v_1$

$$c_2 : x_1 - x_3 + x_4 = 1 \ldots v_2$$
$$c_3 : x_4 + x_5 - x_6 > 0 \ldots v_3$$
$$c_4 : x_2 + x_5 - x_6 = 0 \ldots v_4$$



Použití binarizace

- Konstrukce duálního problému
  - jedna z možných metod binarizace
- Výhody binarizace
  - získáváme unifikovaný tvar CSP problému
  - řada algoritmů navržena pro binární CSP
  - pro výukové účely je vysvětlení na binárních CSP vhodné
    - algoritmy jsou přehlednější a jednodušší na pochopení
    - verze pro nebinární podmínky je často přímým rozšířením obecné verze
    - a tyto algoritmy jsou i aplikovatelné s pomocí binarizace na obecné CSP
- Ale: značné zvětšení velikosti problému

Použití binarizace

- Konstrukce duálního problému
  - jedna z možných metod binarizace
- Výhody binarizace
  - získáváme unifikovaný tvar CSP problému
  - řada algoritmů navržena pro binární CSP
  - pro výukové účely je vysvětlení na binárních CSP vhodné
    - algoritmy jsou přehlednější a jednodušší na pochopení
    - verze pro nebinární podmínky je často přímým rozšířením obecné verze
    - a tyto algoritmy jsou i aplikovatelné s pomocí binarizace na obecné CSP
- Ale: značné zvětšení velikosti problému
- Nebinární podmínky
  - složitější propagační algoritmy
  - lze využít jejich sémantiky pro lepší propagaci
    - příklad: `all_different` vs. množina binárních nerovností

Hranová konzistence

Propagace omezení

- Příklad:
  - proměnné: A,B,C
  - domény: $A \in \{0,1\}$ B=0 $C \in \{0,1,2,3\}$
  - omezení: $A \neq B$, $B \neq C$, $A \neq C$

  $\Longrightarrow$ A=1, B=0, $C \in \{2,3\}$

- Algoritmy pro **propagaci omezení** (**konzistenční algoritmy**)
  - umožňují odstranit nekonzistentní hodnoty z domén proměnných
  - zjednodušují problém
  - udržují ekvivalenci mezi původním a zjednodušeným problémem

Vrcholová konzistence

- **Vrcholová konzistence *(node consistency)* NC**
  - unární podmínky převedeme do domén proměnných
- **Vrchol** reprezentující V_i je **vrcholově konzistentní**:
  - každá hodnota z aktuální domény proměnné V_i splňuje všechny unární podmínky s proměnnou V_i

Vrcholová konzistence

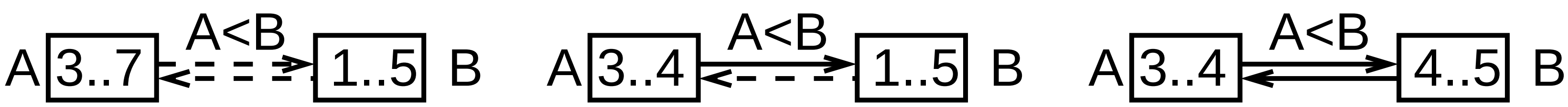
- **Vrcholová konzistence *(node consistency)* NC**
  - unární podmínky převedeme do domén proměnných
- **Vrchol** reprezentující V_i je **vrcholově konzistentní**:
  - každá hodnota z aktuální domény proměnné V_i splňuje všechny unární podmínky s proměnnou V_i
- **CSP problém** je **vrcholově konzistentní**:
  - každý jeho vrchol je vrcholově konzistentní

Hranová konzistence (*Arc Consistency* AC)

- Pouze pro **binární CSP** (až její rozšíření jsou pro nebinární CSP)
  - podmínka odpovídá hraně v grafu podmínek
  - více podmínek na jedné hraně převedeme do jedné podmínky
- **Hrana** (V_i, V_j) je **hranově konzistentní**, právě když
  pro každou hodnotu x z aktuální domény D_i existuje hodnota y v D_j tak, že ohodnocení $[V_i = x, V_j = y]$ splňuje **všechny binární podmínky** nad V_i, V_j.

Hranová konzistence (*Arc Consistency* AC)

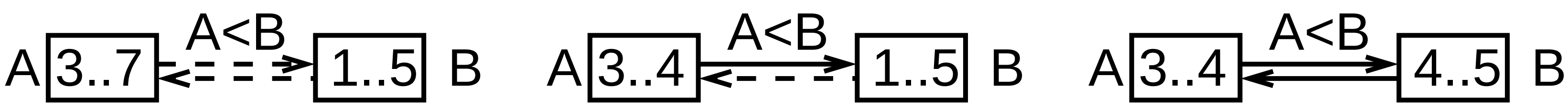
- Pouze pro **binární CSP** (až její rozšíření jsou pro nebinární CSP)
  - podmínka odpovídá hraně v grafu podmínek
  - více podmínek na jedné hraně převedeme do jedné podmínky
- **Hrana** (V_i, V_j) je **hranově konzistentní**, právě když
  pro každou hodnotu x z aktuální domény D_i existuje hodnota y v D_j tak, že
  ohodnocení $[V_i = x, V_j = y]$ splňuje **všechny binární podmínky** nad V_i, V_j.
- Hranová konzistence je **směrová**
  - konzistence hrany (V_i, V_j) nezaručuje konzistenci hrany (V_j, V_i)



konzistence (A,B) konzistence (A,B) i (B,A)

Hranová konzistence (*Arc Consistency* AC)

- Pouze pro **binární CSP** (až její rozšíření jsou pro nebinární CSP)
  - podmínka odpovídá hraně v grafu podmínek
  - více podmínek na jedné hraně převedeme do jedné podmínky
- **Hrana** (V_i, V_j) je **hranově konzistentní**, právě když
  pro každou hodnotu x z aktuální domény D_i existuje hodnota y v D_j tak, že
  ohodnocení $[V_i = x, V_j = y]$ splňuje **všechny binární podmínky** nad V_i, V_j.
- Hranová konzistence je **směrová**
  - konzistence hrany (V_i, V_j) nezaručuje konzistenci hrany (V_j, V_i)



konzistence (A,B)   konzistence (A,B) i (B,A)

- **CSP problém** je **hranově konzistentní**, právě když
  jsou všechny jeho hrany (v obou směrech) hranově konzistentní.

