PA163 Programování s omezujícími podmínkami

podzim 2019

Základní informace

- **Web předmětu:** na IS
  - **průsvitky** průběžně na ISu (interaktivní osnova, učební materiály)
  - **vzorové příklady z řešeními**: ukázky pro písemku i domácí úkoly
- **Ukončení předmětu:**
  - **písemná práce**: 80 bodů
    - cca 7 otázek: přehledové, srovnávací, algoritmy, pojmy, příklady (model)

      cca 25 bodů: příklad(y) návrhu modelu problému – probíráno ve cvičení
    - vzory písemné práce dostupné na webu předmětu
  - **2 domácí úkoly**: 20 bodů
    - za jednu domácí úlohu lze získat až 10 bodů
    - podmínkou je **získání alespoň 8 bodů z celkového počtu 20 bodů za D.Ú.**
  - **bonusové body**: cca 12 bodů
    - 1 bod za aktivní účast na přednášce
  - hodnocení: A více než 90, B 89-80, C 79-70, D 69-60, E 59-50

Literatura

- Dechter, R. **Constraint processing.** Morgan Kaufmann Publishers, 2003.
  - http://www.ics.uci.edu/~dechter/books/
- Tsang, E. **Foundations of Constraint Satisfaction.** Academic Press, 1993.
  - http://www.bracil.net/edward/FCS.html
- Barták, R. **On-line guide to constraint programming.**
  - http://ktilinux.ms.mff.cuni.cz/~bartak/constraints/
- Barták, R. **Programovaní s omezujícími podmínkami**, přednáška na MFF UK.
  - http://kti.ms.mff.cuni.cz/~bartak/podminky/index.html
- Elektronické materiály viz web předmětu

Přehled přednášky

- Úvod
- Konzistenční algoritmy
- Prohledávací algoritmy
- Optimalizační problémy a řešení
- Opakování v příkladech

Omezující podmínky v navazujících přednáškách:

- PA167 Rozvrhování

Cvičení

- **Účast na cvičeních povinná**
  - v případě více než jedné absence nutné zpracovat doplňující příklady
  - při vysokém počtu absencí na cvičení předmět absolvovat nelze
- **Cíl**
  - praktické procvičení příkladů s omezujícími podmínkami u počítačů
- **Obsah**
  - Úvod do programovacího jazyka OPL ILOG od firmy IBM
  - Řešení problémů: globální podmínky, modelování, rozvrhování, prohledávání

Software: IBM ILOG CPLEX Optimization Studio

- Dostupné ke stažení ve Studijních materiálech
- Licence dostupná výhradně pro studenty předmětu! **Šířením porušíte licenci.**
- **Optimization Programming Language (OPL)**
  - přirozený matematický popis optimalizačních modelů
  - vysoko-úrovňová syntaxe s jednoduchým a stručným kódem
  - řešení problémů nejen s omezujícími podmínkami ale i pro matematické programování
  - https://www.ibm.com/analytics/optimization-modeling
- Tutoriály a dokumentace ve Studijních materiálech

  https://is.muni.cz/auth/el/1433/podzim2019/PA163/ilog/ILOG02_course.zip

Optimization Programming Language (OPL)

Společnost Volsay vyrábí sloučeniny NH_3 (amoniak) a NH_4Cl (chlorid amonný).

Volsay má k dispozici:

50 jednotek dusíku (N), 180 jednotek vodíku (H) a 40 jednotek chloru (Cl).

Volsay má zisk 40 EUR za prodej jednotky NH_3 a 50 EUR za jednotku NH_4Cl.

Jak Volsay maximalizuje zisk na základě skladových zásob?

```
using CP;

dvar int+ gas;
dvar int+ chloride;
maximize
   40 * gas + 50 * chloride
subject to {
   gas + chloride <= 50;
   3 * gas + 4 * chloride <= 180;
   chloride <= 40;
};
```

Úvod do programování
s omezujícími podmínkami

Obsah přednášky

- Ukázkový příklad: sudoku
- Základní pojmy: omezení, ...
- Jak řešíme problémy s omezujícími podmínkami
- Příklady a aplikace
- Složitost a úplnost

Sudoku: problém

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

Přiřaď prázdným polím čísla tak, že:

čísla odlišná na řádku, ve sloupci a v bloku

Příklad převzat z přednášky Ch.Schulte, University of Uppsala

Sudoku: řádky

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

Přiřad' prázdným polím čísla tak, že:

čísla odlišná **na řádku**, ve sloupci a v bloku

Sudoku: sloupce

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

Přiřaď prázdným polím čísla tak, že:

čísla odlišná na řádku, **ve sloupci** a v bloku

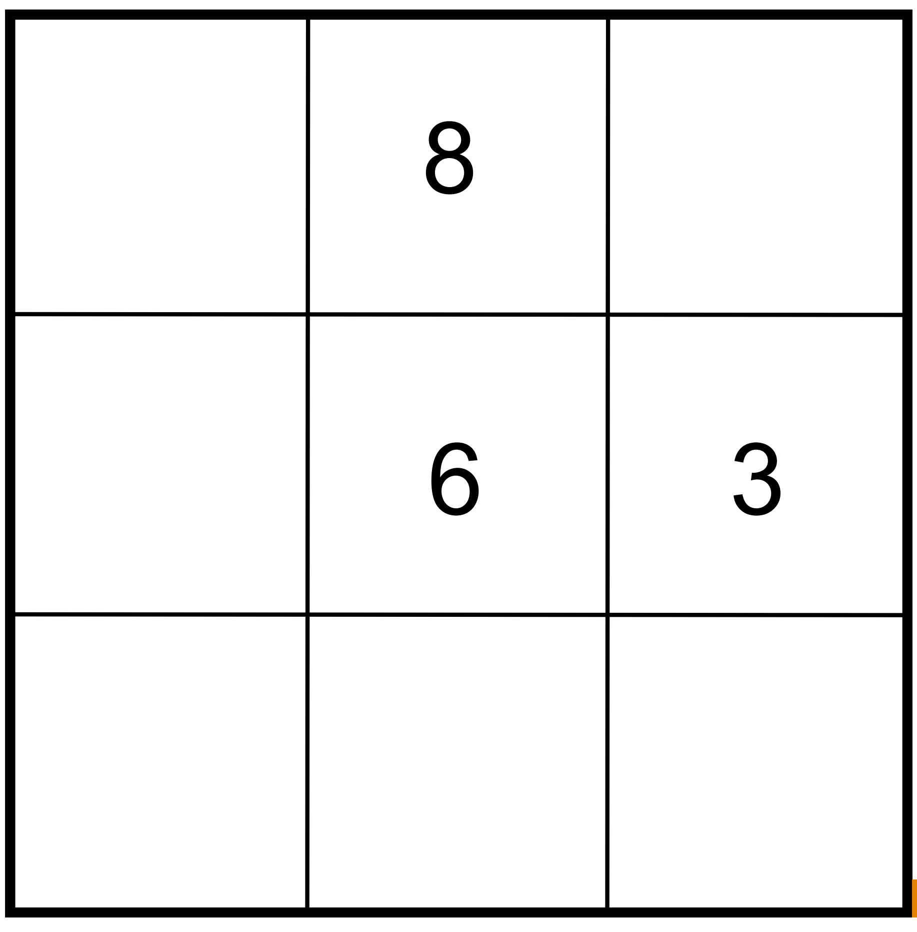
Sudoku: bloky

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

Přiřaď prázdným polím čísla tak, že:

čísla odlišná na řádku, ve sloupci a **v bloku**

Propagace v bloku: vstup



- Žádné pole v bloku nemůže obsahovat čísla 3,6,8

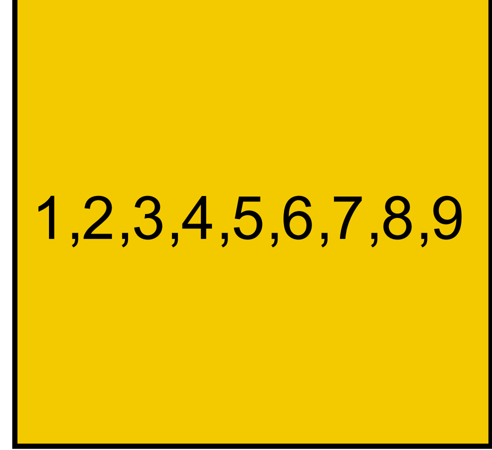
Propagace v bloku: promazání hodnot

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

- Žádné pole v bloku nemůže obsahovat čísla 3,6,8
  - propagace do dalších polí v bloku
- Řádky a sloupce: podobně

Propagace: jedno pole

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



Odstranění čísel z polí tak, že:

čísla odlišná na řádku, ve sloupci a v bloku

Propagace: jedno pole a řádek

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

Odstranění čísel z polí tak, že:

čísla odlišná **na řádku**, ve sloupci a v bloku

Propagace: jedno pole a sloupec

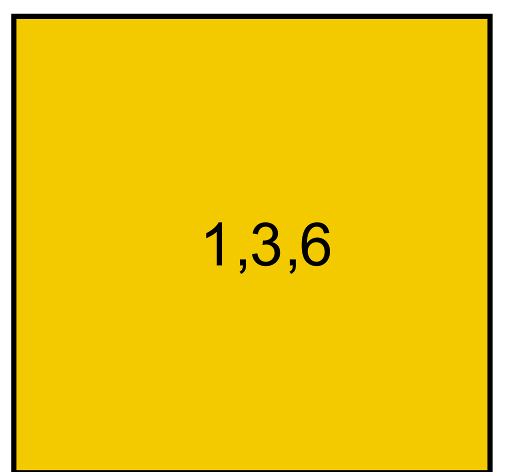
			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

Odstranění čísel z polí tak, že:

čísla odlišná na řádku, **ve sloupci** a v bloku

Propagace: jedno pole a blok

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



Odstranění čísel z polí tak, že:

čísla odlišná na řádku, ve sloupci a **v bloku**

Iterativní propagace

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

- **Iterování propagací** pro řádky, sloupce, bloky
- **Pokud stále zůstává více možností:** prohledávání

Sudoku a programování s omezujícími podmínkami

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

- **Proměnné**: pro každé pole
  - mají **hodnoty**: čísla
  - udržování množiny možných čísel
- **Omezení**: vyjadřují odlišnosti
  - relace mezi proměnnými

- **Modelování**: proměnné, hodnoty, omezení
- **Řešení**: propagace, prohledávání

Omezení (*constraint*)

- Dána
  - množina **(doménových) proměnných** $Y = \{y_1, \ldots, y_k\}$
  - konečná množina hodnot (**doména**) $D = D_1 \cup \ldots \cup D_k$

Omezení (podmínka) c na Y je podmnožina $D_1 \times \ldots \times D_k$ **tj. relace**

  - omezuje hodnoty, kterých mohou proměnné nabývat současně

- Příklad:
  - proměnné: A,B
  - domény: {0,1} pro A {1,2} pro B
  - omezení: A$\neq$B nebo (A,B) $\in$ {(0,1),(0,2),(1,2)}

- Omezení c definováno na proměnných $y_1, \ldots y_k$ je **splněno**, pokud pro hodnoty $d_1 \in D_1, \ldots d_k \in D_k$ platí $(d_1, \ldots d_k) \in c$
  - příklad (pokračování): omezení splněno pro $(0,1), (0,2), (1,2)$, není splněno pro $(1,1)$

Problém splňování podmínek (CSP)

- Dána
  - konečná množina **proměnných** $X = \{x_1, \ldots, x_n\}$
  - konečná množina hodnot (**doména**) $D = D_1 \cup \ldots \cup D_n$
  - konečná množina **omezení** $C = \{c_1, \ldots, c_m\}$
    - omezení je definováno na podmnožině X

Problém splňování podmínek je **trojice** (X, D, C)

(constraint satisfaction problem)

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

Řešení CSP

- **Částečné ohodnocení (přiřazení) proměnných** $(d_1, \dots, d_k), k < n$
  - některé proměnné mají přiřazenu hodnotu
- **Úplné ohodnocení (přiřazení) proměnných** $(d_1, \dots, d_n)$
  - všechny proměnné mají přiřazenu hodnotu
- **Řešení CSP**
  - úplné ohodnocení proměnných, které splňuje všechna omezení
  - $(d_1, \dots, d_n) \in D_1 \times \dots \times D_n$ je **řešení** (X, D, C)
    - pro každé $c_i \in C$ na $x_{i_1}, \dots x_{i_k}$ platí $(d_{i_1}, \dots d_{i_k}) \in c_i$
- Hledáme: **jedno řešení** nebo
  všechna řešení nebo
  optimální řešení vzhledem k účelové funkci, tj. řešíme
  optimalizační problém s podmínkami (constraint optimization problem)

Přístup CP k programování

- **Formulace** daného problému pomocí omezení: **modelování**
- **Řešení** vybrané reprezentace pomocí
  - **doménově specifických metod**
  - **obecných metod**

Obecné metody

- **Algoritmy propagace omezení (konzistenční algoritmy)**
  - umožňují odstranit nekonzistentní hodnoty z domén proměnných
  - zjednodušují problém
  - udržují ekvivalenci mezi původním a zjednodušeným problémem
  - používají se pro výpočet **lokální konzistence**
  - aproximují tak **globální konzistenci**

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

po propagaci: $A = 0,\ B = 1,\ C \in \{1,2\}$

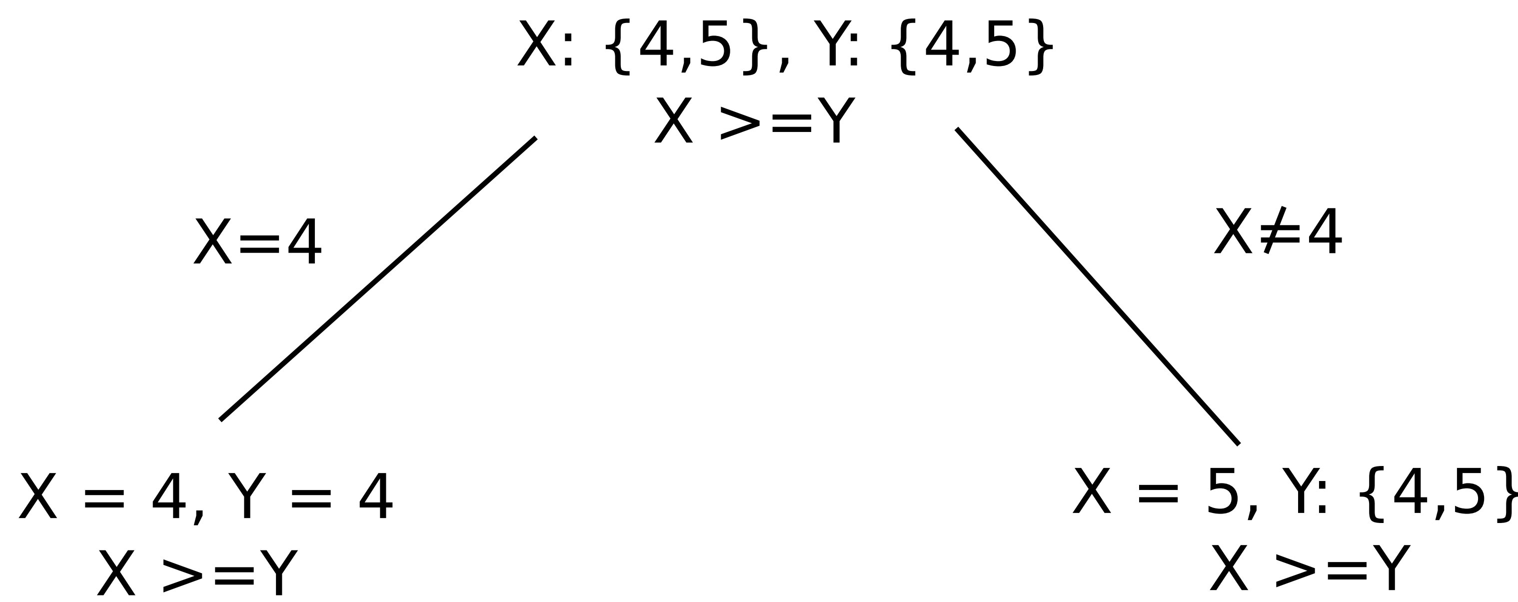
- **Prohledávací algoritmy**
  - prohledávání stavového prostoru řešení
  - příklady: backtracking, metoda větví a mezí

Prohledávání: příklad

Prohledávání pomocí větvení

- vytvoření podproblému s dodatečnou informací

  umožní další propagaci omezení



Doménově specifické metody

- Specializované algoritmy
- Nazývány **řešiče omezení (*constraint solvers*)**
- Příklady:
  - program pro řešení systému lineárních rovnic
  - knihovny pro lineární programování
  - implementace unifikačního algoritmu
- Programování s omezujícími podmínkami
  - široký pojem zahrnující řadu oblastí
  - lineární algebra, globální optimalizace, lineární a celočíselné programování, ...
- **Existence doménově specifických metod**
  $\Longrightarrow$ **použití místo obecných metod**
  - hledání doménově specifických metod tak, aby mohly být použity místo obecných metod

Algebrogram

- Přiřaďte cifry 0, . . . 9 písmenům S, E, N, D, M, O, R, Y tak, aby platilo:
  - ```
      SEND
    + MORE
    ------
     MONEY
    ```
  - různá písmena mají přiřazena různé cifry
  - S a M nejsou 0
- Jediné řešení:
  ```
    9567
  + 1085
  ------
   10652
  ```
- **Proměnné:** S,E,N,D,M,O,R,Y
- **Domény:** 1..9 pro S,M 0..9 pro E,N,D,O,R,Y
  ```

Algebrogram: alternativy pro omezení rovnosti

- 1 omezení rovnosti

```
                 1000*S + 100*E + 10*N + D                    SEND
+                1000*M + 100*O + 10*R + E                  + MORE
= 10000*M + 1000*O + 100*N + 10*E + Y                        MONEY
```

- 5 omezení rovnosti

  použití „přenosových” proměnných P1,P2,P3,P4 s doménami 0..1

```
     D + E = 10*P1 + Y,
P1 + N + R = 10*P2 + E,
P2 + E + O = 10*P3 + N,
P3 + S + M = 10*P4 + O
        P4 = M
```

Algebrogram: alternativy pro omezení nerovnosti

- **28 omezení nerovnosti:** X $\neq$ Y pro X,Y $\in$ {S,E,N,D,M,O,R,Y}
- **1 omezení pro nerovnost**

  pro proměnné $x_1, \ldots, x_n$ s doménami $D_1, \ldots, D_n$:

  all_different $(x_1, \ldots, x_n) := \{(d_1, \ldots, d_n)$, $d_i \neq d_j$ pro $i \neq j\}$

Optimalizační problém s podmínkami (COP)

- Problém splňování podmínek (X, D, C)
- **Účelová funkce** $obj : Sol \rightarrow W$
- **Optimalizační problém s podmínkami (*constraint optimization problem*)**
  - nalezení řešení d pro (X, D, C) takové, že $obj(d)$ je optimální
    - optimální $\equiv$ maximální nebo minimální

Problém batohu (*knapsack problem*)

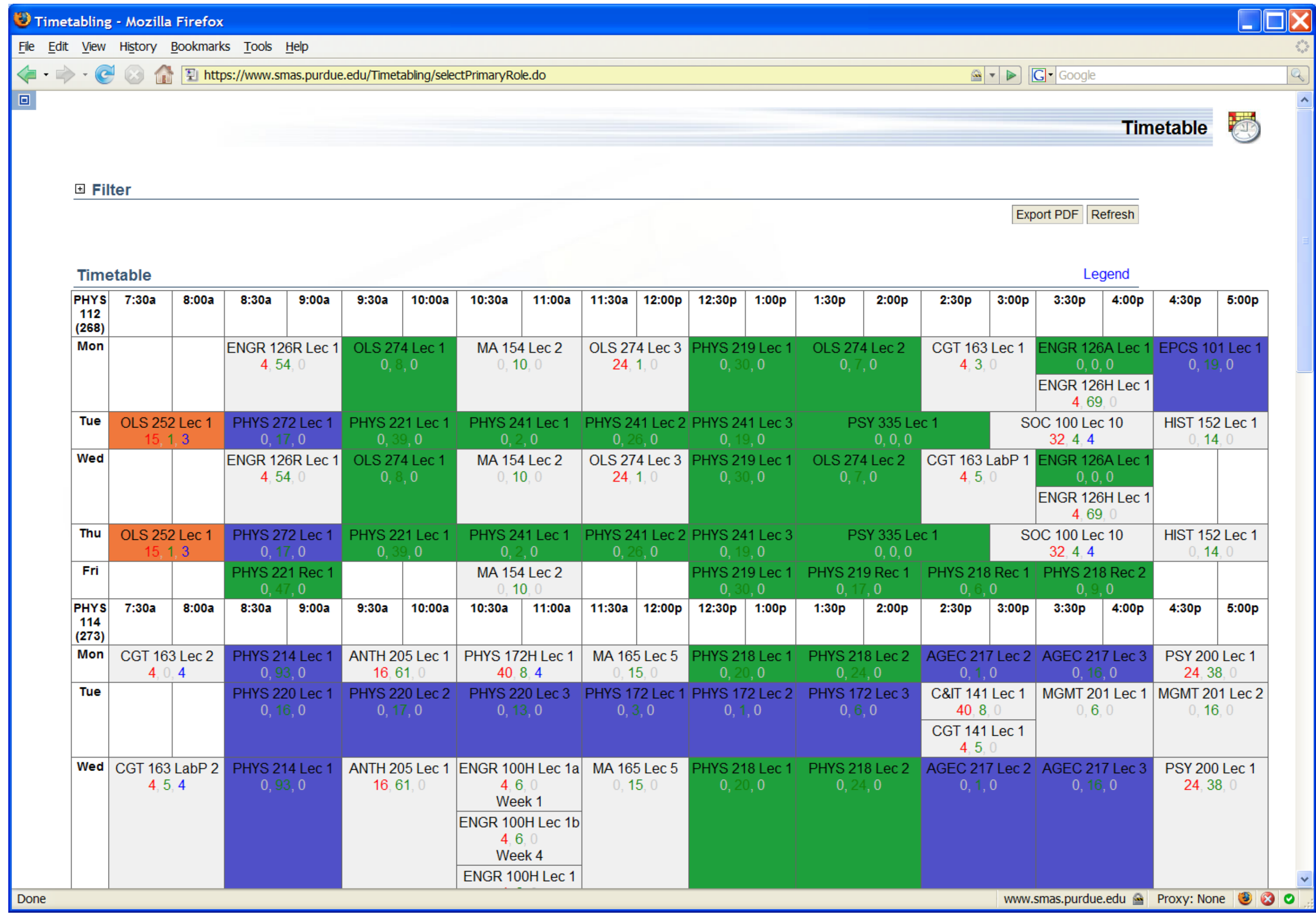
Je dán batoh velikosti m a n předmětů různé velikosti a ceny. Vyberte takové předměty, aby se vešly do batohu a jejich celková cena byla maximální.

- Batoh velikosti m
- Předměty velikosti $v_1, \ldots, v_n$ a ceny $c_1, \ldots, c_n$
- **Proměnné:** $x_1, \ldots, x_n$
- **Domény:** $0..1$
- **Omezení:** $\sum_{i=1}^{n} v_i.x_i \leq m$
- **Účelová funkce:** $\text{maximize} \sum_{i=1}^{n} c_i x_i$

Aplikace: prehled

- Operační výzkum
  - optimalizační problémy
  - rozvrhování, plánování, alokace zdrojů
- Zpracování přirozeného jazyka
  - konstrukce parserů
- Počítačová grafika
  - geometrické vztahy při analýze scény
- Databáze
  - obnovení/zajištění konzistence dat
- Molekulární biologie
  - DNA sekvencování
- ...

Příklad aplikace z MU: univerzitní rozvrhování



rozvrhovací systém Unitime http://www.unitime.org

International Timetabling Competition 2019 co-organized by FI MU: https://www.itc2019.org

Univerzitní rozvrhování: proměnné a omezení

Doménové proměnné

- čas výuky předmětu I: TimeI, hodnoty: možné časy
- místnost výuky předmětu I: RoomI, hodnoty: identifikátory místností

Omezující podmínky

- zakázaný čas: TimeI $\neq$ Prohibited
- minimální velikost místnosti: RoomI $\geq$ ConstSize
  - identifikátory místností uspořádány podle velikosti
  - ConstSize: nejmenší identifikátor místnosti s velikostí Size
- v jedné místnosti v každém čase nejvýše jeden předmět
- jeden vyučující učí nejvýše jeden předmět v každém čase
- ...

Univerzitní rozvrhování: optimalizace

Optimalizační kritéria

- **výuka v preferovaných časech**
  - cena za výuku předmětu I ve vybraném čase: CostTimeI
  - CostTime = CostTime1 + CostTime2 + · · ·
- **výuka v preferovaných místnostech**
  - cena za výuku předmětu I ve vybraném čase: CostRoomI
  - CostRoom = CostRoom1 + CostRoom2 + · · ·
- **dva předměty jednoho studenta by se neměly překrývat**
  - dva předměty I,J zároveň navštěvuje SIJ studentů
  - $\text{CostOverlap} = \sum_{\text{I,J: timeOverlap(I,J)}} \text{SIJ}$
- minimize (WTime * CostTime + WRoom * CostRoom + WOverlap * CostOverlap)

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
- **Lineární nerovnice nad reálnými čísly**
  - lineární programování, simplexová metoda
  - často stačí polynomiální složitost

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 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 prohledávacích algoritmů**
  - úplné algoritmy, příklady: backtracking, generuj & testuj
  - neúplné algoritmy, neprohledávají celý prostor řešení, příklad: omezení času prohledávání

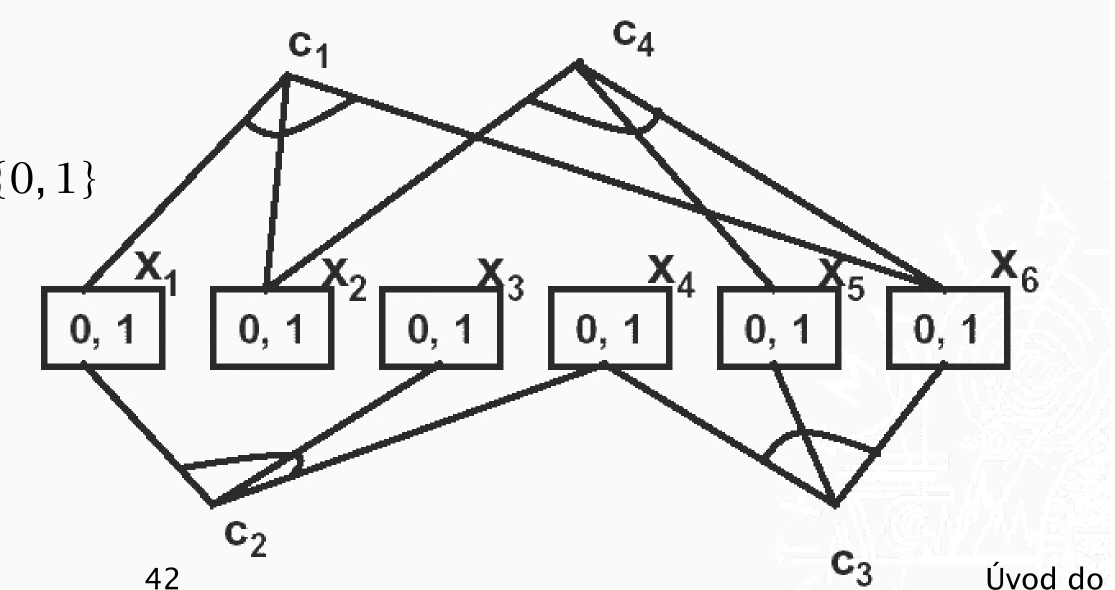
Grafová reprezentace CSP

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

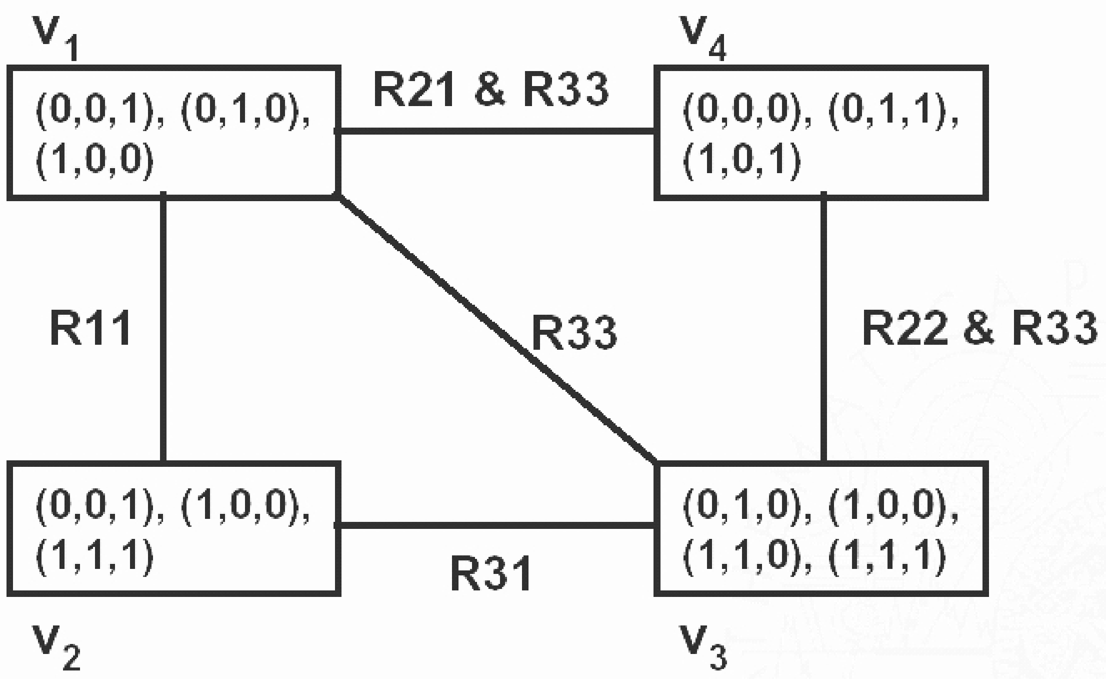
- **Duální problém**: původním omezením odpovídají nové duální proměnné
  - **proměnné:** k-ární podmínku c_i převedeme na duální proměnnou v_i s doménou obsahující konzistentní k-tice
  - **omezení:** pro každou dvojici podmínek c_i a c_j sdílející proměnné zavedeme binární podmínku R_{ij} mezi v_i 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
- 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
- **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 je **směrová**
  - konzistence hrany (V_i, V_j) nezaručuje konzistenci hrany (V_j, V_i)

A 3..7 A<B 1..5 B

A 3..4 A<B 1..5 B
konzistence (A,B)

A 3..4 A<B 4..5 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))
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

A<B B<C (A, B, C)

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

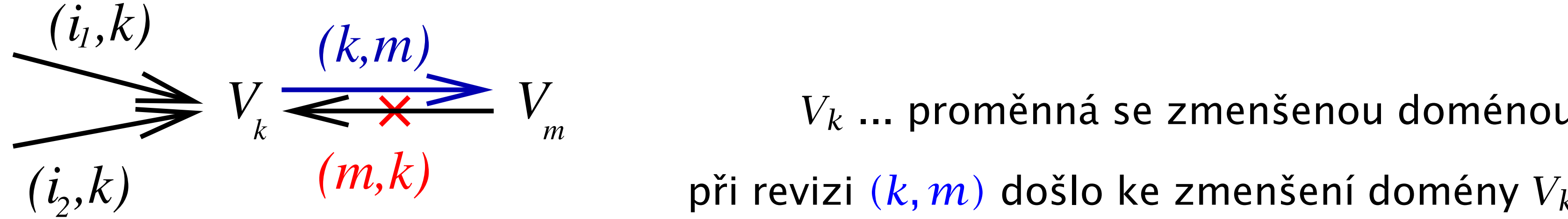
Složitost AC-1

```
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 $O(enk^3)$
  - 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?
  - 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

- 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

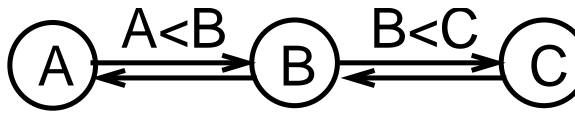
      vyber a smaž (k, m) z Q

      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`

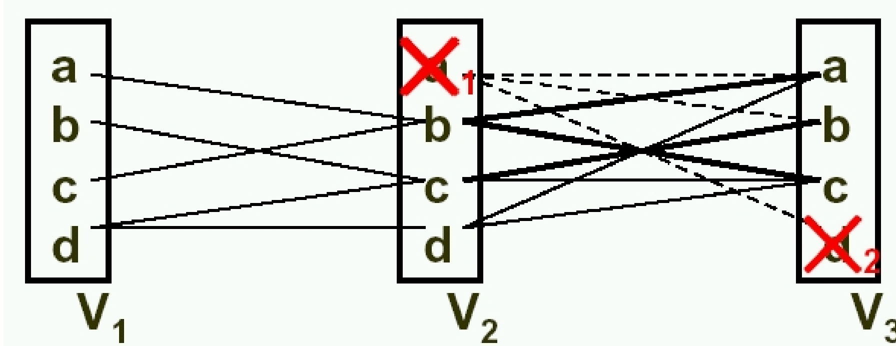
Složitost AC-3

```
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 $O(ek^3)$
  - 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.
  - 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

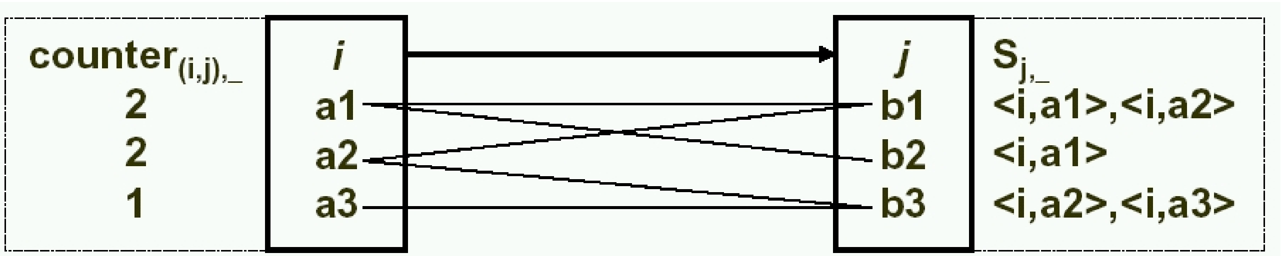
- Udržujeme počet vlastních podpor (víme, co nám chybí).