Algoritmus revize hrany

- Jak udělat hranu (V_i, V_j) hranově konzistentní?
- Z domény D_i vyřadím takové hodnoty x, které nejsou konzistentní s aktuální doménou D_j (pro x neexistuje žádná hodnota y v D_j tak, aby ohodnocení $V_i = x$ a $V_j = y$ splňovalo všechny binární podmínky mezi V_i a V_j)
- procedure revise((i, j))

```
procedure revise((i,j))
Deleted := false
for ∀ x ∈ D_i do
    if neexistuje y ∈ D_j takové, že (x,y) je konzistentní
    then D_i := D_i - {x}
        Deleted := true
    end if
return Deleted
end revise
```

V_1 in 2..4, V_2 in 2..4, $V_1 < V_2$

revise((1,2))

Algoritmus revize hrany

- Jak udělat hranu (V_i, V_j) hranově konzistentní?
- Z domény D_i vyřadím takové hodnoty x, které nejsou konzistentní s aktuální doménou D_j (pro x neexistuje žádná hodnota y v D_j tak, aby ohodnocení $V_i = x$ a $V_j = y$ splňovalo všechny binární podmínky mezi V_i a V_j)
- procedure revise((i, j))

  Deleted := false

  for $\forall x \in D_i$ do

  if neexistuje $y \in D_j$ takové, že (x, y) je konzistentní

  then D_i := D_i − $\{x\}$

  Deleted := true

  end if

  return Deleted

  end revise

V_1 in 2..4, V_2 in 2..4, $V_1 < V_2$

revise((1,2)) smaže 4 z D_1

D_2

Algoritmus revize hrany

- Jak udělat hranu (V_i, V_j) hranově konzistentní?
- Z domény D_i vyřadím takové hodnoty x, které nejsou konzistentní s aktuální doménou D_j (pro x neexistuje žádná hodnota y v D_j tak, aby ohodnocení $V_i = x$ a $V_j = y$ splňovalo všechny binární podmínky mezi V_i a V_j)
- procedure revise((i, j))

```
Deleted := false
for ∀ x ∈ D_i do
    if neexistuje y ∈ D_j takové, že (x,y) je konzistentní
    then D_i := D_i − {x}
         Deleted := true
    end if
return Deleted
end revise
```

V_1 in 2..4, V_2 in 2..4, $V_1 < V_2$

revise((1,2)) smaže 4 z D_1

D_2 se nezmění

- Složitost

Algoritmus revize hrany

- Jak udělat hranu (V_i, V_j) hranově konzistentní?
- Z domény D_i vyřadím takové hodnoty x, které nejsou konzistentní s aktuální doménou D_j (pro x neexistuje žádná hodnota y v D_j tak, aby ohodnocení $V_i = x$ a $V_j = y$ splňovalo všechny binární podmínky mezi V_i a V_j)
- procedure revise((i, j))

```
procedure revise((i,j))
Deleted := false
for ∀ x ∈ D_i do
    if neexistuje y ∈ D_j takové, že (x,y) je konzistentní
    then D_i := D_i − {x}
        Deleted := true
    end if
return Deleted
end revise
```

V_1 in 2..4, V_2 in 2..4, $V_1 < V_2$

revise((1,2)) smaže 4 z D_1

D_2 se nezmění

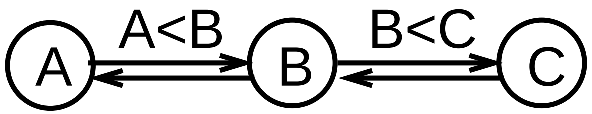
- Složitost $O(k^2)$ (k maximální velikost domény) $\Longleftarrow$ dva cykly: $x \in D_i$ a $y \in D_j$

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany

Algoritmus AC-1

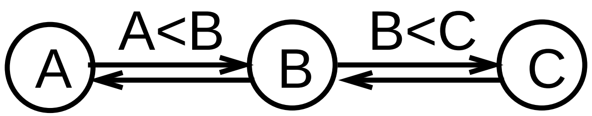
- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



A<B, B<C: ($\underline{3..7}$,1..5,1..5) $\xrightarrow{AB}$

Algoritmus AC-1

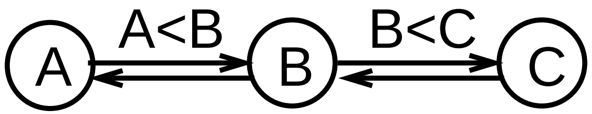
- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



$$A<B,\ B<C:\ (\underline{3..7},1..5,1..5) \overset{AB}{\rightarrow} (3..4,\underline{1..5},1..5) \overset{BA}{\rightarrow}$$

Algoritmus AC-1

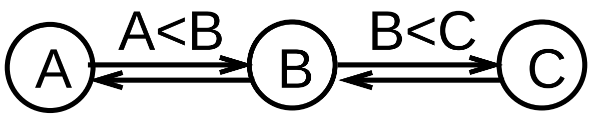
- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



$\mathtt{A<B,\ B<C:}\ (\underline{\mathtt{3..7}},\mathtt{1..5},\mathtt{1..5}) \xrightarrow{\mathrm{AB}} (\mathtt{3..4},\underline{\mathtt{1..5}},\mathtt{1..5}) \xrightarrow{\mathrm{BA}} (\mathtt{3..4},\underline{\mathtt{4..5}},\mathtt{1..5}) \xrightarrow{\mathrm{BC}}$

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany

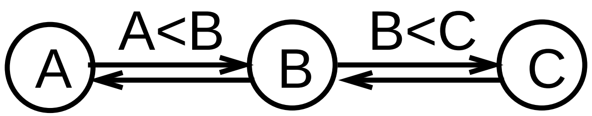


$A<B,\ B<C:\ (\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$

$(3..4,4,\underline{1..5}) \xrightarrow{CB}$

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany

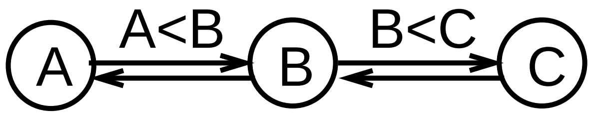


$A<B,\ B<C:\ (\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$

$(3..4,4,\underline{1..5}) \xrightarrow{CB} (\underline{3..4},4,5) \xrightarrow{AB}$

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany

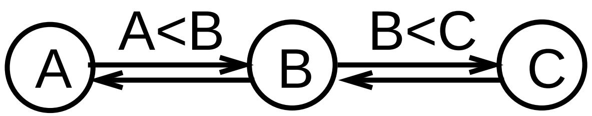


A<B, B<C: $(\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$

$(3..4,4,\underline{1..5}) \xrightarrow{CB} (\underline{3..4},4,5) \xrightarrow{AB} (3,4,5)$

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



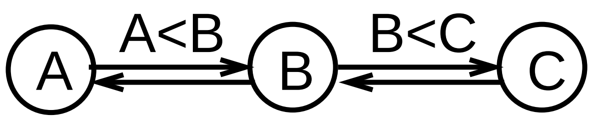
$$A<B,\ B<C:\ (\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$$
$$(3..4,4,\underline{1..5}) \xrightarrow{CB} (\underline{3..4},4,5) \xrightarrow{AB} (3,4,5)$$

- Revize hrany zmenší doménu ⇒ již zrevidované hrany opět nekonzistentní
- Revize hrany opakujeme, dokud dochází ke zmenšení nějaké domény
- procedure AC-1(G)

```
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1
```

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



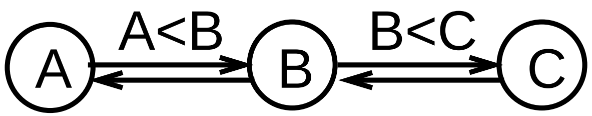
$\texttt{A<B, B<C:}\ (\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$
$(3..4,4,\underline{1..5}) \xrightarrow{CB} (\underline{3..4},4,5) \xrightarrow{AB} (3,4,5)$

- Revize hrany zmenší doménu ⇒ již zrevidované hrany opět nekonzistentní
- Revize hrany opakujeme, dokud dochází ke zmenšení nějaké domény
- procedure AC-1(G)

```
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1                    ⇒ AB, BA, BC, CB,
```

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



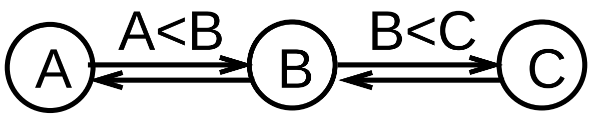
A<B, B<C: $(\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$
$(3..4,4,\underline{1..5}) \xrightarrow{CB} (\underline{3..4},4,5) \xrightarrow{AB} (3,4,5)$