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

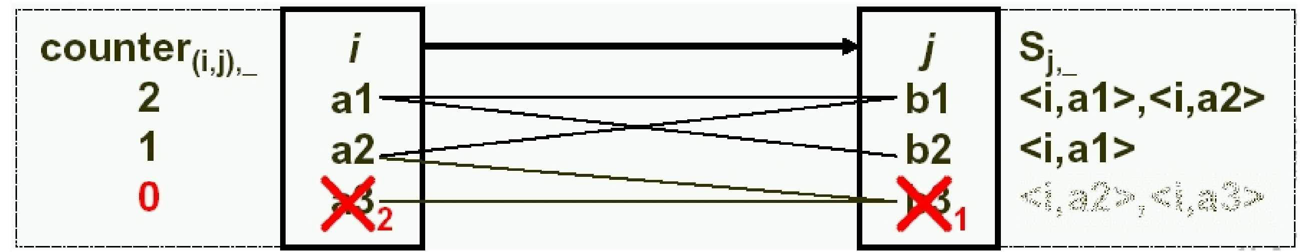
```
procedure initialize(G)
Q := {}, S := {}                          // vyprázdnění datových struktur
for each (V_i,V_j) ∈ hrany(G) do
    for each a ∈ D_i do total := 0
        for each b ∈ D_j do
            if (a,b) konzistentní vzhledem k c_{i,j} then
               total := total+1
               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



- 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⟩}
```

- 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 ∈ D_j then return true
while y := next(y,D_j) ∧ y ≠ nil do
      if (x,y) ∈ 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

Používá verzi AC-3 **s frontou proměnných**

```
procedure AC-2001(G)
Q := { i | i ∈ vrcholy(G) }   % seznam vrcholů pro revizi
while neprazdna(Q) do
    vyber a smaž j z Q
    for ∀i ∈ vrcholy(G) takové, že (i, j) ∈ hrany(G) do
       if REVISE2001(i, j) then
          if D_i = ∅ then return fail
          Q := Q ∪ {i}
return true
procedure REVISE2001(i, j)
DELETED := false
for ∀x ∈ D_i do
   if last((i, x), j) ∉ D_j then
      if ∃y ∈ D_j ∧ y > last((i, x), j) ∧ (x, y) ∈ 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)

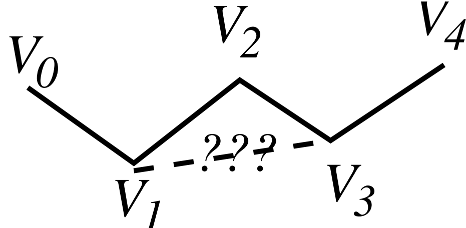
Je hranová konzistence dostatečná (úplná)?

- Použitím AC odstraníme mnoho nekompatibilních hodnot
  - Dostaneme potom řešení problému? NE
  - Víme alespoň zda řešení existuje? NE
- $X, Y, Z \in \{1, 2\}, \quad X \neq Y, \quad Y \neq Z, \quad Z \neq X$
  - 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.
- 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.**
- Důkaz: 1) PC ⇒ cesty délky 2 jsou PC

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

  a) $n = 2$ platí triviálně

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

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

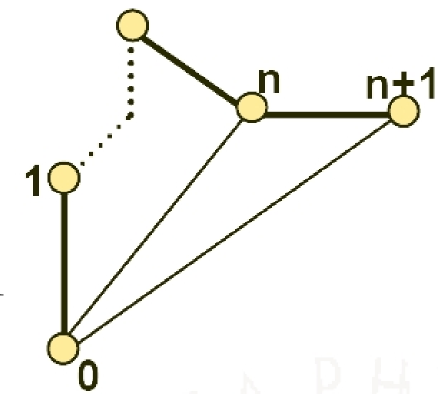
  ii) vezmeme libov. dvě kompatibilní hodnoty $x_0 \in D_0$ a $x_{n+1} \in D_{n+1}$

  (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

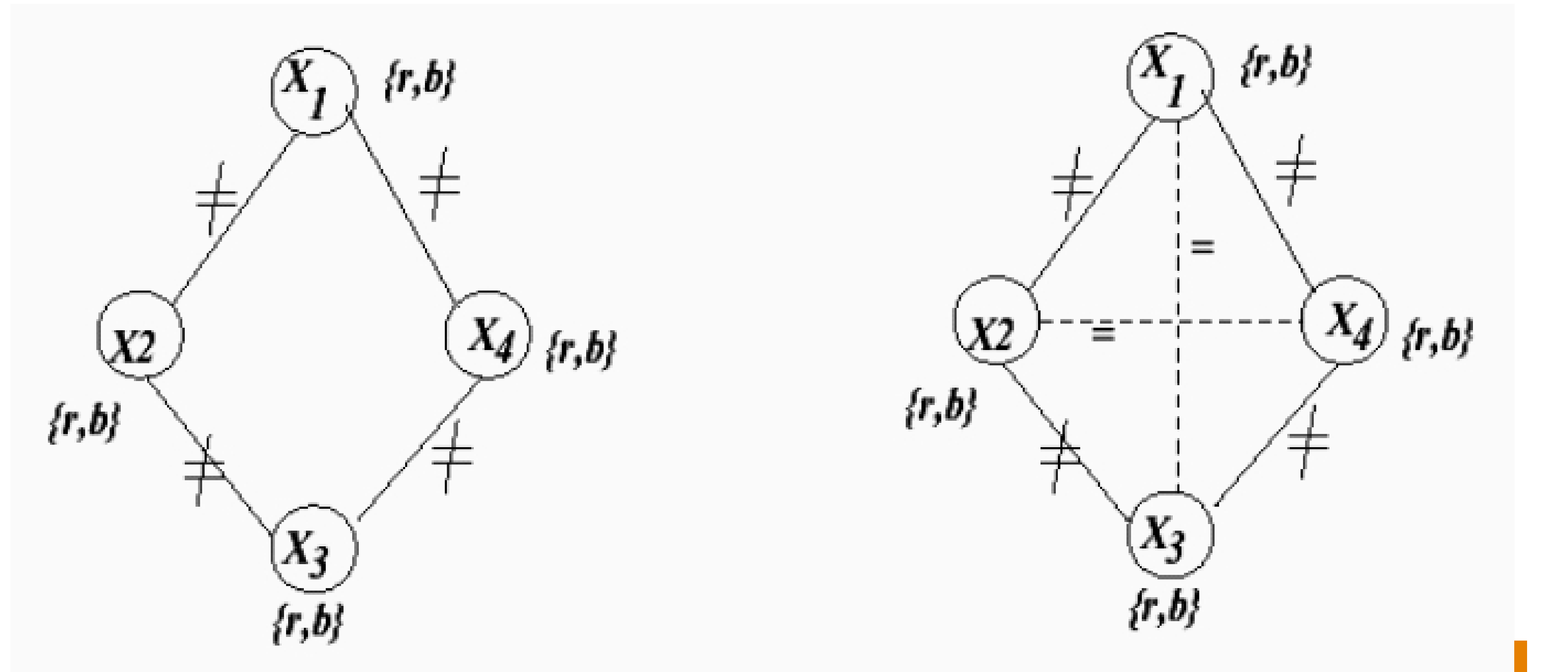
Vztah PC a AC

- PC $\Rightarrow$ 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, zdůvodnění: 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)

Příklad: barvení grafu

Nalezněte obarvení vrcholů grafu tak, aby sousední vrcholy neměly stejnou barvu.



není PC konzistentní

lze přiřadit $X_1 = r, X_3 = b$

je PC konzistentní

Algoritmus revize cesty

- Jak udělat každou cestu z V_i do V_j přes V_m konzistentní?
  - Pro dvojici proměnných V_i a V_j aktualizuji prvky relace R_{ij}
  - Vyřadím dvojici (a,b) z R_{ij}, pokud neexistuje taková hodnota $c \in V_m$, že (a,c) je kozistentní pro R_{im} a (c,b) je kozistentní pro R_{mj}

- ```
  procedure revise-3(i,m,j)
  Deleted := false
  for ∀ (a,b) ∈ R_ij do
      if neexistuje c ∈ D_m takové, že (a,c) ∈ R_im a (c,b) ∈ R_mj
      then smaž (a,b) z R_ij
           Deleted := true
  return Deleted
  end revise-3
  ```

$V_1, V_2 \in \{a,b\}$, $V_3 \in \{a,b,c\}$, $V_1 \neq V_2$, $V_2 \neq V_3$, $V_1 \neq V_3$

revise-3$(1,2,3)$ (V_1, V_3) : ~~aa~~ ~~ab~~ ac

~~ba~~ ~~bb~~ bc

pozor: aa bb nejsou už v relaci R_{13}

- Složitost $O(k^3)$ (k maximální velikost domény) $\Longleftarrow$
  cyklus pro každou dvojici (a,b): $O(k^2)$, vnitřní cyklus přes $c \in D_j$: $O(k)$
  ```

Algoritmus PC-1

- Jak udělat CSP konzistentní po cestě? Provedeme revizi každé cesty délky 2
- Revize cesty odstraní dvojice ⇒ již zrevidované cesty opět nekonzistentní
- Revize cesty opakujeme dokud jsou nějaké dvojice smazány
- Princip je podobný jako u AC-1
- Před spuštěním algoritmu nutno **inicializovat relace** R_{ij} pomocí existujících binárních (i unárních) podmínek
-

```
procedure PC-1(G)
repeat Changed := false
       for m := 1 to n
           for i := 1 to n
               for j := 1 to n
                   Changed := revise-3(i,m,j) or Changed
until not(Changed)
end PC-1
```

Složitost algoritmu PC-1

- Celková složitost $O(n^5k^5)$ odvození podobné jako pro AC-1

  $\Leftarrow$ 3 cykly pro trojice hodnot $O(n^3)$

  $\Leftarrow$ revise-3 $O(k^3)$

  $\Leftarrow$ $O(n^2k^2)$ počet opakování repeat cyklu

  $\Leftarrow$ jeden cyklus smaže (v nejhorším případě) právě jednu dvojici hodnot celkem n^2k^2 hodnot (n^2 dvojic proměnných, k^2 dvojic hodnot pro každou dvojici proměnných)

- Neefektivita PC-1
  - opakovaná revize všech cest, i když pro ně nedošlo k žádné změně
    - při revizi stačí kontrolovat jen zasažené cesty podobně jako pro PC-1
  - cesty stačí brát pouze s jednou orientací ... R_{ij} je totéž co R_{ji}
    - příklad: $V_1, V_2 \in \{a, b\}, V_3 \in \{a, b, c\}, V_1 \neq V_2, V_2 \neq V_{,3} V_1 \neq V_3$

      důsledek revise-3(1, 2, 3) a revise-3(3, 2, 1) je totožný

      $\Longleftarrow$ ke každé dvojici hodnot z V_1, V_3 (V_3, V_1) hledám kompatibilní hodnotu z V_2

Algoritmus PC-2

- Cesty beru pouze s jednou orientací
  - aktualizuji pouze jednu z R_{ij}, R_{ji}
- Do fronty dávám pouze zasažené cesty
  - podobná modifikace jako AC-3
  - revise-3(i, m, j) mění R_{ij}, stačí tedy aktualizovat R_{li} přes j a R_{lj} přes i
- Před spuštěním algoritmu
  - inicializovat relace R_{ij} pomocí existujících binárních (i unárních) podmínek
- procedure PC-2(G)

```
Q := {(i,m,j) | 1 ≤ i < j ≤ n, 1 ≤ m ≤ n, m ≠ i, m ≠ j}  //seznam cest pro revizi
while Q non empty do
     vyber a smaž trojici (i,m,j) z Q
     if revise-3(i,m,j) then
        Q := Q ∪ {(l,i,j)(l,j,i) | 1 ≤ l ≤ n, l ≠ i, l ≠ j}
                //jako u AC přidáváme jen cesty, co ještě nejsou ve frontě
end while
end PC-2
```

Složitost algoritmu PC-2

- Složitost $O(n^3k^5)$
  - každé (i, m, j) může být ve frontě maximálně k^2 krát $\Longrightarrow O(k^2)$

    $\Leftarrow$ když je (i, m, j) přidáno do fronty, R_{ij} bylo redukováno alespoň o jednu hodnotu
  - celkem n^3 trojic $(i, m, j) \Longrightarrow O(n^3)$
  - revise-3 $O(k^3)$
- Algoritmus není optimální podobně jako AC-3
  - existuje algoritmus PC-4 založen na počítání podpor
  - složitost PC-4 je $O(n^3k^3)$, což už je optimální
- Cvičení: řešte následující problém pomocí PC-2 algoritmu

  $V1 \in \{0, 1, 2, 3\}, V2 \in \{0, 1\}, V3 \in \{1, 2\}$

  $V3 = V1 + 1, V2 \neq V3, V1 \neq V2$
  - Nápověda: zkonstruuj iniciální relace R_{ij}

    iniciálně přidej do fronty $(V1, V2, V3), (V1, V3, V2), (V2, V1, V3)$

Omezení PC algoritmů

- Paměťové nároky
  - protože PC eliminuje dvojice hodnot z omezení, potřebuje používat extenzionální reprezentaci omezení (pro každou dvojici hodnot si pamatuji, zda je/není v doméně)
- Poměr výkon/cena
  - PC eliminuje více (nebo stejně) nekonzistencí jako AC
  - poměr výkonu ke zjednodušení problému je ale mnohem horší než u AC
- Změny grafu omezení
  - PC přidává hrany (omezení) i tam, kde původně nebyly a mění tak konektivitu grafu
  - to vadí při dalším řešení problému, kdy se nemohou používat heuristiky odvozené od grafu (resp. dané původním problémem)
- PC stále není dostatečné
  - $V, X, Y, Z \in \{1, 2, 3\}, \quad X \neq Y, \quad Y \neq Z, \quad Z \neq X, \quad V \neq X, \quad V \neq Y, \quad V \neq Z$
    je PC a přesto nemá řešení

Na půli cesty od AC k PC

- Jak oslabit PC, aby algoritmus:
  - neměl paměťové nároky PC
  - neměnil graf podmínek
  - byl silnější než AC?
- **Omezená konzistence po cestě – *Restricted Path Consistency* (RPC)**
  - testujeme PC jen v případě, když je šance, že to povede k **vyřazení hodnoty z domény proměnné**
- Příklad: $V_1, V_2 \in \{a, b\}, V_3 \in \{a, b, c\}, V_1 \neq V_2, V_2 \neq V_3, V_1 \neq V_3$

  je AC ale není PC

  RPC odstraní z domény V_3 hodnoty a, b

Omezená konzistence po cestě (RPC)

- PC hrany se testuje pouze tehdy, pokud vyřazení dvojice může vést k vyřazení některého z prvků z domény příslušné proměnné
- Jak to poznáme? Jedná se o **jedinou vzájemnou podporu**.
- Proměnná V_i je **omezeně konzistentní po cestě** ***(Restricted Path Consistent, RPC)***:
  - každá hrana vedoucí z V_i je hranově konzistentní
  - pro každé $a \in D_i$ platí:

    je-li b jediná podpora a ve vrcholu j, potom v každém vrcholu k (spojenem s i a j) existuje hodnota c tak, že (a, c) a (b, c) jsou kompatibilní s příslušnými podmínkami

- Algoritmus: založen na AC-4 + seznam cest pro PC (viz Barták přednáška)

Propagace pro nebinární omezení

Nebinární omezení

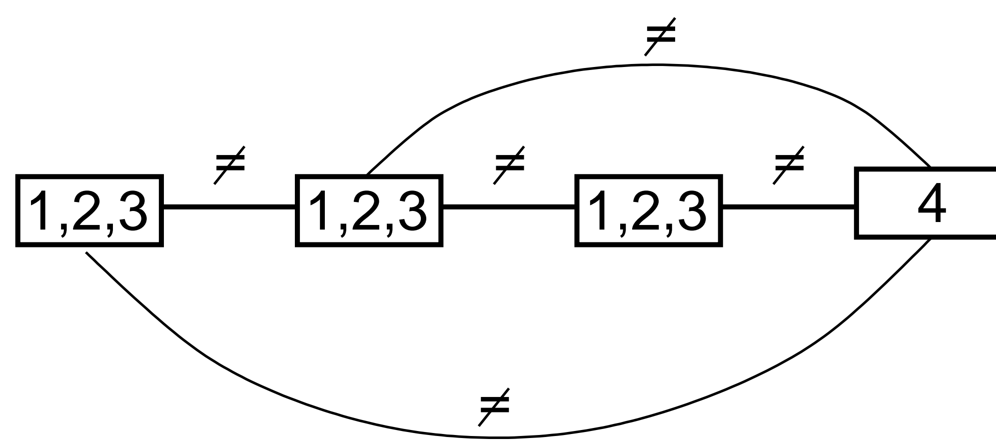
- Zobecnění principů NC, AC, a PC směrem ke **k-konzistenci**
  - zajímavé z pohledu složitosti řešení problémů
  - pracuje se s libovolnými k-ticemi proměnných
  - v praxi se vzhledem k paměťové a časové složitosti nevyužívá
- **Obecné typy konzistence** pro nebinární podmínky
  - pracuje se přímo s n-árními podmínkami
  - příklad: obecná hranová konzistence, konzistence mezí
- **Globální omezení:** specifické typy konzistencí
  - využívá se sémantika omezení, zaměřené opět na konkrétní omezení
  - speciální typy konzistence pro globální omezení (př. `all_different`)
- Pro různé podmínky lze použít různý druh konzistence
  - A<B: hranová konzistence, konzistence mezí

Obsah: propagace pro nebinární omezení

- k-konzistence
- Obecná hranová konzistence
- Konzistence mezí
  - konzistence mezí pro aritmetická omezení
- Globální podmínky
  - klasické typy podmínek a příklady jejich použití
  - propagace pro `all_different` a pro rozvrhování s unárními zdroji
- Obecný konzistenční algoritmus pro nebinární podmínky
  - v jeho rámci lze využívat výše zmíněné typy konzistence

k-konzistence

- Mají AC a PC něco společného?
  - AC: rozšiřujeme jednu hodnotu do druhé proměnné
  - PC: rozšiřujeme dvojici hodnot do třetí proměnné
  - ... můžeme pokračovat
- CSP je **k-konzistentní** právě tehdy, když můžeme libovolné konzistentní ohodnocení (k-1) různých proměnných rozšířit do libovolné k-té proměnné



4-konzistentní graf

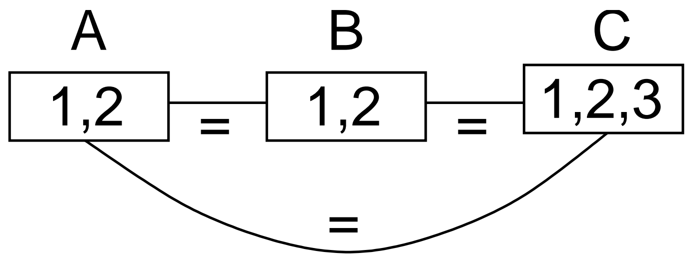
- Pro obecné CSP, tedy i pro **nebinární podmínky**

Silná k-konzistence

3-konzistentní graf

$(1,1)$ lze rozšířit na $(1,1,1)$

$(2,2)$ lze rozšířit na $(2,2,2)$



není 2-konzistentní

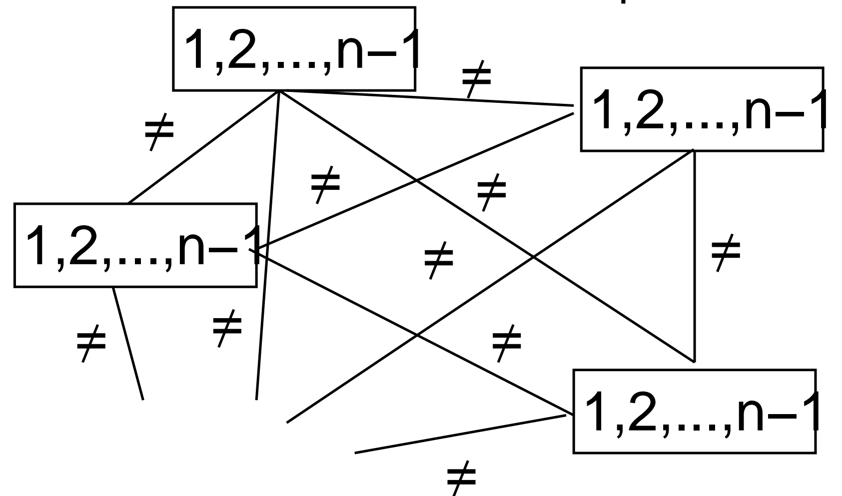
(3) nelze rozšířit

$(1,3)$ ani $(2,3)$ nejsou konzistentní dvojice (nerozšiřujeme je)

- CSP je **silně k-konzistentní** právě tehdy, když je j-konzistentní pro každé j≤k
- Silná k-konzistence ⇒ k-konzistence
- Silná k-konzistence ⇒ j-konzistence $\forall j \leq k$
- k-konzistence $\not\Rightarrow$ silná k-konzistence
- NC = silná 1-konzistence = 1-konzistence
- AC = (silná) 2-konzistence
- PC = (silná) 3-konzistence
- **Cvičení:** uveďte příklad problému, který je 4-konzistentní, ale není 3-konzistentní

Konzistence pro nalezení řešení

- Máme-li graf s n vrcholy, jak silnou konzistenci potřebujeme, abychom přímo našli řešení?
- **CSP vyřešíme bez navracení** vzhledem k uspořádání proměnných $(x_1, \ldots, x_n)$, jestliže pro každé $i \leq n$ může být každé částečné řešení $(x_1, \ldots, x_i)$ konzistentně rozšířeno o proměnnou x_{i+1}.
- Nalezení řešení bez navracení pro libovolné uspořádání proměnných?
  - **silná n-konzistence je nutná pro graf s n vrcholy**
    - n-konzistence nestačí (viz předchozí příklad)
    - silná k-konzistence pro k<n také nestačí



graf s n vrcholy

domény 1..(n-1)

silně k-konzistentní pro každé k<n

přesto nemá řešení

Algoritmy pro dosažení k-konzistence

- Rozšíření revize hrany a revize cesty
  - postupně odstraňujeme prvky z relace nad (k-1) proměnnými
- Aktualizujeme relace nad každou (k-1)-ticí proměnných
  - musíme si pamatovat (k-1)-tice hodnot
- Obecný algoritmus
  - rozšíření AC-1 a PC-1
  - opakování revizí nad (k-1)-ticemi dokud dochází ke změnám
- Velká paměťová i časová složitost
  - v praxi se pro vyšší k nepoužívá
- Algoritmy i složitost viz Dechter: Constraint Processing

Pojmy a značení

- **Rozsah omezení** $scope(c)$

  množina proměnných, na kterých je c definováno

  - příklad: $A, B, C \in \{0, 1, 2\}$

    $scope(A < B) = \{A, B\}$

- **k-tice** $\vec{t}$ hodnot patřících do c: $\vec{t} \in c$

  - příklad (pokračování): $(0, 1) \in (A < B)$

- Proměnná $x \in scope(c)$, k-tice $\vec{t} \in c$

  $\vec{t}[x]$ je hodnota proměnné x v $\vec{t}$

  - příklad (pokračování): $(0, 1)[A] = 0$

Definice obecné hranové konzistence (GAC)

- ***Generalized arc consistency*** (někdy nazývána **doménová konzistence**)
  - pro každou proměnnou z podmínky a každou její hodnotu existuje ohodnocení zbylých proměnných v podmínce tak, že podmínka platí
  - A + B = C, A $\in \{1,2,3\}$, B $\in \{2,3,4\}$, C $\in \{3,\ldots,7\}$ je obecně hranově konzistentní
- **Omezení c je obecně hranově konzistentní**, jestliže má každá hodnota a každé proměnné $x \in scope(c)$ doménovou podporu v c
- Hodnota a proměnné $x \in scope(c)$ má **doménovou podporu** $\vec{t}$ v c, jestliže
  - $\vec{t} \in c$ a platí $a = \vec{t}[x]$
  - pro každé $y \in scope(c)$ platí $\vec{t}[y] \in D_y$
- Příklad: A $\in \{0,1\}$, B$\in \{0,1\}$, C $\in \{1,2\}$, A = B+C,

  0 u A nemá podporu, 1 u A má podporu (1,0,1)
- **CSP je obecně hranově konzistentní** $\Longleftrightarrow$

  všechna jeho omezení jsou obecně hranově konzistentní
- Příklad (pokračování): po GAC dostaneme A=1, B=0, C=1

Konzistence mezí

- ***Bounds consistency* BC**: slabší než obecná hranová konzistence
  - propagace pouze při **změně minimální nebo maximální hodnoty** v doméně proměnné
  - tedy došlo ke **změně mezí domény proměnné**
- **Konzistence mezí pro nerovnice**
  - A > B => min(A) $\geq$ min(B)+1, max(B) $\leq$ max(A)-1
  - příklad: A $\in \{4,\ldots,10\}$, B$\in \{6,\ldots 18\}$, A > B

    min(A) $\geq$ 6+1 $\Rightarrow$ A $\in \{7,\ldots,10\}$

    max(B) $\leq$ 10-1 $\Rightarrow$ B $\in \{6,\ldots,9\}$
- **Cvičení:** napište pravidla pro konzistenci mezí pro uvedená omezení
  - A < B, A $\geq$ B, A $\leq$ B
  - a aplikujte je v případě domén A $\in \{1,\ldots,10\}$, B $\in \{0,\ldots,5\}$
- Jak je to tedy pro nebinární omezení?

Konzistence mezí a aritmetická omezení

- A = B + C => min(A) $\geq$ min(B)+min(C), max(A) $\leq$ max(B)+max(C)
  min(B) $\geq$ min(A)-max(C), max(B) $\leq$ max(A)-min(C)
  min(C) $\geq$ min(A)-max(B), max(C) $\leq$ max(A)-min(B)
  - změna min(A) vyvolá pouze změnu min(B) a min(C)
  - změna max(A) vyvolá pouze změnu max(B) a max(C), ...
- Příklad: A $\in \{1,\dots,6,9,10\}$, B $\in \{1,\dots,10\}$, A = B + 2

  A = B + 2 $\Rightarrow$ min(A)$\geq$1+2, max(A)$\leq$10+2 $\Rightarrow$ A $\in \{3,\dots,6,9,10\}$
  $\Rightarrow$ min(B)$\geq$1-2, max(B)$\leq$10-2 $\Rightarrow$ B $\in \{1,\dots,8\}$

  tj. doména B má pouze změněny meze a hodnoty 5, 6 zůstanou v doméně

  A $\in \{3,\dots,6,9,10\}$, B $\in \{1,\dots,8\}$, A = B + 2 je BC, není GAC, není AC
- **Cvičení:** napište pravidla pro konzistenci mezí pro uvedená omezení
  - A = B - 3, **A = B - C, A = B + C, A < B + C**,
  - a aplikujte je v případě domén A $\in \{1,\dots,10\}$, B $\in \{0,\dots,5\}$, C $\in \{2,3,4\}$

Definice konzistence mezí

- Hodnota a proměnné $x \in scope(c)$ má **intervalovou podporu** $\vec{t}$ v c, jestliže
  - $\vec{t} \in c$ a platí $a = \vec{t}[x]$
    - příklad: $a = 1, \vec{t} = (1, 5, 7)$
  - pro každé $y \in scope(c)$ platí $\vec{t}[y] \in [min(D_y), max(D_y)]$
    - př. (pokrač.) $y = 5, z = 7, \quad D_y = D_z = \{1, 2, 4, 5, 10\}$

      $5 = \vec{t}[y]$ i $7 = \vec{t}[z]$ mají intervalovou podporu
- Srovnání: doménová podpora GAC
  - pro každé $y \in scope(c)$ platí $\vec{t}[y] \in D_y$
    - př. (pokrač.) $7 = \vec{t}[z]$ nemá doménovou podporu
- **Omezení má konzistentní meze**, jestliže každá hodnota a proměnné $x \in scope(c)$ má intervalovou podporu v c
- **CSP má konzistentní meze** $\Longleftrightarrow$ všechna jeho omezení mají konzistentní meze

Globální podmínky

Globální podmínka: definována pro libovolný konečný počet proměnných

Global Constraint Catalog

- http://sofdem.github.io/gccat/
- Nicolas Beldiceanu, Mats Carlsson and Jean-Xavier Rampon
- Popis omezení dostupných v literatuře a v systémech s omezujícími podmínkami
- „The catalog presents a list of 423 global constraints issued from the literature in constraint programming and from popular constraint systems. The semantic of each constraint is given together with some typical usage and filtering algorithms, and with reformulations in terms of graph properties, automata, and/or logical formulae. When available, it also presents some typical usage as well as some pointers to existing filtering algorithms."
- PDF dokument, srpen 2014, cca 4 000 stran

(Globalní podmínky a) rozvrhování

- Všechny proměnné různé
  - `allDifferent`
- Disjunktivní zdroj/rozvrhování
  - `dvar interval`, `dvar sequence`
  - `noOverlap`
- Kumulativní zdroj/rozvrhování
  - `cumuFunction`, `pulse`
- Alternativní zdroje
  - `alternative`

Všechny proměnné různé

- Proměnné v poli `Array` jsou různé
  - `dvar int Array[Interval];`
  - globální podmínka: `allDifferent(Array);`
  - binární podmínky: `for (i, j in Interval : i != j) Array[i] != Array[j];`
- `allDifferent` vs. binární podmínky
  - `allDifferent` má úplnou propagaci
  - binární podmínky mají slabší (neúplnou) propagaci
- Příklad: učitelé musí učit v různé hodiny
  - globální podmínka:

    `Jan = 6, Ota = 2, Anna = 5,`

    `Marie = 1, Petr` $\in \{3,4\}$, `Eva` $\in \{3,4\}$
  - binární podmínky:

    `Jan` $\in \{3,\ldots,6\}$, `Petr` $\in \{3,4\}$, `Anna` $\in \{2,\ldots,5\}$,

    `Ota` $\in \{2,3,4\}$, `Eva` $\in \{3,4\}$, `Marie` $\in \{1,\ldots,6\}$

učitel	min	max
Jan	3	6
Petr	3	4
Anna	2	5
Ota	2	4
Eva	3	4
Marie	1	6

Naivní propagace pro `allDifferent`

učitel	min	max
Jan	3	6
Petr	3	4
Anna	2	5
Ota	2	4
Eva	3	4
Marie	1	6

- **U = {Petr, Ota, Eva}, dom(U) = {2, 3, 4}**:

  `{2,3,4}` nelze pro `Jan, Anna, Marie`

  `Jan` $\in \{5,6\}$, `Anna = 5, Marie` $\in$ `{1,5,6}`

- **Konzistence:** musí platit:

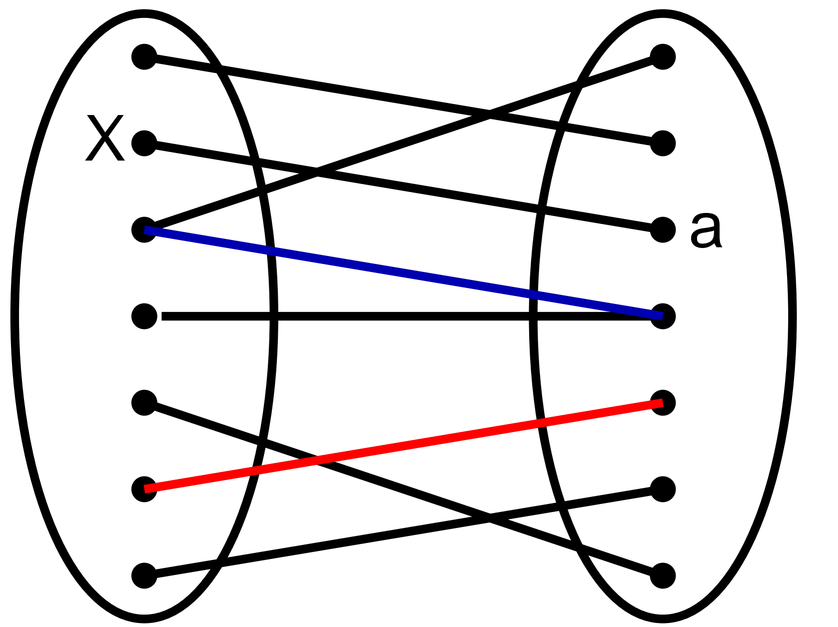
  $\forall \{X_1, \ldots, X_k\} \subset V : card\{D_1 \cup \cdots \cup D_k\} \geq k$

- **Inferenční pravidlo**
  - V: množina všech proměnných
  - $U = \{X_1, \ldots, X_k\}, dom(U) = \{D_1 \cup \cdots \cup D_k\}$
  - $card(U) = card(dom(U)) \Rightarrow \forall v \in dom(U), \forall X \in (V - U), X \neq v$
  - hodnoty v $dom(U)$ jsou pro ostatní proměnné nedostupné

- **Složitost**
  - hledání všech podmnožin množiny n proměnných (naivní) $O(2^n)$

Párování v bipartitních grafech

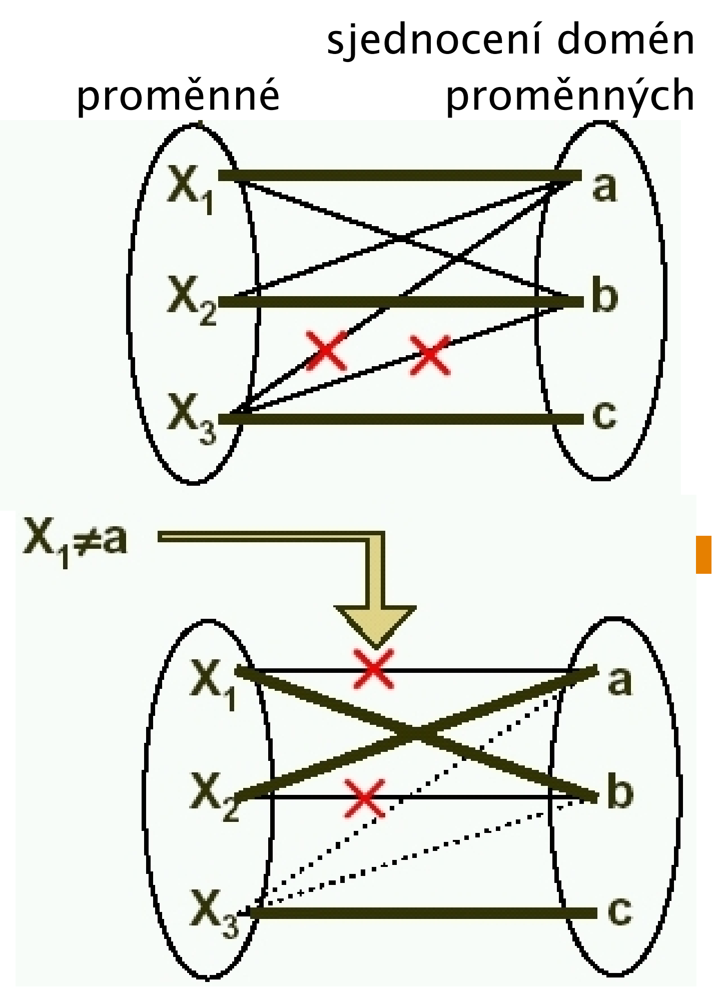
- Efektivní propagaci pro `allDifferent` lze založit na
  párování v bipartitních grafech (Régin 94)
- **Bipartitní graf**
  - uzly grafu rozdělené do dvou množin
  - neexistují hrany mezi uzly jedné množiny
- **Párování** v bipartitních grafech
  - v grafu neexistují dvě hrany,
    které by měly společný vrchol
- **Maximální párování**
  - párování, které má maximální počet hran



- CSP jako bipartitní graf
  - jedna množina vrcholů reprezentuje proměnné
  - druhá množina vrcholů reprezentuje hodnoty proměnných
  - hrana z proměnné X do hodnoty a říká, že proměnná X může nabývat hodnoty a

allDifferent – párování v bipartitních grafech II.

- Inicializace
  - vypočti maximální párování
  - odstraň všechny hrany, které nepatří do žádného maximálního párování
- Propagace zmenšené domény
  - odstraň odpovídající hrany
  - vypočti nové maximální párování
  - odstraň všechny hrany, které nepatří do žádného maximálního párování

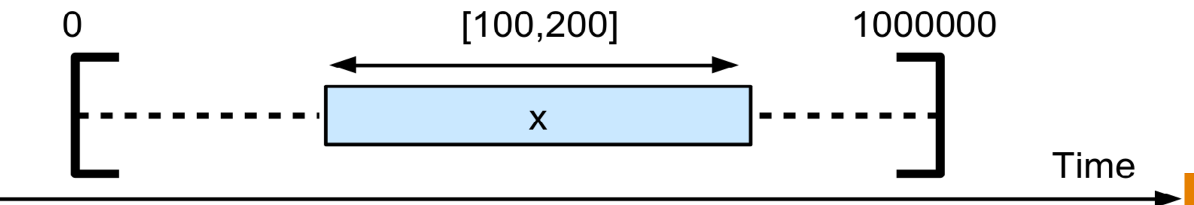


- Algoritmus založen na doménové konzistenci
  - každé maximální párování definuje doménovou podporu
  - složitost $O(n^2k^2)$, n ... počet proměnných, k ... maximální velikost domény
  - efektivnější algoritmus využívá konzistenci mezí – složitost $O(n \log n)$ (Puget 1998)

Intervalové proměnné

Intervalová proměnná: pro modelování časového intervalu (úlohy, aktivity)

- hodnotou intervalové proměnné je celočíselný interval $[start, end)$
- příklad: `dvar interval x in 0..1000000 size 100..200;`



Volitelná intervalová proměnná: pro modelování časového intervalu, který může ale nemusí být přítomen v řešení (**přítomný vs. nepřítomný interval**)

- např. pro modelování volitelných aktivit, které v řešení nemusí být
- příklad: `dvar interval x optional in 0..1000000 size in 100..200;`
- Dom(x) $\subseteq \{\perp\} \cup \{[start, end) | start, end \in \mathbb{Z}, start \leq end\}$

  $\perp$ značí, že interval není přítomen v řešení

Disjunktní/unární rozvrhování

Sekvenční proměnná p

- definována na množině intervalových proměnných x

```
dvar interval x[i in 1..n] ...;
dvar sequence p in x;
```

- **hodnota** intervalové proměnné p je **permutace** přítomných intervalů
  - pozor, permutace t ještě neimplikuje žádné uspořádání v čase!

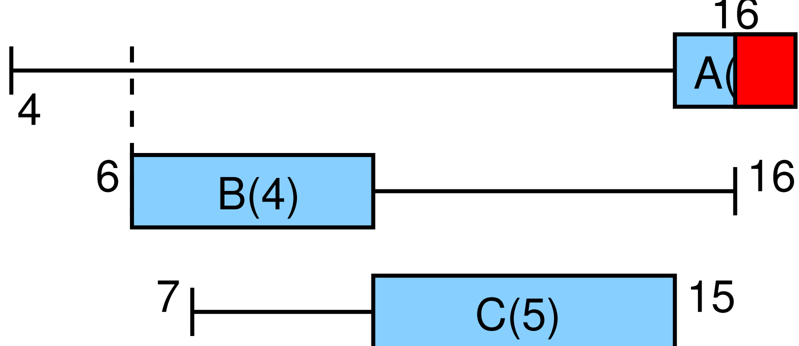
Omezení `noOverlap(p)`

- vyjadřuje, že sekvenční proměnná p reprezentuje řetězec nepřekrývajících se intervalových proměnných
- pro vyjádření rozvrhování na **unárním/disjunktivním zdroji**, kde se intervaly/úlohy/aktivity nepřekrývají
- poznámka: nepřítomné intervaly v řetězci zahrnuty nejsou

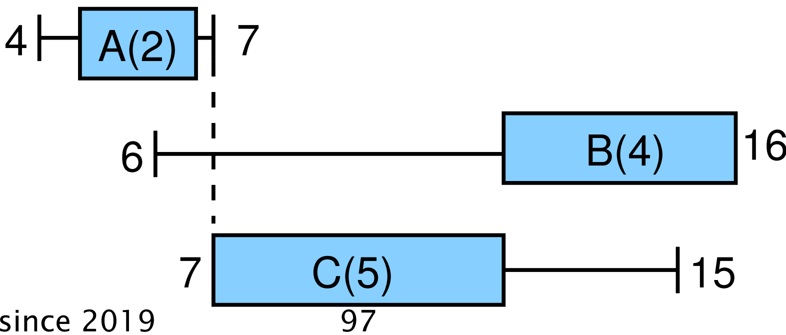
Disjunktivní rozvrhování: princip algoritmu

- Hledání hran *(edge finding)* – Baptiste & Le Pape, 1996
- Hledáme úlohu, která předchází nebo následuje množinu jiných úloh
- `dvar interval A in 4..16 size 2; dvar interval B in 6..16 size 4;`

  `dvar interval C in 7..15 size 5;`
- Co se stane, pokud nebude aktivita A zpracována jako první?



- Pro A,B,C není dost času, a tedy aktivita A musí být první



Další podmínky: precedence

Mezi intervalovými proměnnými můžeme definovat precedenční podmínky:

```
dvar interval i;
dvar interval j;

endBeforeStart(i, j);
endBeforeEnd(i, j);
endAtStart(i, j);
endAtEnd(i, j);
startBeforeStart(i, j);
startBeforeEnd(i, j);
startAtStart(i, j);
startAtEnd(i, j);
```

Poznámka: tyto podmínky platí, pokud jsou oba intervaly přítomné

A dále: logické podmínky

Unární podmínka pro přítomnost intervalu x:

```
presenceOf(x)
```

znamená, že x $\neq \perp$

Příklady:

```
presenceOf(x) == presenceOf(y)  // x přítomen právě tehdy, když je přítomen y
presenceOf(x) => presenceOf(y)  // implikace
presenceOf(x) => !presenceOf(y)
```

Pozor na použití:

- precedenční podmínky: polynomiální složitost
- logické binární podmínky: polynomiální složitost
- precedence + logické binární: NP-těžké!

Výrazy s intervalovými proměnnými

Pro vytváření účelových funkcí nebo definici omezení

```
startOf(x)
endOf(x)
sizeOf(x, V)
```

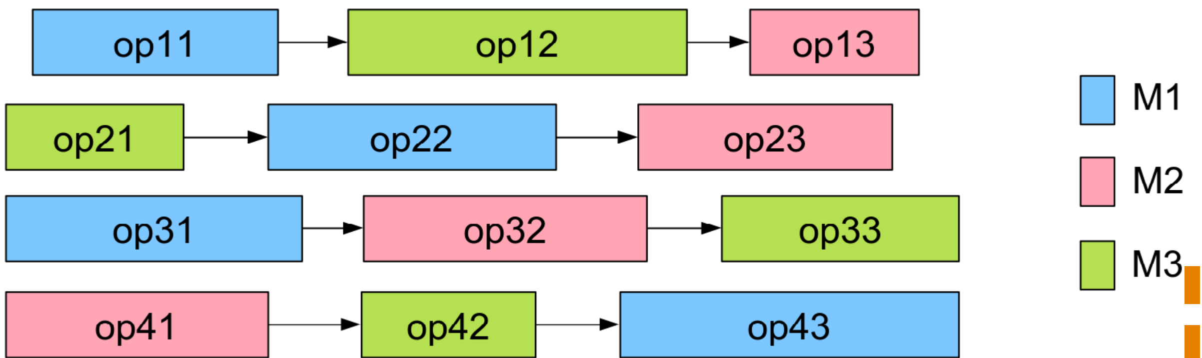
Příklad: minimalizace času dokončení poslední úlohy (tzv. makespanu)

```
minimize max(i in 1..n) endOf(x[i])
```

Příklad: rozvrhování problému job-shop

Job-shop problém:

- problém rozvrhování úloh, které se skládají z nepřekrývajících operací
- každá operace úlohy prováděna na jiném stroji
- pořadí operací úlohy předem určeno
- unární stroje




```
1   dvar interval op[j in Jobs][p in Pos] size Ops[j][p].pt;
2   dvar sequence mchs[m in Mchs] in
3     all(j in Jobs, p in Pos: Ops[j][p].mch == m) op[j][p];
4
5   minimize max(j in Jobs) endOf(op[j][nbPos]);
6   subject to {
7     forall(m in Mchs)
8       noOverlap(mchs[m]);
9     forall(j in Jobs, p in 2..nbPos)
10      endBeforeStart(op[j][p-1],op[j][p]);
11  }
```

```
tuple Oper {
    int mch; // Machine
    int pt; // Processing time
    };
Oper Ops[j in Jobs][m in Mchs] = ...;
př. Ops[1][2]=<3,6>
```

Kumulativní funkce

Hodnota **výrazu kumulativní funkce** reprezentuje vývoj kvantity v čase, která může být inkrementálně změněna (snížena nebo navýšena) intervalovými proměnnými.

Příklady:

- intervaly využívají počty pracovníků
- intervaly využívají skladové zásoby

Intervalové proměnné x[i] přispívají do kumul. funkce po dobu svého provádění

```
int wor[1..5] = [1,3,2,4,1];
cumulFunction y = sum(i in 1..5)  pulse(x[i],wor[i]);
```


Omezení na výrazech kumulativní funkce

```
int h = ...              dvar int h = ...
cumulFunction f= ...     cumulFunction f= ...
f<=h                     f<=h
```

Příklad: RCPSP

Resource-contrained project scheduling problem (RCPSP)

```
tuple Task {
  key int id;
  int     pt;
  int     qty[Resources];
  {int}   succs;
}
{Task} Tasks = ...;
```

a1, a2, a3, a4, a5, a6, a7, a8

R1

R2

```
 1  dvar interval a[i in Tasks] size i.pt;
 2  cumulFunction usage[r in Resources] =
 3    sum(i in Tasks: i.qty[r]>0) pulse(a[i],i.qty[r]);
 4  minimize max(i in Tasks) endOf(a[i]);
 5  subject to {
 6    forall(r in Resources)
 7      usage[r] <= Capacity[r];
 8    forall(i in Tasks, j in i.succs)
 9      endBeforeStart(a[i], a[<j>]);
10  }
```

<j> odkazuje jako klíč na celý tuple

Alternativní podmínka

Pokud je interval x přítomný, pak je právě jeden z intervalů y1,...,yn přítomný a je synchronizován s x (má stejný start a konec)

- `alternative(x, [y1,...,yn])`
- Pokud je x nepřítomné, pak jsou yi také nepřítomné

Rozšíření: právě k intervalů z y1,...,yn je synchronizováno s x

- `alternative(x, [y1,...,yn], k)  //k: celočíselný výraz`

Příklad použití:

- každá intervalová proměnná x[t] vyžaduje nbWorkers[t] přítomných intervalů mezi assigned[t][w] pro pracovníky w (kód ještě nutno rošířit o nepřekrývání pro pracovníky).
- ```
  dvar interval x[t in Tasks] size pt[u];
  int nbWorkers[t in Tasks];
  dvar interval assigned[Tasks][Workers] optional;
  forall (t in Tasks)
          alternative(x[t], all (w in Workers) assigned[t][w], nbWorkers[t]);
  ```
  ```

Konzistenční algoritmus pro nebinární podmínky

- Obecný konzistenční algoritmus
- Varianta AC-3 s **frontou proměnných**

  rozšíření AC-2001 na nebinární podmínky
- Opakovaně se provádí revize podmínek, dokud se mění domény

```
procedure Nonbinary-AC-3-with-Variables(Q)
while Q non empty do
      vyber a smaž Vj ∈ Q
      for ∀ C takové, že Vj ∈ scope(C) do
          W := revise(Vj, C)
             // W je množina proměnných jejichž, doména se změnila
          if ∃ Vi ∈ W taková, že Di = ∅ then return fail
          Q := Q ∪ {W}
end Nonbinary-AC-3-with-Variables
```

Revizní procedura: různé typy konzistence

Speciální `revise` procedury jsou definovány v závislosti na typu omezení,

tj. `revise` procedura může implementovat

- obecnou hranovou konzistenci
- konzistenci mezí
- konzistenční algoritmy pro globální podmínky jako `allDifferent`
- ...

Revizní procedura

- Uživatel má často možnost definovat vlastní `revise` proceduru
- Je potřeba **určit událost, která revizi vyvolá**

  událostí je změna domény proměnné (*suspension*)
  - vyvolání revize pouze při dané změně proměnné
    - při libovolné změně domény (u obecné hranové konzistence)
    - při změně mezí (u konzistence mezí)
    - při instanciaci proměnné
  - tj. pro každou proměnnou lze použít různé události
  - revize jednolivých podmínek jsou realizovány v závislosti na aktivaci odpovídající události (událostmi řízený výpočet)
- Je potřeba **navrhnout propagaci přes podmínku pro danou událost**
  - výsledkem propagace je zmenšení domén proměnných
  - pro jednu podmínku lze mít více propagačních kódů

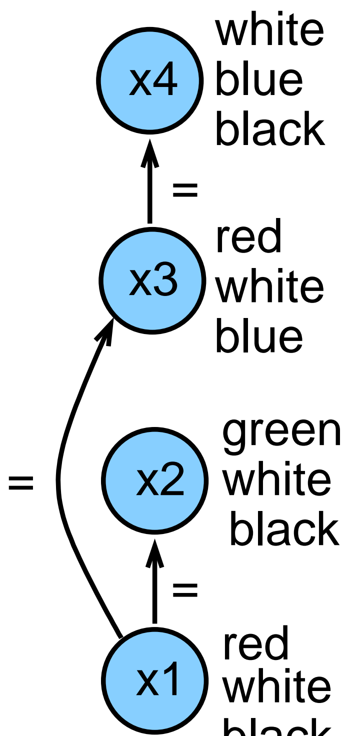
Základní konzistenční algoritmus s událostmi

```
procedure Nonbinary-AC-3-with-Events(Q)
while Q non empty do
      vyber a smaž event(V_j) ∈ Q
      for ∀C takové, že V_j ∈ scope(C) a C čeká na event(V_j) do
          W := revise(event(V_j),C)
           // jsou vyvolány pouze ty revize, které čekají na danou event(V_j)
           // W je množina proměnných jejichž, doména se změnila
          if ∃ V_i ∈ W taková, že D_i = ∅ then return fail
          Q := Q ∪ {W}
end Nonbinary-AC-3-with-Events
```

Směrová konzistence

Směrová hranová konzistence (pro binární CSP)

- CSP je **směrově hranově konzistentní** vzhledem k uspořádání proměnných $(x_1, \ldots, x_n)$, právě když je každá hrana (x_j, x_i) hranově konzistentní pro každé $j \leq i$.



- procedure

  DAC(G, $(x_1, \ldots, x_n)$) *Directed arc consistency* ***DAC***

  for $i = n$ to 1 by -1 do

  for $\forall j < i$ takové, že existuje hrana (x_j, x_i) do

  revise $((j, i))$

  end DAC

- Příklad: $x_1 = x_2$, $x_1 = x_3$, $x_3 = x_4$
  - D1={red,white,black}, D2={green,white,black}, D3={red,white,blue}, D4={white,blue,black}
  - Po DAC: D4={white,blue,black}, D3={white,blue}, D2={green,white,black}, D1= {white}, není AC, ale máme řešení bez navracení vzhledem k (x_1, x_2, x_3, x_4)
- Opakování revizí není nutné, doména revidované proměnné se v dalších cyklech nemění
- Složitost $O(ek^2)$ každá hrana se reviduje jednou se složitostí $O(k^2)$

Směrová i-konzistence

- Algoritmus pro DAC byl pouze pro binární CSP
- i-konzistence vyžaduje uvažování omezení s větším rozsahem proměnných
  - musíme aktualizovat relace až nad $(i-1)$ proměnnými

$\Rightarrow$ uvažujeme obecné CSP (tedy i **nebinární podmínky**)

- CSP je **směrově i-konzistentní** vzhledem k uspořádání proměnných $(x_1, \ldots, x_n)$ právě tehdy, když každých $(i-1)$ proměnných je i-konzistentní vzhledem ke všem proměnným, které je následují v uspořádání.
- CSP je **silně směrově i-konzistentní (DIC-i)** právě tehdy, když je směrově j-konzistentní pro každé $j \leq i$

Vlastnosti DIC-i

- Opakování
  - AC = silná 2-konzistence
  - PC = silná 3-konzistence
- DIC-2 je ekvivalentní DAC
  - u obou uvažujeme unární a binární omezení
  - DAC je definován pouze na binární CSP
  - DIC-2 je sice definován pro obecné CSP, ale je pouze schopen k dané hodnotě proměnné hledat konzistentní hodnotu v doméně další proměnné, nezachytí tedy omezení s více než dvěmi proměnnými
- DIC-3 lze podobně srovnat

  se směrovou konzistencí po cestě (directed path consistency, DPC)
- Algoritmus pro silnou směrovou konzistenci

  viz [Dechter, Constraint Processing]

Řešení bez navracení pomocí DIC-i

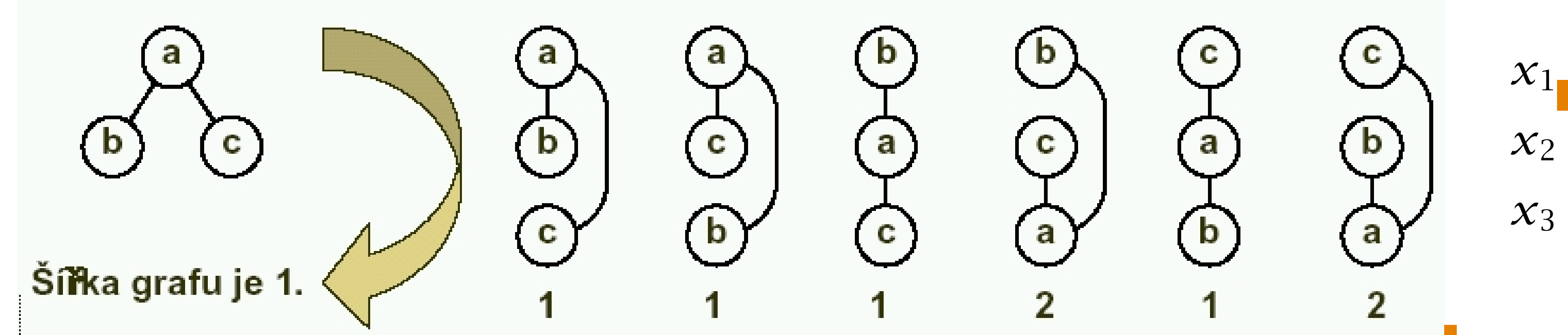
- Opakování:

  CSP vyřešíme bez navracení vzhledem k uspořádání proměnných $(x_1, \ldots, x_n)$, jestliže pro každé $i \leq n$ může být každé částečné řešení $(x_1, \ldots, x_i)$ konzistentně rozšířeno o proměnnou x_{i+1}.
- Proměnná musí být kompatibilní s již ohodnocenými proměnnými, tj. s tolika proměnnými, kolik má „zpětných" hran
- Pro i zpětných hran potřebujeme směrovou $(i + 1)$-konzistenci
- Je-li m maximum počtu zpětných hran pro všechny vrcholy, stačí nám silná směrovou $(m + 1)$-konzistence
- Při různém uspořádání vrcholů je počet zpětných hran různý

⇒ hledáme **uspořádání vrcholů s nejmenším počtem zpětných hran m**

Šířka grafu

- **Uspořádaný graf** je graf s lineárním uspořádáním vrcholů.
- **Šířka vrcholu** v uspořádaném grafu je počet hran vedoucích z tohoto vrcholu do předchozích vrcholů.
- **Šířka uspořádaného grafu** je maximum z šířek jeho vrcholů.
- **Šířka grafu** je minimum z šířek všech jeho uspořádaných grafů.



x_1
x_2
x_3

- procedure MinWidthOrdering((V,E)) (konstruujeme od konce)

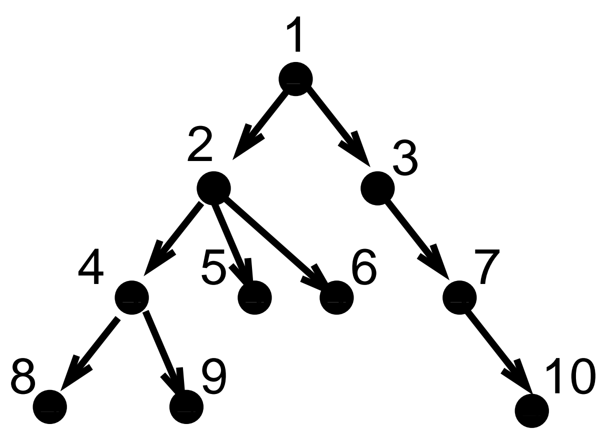
```
Q := {}                          (do vybraného uzlu povede nejméně zpětných hran)
while V not empty do N := vyber a smaž uzel s nejmenším počtem hran z (V,E)
                     zařad' N do Q
return Q
```

Graf podmínek s šířkou 1 = strom

- *Tvrzení:* Graf je strom právě tehdy, když má šířku 1.
- *Důkaz:*
  - Předpoklad: Strom neobsahuje cyklus.

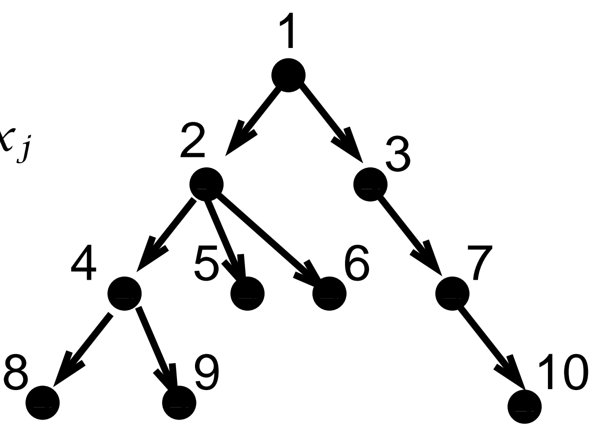
  ⇐ Mějme graf šířky 1. Pokud by obsahoval cyklus, tak bychom pro libovolné uspořádání proměnných měli proměnnou se dvěma rodiči, tj. spor.

  ⇒ Mějme graf bez cyklů. Pak lze vytvořit orientovaný strom s kořenem tak, že všechny hrany směřují z kořenového uzlu. V tomto stromu má každý uzel nejvýše jednoho rodiče. Libovolné uspořádání, ve kterém rodič předchází potomky ve stromě, má šířku 1.



Konzistence pro stromové CSP

- *Tvrzení:* Nechť $d = (x_1, \ldots, x_n)$ je uspořádání stromového grafu podmínek T pro daný CSP. Jestliže T je směrově hranově konzistentní vzhledem k d, pak má CSP řešení bez navracení vzhledem k d.
- *Důkaz:*
  - Uvažujme uspořádání proměnných $d = (x_1, \ldots, x_n)$ s šířkou 1.
  - Předpokládejme, že $x_1, \ldots, x_i$ jsou konzistentně nainstanciovány.
  - Snažíme se nainstanciovat x_{i+1}:
    - d je uspořádaní šířky 1, tedy existuje pouze jeden rodič x_j ($j \leq i$) proměnné x_{i+1}, který může omezovat x_{i+1} :wq
    - (x_j, x_{i+1}) je hranově konzistentní (z DAC), tedy existuje hodnota konzistentní se současným přiřazením x_j
    - tuto hodnotu přiřadíme x_{i+1}



Algoritmus zajištění DAC pro stromové CSP

- `procedure TreeSolving(T)`

  `nalezení uspořádání` $(x_1,...,x_n)$ `s šířkou 1 pomocí stromu T`

  `(rodič předchází potomky)`

  `necht'` $x_{p(i)}$ `označuje rodiče` x_i `v uspořádaném stromu`

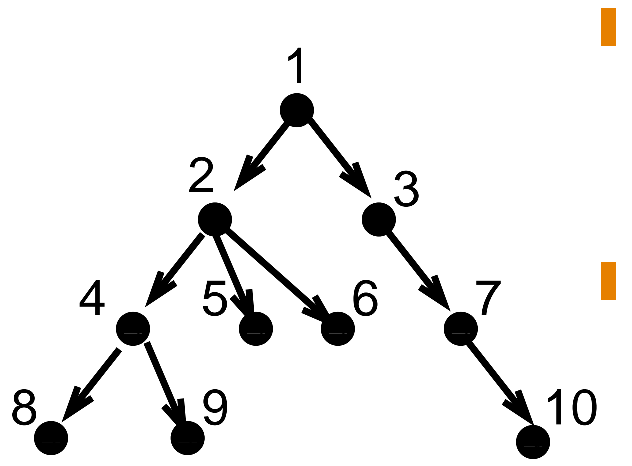
  `for` $i = n$ `to 1 by -1 do`

      `revise(`$(x_{p(i)}, x_i)$`)`

      `if` $D_{p(i)} = \emptyset$ `exit (řešení neexistuje)`

  `end TreeSolving`

  Cvičení: `X1,X2,X3,X4 in 1..5, X1=X2+1, X1<X3, X3<X4`



⇒ Složitost algoritmu $O(nk^2)$

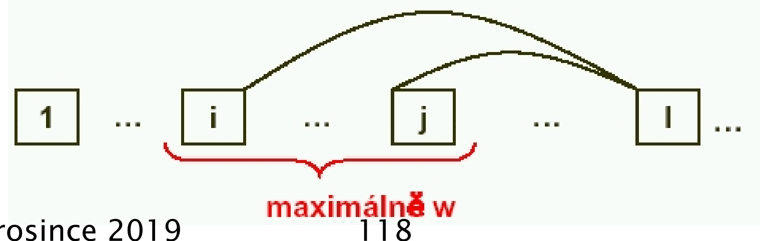
- Algoritmus zajistí DAC pro stromové CSP,

  tj. bude možné nalézt řešení bez navracení

- Pokud aplikujeme DAC vzhledem k uspořádání šířky 1, a pak v opačném směru, tak dosáhneme plné hranové konzistence. viz Barták, přednáška

Šířka grafu a stupeň konzistence

- *Tvrzení:* Nechť je dán CSP, jehož uspořádaný graf podmínek s uspořádáním d má šířku $i-1$. Jestliže je problém silně směrově i-konzistentní, pak je problém řešitelný bez navracení vzhledem k d.
- *Důkaz:*
  - existuje uspořádaný graf s uspořádáním d a šířkou $i-1$
  - speciálně, počet zpětných hran pro každou proměnnou je maximálně $i-1$
  - proměnné ohodnocujeme v pořadí uspořádání grafu
  - nyní, pokud ohodnocujeme proměnnou:
    - musíme najít hodnotu kompatibilní se všemi již ohodnocenými proměnnými, které jsou s proměnnou spojené podmínkou (hranou)
    - nechť takových proměnných je m, potom $m \leq (i-1)$ ($\Leftarrow$ graf má šířku $(i-1)$)
    - graf je směrově i-konzistentní, tedy taková hodnota musí existovat ($\Leftarrow$ z definice)



Adaptivní konzistence

- *Původní intuice:* proměnná musí být kompatibilní s již ohodnocenými proměnnými, tj. s tolika proměnnými, kolik má „zpětných” hran.
  - Je tedy nutná směrová i-konzistence odpovídající šířce grafu?
- *Problém:* algoritmy silné směrové i-konzistence pro $i \geq 3$ mění graf podmínek přidáváním hran, nutná úroveň i se může zvětšovat
- Algoritmus **adaptivní konzistence ADC** pro nalezení řešení bez navracení
  - uvažujme uspořádání proměnných d
  - rekurzivně vytváříme směrovou i-konzistenci (od poslední k první proměnné v d)
  - měníme úroveň i od uzlu k uzlu tak, abychom se **adaptovali aktuální šířce uzlu** v momentě zpracování
- Proč algoritmus funguje?
  - v momentě zpracování uzlu je určena finální šířka uzlu
  - víme tedy, jakou úroveň směrové i-konzistence musíme dosáhnout

Polynomiální CSP

- w šířka grafu, n počet proměnných, k horní mez velikosti domén
- Složitost DIC-i $O(nw^i \cdot (2k)^i)$ důkaz viz Dechter, Constraint Processing
- Časová a prostorová složitost algoritmu adaptivní konzistence je $O(n \cdot (2k)^{w+1})$ a $O(n \cdot k^w)$ důkaz viz Dechter, Constraint Processing
- *Věta:* Třída CSP problémů, jejichž šířka grafu je ohraničena konstantou b je řešitelná v polynomiálním čase a prostoru.
- Použitelnost algoritmu ADC
  - ADC není jen procedura pro rozhodnutí konzistence
  - ADC může sloužit i ke kompilaci
    - ADC transformuje problém na ekvivalentní graf, ze kterého může být každé řešení odvozeno v lineárním čase
    - prostorová složitost ale zůstává

Prohledávání a pohled dopředu

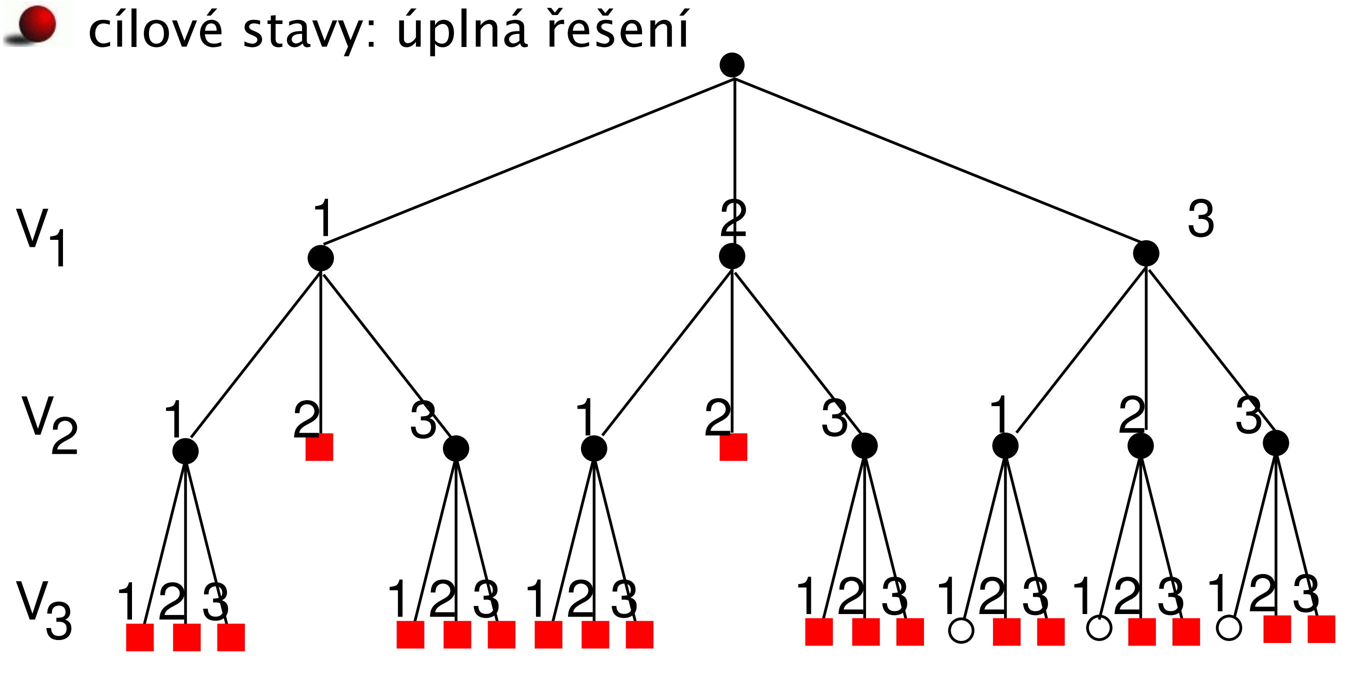
Přehled prohledávacích algoritmů pro CSP

- **Rozšiřování částečného konzistentního přiřazení**
  - stromové prohledávání
    - backtracking
    - pohled dopředu (propagace omezení)
    - pohled zpět (inteligentní vracení)
    - neúplná stromová prohledávání
- **Procházení úplných nekonzistentních přiřazení**
  - generuj a testuj
  - lokální prohledávání
- **Kombinování úplných nekonzistentních přiřazení**
  - *population-based search*

Prohledávací algoritmy pro CSP

Prohledávací algoritmy prochází (traversálně) **graf stavového prostoru**

- uzly grafu (stavy): konzistentní částečné instanciace
- iniciální stav: prázdné přiřazení
- hrany grafu: operátory, které rozšíří částečné řešení $[x_1/a_1, \ldots, x_j/a_j]$ o přiřazení jedné proměnné, které není v konfliktu s dřívějšími přiřazeními, tj. vznikne $[x_1/a_1, \ldots, x_j/a_j, x_{j+1}/a_{j+1}]$
- cílové stavy: úplná řešení



- červené čtverečky: chybný pokus o instanciaci, řešení neexistuje
- nevyplněná kolečka: nalezeno řešení (cílové stavy)
- černá kolečka: vnitřní uzel, máme pouze částečné přiřazení

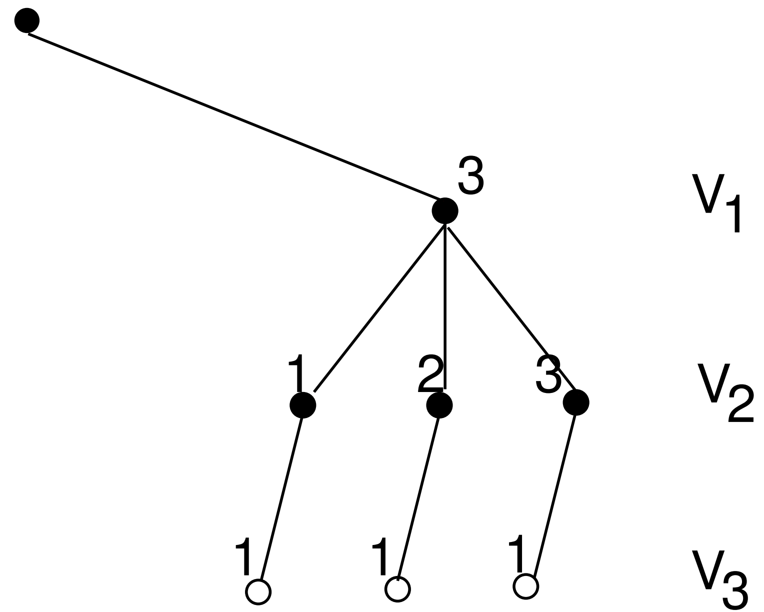
Prohledávací algoritmy pro CSP: vlastnosti

Bod volby: z uzlu grafu vede více hran

- máme na výběr, kterou hodnotu přiřadíme proměnné

CSP má **řešení bez navracení** vzhledem k uspořádání d, jestliže všechny listy v jeho grafu stavového prostoru reprezentují řešení.

- v grafu nejsou žádné slepé větve

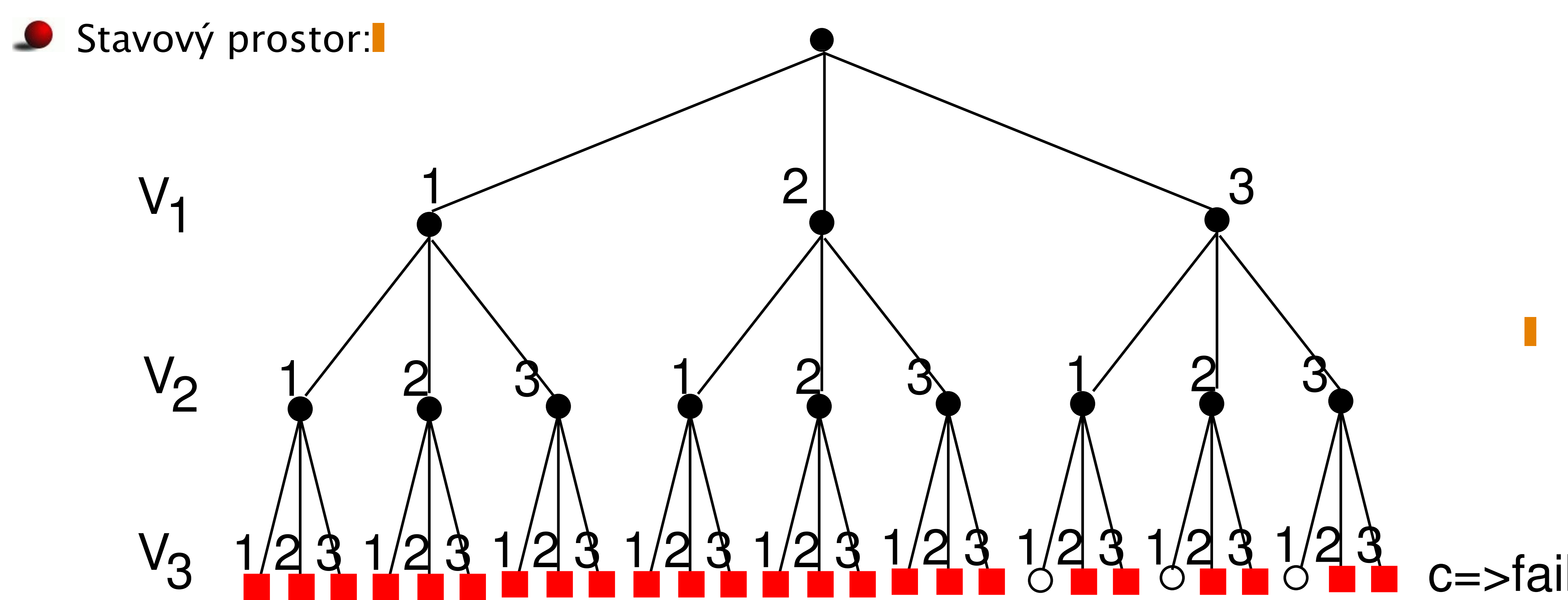


Backtracking (BT)

- **Prohledávání stavového prostoru do hloubky**
- Dvě fáze backtrackingu
  - **dopředná fáze**: proměnné jsou postupně vybírány, rozšiřuje se částečné řešení přiřazením konzistentní hodnoty (pokud existuje) pro další proměnnou
  - **zpětná fáze**: pokud neexistuje konzistentní hodnota pro aktuální proměnnou algoritmus se vrací k předchozí přiřazené hodnotě
- Proměnné dělíme na
  - **minulé** – proměnné, které už byly vybrány (a mají přiřazenu hodnotu)
  - **aktuální** – proměnná, která je právě vybrána a je jí přiřazována hodnota
  - **budoucí** – proměnné, které budou vybrány v budoucnosti

Příklad: backtracking

- Příklad: $V_1, V_2, V_3 \in \{1,2,3\}, \quad c : V_1 = 3 \times V_3$
- Stavový prostor:



Příklady vizualizací viz `http://www.fi.muni.cz/~hanka/vis`

- zpětná vazba k této bakalářská práci?
- viz kontakt na stránkách

Algoritmus backtrackingu

```
procedure Backtracking((X,D,C))
i := 1                          (inicializace čítače proměnných)
D'_i := D_i                     (kopírování domény)
while 1 ≤ i ≤ n
      přiřazení x_i := Select-Value
      if x_i is null            (žádná hodnota nebyla vrácena)
         i := i − 1            (zpětná fáze)
      else i := i + 1          (dopředná fáze)
           D'_i := D_i
if i = 0 return ,,nekonzistentní''
else return přiřazené hodnoty {x_1,...,x_n}
end Backtracking
```

- Algoritmus vrací pouze první řešení
- Série domén D'_i: $\forall\, i : D'_i \subseteq D_i$.

  Select-Value pracuje nad (promazává) D'_i

  Hodnoty v D'_i ještě netestovány vzhledem k aktuálnímu částečnému přiřazení

Uspořádání hodnot pro backtracking

- ```
  procedure Select-Value
  while D'_i is not empty
        vyber a smaž libovolný a ∈ D'_i
        if Consistent(a⃗_{i-1}, x_i = a) return a
  return null
  ```
  (procedure Select-Value; while D'_i is not empty; vyber a smaž libovolný $a \in D'_i$; if Consistent($\vec{a}_{i-1}, x_i = a$) return a; return null)

- Backtracking ověřuje v každém kroku konzistenci podmínek vedoucích z minulých proměnných do aktuální proměnné

- Backtracking tedy zajišťuje konzistenci podmínek
  - na všech minulých proměnných
  - na podmínkách mezi minulými proměnnými a aktuální proměnnou
  ```

Problémy a vylepšení backtrackingu

- ***Thrashing***: opakované objevování stejných nekonzistencí a částečných úspěchů při prohledávání
- **Algoritmy pohledu dopředu** kontrolují podmínky dopředu
  - backtracking odhalí nesplnění podmínky teprve když přiřadí hodnoty jejím proměnným
    příklad: `A,B,C,D,E in 1..10, A=3*E`
    konzistenčními algoritmy lze předem upravit domény A a E
- **Backjumping** se vrací k původci chyby
  - příklad: `A,B,C,D,E in 1..10, A>E`
    backtracking vyzkouší všechny možnosti pro `B,C,D` než odhalí $A \neq 1$
    hned po prvním neúspěšném přiřazení E se lze vrátit k přiřazování A
- **Dynamický backtracking:** změna uspořádání minulých proměnných
- **Neúplná prohledávání**: hledání pouze v některých částech stavového prostoru

Algoritmy pohledu dopředu (*look-ahead*) LA

- Používají propagace omezení
- Vyvolány před přiřazováním hodnoty další proměnné
- Snaží se zjistit, jak rozhodnutí o výběru proměnných a hodnot ovlivní budoucí prohledávání
- Po provedení propagace omezení lze mnohem lépe
  - rozhodnout, **kterou proměnnou přiřadit**
    - většinou lepší přiřadit proměnné, které nejvíce omezují zbytek stavového prostoru
    - příklad: `A,B,C,D,E in 1..10, D=A*B*C*E, E>5`, lépe je začít s E
  - rozhodnout, **kterou hodnotu přiřadit proměnné**
    - snaha o výběr hodnoty, která umožní nejvíce voleb v budoucích přiřazeních
    - příklad: `A,B,C in 1..10, A*B<10, B=C*2`, pro A je lepší vybrat 1
- Vylepšení složitosti nejhoršího případu oproti naivnímu backtrackingu zřídka. Cílem je snaha o efektivní využití algoritmů propagace omezení.

Pohled dopředu pro výběr hodnoty

rozdíly od backtrackingu

```
procedure Look-Ahead((X,D,C))
i := 1                        (inicializace čítače proměnných)
D'_i := D_i pro 1 ≤ i ≤ n     (kopírování domény)
while 1 ≤ i ≤ n
      přiřazení x_i := Select-Value-XXX
      if x_i is null          (žádná hodnota nebyla vrácena)
         i := i − 1           (zpětná fáze)
         nastav všechny D'_k, k > i na jejich hodnotu před poslední instanciací x_i
      else i := i + 1         (dopředná fáze)
if i = 0 return ,,nekonzistentní''
else return přiřazené hodnoty {x_1,...,x_n}
end Look-Ahead
```

- Select-Value-XXX se liší dle typu LA algoritmu
- Ukládáno n kopií každé D'
  - pro každou úroveň ve stavovém prostoru jedna kopie

Kontrola dopředu (*forward checking*) FC

- procedure Select-Value-Forward-Checking

  while D'_i is not empty

  vyber a smaž libovolný $a \in D'_i$

  for $\forall k,\ i < k \leq n$

  for $\forall b \in\ D'_k$

  if not Consistent($\vec{a}_{i-1}, x_i = a, x_k = b$)

  smaž b z D'_k

  if $\exists k, i < k \leq n :\ D'_k$ is empty ($x_i = a$ vede do slepé větve, nevybírej a)

  nastav všechny $D'_k, i < k \leq n$ na hodnotu před výběrem a

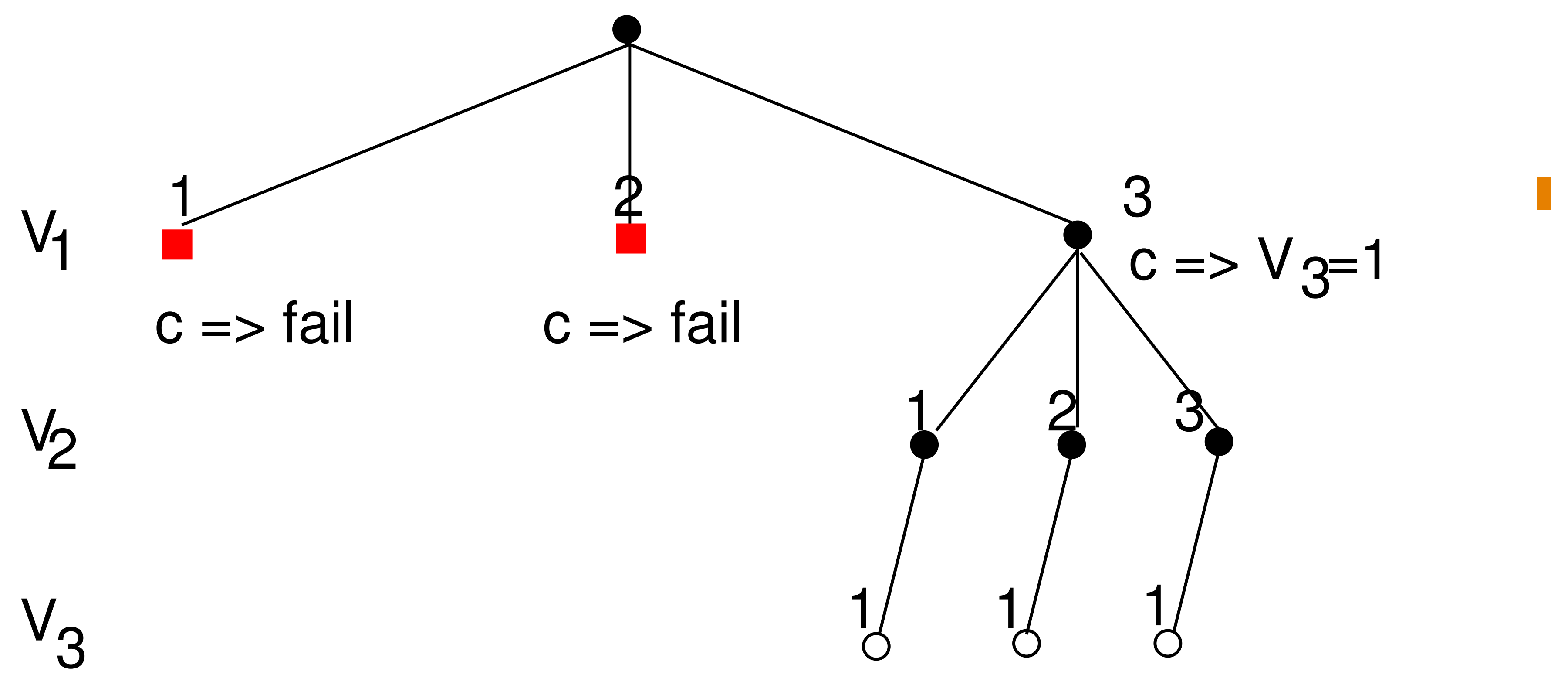
  else return a

  return null

- FC je rozšíření backtrackingu
- FC navíc zajišťuje konzistenci mezi aktuální proměnnou a budoucími proměnnými, které jsou s ní spojeny dosud nesplněnými podmínkami
- Hrany z aktuální proměnné do minulých proměnných není nutno testovat

Příklad: kontrola dopředu

- Příklad: $V_1, V_2, V_3 \in \{1, 2, 3\}, \quad c : V_1 = 3 \times V_3$
- Stavový prostor:



Opravdový úplný pohled dopředu (RFLA)

(Real Full) Look-Ahead, RFLA n. LA

n. Maintaining Arc-Consistency MAC

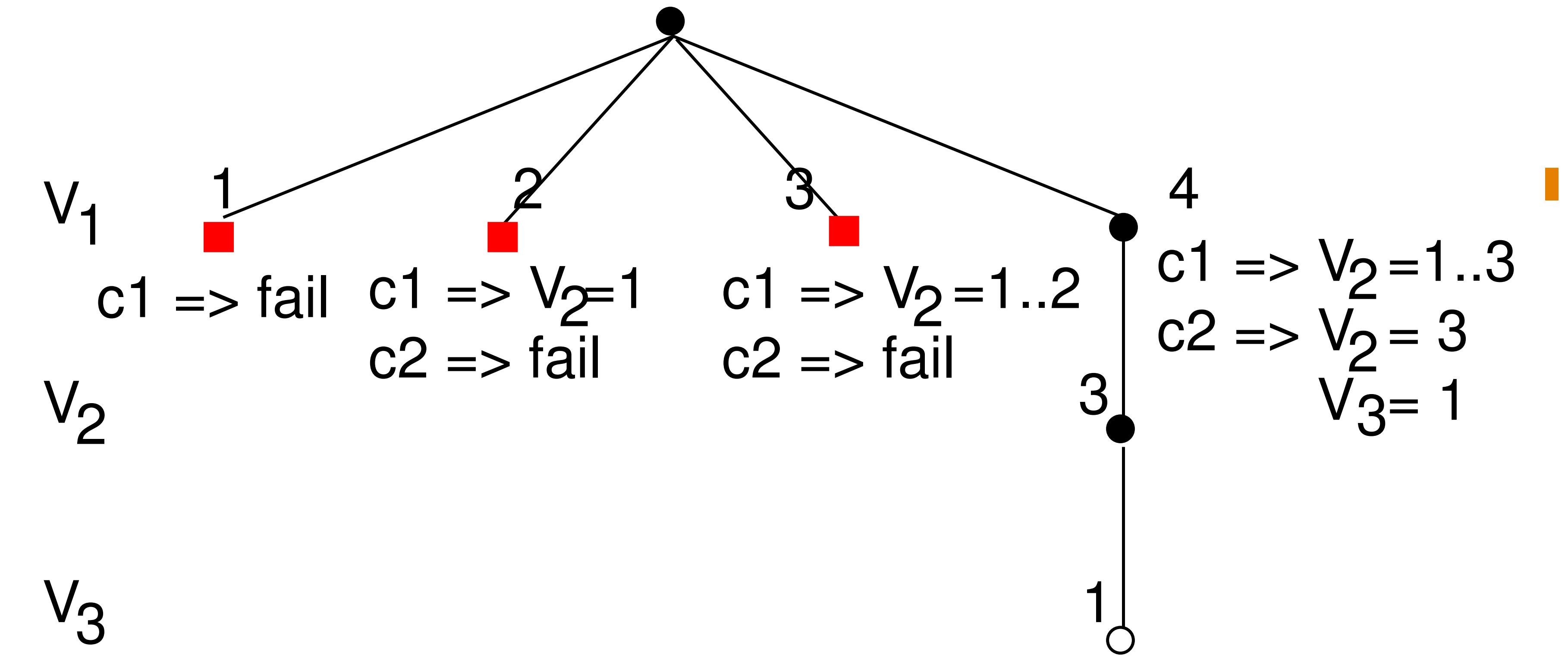
Implementace AC-1 algoritmu

Lze nahradit silnějšími AC algoritmy

```
procedure Select-Value-Look-Ahead
while D'_i is not empty
      vyber a smaž libovolný a ∈ D'_i
      repeat Deleted := false
             for ∀ j, i < j ≤ n
                 for ∀ k, i < k ≤ n
                     for ∀ b ∈ D'_j
                         if neexistuje c ∈ D'_k taková, že
                            Consistent(a⃗_{i-1}, x_i = a, x_j = b, x_k = c)
                         smaž b z D'_j
                         Deleted := true
      until Deleted = false
      if ∃ k, i < k ≤ n takové, že D_k = ∅          (nevybírej a)
         nastav všechny D'_j, i < j ≤ n na hodnotu před výběrem a
      else return a
return null
```

Příklad: pohled dopředu

- Příklad: $V_1, V_2, V_3 \in \{1, 2, 3\}, \quad c1 : V_1 > V_2, \quad c2 : V_2 = 3 \times V_3$
- Stavový prostor:

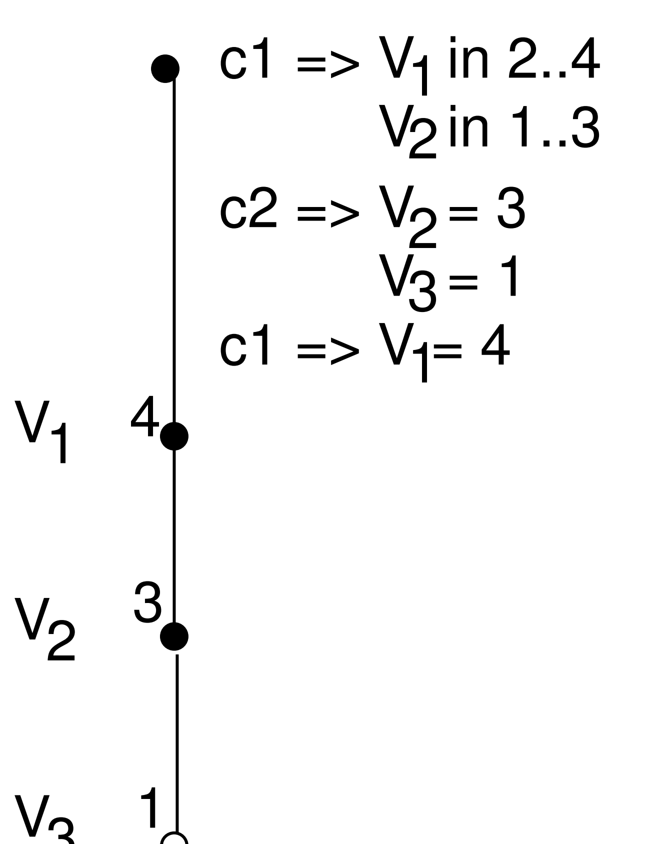


Prohledávání s iniciální konzistencí

Prohledávací algoritmus často aplikován až po **zajištění iniciální konzistence**

- typicky: iniciální konzistence a pak kontrola dopředu nebo

  iniciální konzistence a pak pohled dopředu



Příklad:

- pohled dopředu + iniciální konzistence
- V_1, V_2, V_3 in $1 \ldots 4$

  $c1 : V_1 > V_2, \quad c2 : V_2 = 3 \times V_3$

Srovnání algoritmů

- **Backtracking (BT)** kontroluje v kroku a podmínky

  $c(V_1, V_a), \ldots, c(V_{a-1}, V_a)$

  z minulých proměnných do aktuální proměnné

- **Kontrola dopředu (FC)** kontroluje v kroku a podmínky

  $c(V_{a+1}, V_a), \ldots, c(V_n, V_a)$

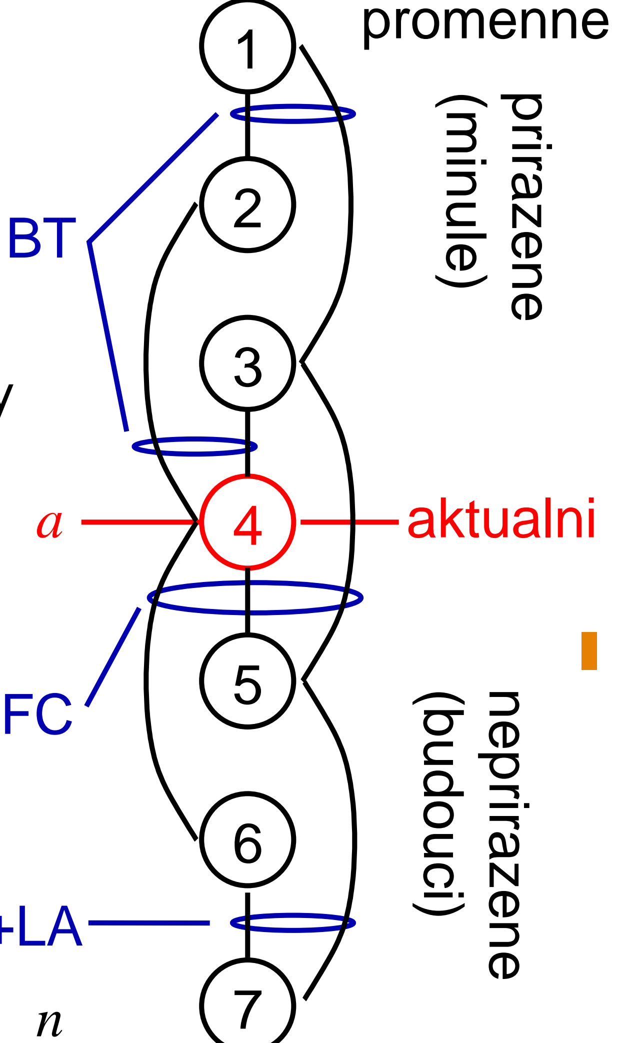
  z budoucích proměnných do aktuální proměnné

- **Pohled dopředu (LA)** kontroluje v kroku a podmínky

  $\forall l(a \leq l \leq n), \forall k(a \leq k \leq n), k \neq l : c(V_k, V_l)$

  z budoucích proměnných do aktuální proměnné

  a mezi budoucími proměnnými



- **Cvičení:** porovnejte jednotlivé algoritmy pro příklad $V_1, V_2, V_3 \in \{1, 2, 3, 4\}$, $V_1 = V_2 * 2, V_3 = V_2 + 1$, nakreslete odpovídající stavový prostor a u každého stavu uveďte, ke kterým propagacím omezení dochází.

Shrnutí: pohledu dopředu pro výběr hodnoty

- **Kontrola dopředu (*forward checking*) FC**
  - FC zajišťuje konzistenci mezi aktuální proměnnou a budoucími proměnnými, které jsou s ní spojeny dosud nesplněnými podmínkami
- **Opravdový úplný pohled dopředu (*real full look-ahead*) RFLA**
  - často se odkazuje: LA algoritmus **RFLA=LA** nebo **Maintaining Arc-Consistency MAC**
  - LA je rozšíření FC, LA zajišťuje hranovou konzistenci
  - LA navíc ověřuje i konzistenci všech hran mezi budoucími proměnnými
- **Úplný pohled dopředu (*full look-ahead*) FLA**
  - pouze jeden průchod `repeat until` cyklem algoritmu
- **Částečný pohled dopředu (*partial look-ahead*) PLA**
  - pouze jeden průchod `repeat until` cyklem algoritmu
  - nahrazuje `for` $\forall k, k \neq j, i < k \leq n$ výrazem `for` $\forall k, j < k \leq n$
    tj. budoucí proměnné porovnávány pouze s těmi, které je následují (DAC)

Výběr hodnoty

Jakým způsobem vybírat ze zbývajích hodnot v doméně proměnné?