- Revize hrany zmenší doménu ⇒ již zrevidované hrany opět nekonzistentní
- Revize hrany opakujeme, dokud dochází ke zmenšení nějaké domény
- procedure AC-1(G)

```
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1                      ⇒ AB, BA, BC, CB,  AB, BA, BC, CB,
```

Algoritmus AC-1

- Jak udělat CSP hranově konzistentní?
- Provedeme revizi každé hrany



$$\texttt{A<B, B<C: } (\underline{3..7},1..5,1..5) \xrightarrow{AB} (3..4,\underline{1..5},1..5) \xrightarrow{BA} (3..4,\underline{4..5},1..5) \xrightarrow{BC}$$
$$(3..4,4,\underline{1..5}) \xrightarrow{CB} (\underline{3..4},4,5) \xrightarrow{AB} (3,4,5)$$

- Revize hrany zmenší doménu ⇒ již zrevidované hrany opět nekonzistentní
- Revize hrany opakujeme, dokud dochází ke zmenšení nějaké domény
- procedure AC-1(G)

```
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1
```

⇒ AB, BA, BC, CB, AB, BA, BC, CB, AB, BA, BC, CB

Složitost AC-1

```
procedure AC-1(G)
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1
```

- k maximální velikost domény, e počet hran, n počet proměnných

Složitost AC-1

```
procedure AC-1(G)
repeat Changed := false
       for ∀ hranu (i, j) ∈ G do
           Changed := revise((i, j)) or Changed
until not(Changed)
end AC-1
```

- k maximální velikost domény, e počet hran, n počet proměnných
- Složitost $O(enk^3)$

Složitost AC-1

```
procedure AC-1(G)
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1
```

- k maximální velikost domény, e počet hran, n počet proměnných
- Složitost $O(enk^3)$
  - cyklus přes všechny hrany $O(e)$

Složitost AC-1

```
procedure AC-1(G)
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1
```

- k maximální velikost domény, e počet hran, n počet proměnných
- Složitost $O(enk^3)$
  - cyklus přes všechny hrany $O(e)$
  - revise $O(k^2)$

Složitost AC-1

```
procedure AC-1(G)
repeat Changed := false
       for ∀ hranu (i,j) ∈ G do
           Changed := revise((i,j)) or Changed
until not(Changed)
end AC-1
```

- k maximální velikost domény, e počet hran, n počet proměnných
- Složitost $O(enk^3)$
  - cyklus přes všechny hrany $O(e)$
  - revise $O(k^2)$
  - jeden cyklus smaže (v nejhorším případě) právě jednu hodnotu z domény proměnné, celkem nk hodnot (každá proměnná má v doméně až k hodnot)
    $\Longrightarrow O(nk)$

Neefektivita AC-1

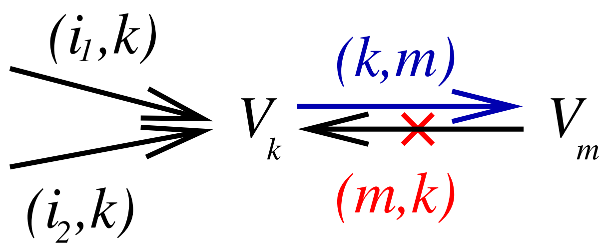
- I když zmenšíme jedinou doménu, provádí se revize všech hran.

  Tyto hrany ale revizí nemusí být vůbec zasaženy.

- Jaké hrany tedy revidovat po zmenšení domény?

Neefektivita AC-1

- I když zmenšíme jedinou doménu, provádí se revize všech hran.
  Tyto hrany ale revizí nemusí být vůbec zasaženy.
- Jaké hrany tedy revidovat po zmenšení domény?
  - ty, jejichž konzistence může být zmenšením domény proměnné narušena
  - jsou to hrany (i, k), které vedou do proměnné V_k se zmenšenou doménou



V_k ... proměnná se zmenšenou doménou

při revizi (k, m) došlo ke zmenšení domény V_k

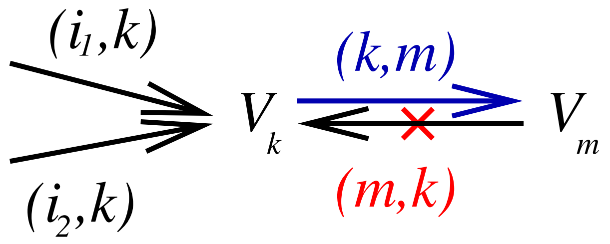
  - hranu (m, k) vedoucí z proměnné V_m, která zmenšení domény způsobila, není třeba revidovat (změna se jí nedotkne)

Neefektivita AC-1

- I když zmenšíme jedinou doménu, provádí se revize všech hran.

  Tyto hrany ale revizí nemusí být vůbec zasaženy.

- Jaké hrany tedy revidovat po zmenšení domény?
  - ty, jejichž konzistence může být zmenšením domény proměnné narušena
  - jsou to hrany (i, k), které vedou do proměnné V_k se zmenšenou doménou



V_k ... proměnná se zmenšenou doménou

při revizi (k, m) došlo ke zmenšení domény V_k

  - hranu (m, k) vedoucí z proměnné V_m, která zmenšení domény způsobila, není třeba revidovat (změna se jí nedotkne)
  - příklad: $V_k < V_m$, V_k in 1..10, V_m in 2..11,

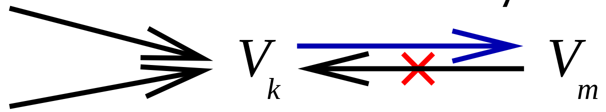
    smazání 11 z V_m, tj. V_k in 1..10, V_m in 2..10,

    důsledek: doména V_k se změní, smaže se 10, tj. V_k in 1..9, V_m in 2..10,

  a teď reaguji na změnu V_k revizemi (V_i, V_k): je tedy zbytečné dělat revizi (V_m, V_k)

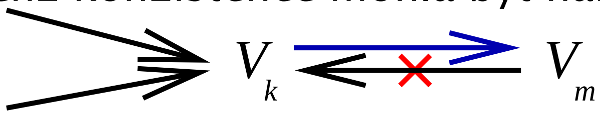
Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

  Q := $\{(i, j) \mid (i, j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

  while Q non empty do

      vyber a smaž (k, m) z Q

      if revise((k, m)) then % přidání pouze hran, které

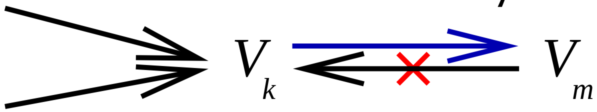
          Q := Q $\cup \{(i, k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

  end while

  end AC-3

Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

  Q := $\{(i,j) \mid (i,j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

  while Q non empty do

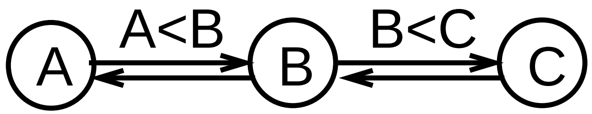
      vyber a smaž (k,m) z Q

      if revise((k,m)) then % přidání pouze hran, které

          Q := Q $\cup$ $\{(i,k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

  end while

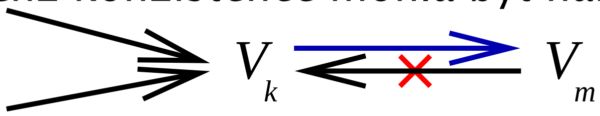
  end AC-3



- Příklad: iniciální fronta: AB, BA, BC, CB;

Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

  Q := $\{(i, j) \mid (i, j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

  while Q non empty do

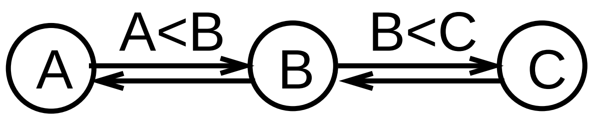
      vyber a smaž (k, m) z Q

      if revise((k, m)) then % přidání pouze hran, které

        Q := Q $\cup \{(i, k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

  end while

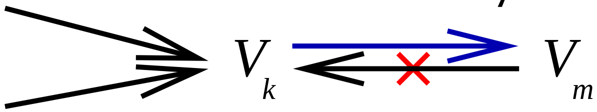
  end AC-3



- Příklad: iniciální fronta: AB, BA, BC, CB; revize AB nepřidá nic (BA odpovídá m,k);

Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

  Q := $\{(i,j) \mid (i,j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

  while Q non empty do

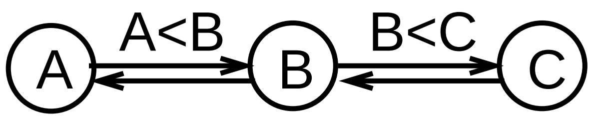
      vyber a smaž (k,m) z Q

      if revise((k,m)) then % přidání pouze hran, které

        Q := Q $\cup$ $\{(i,k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

  end while

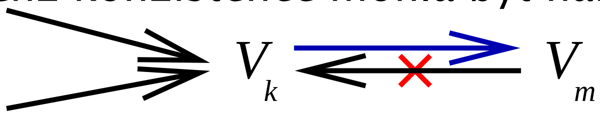
  end AC-3



- Příklad: iniciální fronta: AB, BA, BC, CB; revize AB nepřidá nic (BA odpovídá m,k); revize BA nepřidá nic (AB odpovídá m,k a CB už je ve frontě);

Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

Q := $\{(i,j) \mid (i,j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

while Q non empty do

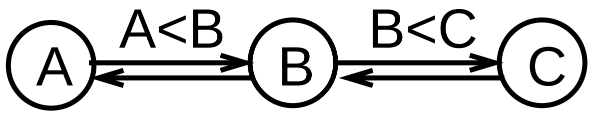
  vyber a smaž (k,m) z Q

  if revise((k,m)) then % přidání pouze hran, které

    Q := Q $\cup \{(i,k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

end while

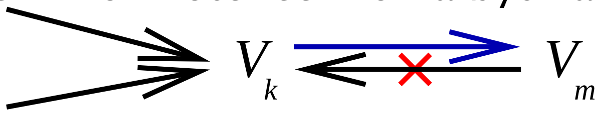
end AC-3



- Příklad: iniciální fronta: AB, BA, BC, CB; revize AB nepřidá nic (BA odpovídá m,k); revize BA nepřidá nic (AB odpovídá m,k a CB už je ve frontě); revize BC přidá AB (CB odpovídá m,k);

Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

  Q := $\{(i,j) \mid (i,j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

  while Q non empty do

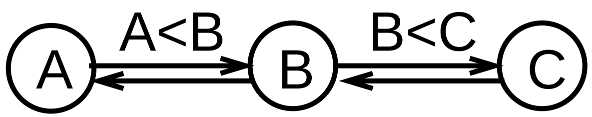
      vyber a smaž (k,m) z Q

      if revise((k,m)) then % přidání pouze hran, které

          Q := Q $\cup$ $\{(i,k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

  end while

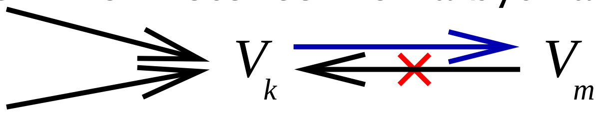
  end AC-3



- Příklad: iniciální fronta: AB, BA, BC, CB; revize AB nepřidá nic (BA odpovídá m,k); revize BA nepřidá nic (AB odpovídá m,k a CB už je ve frontě); revize BC přidá AB (CB odpovídá m,k); revize CB nepřidá nic (BC odpovídá m,k);

Algoritmus AC-3

- Opakování revizí můžeme dělat pomocí **fronty**
  - stačí jediná fronta hran, které je potřeba (znova) zrevidovat
  - přidáváme tam jen hrany, jejichž konzistence mohla být narušena zmenšením domény



- procedure AC-3(G)

  Q := $\{(i,j) \mid (i,j) \in$ hrany(G), $i \neq j\}$ % seznam hran pro revizi

  while Q non empty do

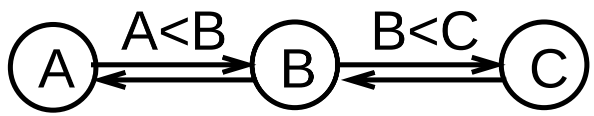
  vyber a smaž (k,m) z Q

  if revise((k,m)) then % přidání pouze hran, které

  Q := Q $\cup$ $\{(i,k) \in$ hrany(G), $i \neq k$, $i \neq m\}$ % dosud nejsou ve frontě

  end while

  end AC-3



- Příklad: iniciální fronta: AB, BA, BC, CB; revize AB nepřidá nic (BA odpovídá m,k); revize BA nepřidá nic (AB odpovídá m,k a CB už je ve frontě); revize BC přidá AB (CB odpovídá m,k); revize CB nepřidá nic (BC odpovídá m,k); revize AB nepřidá nic (BA odpovídá m,k)
- Jaké budou domény A,B,C po AC-3 pro: `A,B,C in 1..10, A < B + 1, C < B`

Složitost AC-3

```
procedure AC-3(G)
Q := {(i,j) | (i,j) ∈ hrany(G), i ≠ j}     // seznam hran pro revizi
while Q non empty do
     vyber a smaž (k,m) z Q
     if revise((k,m)) then
        Q := Q ∪ {(i,k) ∈ hrany(G), i ≠ k, i ≠ m}
```

- k maximální velikost domény, e počet hran

Složitost AC-3

```
procedure AC-3(G)
Q := {(i,j) | (i,j) ∈ hrany(G), i ≠ j}     // seznam hran pro revizi
while Q non empty do
     vyber a smaž (k,m) z Q
     if revise((k,m)) then
        Q := Q ∪ {(i,k) ∈ hrany(G), i ≠ k, i ≠ m}
```

- k maximální velikost domény, e počet hran
- Složitost $O(ek^3)$

Složitost AC-3

```
procedure AC-3(G)
Q := {(i,j) | (i,j) ∈ hrany(G), i ≠ j}     // seznam hran pro revizi
while Q non empty do
     vyber a smaž (k,m) z Q
     if revise((k,m)) then
        Q := Q ∪ {(i,k) ∈ hrany(G), i ≠ k, i ≠ m}
```

- k maximální velikost domény, e počet hran
- Složitost $O(ek^3)$
  - revise $O(k^2)$

Složitost AC-3

```
procedure AC-3(G)
Q := {(i,j) | (i,j) ∈ hrany(G), i ≠ j}     // seznam hran pro revizi
while Q non empty do
     vyber a smaž (k,m) z Q
     if revise((k,m)) then
        Q := Q ∪ {(i,k)  ∈ hrany(G), i ≠ k, i ≠ m}
```