- Triviální výběr hodnoty
  - doména procházena ve vzrůstajícím pořadí
  - doména procházena v klesajícím pořadí
- Obecný princip výběru hodnoty: **první úspěch *(succeed first)***
  - volíme pořadí tak, abychom výběr nemuseli opakovat
  - ?- $A, B, C \in \{1, 2\}, A = B + C$

    optimální výběr $A = 2, B = 1, C = 1$ je bez backtrackingu

Pohled dopředu pro dynamický výběr hodnoty

- Výběr hodnoty v doméně proměnné
  - zatím jsme vybírali první konzistentní hodnotu - jednalo se o **statický výběr hodnoty**
  - místo toho zkusíme proměnné přiřadit každou z hodnot v doméně a vyzkoušíme efekt použití FC nebo jiného LA algoritmu
- **Minimální konflikt**
  - výběr hodnoty, která smaže nejmenší počet hodnot z domén budoucích proměnných
  - experimentálně: tato heuristika vykazuje velmi dobré výsledky
- **Maximální velikost domény**
  - výběr hodnoty, která způsobí vytvoření největší minimální velikost domény mezi všemi budoucími proměnnými
  - intuice: proměnné, které mají malé domény, způsobí pravděpodobněji nekonzistenci

Výběr proměnných: statický

Uspořádání (výběr) proměnných má velký vliv na velikost stavového prostoru

- **Statické uspořádání proměnných:** výběr proměnných dán předem
  - triviálně: výběr nejlevější proměnné (tj. proměnné s nejmenším indexem)
  - empirické i teoretické studie ukázaly, že některá statická uspořádání vedou obecně ke zmenšení velikost stavového prostoru
- **Maximální kardinalita**
  - první proměnnou (uzel grafu) vybereme náhodně
  - každý uzel je spojen s maximálním počtem už uspořádaných (= dříve vybraných) vrcholů
  - proměnné vybíráme v pořadí dle tohoto počtu vrcholů
- **Minimální šířka**
  - proměnné uspořádány tak, aby byla minimalizována šířka grafu (minimalizován počet zpětných hran)
  - odzadu vybíráme proměnné s nejmenším počtem hran v aktualizovaném grafu (algoritmus viz dříve)

Výběr proměnných: dynamický

- **Dynamické uspořádání proměnných:** výběr proměnných počítán až v průběhu prohledávání
  - lze využít (výpočtů) algoritmů pohledu dopředu
- ***First-fail***
  - velmi často používaná metoda (vhodná jako default)
  - výběr proměnné, která nejvíce omezí zbytek stavového prostoru
  - výbereme proměnnou **s nejmenší doménou**
  - později už by mohlo být těžší pro tuto proměnnou nalézt správnou hodnotu
  - kombinace first-fail a maximální kardinality
  - $A, B, C \in \{1, 2, 3\}, A < 3, A = B + C$ nejlépe je začít s výběrem A

Pohled dopředu pro dynamický výběr proměnné

procedure Var-Ord-Look-Ahead((X, D, C)) — rozdíly od Look-Ahead

i := 1 (inicializace čítače proměnných)

D'_i := D_i pro $1 \leq i \leq n$ (kopírování domény)

s := $\min_{1 \leq j \leq n} \| D'_j \|$ (nalezni proměnnou s nejmenší doménou)

$x_1 := x_s$ (přeskládej proměnné tak, aby x_s byla první)

while $1 \leq i \leq n$

   přiřazení x_i := Select-Value-XXX

   if x_i is null (žádná hodnota nebyla vrácena)

      i := $i - 1$ (zpětná fáze)

      nastav všechny $D'_k, k > i$ na jejich hodnotu před poslední instanciací x_i

   else if $i < n$

         s :=$\min_{i < j \leq n} \| D'_j \|$ (nalezni budoucí proměnnou s nejmenší doménou)

         $x_{i+1} := x_s$ (přeskládej proměnné tak, aby x_s následovala x_i)

         i := $i + 1$ (dopředná fáze)

if $i = 0$ return „nekonzistentní“

else return přiřazené hodnoty $\{x_1, \ldots, x_n\}$

end Var-Ord-Look-Ahead

Silnější pohled dopředu

- Algoritmy pohledu dopředu zatím uvažovaly pouze hranovou konzistenci
  - vede k mazání hodnot z domény
- Po každém přiřazení hodnoty proměnné lze vyžadovat dosažení vyššího stupně konzistence
  - navíc jsou přidávána i nová omezení
- Kdy má cenu vyšší úrovně konzistence uvažovat?
  - vhodné pro propagaci nad podmnožinou proměnných
    - globální podmínky
  - nebo při rozhodování o výběru hodnoty
    - nepřidáváme nová omezení

Doplňkové materiály

Course on Constraint Programming and Scheduling **(10 hodin)**

- bakalářský kurz vyučovaný v angličtině na Hochschule Konstanz, Technik, Wirtschaft,und Gestaltung (Německo) v květnu 2009
- `http://www.fi.muni.cz/~hanka/konstanz09/`
- propagace, prohledávání, modelování, příklady
- omezující podmínky
- použití omezujících podmínek pro rozvrhování

Část přednášky ***Constraint Programming: Search***

- algoritmy v rekurzivním pseudokódu
- příklady prohledávání stavového prostoru včetně řešení

Pohled zpět

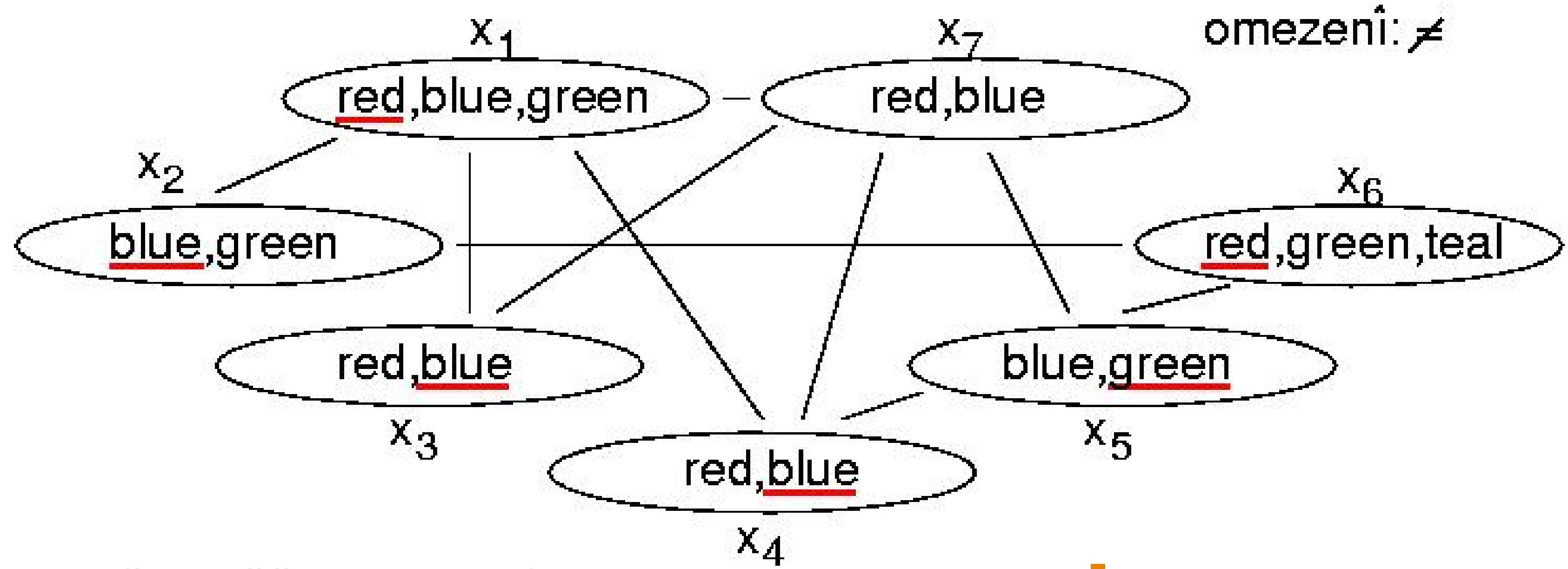
Přehled algoritmů pohledu zpět *look-back*

- **Algoritmy skoku zpět (*backjumping*)**
  - při zpětném průchodu se nevracíme pouze jeden krok jako algoritmus backtrackingu
  - snažíme se vracet co nejdále až ke zdroji chyby
- **Algoritmy učení (*constraint recording, no-good learning*)**
  - *no-good* = chybné částečné přiřazení
  - přidáme nová omezení, která zakazují nalezená chybná přiřazení
- **Dynamický backtracking**
  - při skoku zpět se snažíme nezapomínat už udělanou práci
  - měníme hodnoty pouze u minulých proměnných s konfliktem (ostatní neměníme)
  - realizace: změníme uspořádání proměnných
- **Backmarking**
  - pamatuje si, kde testy na konzistenci neuspěly
  - eliminuje opakování dříve provedených konzistenčních testů

Konfliktní množina

pevné uspořádání proměnných $(x_1, \ldots, x_n)$

- Mějme částečné konzistentní přiřazení $\vec{a} = (a_{i_1}, \ldots, a_{i_k})$ libovolné podmnožiny proměnných a uvažujme dosud nepřiřazenou proměnnou x. Jestliže neexistuje hodnota b z domény x tak, aby $(\vec{a}, x = b)$ bylo konzistentní, říkáme, že $\vec{a}$ je **konfliktní množina** x a nebo, že $\vec{a}$ je **v konfliktu s** x.

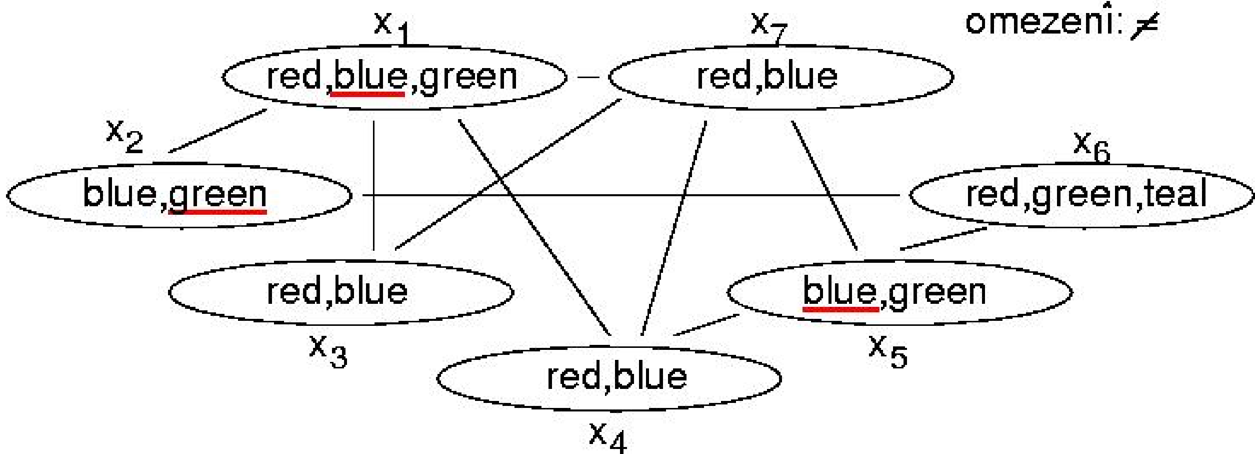


  - vyznačené přiřazení je konfliktní množina vzhledem k x_7

- Pokud $\vec{a}$ neobsahuje j-tici ($j < k$), která je v konfliktu s x, pak nazýváme $\vec{a}$ **minimální konfliktní množina**. $\{x_1/red, x_3/blue\}$: min.konf.množina vzhledem k x_7

Chybné přiřazení

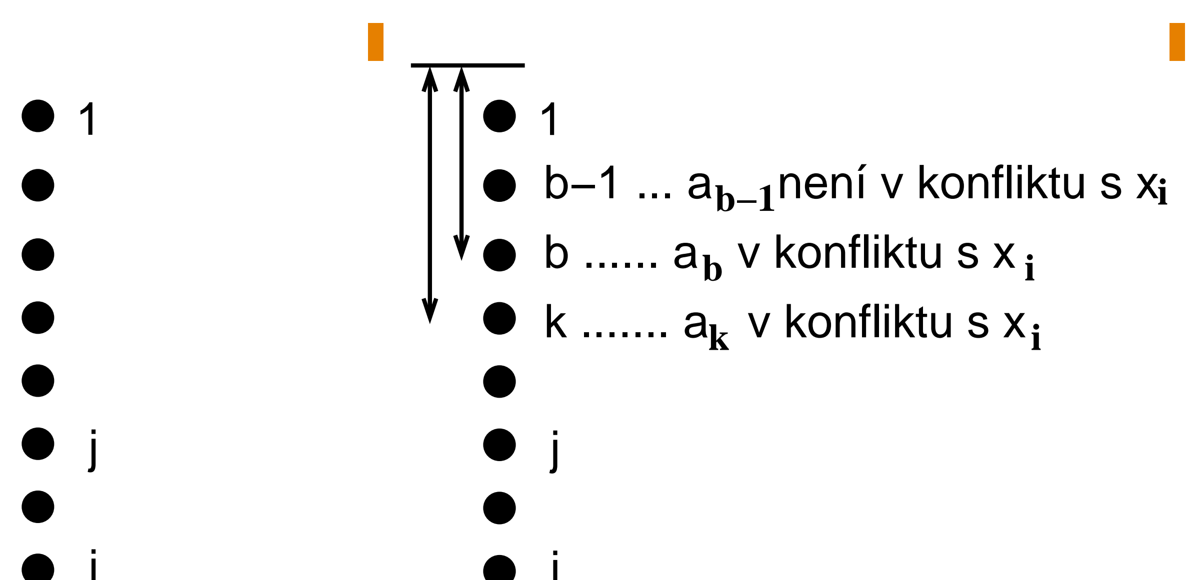
- Mějme částečné konzistentní přiřazení $\vec{a}_{i-1} = (a_1, \ldots, a_{i-1})$. Jestliže je $\vec{a}_{i-1}$ v konfliktu s x_i, pak ho nazýváme **list slepé větve**.
  - přiřazení v předchozím příkladu je list slepé větve
- Částečné přiřazení $\vec{a}$, které se nevyskytuje v žádném řešení CSP, se nazývá **chybné přiřazení (*no-good*)**.
  - konfliktní množiny jsou chybná přiřazení



  - $[x_1$/blue,x_2/green,x_5/blue$]$ je chybné přiřazení, ale nepatří do žádné konfliktní množiny
  - konflitní množina = chybné přiřazení vzhledem k jediné proměnné
- **Minimální chybná přiřazení** neobsahují chybná přiřazení méně proměnných.

Konfliktní proměnná

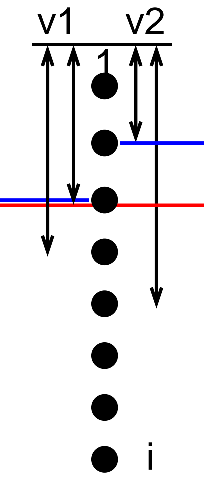
- Mějme list slepé větve $\vec{a}_{i-1}$. Proměnná x_j $(j < i - 1)$ je **bezpečná**, pokud je $\vec{a}_j$ chybné přiřazení.
  - také říkáme, že skok z x_i na x_j je bezpečný — nevynecháme žádné řešení
- Mějme list slepé větve $\vec{a}_{i-1}$. Proměnná x_b je **konfliktní proměnná (*culprit*)** vzhledem k $\vec{a}_{i-1}$, jestliže $b = \min\{k < i \mid \vec{a}_k$ v konfliktu s $x_i\}$.



- *Tvrzení:* Skok ke konfliktní proměnné je bezpečný.

Gaschnigův skok zpět

- ***Gaschnig's backjumping* (GBJ)**
  - když nedokážeme přiřadit hodnotu proměnné x_i,
    vracíme se zpět (skáčeme zpět) ke konfliktní proměnné
  - určení konfliktní proměnné
    - pro každou hodnotu vi ∈ x_i nalezneme proměnnou s nejmenším indexem,
      která je s x_i/vi v konfliktu
      (přiřazení této proměnné musíme určitě změnit, aby šla hodnota vi použít)
    - konfliktní proměnná je ta z nich, která má největší index
      (když její hodnotu změníme, můžeme zrušit konflikt s odpovídající hodnotou)



Algoritmus GBJ

Každá proměnná x_i uchovává v $latest_i$ index konfliktní proměnné

rozdíly od backtrackingu

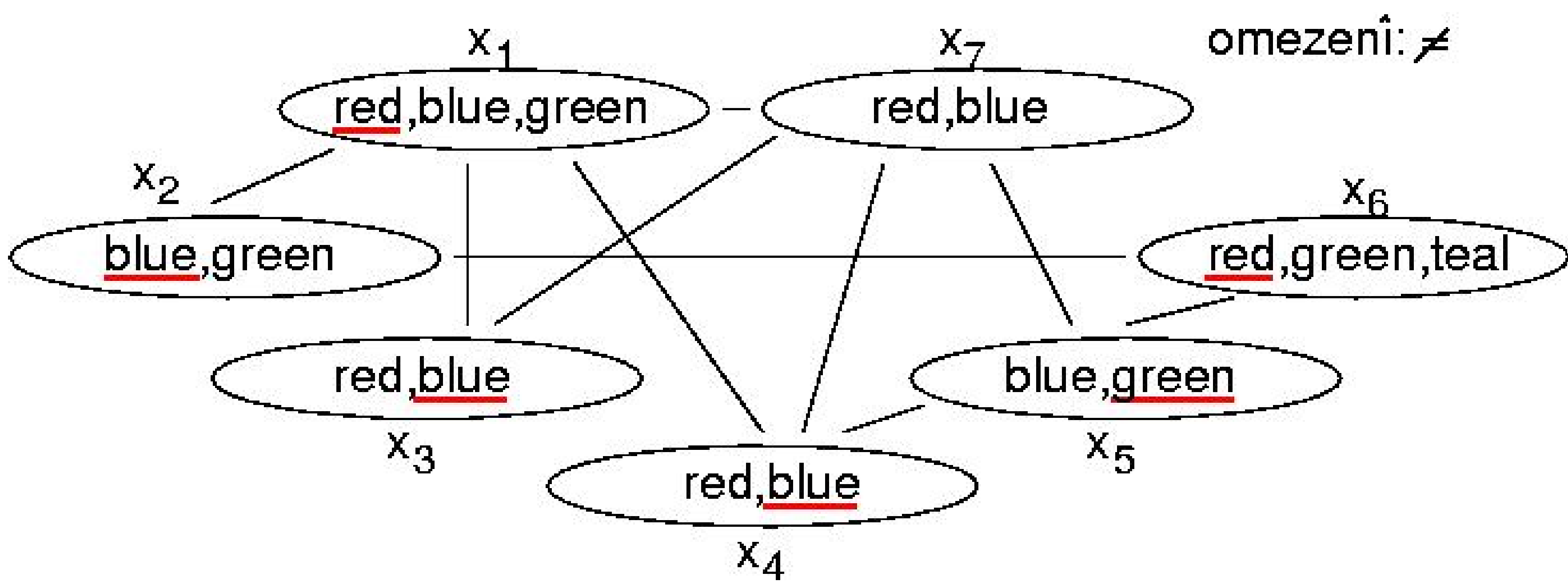
```
procedure GBJ((X,D,C))
i := 1                       (inicializace čítače proměnných)
D'_i := D_i                  (kopírování domény)
latest_i := 0                (inicializace čítače na konfliktní proměnnou
while 1 ≤ i ≤ n
      přiřazení x_i := Select-Value-GBJ
      if x_i is null         (žádná hodnota nebyla vrácena)
         i := latest_i       (skok zpět)
      else i := i+1          (dopředná fáze)
           D'_i := D_i
           latest_i := 0
if i = 0 return ,,nekonzistentní''
else return přiřazené hodnoty {x_1,...,x_n}
end GBJ
```

GBJ: výběr hodnoty

```
procedure Select-Value-GBJ
while D'_i is not empty
      vyber a smaž libovolný v ∈ D'_i
      consistent := true
      k := 1
      while k < i ∧ consistent
            if k > latest_i
               latest_i := k
            if not Consistent(v⃗_k, x_i = v)
               consistent := false
            else k := k+1
      if consistent
         return v
return null              (v doméně x_i neexistuje konzistentní hodnota)
```

v1 v2
1
i

GBJ: příklad



- vyznačené přiřazení zároveň říká, že předchozí hodnoty už jsme vyzkoušeli (a vyřadili)
- x_7 = red vyřadí x_1 (tj. $latest_7 = 1$), x_7 = blue vyřadí $x_3 \Longrightarrow$ celkem $latest_7 = 3$
- vracíme se ke konfliktní proměnné x_3, ta už má ale prázdnou doménu
- $latest_3 = 2$, vracíme se tedy k x_2
  - x_3=red dala $latest_3$ na 1, x_3=blue dala $latest_3$ na 2
- lépe (až k x_1) se ale vrátit neumíme, GBJ se v tomto okamžiku vrací k předchozí proměnné

Konflikty řízený skok zpět

Conflict-directed backjumping CBJ

- Při skoku zpět na proměnnou x_j z listu slepé větve nemusíme nalézt v doméně x_j žádné hodnoty pro přiřazení. $\vec{a}_{j-1}$ se pak nazývá **vnitřní slepá větev**.
- CBJ skáče zpět v listu slepé větve i ve vnitřní slepé větvi
  - udržujeme J_i **množinu skoků zpět** pro každou proměnnou pomocí nesplněných omezení
  - proměnná $x_j (j < i)$ je **bližší** k x_i než $x_k (k < i)$, jestliže $j > k$, naopak x_k je **vzdálenější**
  - omezení R je **vzdálenější než** omezení S, jestliže je nejbližší proměnná v rozsahu R vzdálenější než nejbližší proměnná v rozsahu S (pokud totožné proměnné, rozhodují další proměnné). Pro uspořádání $(x_1, x_2, \ldots)$:
    - rozsah R_1: (x_3, x_5, x_7), rozsah S_1: (x_2, x_6, x_8, x_9) R_1 je vzdálenější než S_1 od x_{10}
      rozsah R_2: (x_3, x_5, x_8, x_9), rozsah S_2: (x_2, x_6, x_8, x_9) R_2 je vzdálenější než S_2 od x_{10}
  - mezi nesplněnými omezeními vybereme to nejvzdálenější (ostatní omezení neumožní nejdelší možný skok zpět)
  - skočíme zpět na nejbližší proměnnou v tomto omezení (je bezpečné změnit proměnnou, která byla nejpozději přiřazená)

CBJ: výběr hodnoty

```
procedure Select-Value-CBJ
while D'_i is not empty
      vyber a smaž libovolný v ∈ D'_i
      consistent := true
      k := 1
      while k < i ∧ consistent
            if Consistent(v_k, x_i = v)
               k := k + 1
            else R := nejvzdálenější nesplněné omezení
                 přidej proměnné v rozsahu R vyjma x_i do J_i
                 consistent := false
      if consistent
         return v
return null              (v doméně x_i neexistuje konzistentní hodnota)
```

Where: D'_i, $v \in D'_i$, $k < i \wedge$ consistent, Consistent$(\vec{v}_k, x_i = v)$, $k := k + 1$, R, x_i, J_i.

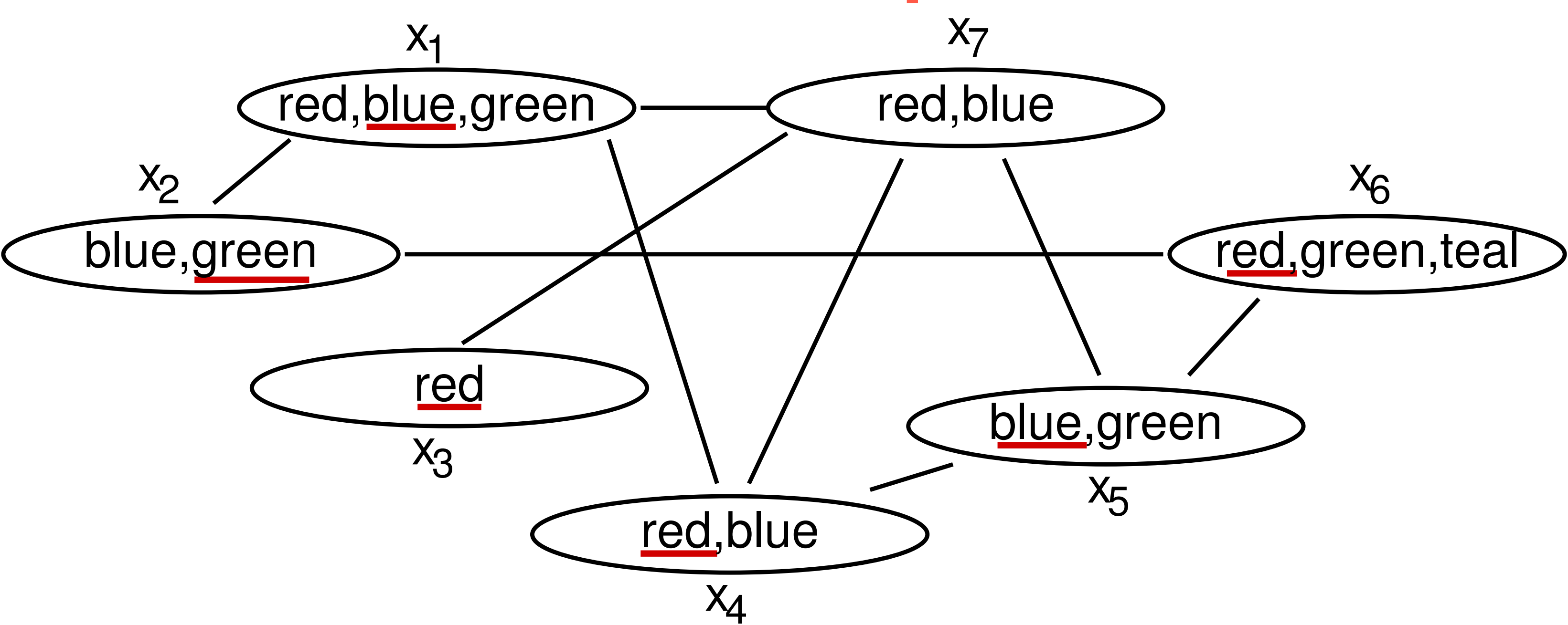
Algoritmus CBJ

```
procedure CBJ((X,D,C))                                      rozdíly od backtrackingu
i := 1                                                      (inicializace čítače proměnných)
D'_i := D_i                                                 (kopírování domény)
J_i := ∅                                                    (inicializace množiny skoků zpět)
while 1 ≤ i ≤ n
      přiřazení x_i := Select-Value-CBJ
      if x_i is null                                        (žádná hodnota nebyla vrácena)
         ip := i
         i := index poslední proměnné v J_i                 (skok zpět)
         J_i := J_i ∪ J_ip − {x_i}                          (spoj množiny skoku zpět)
                 (takto upravená J_i se použije ve vnitřní slepé větvi)
      else i := i + 1                                       (dopředná fáze)
           D'_i := D_i
           J_i := ∅
if i = 0 return ,,nekonzistentní''
else return přiřazené hodnoty {x_1,...,x_n}
end CBJ
```

- k
- j
- i
- ip

(k, j, i → J_{ip})

CBJ: příklad



- před vstupem do Select-Value-CBJ pro proměnnou x_7 je $J_7 = \emptyset$
- $x_3 \neq x_7$ je nejvzdálenější omezení, které vyřadilo $x_7 =$red, tedy $J_7 = \{x_3\}$
- $x_1 \neq x_7$ je nejvzdálenější omezení, které vyřadilo $x_7 =$blue, tedy $J_7 = \{x_1, x_3\}$
- skáčeme zpět na poslední proměnnou v J_7, tedy na x_3
- do J_3, která byla zatím prázdná, přidáme x_1
- doména x_3 je prázdná, v J_3 je jediná proměnná x_1 a na tu se vracíme

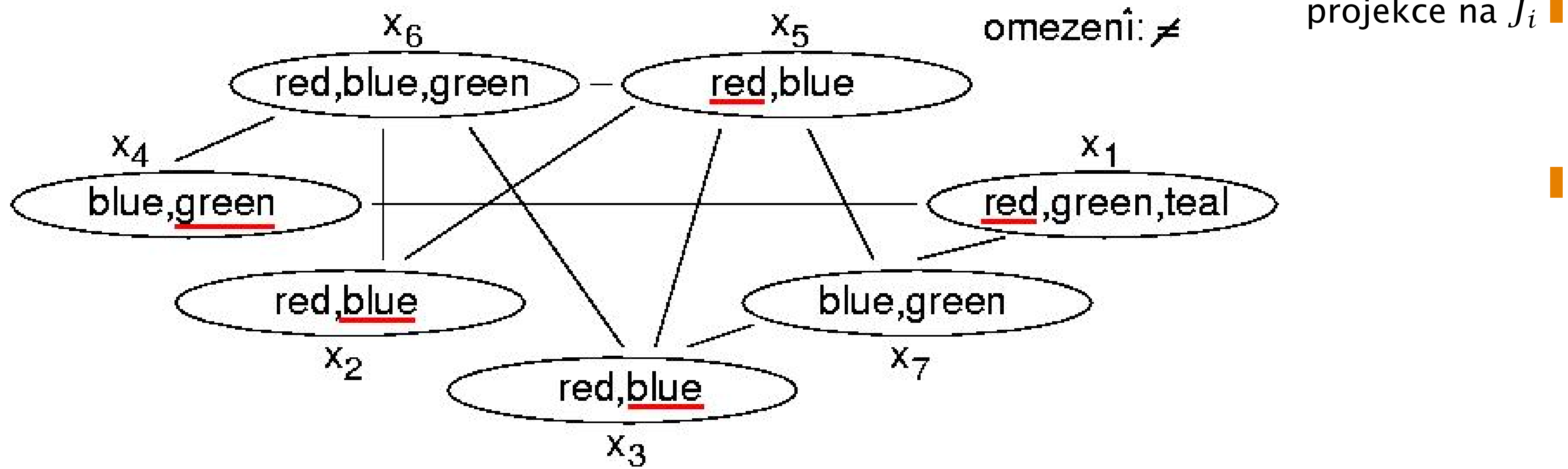
Algoritmy učení

- Množiny skoků zpět jsou chybná přiřazení vypočítaná během prohledávání
- Tato chybná přiřazení se mohou vyskytovat i později v jiných cestách stromu prohledávání a jsou znovu počítána
- Přidáme chybná přiřazení jako nová omezení při detekci slepé větve
  - nemá cenu uchovávat celou slepou větev $\vec{a}_i$ v proměnné x_{i+1}
    na toto přiřazení už později nenarazíme
  - uchováme chybná přiřazení na podmnožině proměnných $\{x_1, \ldots, x_i\}$
- Prořezávání stavového prostoru
  - čím menší chybná přiřazení se podaří uchovat, tím rychlejší bude prohledávání
- Zvětšování množiny omezení
  - cena za prořezávání stavového prostoru nesmí přerůst cenu za zvětšování množiny omezení

Učení skoku zpět (*jumpback learning*)

- Využijeme chybná přiřazení, která jsme se naučili v CBJ
- Algoritmus učení skoku zpět ≡ CBJ + přidávání nových omezení
- Po dosažení listu slepé větve $\vec{a}_{i-1}$ přidáme omezení zakazující $\vec{a}_{i-1}[J_i]$

projekce na J_i



  - po dosažení listu slepé větve v x_6 je $J_6 = \{x_2, x_4, x_5\}$
    $\Leftarrow$ red vyřadila $x_6 \neq x_5$, blue vyřadila $x_6 \neq x_2$, green vyřadila $x_6 \neq x_4$
  - $\vec{a}_5[J_6]$ tedy zakazuje přiřazení $[\langle x_2$,blue$\rangle, \langle x_4$,green$\rangle, \langle x_5$,red$\rangle]$
  - později při procházení stromu lze např. ze znalosti x_2 =blue, x_4 =green odvodit x_5 ≠red

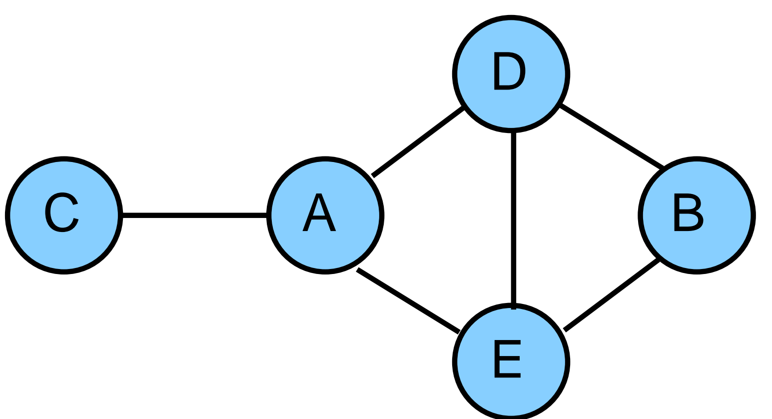
Další rozšíření backtrackingu

Problémy skoku zpět

- Při skoku zpět zapomínáme už udělanou práci
- Příklad:

  Obarvěte graf třemi barvami tak, že mají sousední vrcholy různou barvu

  (uvedené hodnoty barev jsme už vyzkoušeli)



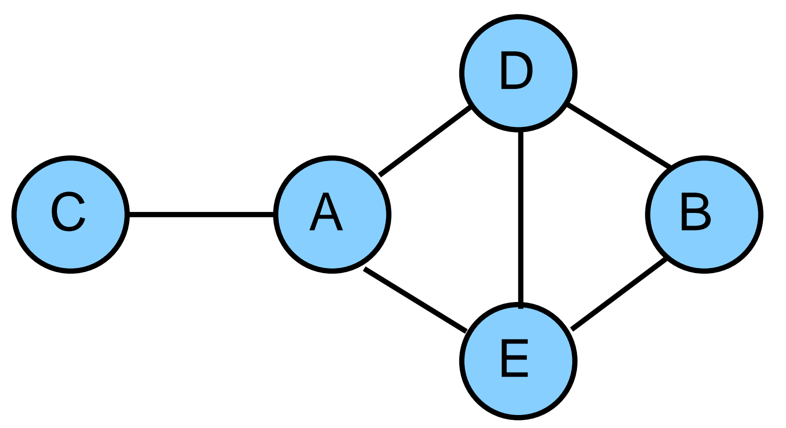
Vrchol	Barva	
A	1	1
B	2	2 1
C	1 2	1 2
D	1 2 3 ⇒ skok na B	1 2
E	1 2 3 ⇒ zpět na D	1 2 3

Při druhém ohodnocení C děláme zbytečnou práci,

stačilo nechat původní hodnotu 2, změnou B se nic neporušilo

Dynamický backtracking: příklad

- Stejný graf (A,B,C,D,E), stejné barvy (1,2,3), ale jiný postup



Skok zpět

\+ pamatování si důvodu konfliktu

\+ přenos důvodu konfliktu

\+ změna pořadí proměnných

= **dynamický backtracking**

Vrchol	1	2	3	Vrchol	1	2	3	Vrchol	1	2	3
A	o			A	o			A	o		
B		o		B		o		C	A	o	
C	A	o		C	A	o		B	o	A	
D	A	B	o	D	A	B	AB	D	A	o	
E	A	B	D	E	A	B		E	A	D	o

vybraná barva: o

důvod konfliktu: AB

skok zpět (na D) + přenos důvodu chyby (AB)

skok zpět (na B) + přenos důvodu chyby (A) + změna pořadí (B,C)

- Vrchol C, resp. celý graf, který na něm případně visel není nutno přebarvovat

Dynamický backtracking: algoritmus

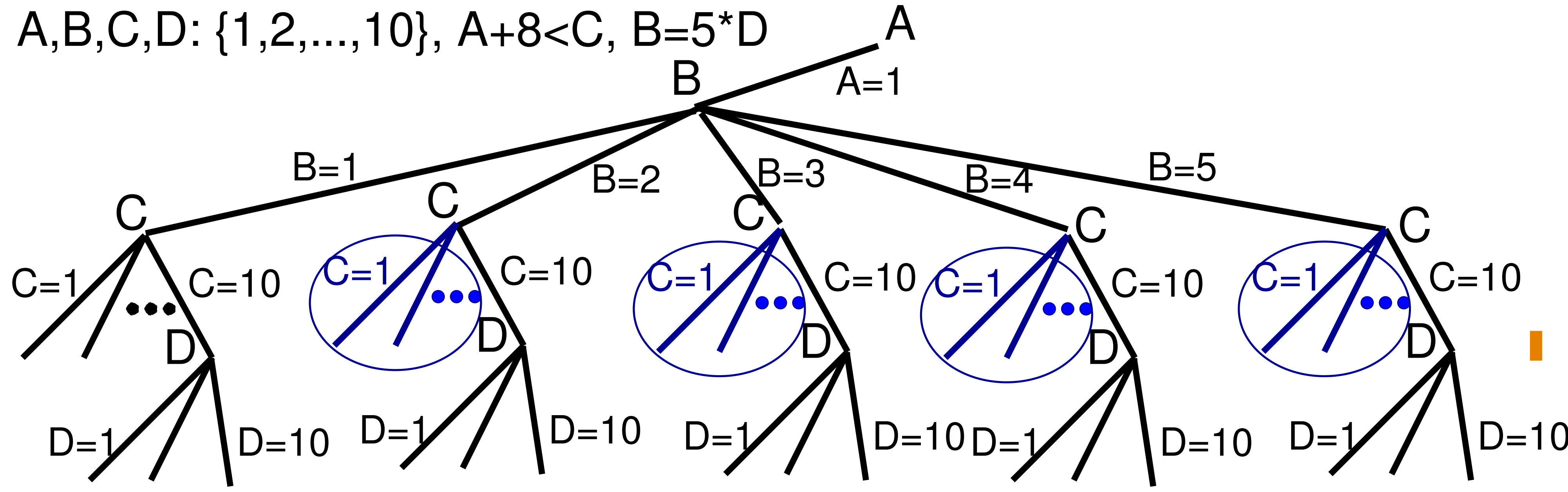
```
procedure DB(Variables, Constraints)
Labelled := ∅; Unlabelled := Variables
while Unlabelled <> ∅ do
    vyber X z Unlabelled
    ValuesX := DX - hodnoty nekonzistentní s Labelled použitím Constraints
    if ValuesX = ∅
    then necht’ E je vysvětlení konfliktu (množina konfliktních proměnných)
        if E = ∅ then failure
        else necht’ Y je nejbližší proměnná v E
            zruš přiřazení Y (z Labelled) s vysvětlením E-Y
            smaž všechna vysvětlení zahrnující Y
    else vyber V z ValuesX
        Unlabelled := Unlabelled - {X}
        Labelled := Labelled ∪ {X/V}
return Labelled
```

Nevýhody algoritmu:

přeuspořádáváním proměnných rušíme efekt heuristik výběru proměnných

Redundance backtrackingu

- **Redundantní práce:** opakování výpočtu, jehož výsledek už máme k dispozici



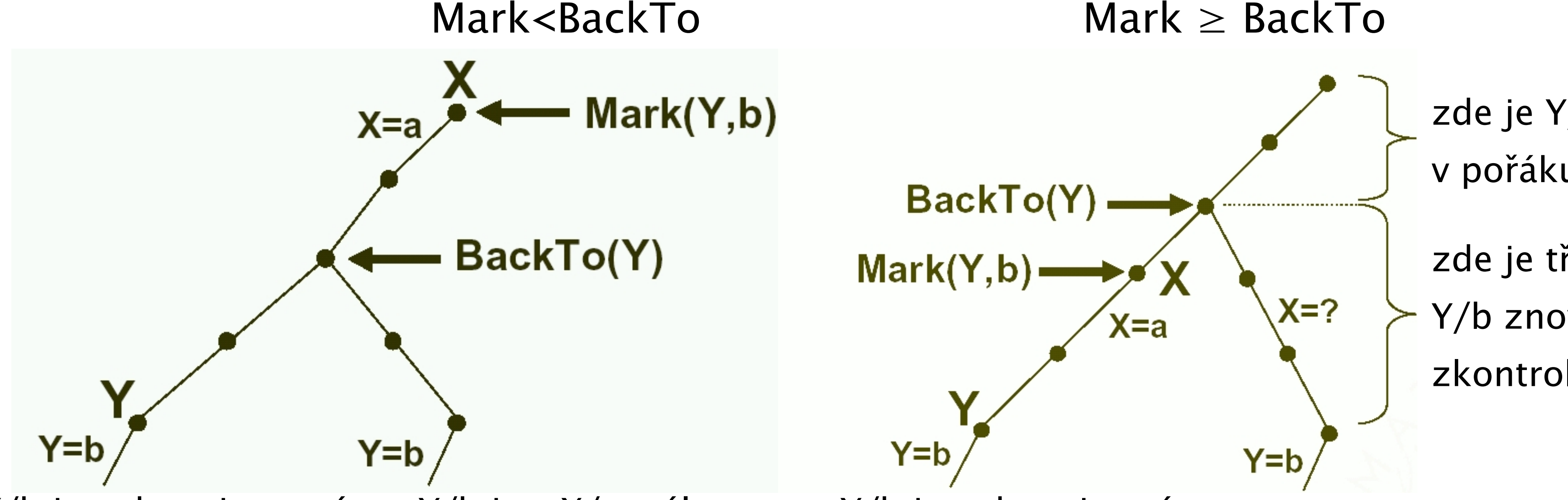
Změna B neovlivňuje hodnotu C ⇒

není potřeba znova procházet podstromy C=1,. . . ,C=9

- **Backmarking**
  - pamatuje si, kde testy na konzistenci neuspěly
  - eliminuje opakování dříve provedených konzistenčních testů

Základy backmarkingu

- **Mark(Y, b)**: u každé hodnoty b z domény Y pamatuje nejvzdálenější konflikt
  - konflikt = proměnná x_p taková, že $\vec{a_p}$ je v konfliktu s $Y = b$
- **BackTo(Y)**: u každé proměnné Y pamatuje nejvzdálenější návrat
  - návrat = proměnná, jejíž hodnota se změnila od poslední instanciace Y



Y/b je nekonzistentní s X/a (konzistentní se vším nad X)

Y/b je s X/a stále nekonzistentní, Y=b tedy nezkoušíme přiřazovat

Y/b je nekonzistení s X/a (ale je konzistentní se vším předtím)

Backmarking: příklad

Problém N královen

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

1. Dámy přiřazujeme po řádcích, tj. pro každou dámu hledáme sloupec
2. Vedle šachovnice píšeme úrovně návratu (BackTo). Na začátku všude 1.
3. Do políčka zapisujeme čísla nejvzdálenějších konfliktních dam (Mark). Na začátku všude 1.
4. Dámu v řádku 6 nelze přiřadit.
5. Vracíme se na 5, opravíme BackTo.
6. Když znova přijdeme na 6, všechny pozice jsou stále špatné (Mark<BackTo)

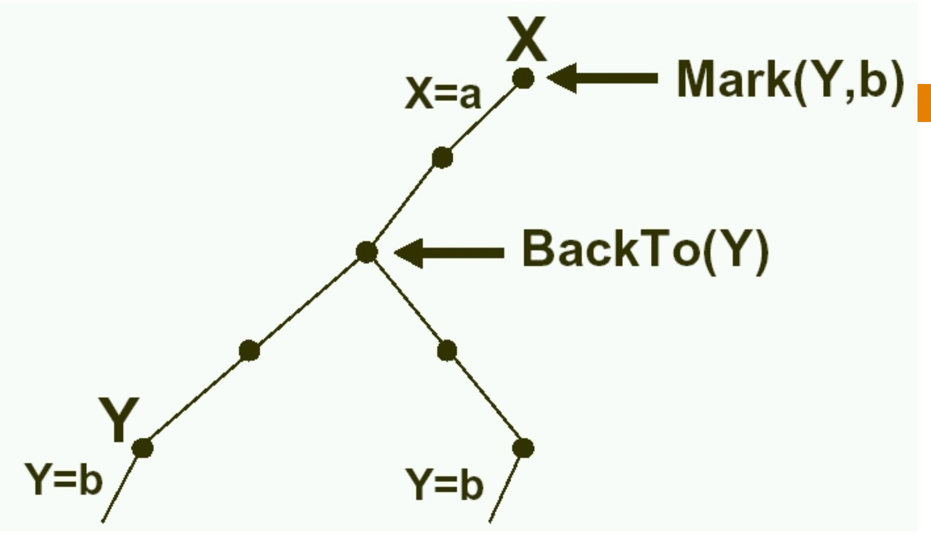
Backmarking lze kombinovat s backjumpingem (zdarma)

Algoritmus backmarkingu

```
procedure Backmarking((X,D,C))                                    rozdíly od backtrackingu
Mark(x_i,v) := 0,  BackTo(x_i) := 0 pro ∀i a ∀v   (inicializace datových struktur)
i := 1                                            (inicializace čítače proměnných)
D'_i := D_i                                       (kopírování domény)
while 1 ≤ i ≤ n
      přiřazení x_i := Select-Value-Backmarking
      if x_i is null                              (žádná hodnota nebyla vrácena)
         for ∀j : i < j ≤ n                       (úprava BackTo pro budoucí proměnné)
             if i < BackTo(x_j) then BackTo(x_j) := i   (i je nový nejvzdálenější návrat)
         BackTo(x_i) := i − 1
         i := i − 1                               (zpětná fáze)
      else i := i + 1                             (dopředná fáze)
           D'_i := D_i
if i = 0 return ,,nekonzistentní''
else return přiřazené hodnoty {x_1,...,x_n}
end Backmarking
```

Uspořádání hodnot pro backmarking

```
procedure Select-Value-Backmarking
smaž z D'_i všechna v taková, že Mark(x_i,v)<BackTo(x_i)
while D'_i is not empty
      vyber a smaž libovolný v ∈ D'_i
      consistent := true
      k := BackTo(x_i)
      while k < i ∧ consistent
            if not Consistent(a⃗_k, x_i = v)
               Mark(x_i,v) := k
               consistent := false
            else k := k + 1
      if consistent
         Mark(x_i,v) := i
         return v
return null
```



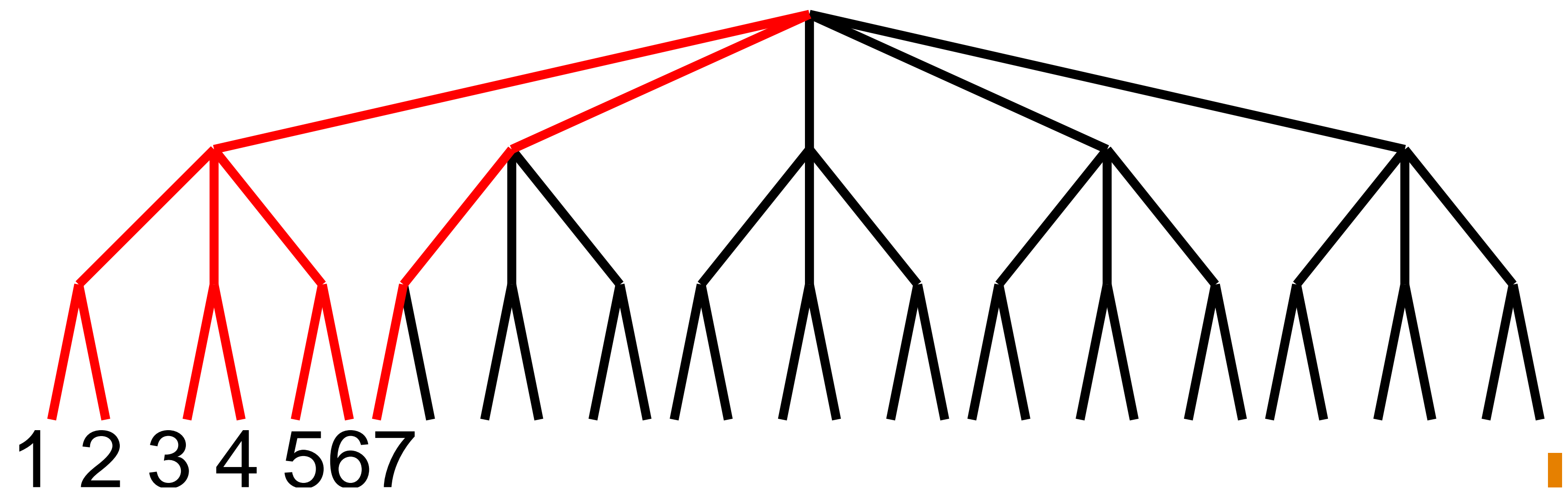
Neúplná stromová prohledávání

Neúplné prohledávání do hloubky

- ***Depth first search (DFS)***: odpovídá algoritmu **backtrackingu**
- Reálné problémy mají často tak velké prostory možných ohodnocení, že není možné je celé prohledat.
- Je možné prohledat jen omezený prostor

⇒ Neúplná stromová prohledávání

- **Neúplné prohledávání do hloubky**



Neúplná stromová prohledávání

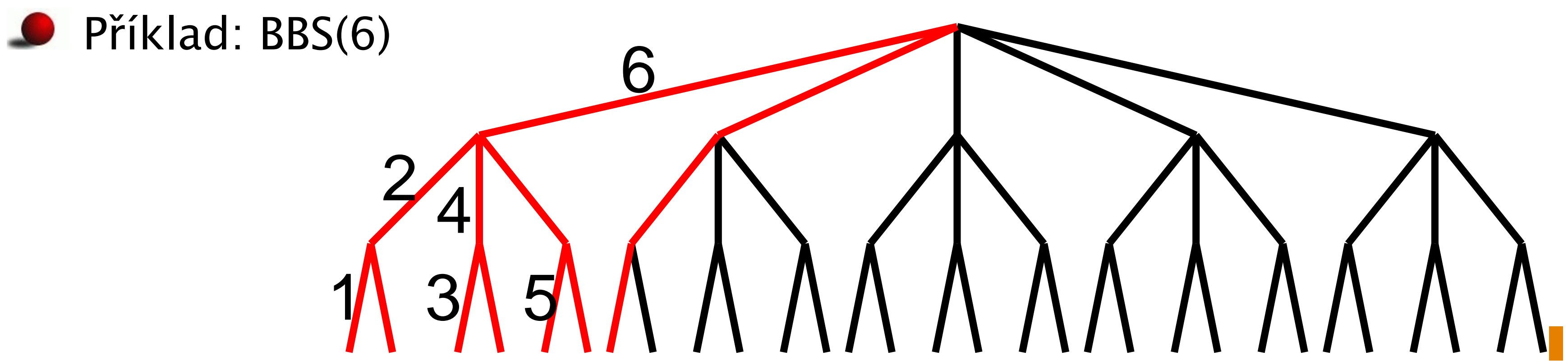
- Neprohledáváme celý stavový prostor
- Nemáme záruku, že řešení neexistuje, i když ho algoritmus nenalezne
  - ztráta úplnosti
  - u některých algoritmů lze *obecně* zaručit úplnost,
    i když s vyšší složitostí než měl původní algoritmus
- V řadě případů najdeme řešení rychleji
- Neúplné algoritmy často odvozeny od algoritmu úplného (DFS)
  - **přerušení běhu algoritmu (*cutoff*)**
    - po vyčerpání přiděleného **prostředku** (čas, počet návratů, . . .) algoritmus přerušíme
    - prostředek může být **globální** (pro celý strom) i **lokální** (pro daný podstrom nebo uzel)
  - **opakovaní běhu algoritmu (*restart*)**
    - běh předešlého neúplného prohledávání opakujeme s jiným nastavením parametrů
    - při opakování běhu lze využívat algoritmy učení

Randomizovaný backtracking

- **Časově omezený backtracking** (přerušení)
  - běh (úplného) algoritmu ukončíme po zadaném časovém intervalu (prostředek=čas)
  - časový interval lze pro další běhy zvětšit
    - $\Rightarrow$ při dostatečném počtu kroků máme úplný algoritmus
- **Náhodný výběr hodnot a proměnných** (opakování)
  - pokud máme možnost volby při výběru hodnot nebo proměnných (vzhledem k dané heuristice uspořádání hodnot a proměnných) náhodně zvolíme některou z nich
- **Randomizovaný backtracking s učením**
  - chybná přiřazení předchozích běhů uchováme a využíváme
  - takto lze také dosáhnout úplnosti, protože chybných přiřazení je konečně mnoho

Omezení počtu návratů

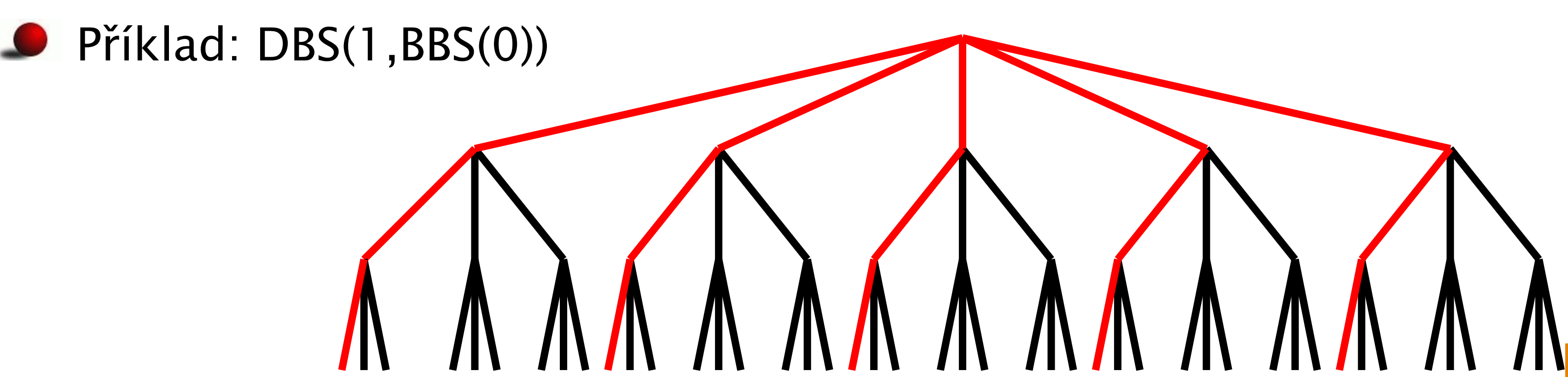
- ***Bounded-backtrack search*** **(BBS)**
- Omezený počet návratů **(přerušení)**
  - návrat do bodů volby, kde už nelze vybrat novou hodnotu nezapočítáváme
  - „omezený počet navštívených listů"
- Pro úplnost: při neúspěchu zvětšíme počet návratů o jedna **(opakování)**
- Příklad: BBS(6)



- Implementace
  - počítáme počet návratů (neúspěchů)
  - při překročení meze se prohledávání ukončí

Omezení hloubky

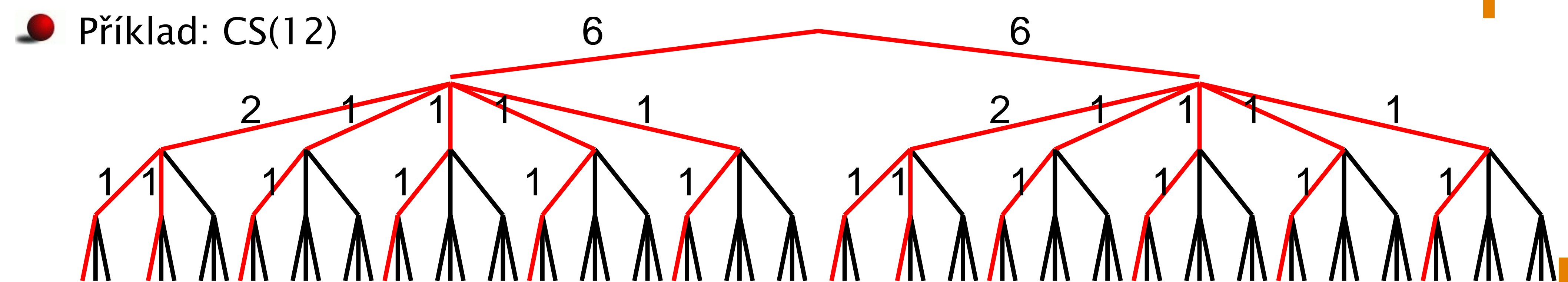
- ***Depth-bounded search*** **(DBS)**
- Omezíme hloubku prohledávaného stromu **(přerušení)**
  - do dané hloubky stromu se zkouší všechny alternativy
  - ve zbytku stromu se může použít jiná neúplná metoda
- Pro úplnost: při neúspěchu zvětšíme hloubku prohledávání o jedna **(opakování)**
- Příklad: DBS(1,BBS(0))



- Implementace
  - udržujeme pořadové číslo přiřazované proměnné
  - je-li pořadové číslo větší než daná mez, zkouší se pouze jedna alternativa – BBS(0)

Prohledávání s kreditem

- *Credit search* (CS)
- Omezený kredit (počet návratů) pro prohledávání **(přerušení)**
  - kredit se rovnoměrně dělí mezi alternativní větve prohledávání
  - jednotkový kredit zakazuje možnost volby (hodnoty), tj. pokračujeme pouze bez návratů
- Pro úplnost: při neúspěchu navýšíme kredit o jedna **(opakování)**
- Příklad: CS(12)

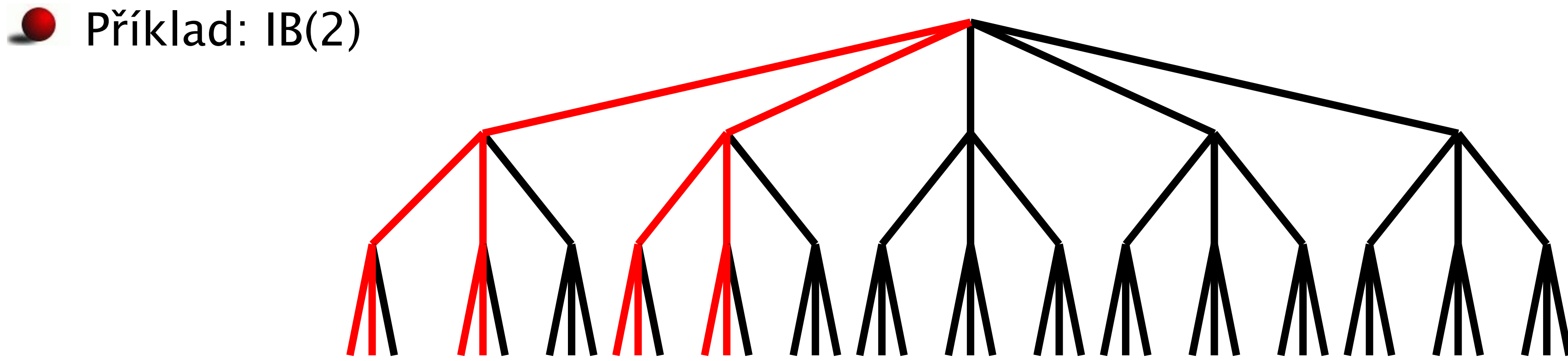


- Implementace
  - v každém uzlu se nejednotkový kredit (rovnoměrně) rozdělí mezi alternativní podstromy
  - při jednotkovém kreditu se neberou alternativy (tj. při neúspěchu končíme)

Iterativní rozšiřování

- *Iterative broadening* IB
- Omezený maximální počet voleb (hodnot)

  při každém výběru proměnné **(přerušení)**
  - v každém bodě volby větvení omezeno konstantou
  - při překročení max. počtu voleb pokračujeme předchozím bodem volby
- Úplnost: při neúspěchu zvýšíme povolený počet voleb o jedna **(opakování)**
- Příklad: IB(2)



- Implementace
  - po výběru proměnné umožníme pouze výběr určeného počtu jejích hodnot

Stromová prohledávání a heuristiky

- Při řešení reálných problémů často existuje nápověda odvozená ze zkušeností s „ručním" řešením problému
- Heuristiky – radí, jak pokračovat v prohledávání
  - doporučují hodnotu pro přiřazení
  - často vedou přímo k řešení
- Co dělat, když heuristika neuspěje?
  - DFS se stará hlavně o konec prohledávání (spodní část stromu)
  - DFS tedy spíše opravuje poslední použité heuristiky než první
  - DFS předpokládá, že dříve použité heuristiky ho navedly dobře
- Pozorování:
  - počet porušení heuristiky na úspěšné cestě je malý
  - heuristiky jsou méně spolehlivé na začátku prohledávání než na jeho konci (na konci máme více informací a méně možností)

Zotavení se z chyb heuristiky

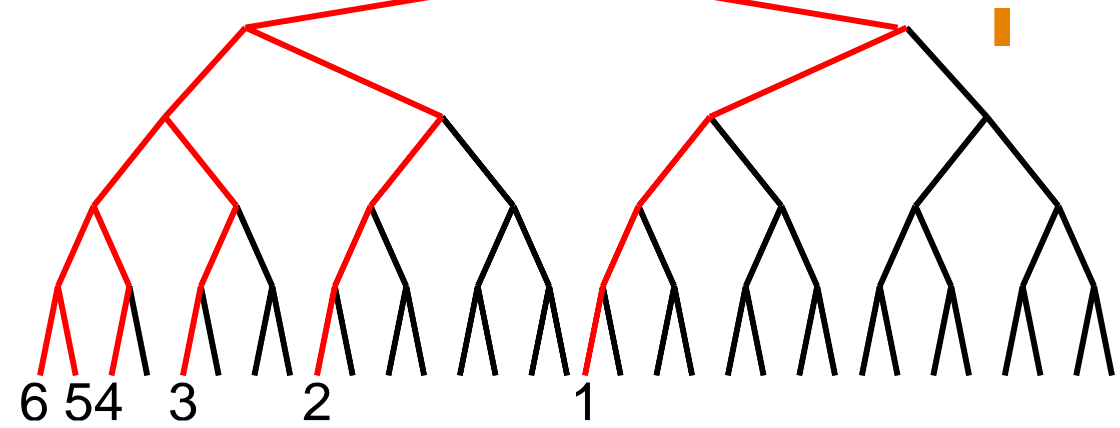
- Heuristika doporučuje hodnotu pro přiřazení
- **Diskrepance** = porušení heuristiky
  - použita jiná hodnota, než doporučila heuristika
- Pozorování: málo chyb heuristiky na cestě k řešení

  $\Rightarrow$ cesty s méně diskrepancemi jsou prozkoumány dříve
- Pozorování: chyby heuristiky hlavně na začátku cesty

  $\Rightarrow$ cesty s diskrepancemi na začátku jsou prozkoumány dříve

Omezené diskrepance

- ***Limited discrepancy search*** **(LDS)**
- Omezený maximální počet diskrepancí na cestě **(přerušení)**
  - cesty s diskrepancemi na začátku jsou prozkoumány dříve
- Při neúspěchu navýšíme počet povolených diskrepancí o jedna **(opakování)**
  - tj. nejprve jdeme tak, jak doporučuje heuristika; potom jdeme po cestách s maximálně jednou diskrepancí; pak maximálně se dvěma diskrepancemi, atd.
- Příklad: LDS-PROBE(1), heuristika doporučuje vydat se levou větví



- **Diskrepance pro nebinární domény**
  - nedoporučené hodnoty se berou jako jedna diskrepance (zde)
  - výběr každé další hodnoty proměnné je jedna diskrepance (tj. třetí hodnota = dvě diskrepance, čtvrtá hodnota = tři diskrepance, . . .)