- k maximální velikost domény, e počet hran
- Složitost $O(ek^3)$
  - revise $O(k^2)$
  - celkem e hran/omezení $O(e)$

Složitost AC-3

```
procedure AC-3(G)
Q := {(i,j) | (i,j) ∈ hrany(G), i ≠ j}     // seznam hran pro revizi
while Q non empty do
     vyber a smaž (k,m) z Q
     if revise((k,m)) then
        Q := Q ∪ {(i,k)  ∈ hrany(G), i ≠ k, i ≠ m}
```

- k maximální velikost domény, e počet hran
- Složitost $O(ek^3)$
  - revise $O(k^2)$
  - celkem e hran/omezení $O(e)$
  - každé omezení může být ve frontě maximálně $2k$ krát $\Longrightarrow O(k)$
    - jakmile je omezení přidáno do fronty, doména (k) jedné z jeho dvou proměnných (2) byla redukována alespoň o jednu hodnotu
- Technika AC-3 je dnes asi nejpoužívánější, ale stále není optimální

Podpora hodnoty

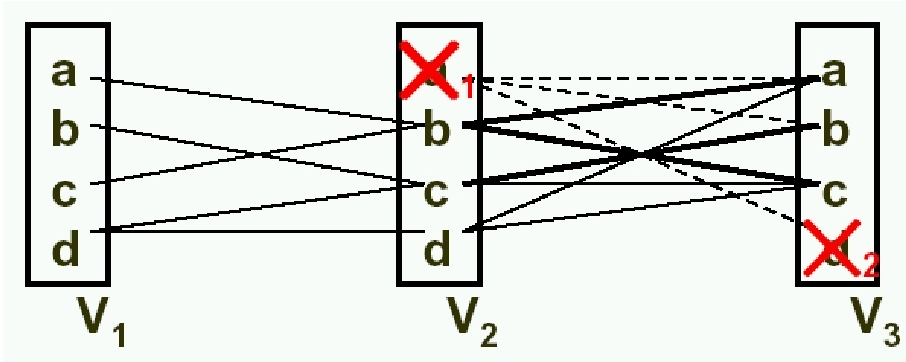
- AC-3: při každé revizi hrany testujeme množství dvojic hodnot na konzistenci vzhledem k podmínce.

  Tyto testy znova opakujeme při každé další revizi hrany.

Podpora hodnoty

- AC-3: při každé revizi hrany testujeme množství dvojic hodnot na konzistenci vzhledem k podmínce.

  Tyto testy znova opakujeme při každé další revizi hrany.

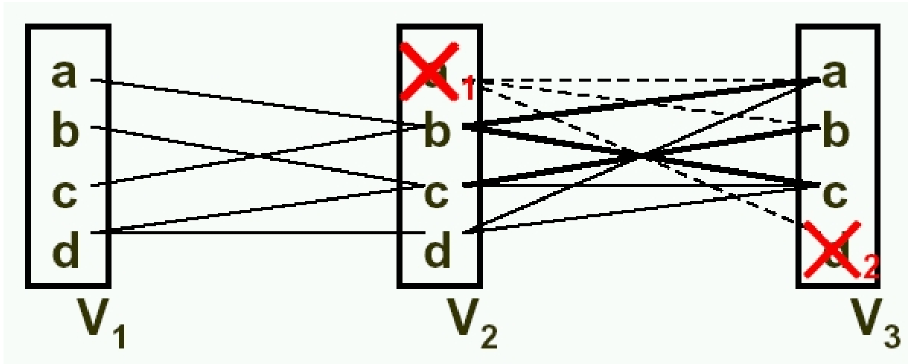


Podpora hodnoty

- AC-3: při každé revizi hrany testujeme množství dvojic hodnot na konzistenci vzhledem k podmínce.

  Tyto testy znova opakujeme při každé další revizi hrany.

  - Při revizi hrany (V_2, V_1) vyřadíme hodnotu a z domény proměnné V_2.
  - Nyní musíme prozkoumat doménu V_3, zda některá z hodnot a, b, c, d neztratila svoji podporu ve V_2.

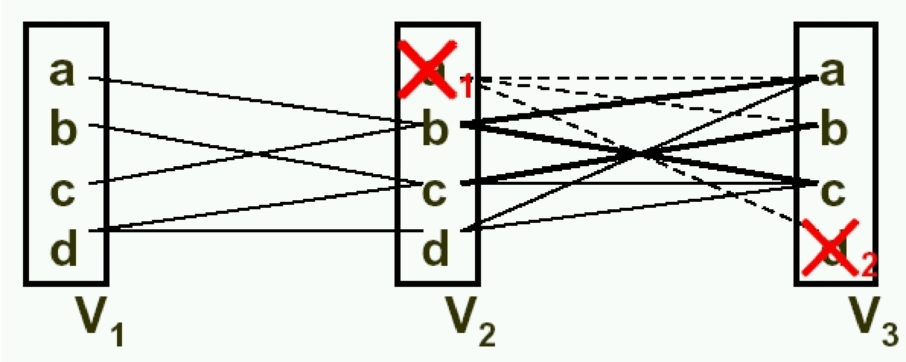


Podpora hodnoty

- AC-3: při každé revizi hrany testujeme množství dvojic hodnot na konzistenci vzhledem k podmínce.

  Tyto testy znova opakujeme při každé další revizi hrany.

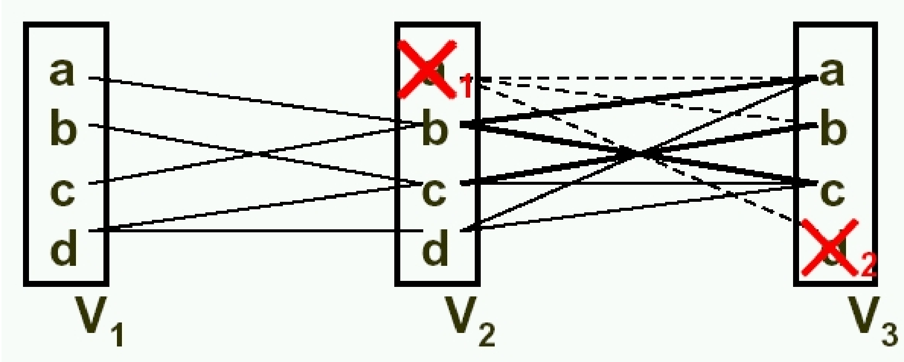
  - Při revizi hrany (V_2, V_1) vyřadíme hodnotu a z domény proměnné V_2.
  - Nyní musíme prozkoumat doménu V_3, zda některá z hodnot a, b, c, d neztratila svoji podporu ve V_2.



  - Hodnoty a, b, c není třeba znova kontrolovat $\Longleftarrow$ mají ve V_2 také jinou podporu než a.

Podpora hodnoty

- AC-3: při každé revizi hrany testujeme množství dvojic hodnot na konzistenci vzhledem k podmínce.
  Tyto testy znova opakujeme při každé další revizi hrany.



  - Při revizi hrany (V_2, V_1) vyřadíme hodnotu a z domény proměnné V_2.
  - Nyní musíme prozkoumat doménu V_3, zda některá z hodnot a, b, c, d neztratila svoji podporu ve V_2.
  - Hodnoty a, b, c není třeba znova kontrolovat $\Longleftarrow$ mají ve V_2 také jinou podporu než a.