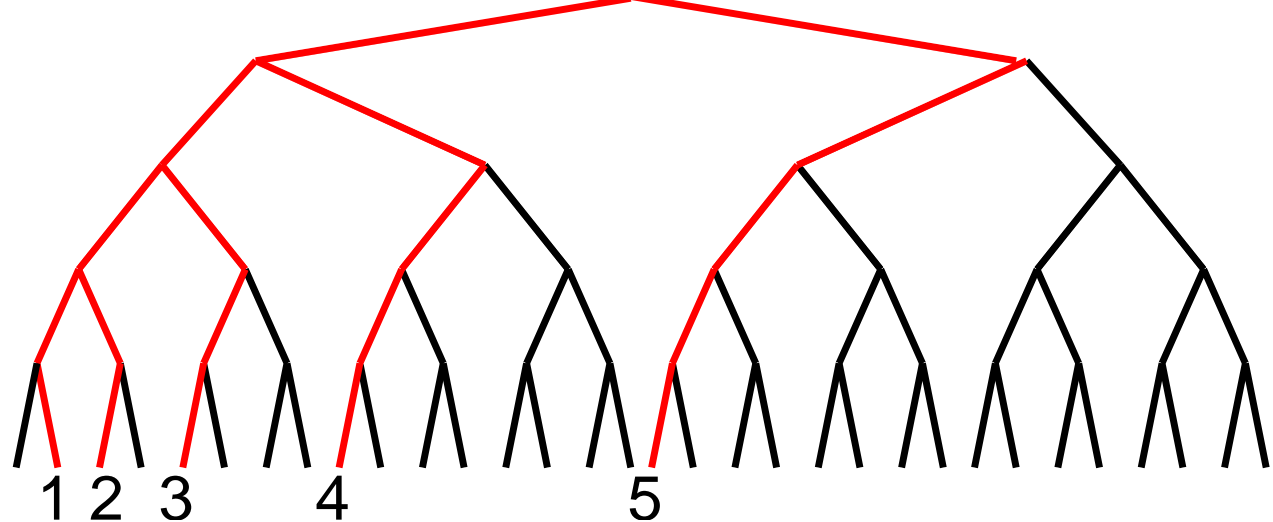
Algoritmus LDS

```
procedure LDS(Variables,Constraints)
  for D=0 to |Variables| do                 % D určuje max. počet povolených diskrepancí
     R := LDS-PROBE(Variables,{},Constraints,D)
     if R ≠ fail then return R
return fail

procedure LDS-PROBE(Unlabelled,Labelled,Constraints,D)
  if Unlabelled = {} then return Labelled
  select X ∈ Unlabelled
  Values_X := D_X - {values inconsistent with Labelled using Constraints}
  if Values_X = {} then return fail
  else select HV ∈ Values_X using heuristic
      if D>0 then
        for ∀V ∈ Values_X-{HV} do
           R := LDS-PROBE(Unlabelled-{X}, Labelled ∪ {X/V}, Constraints, D-1)
           if R ≠ fail then return R
      return LDS-PROBE(Unlabelled-{X}, Labelled ∪ {X/HV}, Constraints, D)
end LDS-PROBE
```

Omezené diskrepance – zlepšení

- LDS v každé další iteraci prochází i větve z předchozí iterace, tj. opakuje již provedený výpočet a navíc se v rámci iterace musí vracet do již prošlých částí
- *Improved limited discrepancy search (ILDS)*
  - daný počet diskrepancí na cestě (přerušení)
    - cesty s diskrepancemi na konci prozkoumány dříve
- Při neúspěchu navýšíme počet diskrepancí o jedna (opakování)
- Příklad: ILDS-PROBE(1), heuristika doporučuje vydat se levou větví



Algoritmus ILDS

procedure ILDS(Variables,Constraints) % analogie LDS(Variables,Constraints)

procedure ILDS-PROBE(Unlabelled,Labelled,Constraints,D) Rozdíly od LDS

if Unlabelled = {} then return Labelled

select X $\in$ Unlabelled

Values$_X$:= D$_X$ - {values inconsistent with Labelled using Constraints}

if Values$_X$ = {} then return fail

else select HV $\in$ Values$_X$ using heuristic

if D<|Unlabelled| then

R := ILDS-PROBE(Unlabelled-{X}, Labelled $\cup$ {X/HV}, Constraints, D)

if R $\neq$ fail then return R

if D>0 then

for $\forall$V $\in$ Values$_X$-{HV} do

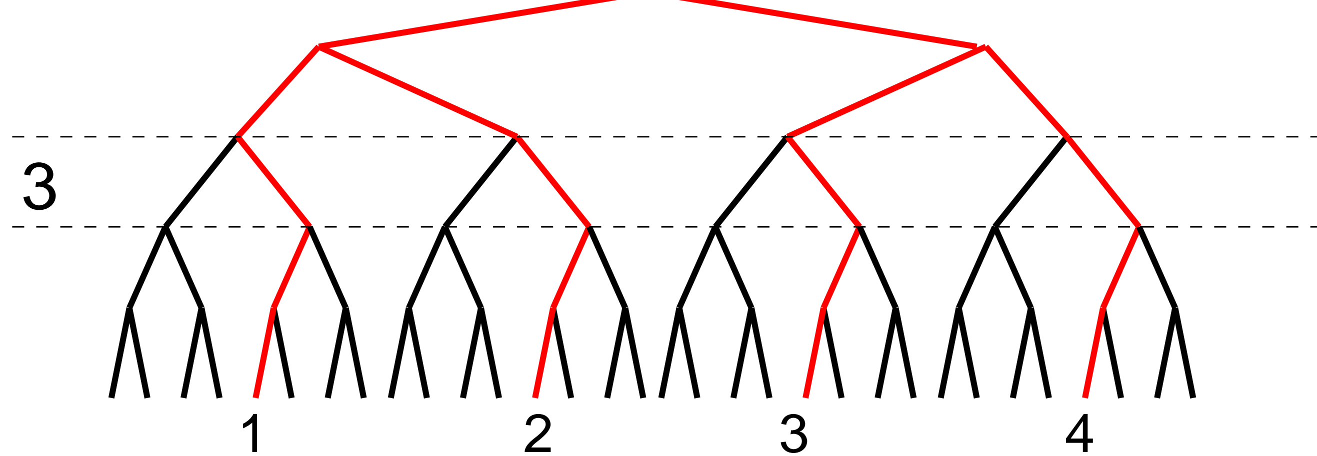
R := ILDS-PROBE(Unlabelled-{X}, Labelled $\cup$ {X/V}, Constraints, D-1)

if R $\neq$ fail then return R

end ILDS-PROBE

Hloubkou omezené diskrepance

- ILDS bere cesty s diskrepancemi na konci dříve
- *Depth-bounded discrepancy search (DDS)*
- Diskrepance povoleny pouze do dané hloubky **(přerušení)**
  - v této hloubce je vždy diskrepance, tj. zabrání se procházení větví z předchozí iterace
  - hloubka zároveň omezuje maximální počet diskrepancí
  - cesty s diskrepancemi na začátku prozkoumány dříve
- Při neúspěchu navýšíme hloubku o jedna **(opakování)**
- Příklad: DDS(3), heuristika doporučuje vydat se levou větví



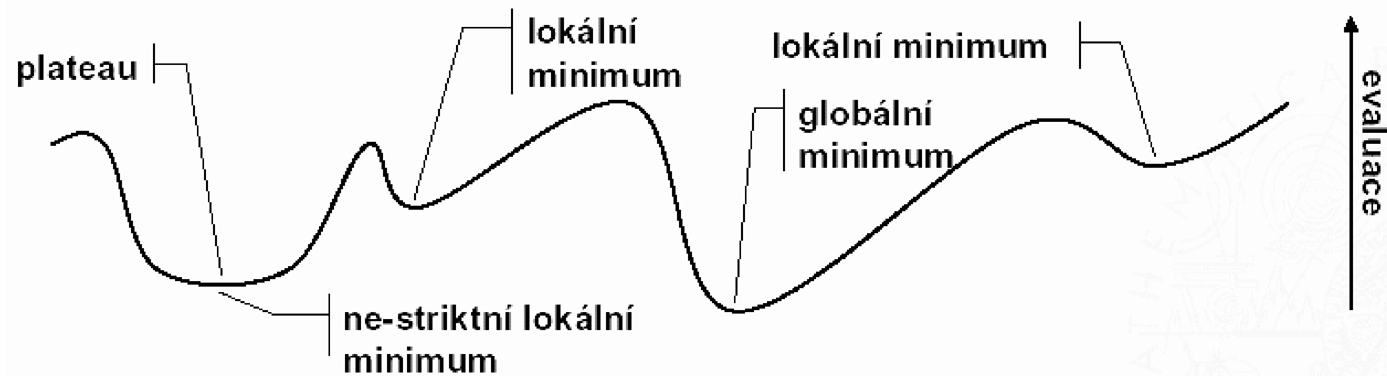
Lokální prohledávání

Lokální prohledávání (LS)

- Princip
  - pracuje s **úplnými nekonzistentními přiřazeními proměnných**
  - snaží se **lokálními opravami** snížit počet konfliktů
- Příklad: umístění n dam na šachovnici
  - iniciální přiřazení umístí každou královnu do každého sloupce a řádku bez ohledu na diagonální omezení
  - přesunujeme královnu v jejím sloupci do jiného řádku tak, abychom odstranili co nejvíce konfliktů
  - LS vyřeší řádově větší problémy než algoritmy založené na prohledávání do hloubky
- Přibližná metoda prohledávání
  - neúplná metoda
  - nezaručuje nalezení (vyloučení existence) řešení i když existuje (neexistuje)
  - malé paměťové nároky

Terminologie lokálního prohledávání

- **Stav θ:** ohodnocení všech proměnných
- **Evaluace E:** hodnota objektivní funkce (počet nesplněných podmínek=**počet konfliktů**)
- **Globální optimum:** stav s nejlepší evaluací
- **Lokální změna:** změna hodnoty (jedné) proměnné
- **Okolí o:** množina stavů lišící se od daného stavu o jednu lokální změnu
- **Lokální optimum:** stav, v jehož okolí jsou stavy s horší evaluací; není globálním optimum
- **Striktní lokální optimum:** stav, v jehož okolí jsou pouze stavy s horší evaluací;
  není globálním optimum
- **Ne-striktní lokální optimum:** stav, v jehož okolí exisují stavy se stejnou evaluací;
  není globálním optimum
- **Plateau:** množina stavů se stejnou evaluací



Algoritmus lokálního prohledávání

- Algoritmy lokálního prohledávání mají společnou kostru

```
procedure LS(MaxPokusu,MaxZmen)                    parametry algoritmu
θ := náhodné ohodnocení proměnných
for i := 1 to MaxPokusu while GPodminka do
    for j := 1 to MaxZmen while LPodminka do
        if E(θ)=0 then
            return θ
        vyber δ ∈ o(θ)
        if akceptovatelne(δ) then
            θ := δ
    θ := novyStav(θ)
return nejlepší θ
```

- Jak stanovit MaxPokusu,MaxZmen?
  - pokračovat dokud existuje přiřazení s lepší evaluací
  - restart (MaxPokusu>1) v některých případech diskutabilní

Metoda největšího stoupání (*hill climbing*) HC

- Začíná v náhodně vybraném stavu
- Hledá vždy nejlepší stav v okolí
- Okolí = hodnota libovolné proměnné je změněna, velikost okolí $n \times (d-1)$
- Útěk ze striktního lokálního minima pomocí restartu

```
procedure HC(MaxZmen)
restart: θ := náhodné ohodnocení proměnných
         for j := 1 to MaxZmen do
             if E(θ) = 0 then
                return θ
             if θ je striktní lokální optimum then
                goto restart
             else θ := stav z o(θ) s nejlepší evaluací
         goto restart
end HC
```

Metoda minimalizace konfliktů (MC)

- Okolí u HC je poměrně velké: $n \times (d-1)$
  - ale pouze změna konfliktní proměnné může přinést zlepšení
  - konfliktní proměnná = vystupuje v některých nesplněných podmínkách
- MC mění pouze konfliktní proměnné
  - okolí = hodnota zvolené proměnné je změněna, velikost okolí $d-1$

```
procedure MC(MaxZmen)
θ := náhodné ohodnocení proměnných
PocetZmen := 0
while E(θ) > 0 ∧ PocetZmen<MaxZmen do
      vyber náhodně konfliktní proměnnou V
      vyber hodnotu a, která minimalizuje počet konfliktů pro V
      if a≠současná hodnota V then
         přiřad’ a do V
         PocetZmen := PocetZmen+1
return θ
```

neřešíme únik z lokálního optima

Náhodná procházka (*Random Walk*) RW

- Jak **uniknout z lokálního optima bez restartu**?
  - přidáním „šumu” do algoritmu
- Pokud se dostaneme do lokálního optima
  - náhodně zvolíme stav z okolí a pokračujeme jím
  - tato metoda samostatně k řešení nepovede
  - potřebuje další směrování v prohledávacím prostoru
    - lze kombinovat s HC i MC
- **RW je kombinováno s heuristikou pomocí pravděpodobnostního rozložení**
  - pravděpodobnost náhodného kroku je p
  - pravděpodobnost použití směrové heuristiky je $1 - p$
  - náhodná procházka tedy nahrazuje restart
    - únik z lokálního minima prostřednictvím náhodného výběru

Minimalizace konfliktů s náhodnou procházkou

Rozdíly od MC

```
procedure MCRW(MaxZmen,p)
θ := náhodné ohodnocení proměnných
PocetZmen := 0
while E(θ) > 0 ∧ PocetZmen<MaxZmen do
      vyber náhodně konfliktní proměnnou V
      generuj náhodné číslo Pravdepodobnost ∈ [0,1]
      if Pravdepodobnost ≥ (1 − p)
         vyber náhodně hodnotu a pro V
      else vyber hodnotu a, která minimalizuje počet konfliktů pro V
      if a≠současná hodnota V then
         přiřaď a do V
         PocetZmen := PocetZmen+1
return θ
```

$0.02 \le p \le 0.1$

Největší stoupání s náhodnou procházkou

Rozdíly od MCRW

```
procedure HCRW(MaxZmen,p)
θ := náhodné ohodnocení proměnných
PocetZmen := 0
while E(θ) > 0 ∧ PocetZmen<MaxZmen do
      generuj náhodné číslo Pravdepodobnost ∈ [0,1]
      if Pravdepodobnost ≥ (1 − p)
         vyber náhodně konfliktní proměnnou V
         vyber náhodně hodnotu a pro V
      else vyber ⟨V,a⟩ s nejlepší evaluací
      if a≠současná hodnota V then
         přiřad' a do V
         PocetZmen := PocetZmen+1
return θ
```

Tabu seznam

- Setrvání v lokálním optimu je speciálním případem cyklu
- Jak se obecně zbavit cyklů?
  - stačí si pamatovat předchozí stavy a zakázat opakování stavu
    - paměťově příliš náročné (mnoho stavů)
  - můžeme si zapamatovat pouze několik posledních stavů (zabrání krátkým cyklům)
- **Tabu seznam** = seznam **tabu (zakázaných) stavů**
  - stav lze popsat význačným atributem (není nutné uchovávat celý stav)
    - (proměnná,hodnota): zachycuje změnu stavu (uložíme původní hodnoty)
  - tabu seznam má **fixní délku (*tabu tenure*)**
    - „staré" stavy ze seznamu vypadávají s přicházejícími novými stavy
- **Aspirační kritérium** = odtabuizování stavu
  - do stavu lze přejít, i když je v tabu seznamu (např. krok vede k celkově lepšímu stavu)
- Tabu seznam je používán samostatně i v kombinaci s jinými metodami LS

Algoritmus prohledávání s tabu seznamem

- Tabu seznam zabraňuje krátkému cyklení
- Povoleny jsou pouze tahy mimo tabu seznam a tahy splňující aspirační kritérium
- **Tabu seznam (TS) v kombinaci s metodou stoupání (HC)**:

```
procedure TSHC(MaxZmen)
θ := náhodné ohodnocení proměnných
PocetZmen := 0
while E(θ) > 0 ∧ PocetZmen<MaxZmen do
      vyber ⟨V,a⟩ s nejlepší evaluací tak, že
            není v tabu seznamu a nebo splňuje aspirační kriterium
      přidej ⟨V,c⟩ do tabu seznamu, kde c je současná hodnota V
      smaž nejstarší položku v tabu seznamu
      přiřaď a do V
      PocetZmen := PocetZmen+1
return θ
```

Výběr souseda: přehled

- **Metoda stoupání (HC)**
  - soused s nejlepší evaluací vybrán
- **Tabu prohledávání (TS+HC)**
  - soused s nejlepší evaluací vybrán (metoda stoupání)
  - sousedé z tabu seznamu nemohou být vybráni
- **Minimální konflikt (MC)**
  - soused je omezen na náhodně vybranou konfliktní proměnnou
  - výběr její hodnoty s nejlepší evaluací
- **Náhodná procházka (RW)**
  - soused vybrán náhodně
- **Min. konflikt (metoda stoupání) s náhodnou procházkou MC+RW (HC+RW)**
  - s malou pravděpodobností: náhodný výběr souseda
  - jinak: minimální konflikt (metoda stoupání)

Simulované žíhání (*simulated annealing*) SA

- Myšlenka: **simulace procesu ochlazování kovů**
  - na začátku při vyšší teplotě atomy více kmitají a pravděpodobnost změny krystalické mřížky je vyšší
  - postupným ochlazováním se atomy usazují co „nejlepší polohy" s nejmenší energií a pravděpodobnost změny je menší

  ⇒ na začátku je tedy pravděpodobnost toho, že akceptujeme zhoršování řešení, vyšší
- **Akceptování nového stavu**
  - vždy při zlepšení
  - při zhoršení pouze za dané pravděpodobnosti, která klesá se snížením teploty
- **Cykly algoritmu**
  - vnější: simulace procesu ochlazování snižováním **teploty** T

    čím nižší bude teplota, tím nižší bude pravděpodobnost akceptování zhoršení
  - vnitřní: počítáme, kolikrát jsme neakceptovali zhoršení (dán limit `MaxIter`)

Metropolisovo kritérium

Rozdíl mezi kvalitou nového (δ) a existujícího (θ) řešení

- $\Delta E = E(\delta) - E(\theta)$
- E (chybovost) musí být minimalizováno

Metropolisovo kritérium

- lepší (někdy případně i stejně kvalitní) řešení akceptováno: $\Delta E > 0$
- **horší řešení ($\Delta E > 0$) akceptováno pokud**

$$U < e^{-\Delta E/T}$$

- U náhodné číslo z intervalu $(0, 1)$
- pomůcka: porovnej $e^{-10/100}$ vs. $e^{-100/100}$ a $e^{-10/100}$ vs. $e^{-10/1}$

Algoritmus simulovaného žíhání

```
procedure SA( TInit, TEnd, MaxIter )
θ := náhodné ohodnocení proměnných
α := θ                                   (dosud nejlepší nalezené přiřazení)
for T := TInit to TEnd
    PocetChyb := 0
    while PocetChyb < MaxIter
        vyber lokální změnu z θ do δ
        if E(δ) < E(θ) then                          (akceptuj přiřazení)
           θ := δ
           if E(θ) < E(α) then
              α := θ                                 (nové optimum)
        else generuj náhodné číslo p ∈ [0,1)
             if p < e^((E(θ) − E(δ))/T) then         (akceptuj přiřazení)
                θ := δ
             else PocetChyb := PocetChyb + 1         (neakceptuj přiřazení)
    T := max(round(0.8×T),TEnd)                      (sniž teplotu)
end SA
```

Lokální prohledávání pro SAT problém

- **SAT problém**: splnitelnost logické formule v konjunktivní normální formě
  - **CNF** = konjunkce klauzulí
  - **klauzule** = disjunkce literálů (podmínka)
  - **literál** = atom nebo negace atomu
  - příklad: $(A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee \neg A)$

  $\Longrightarrow$ CSP nad disjunkcemi boolean proměnných
- SAT je NP-úplný
- LS řeší poměrně velké formule
  - formulace LS je velice přirozená a jeho použití je velice populární
  - lokální změna je **překlápěním (*flipping*)** hodnoty proměnné
- Algoritmus **GSAT** (*greedy SAT*)
  - postupné překlápění proměnných
  - evaluace udává, jaký je (vážený) počet nesplněných klauzulí

Algoritmus GSAT

```
procedure GSAT(A,MaxPokusu,MaxZmen)    (A je CNF formule)
for i := 1 to MaxPokusu do
    θ := náhodné ohodnocení proměnných
    for j := 1 to MaxZmen do
        if A je splnitelná pomocí θ then
           return θ
        V := proměnná, jejíž změna hodnoty
             nejvíce sníží počet nesplněných klauzulí
        θ := přiřazení, které se liší od θ změnou hodnoty V
return nejlepší θ
```

- Příklad: $\{\neg C, \quad \neg A \vee \neg B \vee C, \quad \neg A \vee D \vee E, \quad \neg B \vee \neg C\}$
  - iniciální přiřazení (všechny proměnné mají hodnotu 1) nesplňuje první a poslední klauzuli
  - změna A, D, E nemá vliv na počet nesplněných klauzulí
  - změna C na 0 splní první i poslední klauzuli ale nesplní $\neg A \vee \neg B \vee C$ (evaluace=1)
  - změna B má evaluaci 1 také, pokud ale změníme C a pak B, pak dostáváme evaluaci 0

Heuristiky pro GSAT

- GSAT lze kombinovat s různými heuristikami, které zvyšují jeho efektivitu
  - obzvláště při řešení strukturovaných problémů
- Použití **náhodné procházky spolu s minimalizací konfliktů**
- **Vážení klauzulí**
  - některé klauzule zůstávají po řadu iterací nesplněné, klauzule tedy mají různou důležitost
  - splnění „těžké" klauzule lze preferovat přidáním váhy
  - váhu může systém odvodit
    - na začátku mají všechny klauzule stejnou váhu
    - po každém pokusu zvyšujeme váhu u nesplněných klauzulí
- **Průměrování řešení**
  - standardně každý pokus začíná z náhodného řešení
  - společné části předchozích řešení lze zachovat
    - restartovací stav se vypočte ze dvou posledních výsledků bitovým srovnáním
      stejné bity zachovány, ostatní nastaveny náhodně

Hybridní prohledávání

- Příklady kombinace lokálního prohledávání
  - prohledává úplná nekonzistentní přiřazení

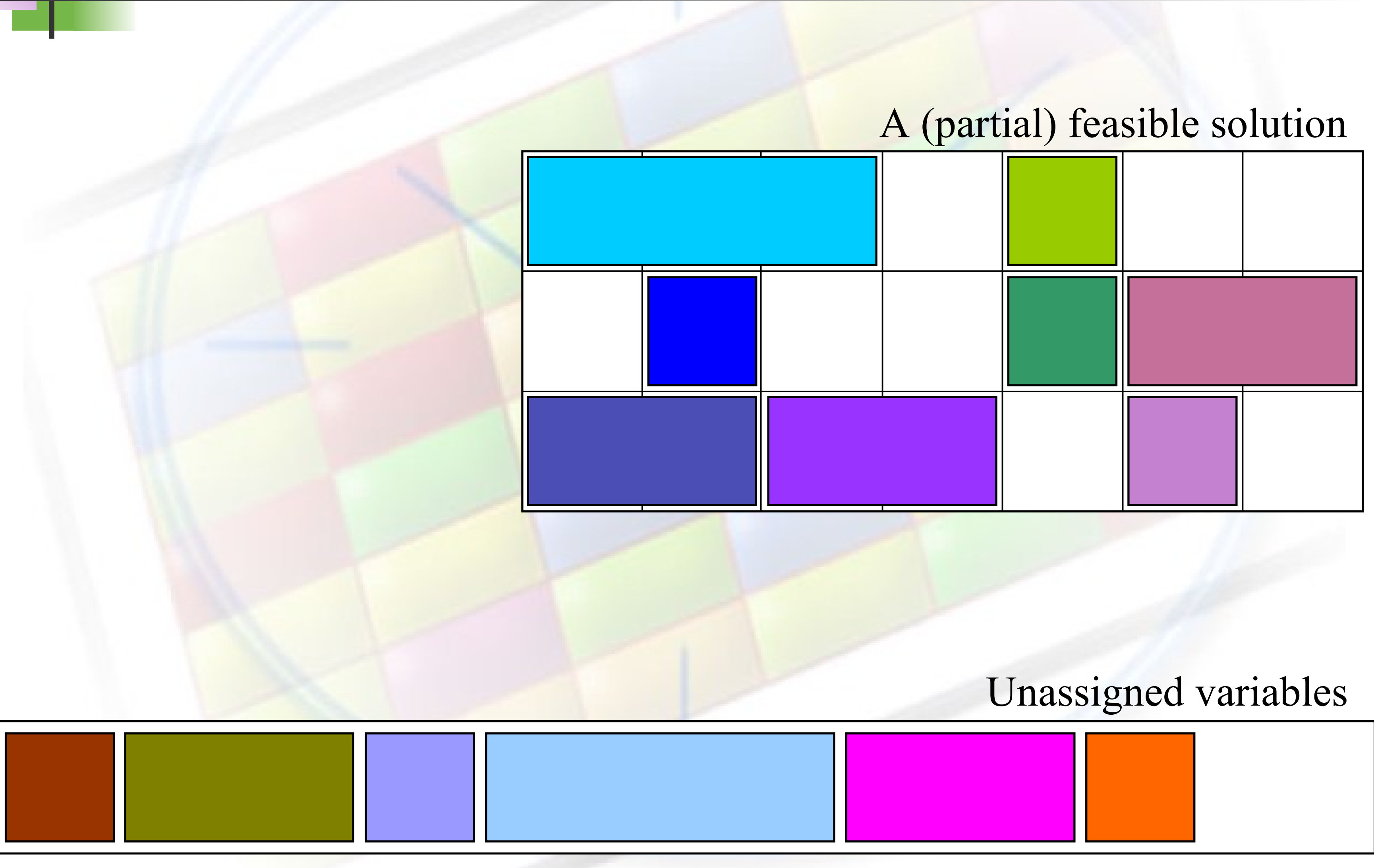
  a stromového prohledávání
  - rozšiřuje částečné konzistentní přiřazení

1. Lokální prohledávání **před nebo po** stromovém prohledávání např:
   - před: lokální prohledávání nám poskytne heuristiku na uspořádání hodnot
   - po: lokálním prohledáváním se snažíme lokálně vylepšit spočítané řešení (optimalizace)
2. **Stromové prohledávání** je doplněno lokálním prohledávání
   - v listech prohledávacího stromu i v jeho uzlech
     - např. lokální prohledávání pro výběr hodnoty nebo vylepšení spočítaného řešení
3. **Lokální prohledávání** je doplněno stromovým prohledáváním
   - pro výběr souseda z okolí a nebo pro prořezávání stavového prostoru

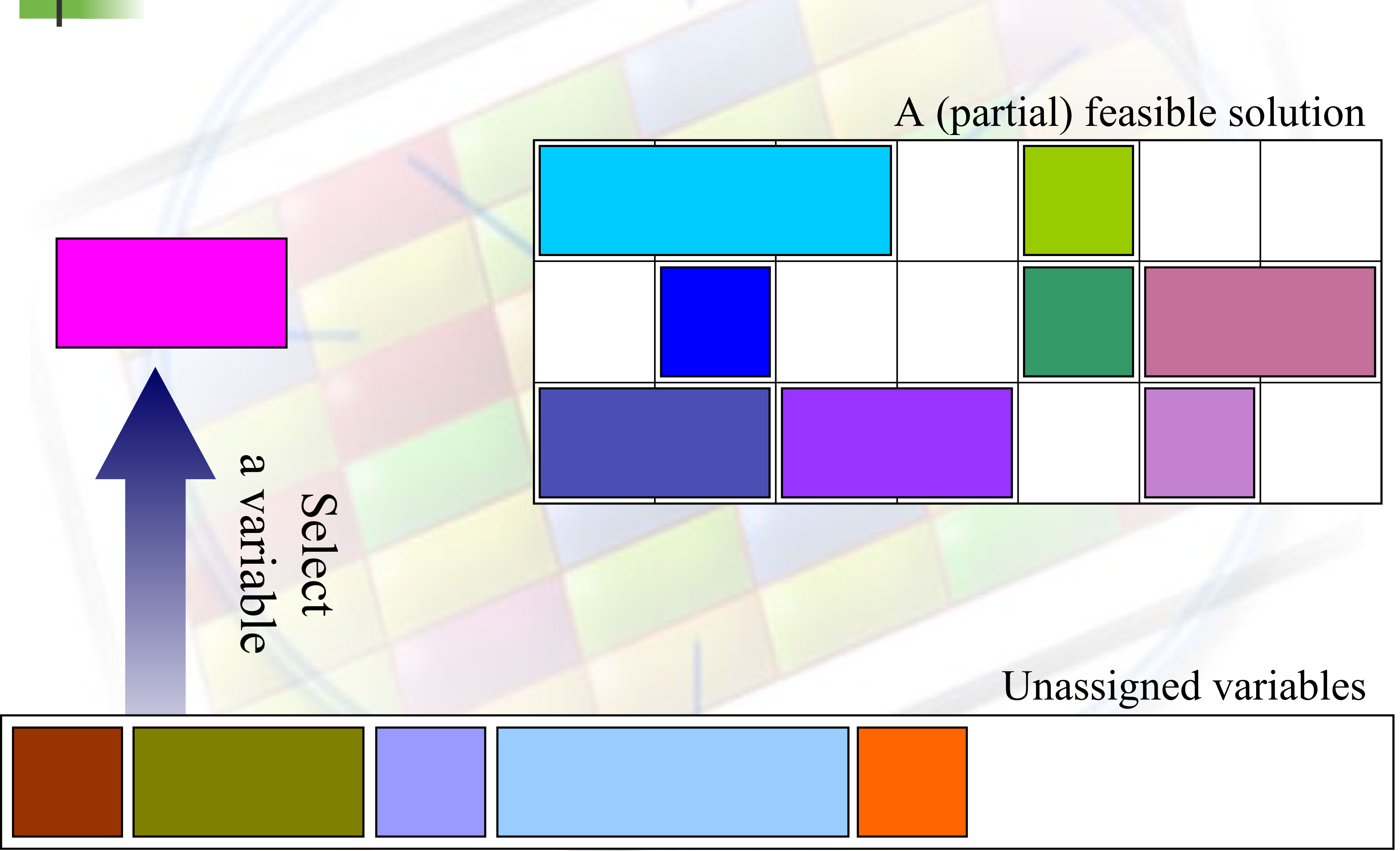
Iterativní dopředné prohledávání

- ***Iterative Forward Search (IFS)***
- **Hybridní prohledávání: konstruktivní nesystematický algoritmus**
  - pracuje nad **modelem s pevnými a měkkými omezujícími podmínkami**
  - pevné podmínky: musí být splněny
  - měkké podmínky: reprezentují účelové funkce, jejichž vážený součet je minimalizován
- Pracuje s **konzistentními přiřazeními**
- Základní myšlenky (blízký dynamickému backtrackingu)
  - začíná s prázdným přiřazením
  - vybere novou proměnnou k přiřazení
  - pokud nalezne konflikt,
    zruší přiřazení všech proměnných v konfliktu s vybranou proměnnou
  - výběr hodnot pomocí **konfliktní statistiky**
  - výběr proměnných není pro algoritmus kritický,
    protože lze proměnné přiřadit opakovaně

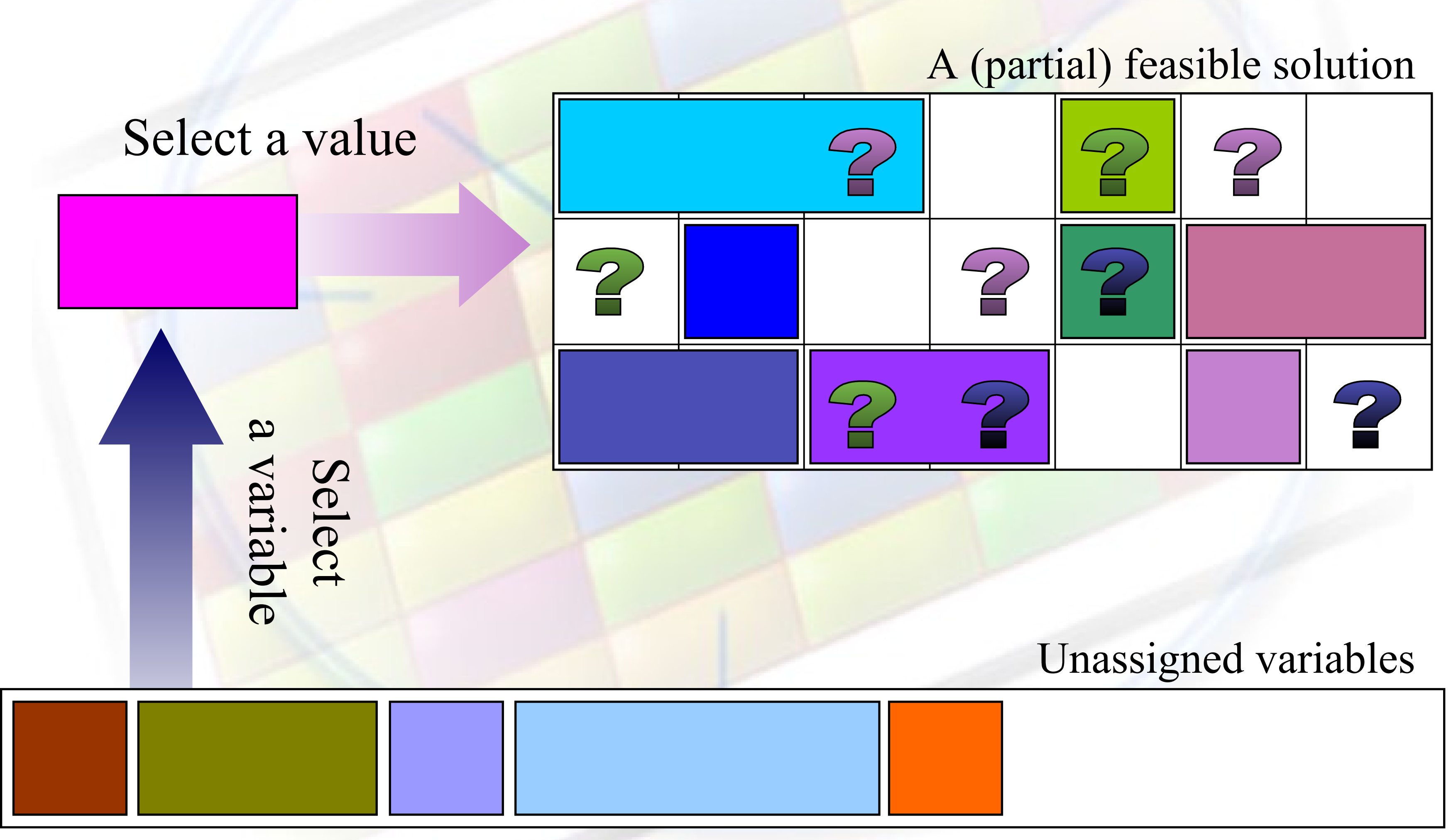
Iterative Forward Search Algorithm



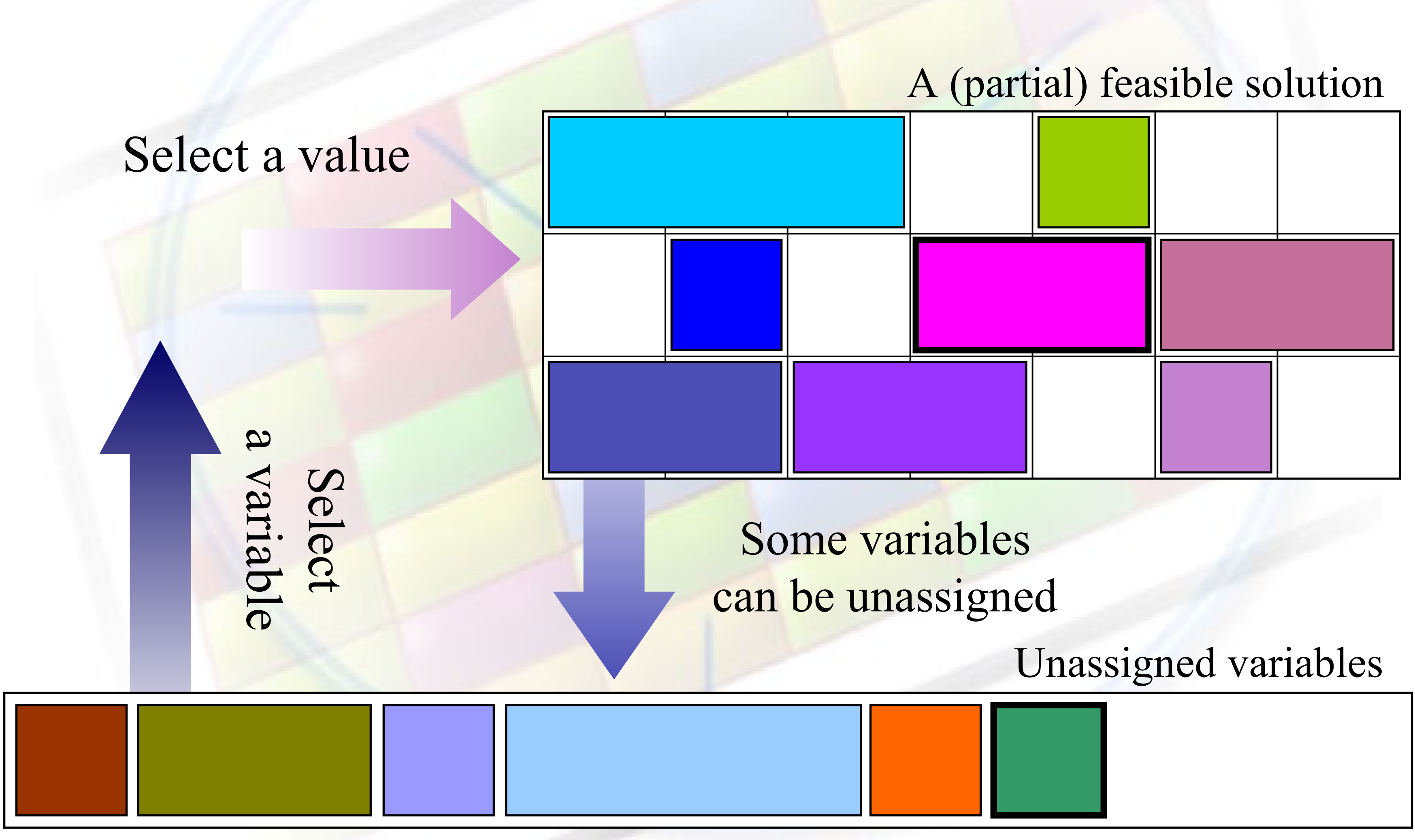
Iterative Forward Search Algorithm



Iterative Forward Search Algorithm



Iterative Forward Search Algorithm



IFS: algoritmus

```
procedure IFS()
iteration = 0;                % čítač iterací
current = ∅;                  % aktuální řešení
best = ∅;                     % nejlepší řešení
while canContinue (current, iteration) do
    iteration = iteration + 1;
    variable = selectVariable (current);
    value = selectValue (current, variable);
    unassign(current, conflictingVariables(current, variable, value));
    assign(current, variable, value);
    if better (current, best) then best = current
return best
end IFS procedure
```

conflictingVariables: kontroluje konzistenci tak, aby byla splněna omezení
na přiřazených proměnných, a vrací konfliktní proměnné

unassign: odstraní přiřazení konfliktních proměnných

IFS: konfliktní statistika

- Předpoklad: při výběru hodnoty a proměnné A

  je nutné zrušit přiřazení hodnoty b proměnné B, tj. $[A = a \rightarrow \neg\ B = b]$