- **Podpora** (*support*) pro $a \in D_i = \{\langle j, b\rangle \mid b \in D_j, (a, b) \in c_{i,j}\}$
  - $a \in D_2$ má podpory $\{\langle 3, a\rangle, \langle 3, b\rangle, \langle 3, d\rangle\}$
  - $d \in D_3$ má pouze jedinou podporu $\{\langle 2, a\rangle\}$
  - $b \in D_2$ má podpory $\{\langle 1, a\rangle, \langle 1, c\rangle, \langle 3, a\rangle, \langle 3, c\rangle\}$

- Podpory spočítáme jen jednou. Při opakovaných revizích je budeme používat.

Algoritmus na inicializaci podpor

- Udržujeme seznam hodnot, které sami podporujeme (víme komu říct, když zmizíme).

  S$_{j,b}$: množina dvojic $\langle i, a \rangle$, pro které je $\langle j, b \rangle$ podporou

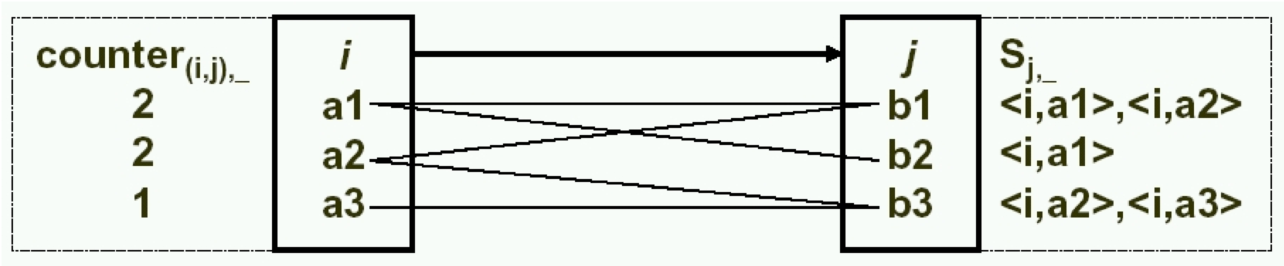
- Udržujeme počet vlastních podpor (víme, co nám chybí).

  counter[$(i, j), a$**]:** počet podpor, které má hodnota $a \in D_i$ u proměnné V_j

```
procedure initialize(G)
Q := {}, S := {}                        // vyprázdnění datových struktur
for each (V_i,V_j) ∈ hrany(G) do
    for each a ∈ D_i do total := 0
        for each b ∈ D_j do
            if (a,b) konzistentní vzhledem k c_i,j then
               total := total+1
               S_j,b := S_j,b ∪ {⟨i,a⟩}
        counter[(i,j),a] := total
        if counter[(i,j),a] = 0 then smaž a z D_i
                                     Q := Q ∪ {⟨i,a⟩}
return Q    // Q je fronta se smazanými hodnotami
```

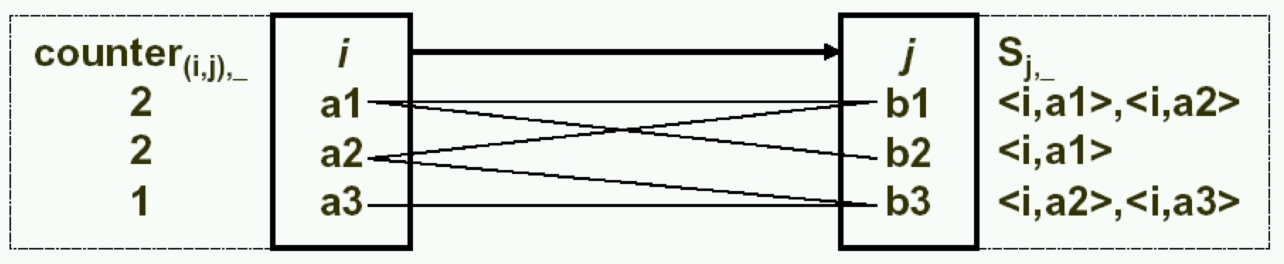
Aktualizace podpor během výpočtu

- Situace po zpracování hrany (V_i, V_j) algoritmem initialize



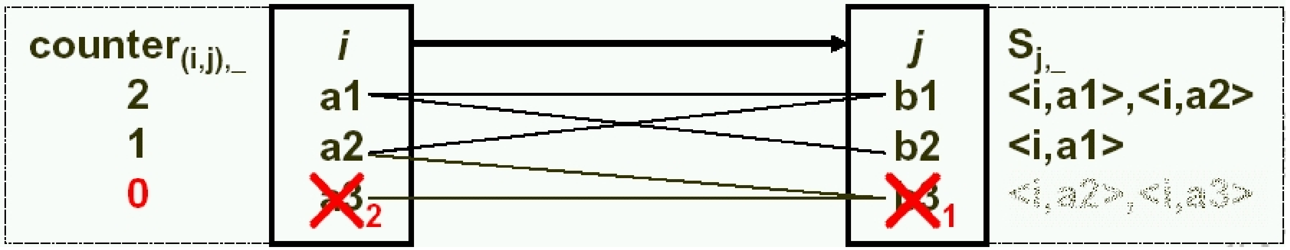
Aktualizace podpor během výpočtu

- Situace po zpracování hrany (V_i, V_j) algoritmem initialize



- Využití struktur s podporami:

1. Předpokládejme, že $b3$ je vyřazena z domény V_j.
2. Zjistíme v $S_{j,b3}$, pro které hodnoty je $b3$ podporou (tj. $\langle i, a2\rangle, \langle i, a3\rangle$).
3. Snížíme `counter` u těchto hodnot (odstraníme jim jednu podporu).
4. Pokud je některý `counter` vynulován ($a3$), potom příslušnou hodnotu vyřadíme z domény a pokračujeme s ní od kroku (1).



Algoritmus AC-4

```
procedure AC-4(G)
Q := initialize(G)
while not empty(Q) do
      vyber a smaž libovolný ⟨j,b⟩ ∈ Q
      for each ⟨i,a⟩ ∈ S_{j,b} do
          counter[(i,j),a] := counter[(i,j),a] - 1
          if (counter[(i,j),a] = 0) ∧ (a ∈ D_i) then
             smaž a z D_i
             Q := Q ∪ {⟨i,a⟩}
```

Algoritmus AC-4

```
procedure AC-4(G)
Q := initialize(G)
while not empty(Q) do
      vyber a smaž libovolný ⟨j,b⟩ ∈ Q
      for each ⟨i,a⟩ ∈ S_{j,b} do
          counter[(i,j),a] := counter[(i,j),a] - 1
          if (counter[(i,j),a] = 0) ∧ (a ∈ D_i) then
             smaž a z D_i
             Q := Q ∪ {⟨i,a⟩}
```

- Složitost $O(ek^2)$ $\Longrightarrow$ algoritmus optimální v nejhorším případě
  - složitost `initialize`

Algoritmus AC-4

```
procedure AC-4(G)
Q := initialize(G)
while not empty(Q) do
      vyber a smaž libovolný ⟨j,b⟩ ∈ Q
      for each ⟨i,a⟩ ∈ S_{j,b} do
          counter[(i,j),a] := counter[(i,j),a] - 1
          if (counter[(i,j),a] = 0) ∧ (a ∈ D_i) then
             smaž a z D_i
             Q := Q ∪ {⟨i,a⟩}
```

- Složitost $O(ek^2)$ $\Longrightarrow$ algoritmus optimální v nejhorším případě
  - složitost `initialize` $O(ek^2)$ $\Longleftarrow$ $(V_i, V_j) \in$ hrany(G), $a \in D_i$, $b \in D_j$
  - složitost while cyklu

Algoritmus AC-4

```
procedure AC-4(G)
Q := initialize(G)
while not empty(Q) do
      vyber a smaž libovolný ⟨j,b⟩ ∈ Q
      for each ⟨i,a⟩ ∈ S_{j,b} do
          counter[(i,j),a] := counter[(i,j),a] - 1
          if (counter[(i,j),a] = 0) ∧ (a ∈ D_i) then
             smaž a z D_i
             Q := Q ∪ {⟨i,a⟩}
```

- Složitost $O(ek^2)$ $\Longrightarrow$ algoritmus optimální v nejhorším případě
  - složitost `initialize` $O(ek^2)$ $\Longleftarrow$ $(V_i, V_j) \in$ hrany(G), $a \in D_i$, $b \in D_j$
  - složitost while cyklu $O(ek^2)$:

    postupně musím odebrat všechny podpory a těch je $O(ek^2)$

Algoritmus AC-4

```
procedure AC-4(G)
Q := initialize(G)
while not empty(Q) do
      vyber a smaž libovolný ⟨j,b⟩ ∈ Q
      for each ⟨i,a⟩ ∈ S_{j,b} do
          counter[(i,j),a] := counter[(i,j),a] - 1
          if (counter[(i,j),a] = 0) ∧ (a ∈ D_i) then
             smaž a z D_i
             Q := Q ∪ {⟨i,a⟩}
```

- Složitost $O(ek^2) \Longrightarrow$ algoritmus optimální v nejhorším případě
  - složitost initialize $O(ek^2) \Longleftarrow (V_i, V_j) \in$ hrany(G), $a \in D_i$, $b \in D_j$
  - složitost while cyklu $O(ek^2)$:

    postupně musím odebrat všechny podpory a těch je $O(ek^2)$
- Paměťová náročnost, není nejlepší v průměrném případě (inicializace zůstává)

Další AC algoritmy

- Existuje řada dalších algoritmů pro zajištění hranové konzistence
  - AC-5, AC-6, AC-7, ...
- **AC-6** *(Bessière, Cordier)*
  - zlepšuje paměťovou náročnost a průměrný čas AC-4
  - drží si pouze jednu podporu, další podpory hledá až při ztrátě aktuální podpory
- **AC-3.1**: AC-3 hledá podpory vždy od začátku, jak to vylepšit?
- **AC-2001**: AC-3 s frontou proměnných místo fronty omezení
- Porovnání:
  - AC-3 není (teoreticky) optimální
  - AC-4 je (teoreticky) optimální, ale (prakticky) pomalý
  - AC-6/7 jsou (prakticky) rychlejší než AC-4, ale složité
  - AC-2001: v praxi je časté využití fronty proměnných

AC-3.1: optimální algoritmus AC-3

- Co je na AC-3 neefektivní?

  hledání podpor v REVISE, které vždy začíná od nuly!

  if „neexistuje $y \in D_j$ takové, že (x, y) je konzistentní” then

- **AC-3.1** *(Zhang, Yap)*
  - běh stejný jako u AC-3
  - pro každou hodnotu si pamatuje poslední nalezenou podporu (last) v každém směru a hledání začíná u ní

procedure EXIST((i,x),j)

y := $last((i,x),j)$

if $y \in D_j$ then return true

while y := $next(y,D_j)$ $\wedge$ $y \neq nil$ do

    if $(x,y) \in c_{i,j}$ then   % $c_{i,j}$ omezení s proměnými i, j

        $last((i,x),j)$:= y

        return true

return false

AC-2001: jiný optimální algoritmus AC-3

procedure AC-2001(G) — Používá verzi AC-3 **s frontou proměnných**

Q := $\{ i \mid i \in vrcholy(G)\}$ % seznam vrcholů pro revizi

while neprazdna(Q) do

    vyber a smaž j z Q

    for $\forall i \in vrcholy(G)$ takové, že $(i, j) \in hrany(G)$ do

        if REVISE2001(i, j) then

            if $D_i = \emptyset$ then return fail

            Q := $Q \cup \{i\}$

return true

AC-2001: jiný optimální algoritmus AC-3

Používá verzi AC-3 s frontou proměnných

procedure AC-2001(G)

Q := $\{i \mid i \in vrcholy(G)\}$ % seznam vrcholů pro revizi

while neprazdna(Q) do

  vyber a smaž j z Q

  for $\forall i \in vrcholy(G)$ takové, že $(i, j) \in hrany(G)$ do

    if REVISE2001(i, j) then

      if $D_i = \emptyset$ then return fail

      Q := $Q \cup \{i\}$

return true

procedure REVISE2001(i, j)

DELETED := false

for $\forall x \in D_i$ do

  if $last((i,x),j) \notin D_j$ then

    if $\exists y \in D_j \wedge y > last((i,x),j) \wedge (x,y) \in c_{i,j}$

    then $last((i,x),j)$:= y

    else smaž x z D_i; DELETED := true

return DELETED

Algoritmus fakticky pracuje s rozdílovými množinami, tj. pro každou proměnnou si pamatuje, jaké hodnoty byly smazány z domény od poslední revize (podobně jako AC-3.1)

Je hranová konzistence dostatečná (úplná)?

- Použitím AC odstraníme mnoho nekompatibilních hodnot
  - Dostaneme potom řešení problému? NE
  - Víme alespoň zda řešení existuje? NE

Je hranová konzistence dostatečná (úplná)?

- Použitím AC odstraníme mnoho nekompatibilních hodnot
  - Dostaneme potom řešení problému? NE
  - Víme alespoň zda řešení existuje? NE
- $X, Y, Z \in \{1, 2\}, \quad X \neq Y, \quad Y \neq Z, \quad Z \neq X$
  - hranově konzistentní
  - nemá žádné řešení

Je hranová konzistence dostatečná (úplná)?

- Použitím AC odstraníme mnoho nekompatibilních hodnot
  - Dostaneme potom řešení problému? NE
  - Víme alespoň zda řešení existuje? NE
- $X, Y, Z \in \{1, 2\}, \quad X \neq Y, \quad Y \neq Z, \quad Z \neq X$
  - hranově konzistentní
  - nemá žádné řešení
- Jaký je tedy význam AC?
  - někdy dá řešení přímo
    - nějaká doména se vyprázdní ⇒ řešení neexistuje
    - všechny domény jsou jednoprvkové ⇒ máme řešení
  - v obecném případě se alespoň zmenší prohledávaný prostor

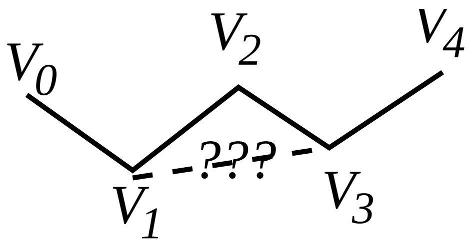
Konzistence po cestě

Konzistence po cestě (PC *path consistency*)

- Příklad: $X, Y, Z \in \{1, 2\}, \quad X \neq Y, \quad Y \neq Z, \quad Z \neq X$
- Jak posílit konzistenci? Budeme se zabývat několika podmínkami najednou.
- **Cesta ($V_0, V_1, \ldots, V_m$) je konzistentní** právě tehdy, když pro každou dvojici hodnot $x \in D_0$ a $y \in D_m$ splňující binární podmínky na hraně V_0, V_m existuje ohodnocení proměnných $V_1, \ldots V_{m-1}$ takové, že všechny binární podmínky mezi sousedy V_j, V_{j+1} jsou splněny.
- CSP je **konzistentní po cestě**, právě když jsou všechny cesty konzistentní.

Konzistence po cestě (PC *path consistency*)

- Příklad: $X, Y, Z \in \{1, 2\}, \quad X \neq Y, \quad Y \neq Z, \quad Z \neq X$
- Jak posílit konzistenci? Budeme se zabývat několika podmínkami najednou.
- **Cesta ($V_0, V_1, \ldots, V_m$) je konzistentní** právě tehdy, když pro každou dvojici hodnot $x \in D_0$ a $y \in D_m$ splňující binární podmínky na hraně V_0, V_m existuje ohodnocení proměnných $V_1, \ldots V_{m-1}$ takové, že všechny binární podmínky mezi sousedy V_j, V_{j+1} jsou splněny.
- CSP je **konzistentní po cestě**, právě když jsou všechny cesty konzistentní.
- Definice PC nezaručuje, že jsou splněny všechny podmínky nad vrcholy cesty, zabývá se pouze podmínkami mezi sousedy



PC a cesty délky 2

- Zjišťování konzistence všech cest není efektivní
- Stačí ověřit konzistenci cest délky 2
- Věta: **CSP je PC právě tehdy, když každá cesta délky 2 je PC.**

PC a cesty délky 2

- Zjišťování konzistence všech cest není efektivní
- Stačí ověřit konzistenci cest délky 2
- Věta: **CSP je PC právě tehdy, když každá cesta délky 2 je PC.**
- Důkaz: 1) PC ⇒ cesty délky 2 jsou PC

  2) cesty délky 2 jsou PC ⇒ $\forall n$ cesty délky n jsou PC ⇒ PC   indukcí podle délky cesty

  a) $n = 2$ platí triviálně

  b) $n + 1$ (za předpokladu, že platí pro n)

  i) vezmeme libovolných $n + 2$ vrcholů $V_0, V_1, \ldots, V_{n+1}$

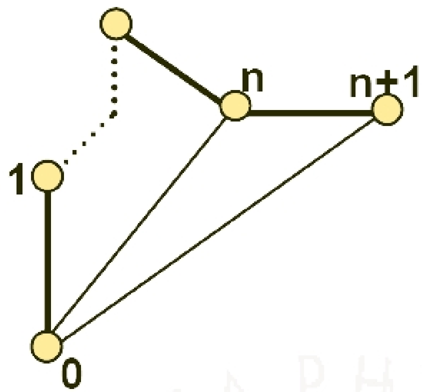
  ii) vezmeme libovolné dvě kompatibilní hodnoty $x_0 \in D_0$ a $x_{n+1} \in D$

  (kompatibilní = splňující všechny bin.podmínky mezi x_0 a x_{n+1})

  iii) podle a) jsou všechny cesty délky 2 PC, a tedy i V_0, V_n, V_{n+1} je PC

  najdeme tedy $x_n \in D_n$ tak, že (x_0, x_n) a (x_n, x_{n+1}) jsou konzistentní

  iv) podle indukčního kroku najdeme zbylé hodnoty na cestě $V_0, V_1, \ldots, V_n$



PC a cesty délky 2

- Zjišťování konzistence všech cest není efektivní
- Stačí ověřit konzistenci cest délky 2
- Věta: **CSP je PC právě tehdy, když každá cesta délky 2 je PC.**
- Důkaz: 1) PC $\Rightarrow$ cesty délky 2 jsou PC

  2) cesty délky 2 jsou PC $\Rightarrow \forall n$ cesty délky n jsou PC $\Rightarrow$ PC indukcí podle délky cesty

  a) $n = 2$ platí triviálně

  b) $n + 1$ (za předpokladu, že platí pro n)

  i) vezmeme libovolných $n + 2$ vrcholů $V_0, V_1, \ldots, V_{n+1}$

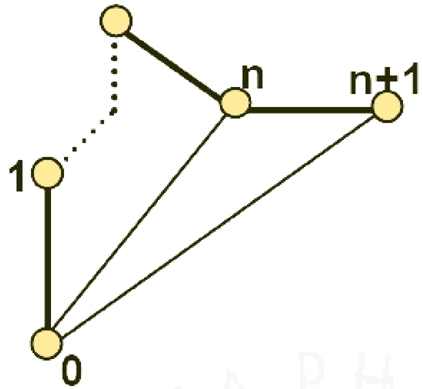
  ii) vezmeme libovolné dvě kompatibilní hodnoty $x_0 \in D_0$ a $x_{n+1} \in D$

  (kompatibilní = splňující všechny bin.podmínky mezi x_0 a x_{n+1})

  iii) podle a) jsou všechny cesty délky 2 PC, a tedy i V_0, V_n, V_{n+1} je PC

  najdeme tedy $x_n \in D_n$ tak, že (x_0, x_n) a (x_n, x_{n+1}) jsou konzistentní

  iv) podle indukčního kroku najdeme zbylé hodnoty na cestě $V_0, V_1, \ldots, V_n$



- Definici PC lze tedy upravit tak, že vyžadujeme pouze konzistenci cest délky 2

Vztah PC a AC

- PC ⇒ AC
  - pokud je cesta (i, j, i) konzistentní (PC),
    pak je i hrana (i, j) konzistentní (AC), tj. z PC tedy plyne AC
    - PC: ke každé „dvojici hodnot” pro i, i najdu hodnotu v j

    ⇒ AC: ke každé hodnotě v i tedy najdu hodnotu v j

Vztah PC a AC

- PC $\Rightarrow$ AC
  - pokud je cesta (i, j, i) konzistentní (PC),
    pak je i hrana (i, j) konzistentní (AC), tj. z PC tedy plyne AC
    - PC: ke každé „dvojici hodnot” pro i, i najdu hodnotu v j

    $\Rightarrow$ AC: ke každé hodnotě v i tedy najdu hodnotu v j
- AC $\not\Rightarrow$ PC
  - příklad: X, Y, Z $\in \{1, 2\}$, X $\neq$ Y, Y $\neq$ Z, Z $\neq$ X
    - je AC, ale není PC, X=0, Y=1 nelze rozšířit po cestě (X,Z,Y)

Vztah PC a AC

- PC $\Rightarrow$ AC
  - pokud je cesta (i, j, i) konzistentní (PC),
    pak je i hrana (i, j) konzistentní (AC), tj. z PC tedy plyne AC
    - PC: ke každé „dvojici hodnot" pro i, i najdu hodnotu v j

    $\Rightarrow$ AC: ke každé hodnotě v i tedy najdu hodnotu v j
- AC $\not\Rightarrow$ PC
  - příklad: X, Y, Z $\in \{1, 2\}$, X $\neq$ Y, Y $\neq$ Z, Z $\neq$ X
    - je AC, ale není PC, X=0, Y=1 nelze rozšířit po cestě (X,Z,Y)
- AC vyřazuje nekompatibilní prvky z domény proměnných. Co dělá PC?
  - PC vyřazuje dvojice hodnot
  - PC si pamatuje podmínky explicitně
  - PC si pamatuje, které dvojice hodnot jsou v relaci
  - PC dělá všechny relace nad dvojicemi implicitní (A<B, B<C $\Rightarrow$ A+1<C)