- V průběhu výpočtu si tedy lze pamatovat:

$$A = a \quad \Rightarrow \quad 3 \times \neg\ B = b, \quad 4 \times \neg\ B = c, \quad 1 \times \neg\ C = a, \quad 120 \times \neg\ D = a$$

- **Při výběru hodnoty**
  - vybíráme hodnoty s nejnižším počtem konfliktů vážených jejich frekvencí
    - konflikt započítáme pouze tehdy, pokud to vede k odstranění přiřazení
  - př. $A = a \quad \Rightarrow \quad 3 \times \neg\ B = b, \quad 4 \times \neg\ B = c, \quad 1 \times \neg\ C = a, \quad 120 \times \neg\ D = a$

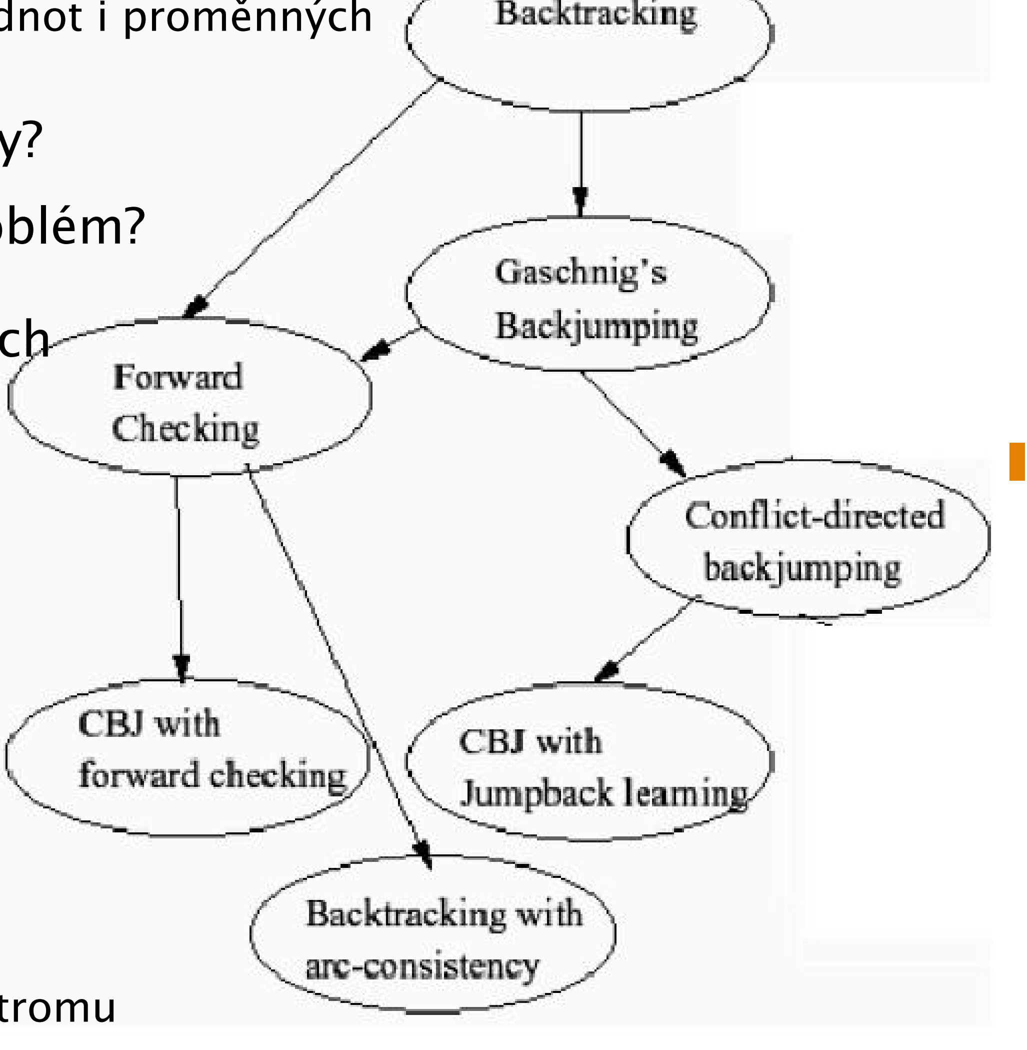
    $A = b \quad \Rightarrow \quad 1 \times \neg\ B = a, \quad 3 \times \neg\ B = b, \quad 2 \times \neg\ C = a$

    Máme přiřazení $B = c, C = a, D = b$ a vybíráme hodnotu pro A:
    - nechť A/a vede ke konfliktu s B/c: vyhodnoceno jako 4
      - není konflikt s C/a, tak se nezapočítává
    - nechť A/b vede ke konfliktu s C/a: vyhodnoceno jako 2
    - tj. vybereme hodnotu b pro proměnnou A

Srovnání prohledavacích algoritmů

Srovnání prohledávacích algoritmů

- $A \rightarrow B$ znamená, že uzly prohledávacího stromu B jsou i v stromě A
  - za předpokladu stejného uspořádání hodnot i proměnných
- Existuje srovnání i pro další algoritmy?
  Jaké algoritmy používat pro daný problém?
- **Experimentální porovnání** na různých sadách problémů ***(benchmarks)***
  - reálné problémy
  - nahodně vygenerované problémy
  - aplikačně založené náhodné problémy
- Kriteria
  - CPU čas
  - velikost generovaného prohledávacího stromu
  - počet volání procedury (např. `Consistent`)



Experimenty na reálných problémech

- Sady reálných problémů ***(benchmarks)***, na kterých lze algoritmy porovnávat
- CSPLib `http://www.csplib.org`
  - knihovna problémů pro omezující podmínky (otevřená pro nové problémy)
  - kolem 130 problémů z oblasti jako je rozvrhování, návrh a konfigurace, kombinatorika, bioinformatika, hry
  - popis problému, reference na jeho řešení, data, výsledky
    někdy i řešení nebo podrobné studie různých možností řešení
  - příklady
    - dopravní signalizace v čase na zadaných křižovatkách, výrobní linka, problém batohu, sledování cíle v distribuovaných senzorických sítích, ...
- Problém: výsledky lze stále velice obtížně zobecnit na další problémy
  - pro jeden problém je lepší jeden algoritmus, pro další problém jiný algoritmus

Náhodné problémy

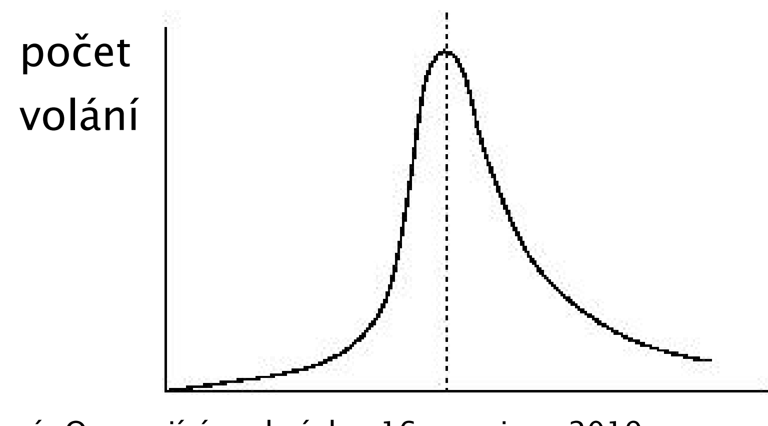
- Algoritmy porovnávány na umělých, náhodně vygenerovaných problémech
  - lze generovat problémy různé obtížnosti (**fáze přechodu**)
  - libovolný počet datových instancí
  - lze testovat, co se stane (např. s parametry algoritmu) při změnách problému
- **Náhodné binární CSP (*random binary CSP*)**
  - parametry (n, m, p_1, p_2)
  - n počet proměnných
  - m počet hodnot v doméně proměnných
  - p_1 pravděpodobnost, že existuje omezení na páru proměnných
  - p_2 pravděpodobnost, že omezení povoluje daný pár hodnot

Aplikačně založené náhodné problémy

- Identifikace problémové domény
  - lze definovat parametrizovatelné problémy
  - problémy mají přitom specifickou (z aplikace vycházející) strukturu
  - problémy lze náhodně generovat s různým nastavením parametrů
- Výhody
  - zaměřené na reálné problémy
  - generování řady problémů umožňuje statistické porovnání
- Příklad: **shop scheduling** problémy
  - m strojů
  - n úloh, každá úloha se skládá z m operací prováděných na odlišných strojích
  - operace jedné úlohy nesmí být prováděny zároveň
  - podmínky na sekvencování operací úlohy (žádné, dáno pořadí, stejné pořadí pro všechny)
  - minimalizace dokončení poslední úlohy, minimalizace největšího zpoždění úlohy, ...

Fáze přechodu

- Náhodný k-SAT problém
  - formule pevné délky jsou generovány výběrem m klauzulí
  - každá klauzule délky k je uniformně náhodně generována z množiny všech klauzulí
- **Obtížnost nalezení řešení**
  - při malém počtu klauzulí je většina problémů splnitelná a snadno řešitelná
  - při velkém počtu klauzulí je detekována snadno nesplnitelnost většiny problémů
  - nalezení řešení je nejobtížnější za předpokladu, že cca 50 % problémů je splnitelných
- Fenomén **fáze předchodu (*phase transition*)**
  - fáze přechodu z obtížně řešitelných problémů na snadno řešitelné problémů



Využití fáze přechodu:

lze generovat problémy různé obtížnosti

poměr počtu klauzulí vůči počtu proměných

Optimalizace & soft omezení: modely

Optimalizační problém s podmínkami (COP)

- Problém splňování podmínek (X, D, C)
  - proměnné $X = \{x_1, \ldots, x_n\}$
  - domény $D = \{D_1, \ldots, D_n\}$
  - omezení $C = \{C_1, \ldots, C_n\}$
    - C_i je definováno na $Y_i \subseteq X$, $Y_i = \{x_{i_1}, \ldots, x_{i_k}\}$
    - C_i je podmnožina $D_{i_1} \times \ldots \times D_{i_k}$
- **Objektivní funkce** $obj : Sol \rightarrow W$
- Základní definice:

  Optimalizační problém s podmínkami (*constraint optimization problem*)

  - nalezení řešení $\vec{d}$ pro (X, D, C) takové, že $obj(\vec{d})$ je optimální
    - optimální $\equiv$ maximální nebo minimální

COP: operační výzkum

- **Pevné (hard, required) omezení** $C_h = \{C_1, \dots, C_n\}$ relace
  - C_i je definováno na $Y_i \subseteq X$, $Y_i = \{x_{i_1}, \dots, x_{i_k}\}$
  - C_i je podmnožina $D_{i_1} \times \dots \times D_{i_k}$
- **Měkké (soft) omezení** $C_s = \{F_1, \dots, F_l\}$ funkce
  - F_j je definované nad $Q_j \subseteq X$, $Q_j = \{x_{j_1}, \dots, x_{j_l}\}$
  - F_j je funkce do reálných čísel $D_{j_1} \times \dots \times D_{j_l} \rightarrow \mathbb{R}^+$
- **Optimalizační problém s podmínkami (COP):** (X, D, C_h, C_s)
- **Objektivní funkce** zjednodušení na $\sum$

$$F(\vec{d}) = \sum_{j=1}^{l} F_j(\vec{d}[Q_j])$$

$\vec{d}[Q_j] \dots$ projekce $\vec{d}$ na Q_j

- **Řešení COP:** $\vec{d}^o$ splňující všechna omezení z C_h tak, že
  $F(\vec{d}^o) = \max_{\vec{d}} F(\vec{d})$ nebo $F(\vec{d}^o) = \min_{\vec{d}} F(\vec{d})$

Použití soft omezení

- Problémy optimalizační, příliš podmíněné, špatně definované problémy, ...
- Fuzzy preference, pravděpodobnosti, ceny, váhy, úrovně požadavků,...
- **Příliš podmíněné problémy:** řešení CSP neexistuje

  F_1 `Prednaska < Cviceni` @ 10 tj. `pokud Prednaska<Cviceni pak` F_1=0
  `pokud Prednaska≥Cviceni pak` F_1=10

  F_2 `Prednaska in 4..5` @ 6 tj. `pokud Prednaska∈{4,5} pak` F_2=0
  `pokud Prednaska∉{4,5} pak` F_2=6

  F_3 `Cviceni in 1..4` @ 4

- **Problémy s nejistotou**
  - Je 0.7 nezbytné, abych přišla do středy. po..st 0.7, ct..ne 0
  - Je nezbytné, abych nepřišla příliš později než ve středu. po..st 0.7, ct 0.5, pa 0.3, so..ne 0
- **Špatně definované problémy:** není zřejmé, která omezení definují CSP

  `Zitra = pekne` @ 80%

  `Zitra = zamraceno` @ 30%

Přístupy pro soft omezení

- Vybrané přístupy
  - základní: MAX-CSP, omezení s váhami, fuzzy omezení
  - zobecňující: omezení nad polookruhy (*semiring-based*)
- Rozlišení systémů na základě: (v závorkách popis pro CSP)
  - **omezení** – rozšíření klasického omezení (c = relace)
  - **problém** – rozšíření CSP (trojice (V, D, C))
  - **úroveň splnění** – jak přiřazení hodnot splňuje problém ($\bigwedge c\theta$)
  - **řešení** – které přiřazení je (optimálním) řešením (splňují všechna omezení)
  - **úroveň konzistence problému** – jak je možné nejlépe splnit problém
    tj. jak (optimální) řešení splňuje problém (true)

Omezení s váhami, MAX-CSP

- **Omezení s váhami**:
  - Váha/cena spojená s každým omezením
  - Omezení – dvojice $(c, w(c))$
  - Problém – trojice (V, D, C_w)
  - Úroveň splnění – funkce na množině přiřazení $\omega(\theta) = \sum_{\theta \models \neg c} w(c)$

    $\Rightarrow$ **součet vah nesplněných omezení**
  - Řešení – přiřazení θ s **minimální** $\omega(\theta)$
  - Úroveň konzistence – úroveň splnění řešení

- **MAX-CSP (maximální CSP)**
  - Váha je rovna jedné $\Rightarrow$ **maximalizace počtu splněných omezení**

Omezení s váhami: příklad

```
Prednaska < Cviceni @ 10
Prednaska in 4..5 @ 6
Cviceni in 1..4 @ 4
```

- Přiřazení: σ = {Prednaska/3, Cviceni/4}
- Úroveň splnění σ odpovídá součtu vah nesplněných omezení při σ, tj. 6
- Řešení: θ = {Prednaska/4, Cviceni/5}
- Úroveň splnění θ: 4
- Úroveň konzistence odpovídá úrovni splnění θ: 4

Fuzzy CSP

- **Fuzzy množiny**: příslušnost prvku k množině zadána číslem z intervalu $[0, 1]$
- **Fuzzy omezení:** fuzzy relace $\mu_c(d_1, \ldots, d_k) \in \langle 0, 1 \rangle$ udává **úroveň preference**

$D_A = D_B = \{1, 2, 3\}$

```
c1: A = 1 @(1,0.7)  tj. pokud A=1, pak μc1=1
                        pokud A≠1, pak μc1=0.7
c2: min( abs(A - B) ), abs(A - B) = 0 => @1   tj. μc2=1
                                  = 1 => @0.5 tj. μc2=0.5
                                  = 2 => @0.1 tj. μc2=0.1
c3: max(A + B) @(A + B)/10   tj. μc3=(A + B)/10
```

- **Fuzzy CSP** (X, D, C_f)
  - C_f je množina fuzzy omezení
  - X uspořádaná množina proměnných

Kombinace a projekce omezení

- **Projekce n-tic** $(d_1, \ldots, d_l) \downarrow^Y_X$ příklad: $(1,2,3,4,5) \downarrow^{(A,B,C,D,E)}_{(D,A,E)} = (4,1,5)$
- **Kombinace** $c = c_X \oplus c_Y$, $dom(c) = Z = X \cup Y$ c, c_X, c_Y omezení nad Z, X, Y

$$\mu_c(d_1, \ldots, d_k) = \min(\mu_{c_X}((d_1, \ldots, d_k) \downarrow^Z_X), \mu_{c_Y}((d_1, \ldots, d_k) \downarrow^Z_Y))$$

  - udává, jaká je úroveň splnění všech přiřazení Z vzhledem k c_X a c_Y
  - příklad (pokračování): kombinace c1 ⊕ c2 ⊕ c3 pro $(1,3)$

$$\mu_{c1 \oplus c2 \oplus c3}(1,3) = \min(\mu_{c1}((1)), \mu_{c2}((1,3)), \mu_{c3}((1,3))) = \min(1, 0.1, 0.4) = 0.1$$

- **Projekce** $c = c_Y \Downarrow_X$, $dom(c) = X, X \subseteq Y$ c, c_X, c_Y omezení nad X, X, Y

$$\mu_c(d_{x1}, \ldots, d_{xk}) = \max_{((d_{y1},\ldots,d_{yl}) \in D_{y1} \times \cdots \times D_{yl}) \wedge ((d_{y1},\ldots,d_{yl}) \downarrow^Y_X = (d_{x1},\ldots,d_{xk}))} \mu_{c_Y}(d_{y1}, \ldots, d_{yl})$$

  - udává, jaká je úroveň splnění všech přiřazení X vzhledem k c_Y
  - příklad (pokračování): projekce c3 $\Downarrow_{(B)}$ na (1)

$$\mu_{c3 \Downarrow_{(B)}}(1) = \max(\mu_{c3}(1,1), \mu_{c3}(2,1), \mu_{c3}(3,1)) = \max(0.2, 0.3, 0.4) = 0.4$$

Řešení fuzzy CSP

- **Úroveň splnění** přiřazení $(d_1,\ldots,d_n)$ dána jako $\mu_{\bigoplus C}(d_1,\ldots,d_n)$
- **Řešení** – přiřazení $(d_1,\ldots,d_n)$ takové, že

$$\max_{(d_1,\ldots,d_n)\in D_1\times\cdots\times D_n} \mu_{\bigoplus C}(d_1,\ldots,d_n) = \mathbb{C}(P_\mu)$$

  - příklad: $\bigoplus C$ = c1 $\oplus$ c2 $\oplus$ c3 pro všechna (A, B)

    $(3,3)\ldots 0.6$, $(2,2)\ldots 0.4$, $(1,1)\ldots 0.2$

    $(2,3)$ a $(3,2)\ldots 0.5$, $(2,1)$ a $(1,2)\ldots 0.3$

    $(1,3)$ a $(3,1)\ldots 0.1$ $\Rightarrow$ $(3,3)$ je řešení, $\mathbb{C}(P_\mu) = 0.6$

- **Úroveň nekonzistence** $1 - \mathbb{C}(P_\mu)$
- $\mathbb{C}(P_\mu)$ je **úroveň konzistence**

  také jako projekce na prázdnou množinu proměnných $\bigoplus C \Downarrow_\emptyset$

Příklad: řešení fuzzy CSP

- Viz dříve: $\bigoplus C$ = c1 $\oplus$ c2 $\oplus$ c3 pro všechna (A, B)

  $(3,3)\ldots 0.6,\ (2,2)\ldots 0.4,\ (1,1)\ldots 0.2,$

  $(2,3)$ a $(3,2)\ldots 0.5,\ (2,1)$ a $(1,2)\ldots 0.3,\ (1,3)$ a $(3,1)\ldots 0.1$

- $\bigoplus C = \bigoplus C \Downarrow_{\{A,B\}}$

  $\bigoplus C \Downarrow_{\{A,B\}} (3,3) = \max(\mu_{\bigoplus C}(3,3)) = 0.6,$

  $\bigoplus C \Downarrow_{\{A,B\}} (2,2) = 0.4, \ldots$

- $\bigoplus C \Downarrow_{\{A\}}$

  $\bigoplus C \Downarrow_{\{A\}} (1) = \max(\mu_{\bigoplus C}(1,1), \mu_{\bigoplus C}(1,2), \mu_{\bigoplus C}(1,3)) = \max(0.2, 0.3, 0.1) = 0.3$

  $\bigoplus C \Downarrow_{\{A\}} (2) = \max(\mu_{\bigoplus C}(2,1), \mu_{\bigoplus C}(2,2), \mu_{\bigoplus C}(2,3)) = \max(0.3, 0.4, 0.5) = 0.5$

  $\bigoplus C \Downarrow_{\{A\}} (3) = \max(\mu_{\bigoplus C}(3,1), \mu_{\bigoplus C}(3,2), \mu_{\bigoplus C}(3,3)) = \max(0.1, 0.5, 0.6) = 0.6$

- $\bigoplus C \Downarrow_{\emptyset}$

  $$\bigoplus C \Downarrow_{\emptyset} () = \max(\mu_{\bigoplus C}(1,1), \mu_{\bigoplus C}(1,2), \mu_{\bigoplus C}(1,3), \mu_{\bigoplus C}(2,1), \mu_{\bigoplus C}(2,2), \mu_{\bigoplus C}(2,3), \mu_{\bigoplus C}(3,1), \mu_{\bigoplus C}(3,2), \mu_{\bigoplus C}(3,3)) = 0.6$$

Omezení nad polookruhy

- *Semiring-based CSP*
- c-**polookruh** $\langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$
  - $\mathcal{A}$ množina polookruhu (množina preferencí)
  - $\mathbf{0} \in \mathcal{A}$ (úplné nesplnění), $\mathbf{1} \in \mathcal{A}$ (úplné splnění)
  - $+$ komutativní, idempotentní, asociativní operace
    s jednotkovým prvkem $\mathbf{0}$, s absorbujím prvkem $\mathbf{1}$
  - $\times$ komutativní, asociativní operace, distributivní nad $+$,
    s jednotkovým prvkem $\mathbf{1}$, s absorbujím prvkem $\mathbf{0}$
- **$\times$ se používá ke kombinaci preferencí několika omezení** $\quad$ $\min$ u fuzzy CSP
  $\mathbf{0}$ minimum (nesplnění), $\mathbf{1}$ maximum (splnění)
- **Částečné uspořádání $\leq_S$**: $a \leq_S b$ právě tehdy, když $a + b = b$
  používá se k výběru „lepšího” přiřazení $\quad$ $\max$ u fuzzy CSP

Instance omezení nad polookruhy

Přístup	$\mathcal{A}$	$+$ uspořádání	$\times$ kombinace	$\mathbf{0}$	$\mathbf{1}$
CSP	$\{0,1\}$	$\vee$	$\wedge$	0	1
Fuzzy CSP	$\langle 0,1\rangle$	$\max$	$\min$	0	1
CSP s váhami	$\mathbb{N} \cup \{+\infty\}$	$\min$	$+$	$+\infty$	0

- Důležité vlastnosti:
  - **striktní monotonie**: $\forall a,b,c \in \mathcal{A} : ((a < c) \wedge (b \neq \mathbf{0})) \Rightarrow ((a \times b) < (c \times b))$
    fakt, že něco lze lokálně zlepšit nelze globálně ignorovat (platí pro CSP s váhami)
    př. $a = 0.3, c = 0.4, b = 0.2$ pro fuzzy CSP: není striktní monotonie
  - **idempotence**: $\forall a \in \mathcal{A} : a \times a = a$ (platí pro fuzzy CSP)
    omezení, které je v problému obsaženo, může být do něj přidáno beze změny významu
    př. $x > 1@10, x > 1@10, x = 0@\infty$, přiřazení $x = 0$ má pro CSP s váhami úroveň konz. 20
  - striktní monotonie a idempotence $\times$ zároveň pouze pro dvouprvkové $\mathcal{A}$, tj. jen pro CSP
- Existující vztahy: CSP $\equiv$ fuzzy CSP na dvouprvkové $\mathcal{A}$
  fuzzy CSP lze polynomiálně transformovat na CSP s váhami

Definice omezení nad polookruhy

- **Systém** (S, D, V):
  S c-polookruh, D konečná doména, V uspořádaná množina proměnných
- **Soft omezení** (def, con): rozsah omezení $con \subseteq V$, $def : D^{|con|} \rightarrow \mathcal{A}$
- **Soft problém** je (C, con) nad (S, D, V), kde $con \subseteq V$, C množina omezení
- **Projekce n-tic** $\vec{t} \downarrow^X_Y$
  - **příklad:** $(1, 2, 3, 4, 5) \downarrow^{(A,B,C,D,E)}_{(D,A,E)} = (4, 1, 5)$
- **Kombinace** $c = c_1 \otimes c_2$ $\quad c_1 = (def_1, con_1)$ and $c_2 = (def_2, con_2)$
  $c = (def, con)$, $con = con_1 \cup con_2$,

$$def(\vec{t}) = def_1(\vec{t} \downarrow^{con}_{con_1}) \times def_2(\vec{t} \downarrow^{con}_{con_2})$$

  - **příklad:** CSP s váhami: $\mathcal{A} = \mathbb{N} \cup \{\infty\}$, $+ \equiv \min$, $\times \equiv +$, $+\infty$, 0
    zadáno omezení c_1 na xy: aa 2, ab 4, ba 1, bb 0,
    zadáno omezení c_2 na x: a 0, b 1
    kombinace $c = c_1 \otimes c_2$: aa 2(=2+0) ab 4(=4+0) ba 2(=1+1) bb 1(=0+1)

Řešení pro omezení nad polookruhy

- **Projekce** $c \Downarrow_I$ $\quad c = (def, con),\ I \subseteq V\ c' = (def', con'),\ con' = con \cap I$

$$def'(\vec{t}') = \sum_{\vec{t}/\vec{t}\downarrow^{con}_{I\cap con}=\vec{t}'} def(\vec{t})$$

  - příklad (pokračování): c_1 na xy: aa 2, ab 4, ba 1, bb 0, projekce $c_1 \Downarrow_{\{x\}}$: a 2, b 0

- **Úroveň splnění** problému $P = (C, con)$ udává omezení

$$Sol(P) = (\bigotimes C) \Downarrow_{con}$$

  - kombinace všech omezení v C a následovně projekce na proměnné v con
  - pro každé přiřazení proměnných v con

    vrací omezení $Sol(P)$ jeho úroveň splnění

  - příklad (pokračování): c_1 na xy: aa 2, ab 4, ba 1, bb 0 $\quad c_2$ na x: a 0, b 1

    problém $P_1 = (\{c_1, c_2\}, \{x, y\})$: $\quad Sol(P_1) = (c_1 \otimes c_2) \Downarrow_{\{x,y\}}$: aa 2, ab 4, ba 2, bb 1

    problém $P_2 = (\{c_1, c_2\}, \{x\})$: $\quad Sol(P_2) = (c_1 \otimes c_2) \Downarrow_{\{x\}}$: a 2, b 1

- **Úroveň konzistence:** $blevel(P) = Sol(P) \Downarrow_{\emptyset}$
  - příklad (pokračování): $blevel(P_1) = blevel(P_2) = 1$

Optimalizace & soft omezení: algoritmy

Soft propagace

- Klasická propagace: eliminace nekonzistentních hodnot z domén proměnných
- **Soft propagace**: propagace preferencí (cen) nad k-ticemi hodnot proměnných
  - snaha o získání realističtějších preferencí
  - výpočet realističtějších příspěvků pro cenovou funkci
- C je **soft k-konzistentní**, jestliže pro všechny podmnožiny $(k-1)$ proměnných W a libovolnou další proměnnou y platí

$$\otimes\{c_i \mid c_i \in C \wedge con_i \subseteq W\} = (\otimes\{c_i \mid c_i \in C \wedge con_i \subseteq (W \cup \{y\})\}) \Downarrow_W$$

  - con_i označuje proměnné v omezení c_i
  - uvažování (*def*) pouze omezení ve W stejné jako: uvažování (*def*) omezení ve W a omezení spojující y s W, s následnou projekcí na W

Soft hranová konzistence (SAC)

- $k = 2, W = \{x\} : c_x = (\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_{\{x\}}$
- CSP: libovolná hodnota v doméně x může být rozšířena o hodnotu v doméně y tak, že je c_{xy} splněno
  - hodnoty a v doméně x, které nelze rozšířit na y, mají $def((a)) = \mathbf{0}$
- Fuzzy CSP: úroveň preference všech hodnot x v c_x je stejná jako úroveň preference daná $(\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_{\{x\}}$
- Příklad na fuzzy CSP: $\mathcal{A} = \langle 0, 1 \rangle, + \equiv \max, \times \equiv \min, 0, 1$
  - $x, y \in \{a, b\}$
  - $c_x : a \ldots 0.9, b \ldots 0.1,\ c_y : a \ldots 0.9, b \ldots 0.5,\ c_{xy} : aa \ldots 0.8,\ ab \ldots 0.2,\ ba \ldots 0,\ bb \ldots 0$
  - není SAC: c_x dává 0.9 na $x = a$, ale $(\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_{\{x\}}$ dává 0.8 na $x = a$
    $\otimes\{c_y, c_{xy}, c_x\}$ dává na $(a, a) : \min(0.9, 0.8, 0.9) = 0.8$
    $\otimes\{c_y, c_{xy}, c_x\}$ dává na $(a, b) : \min(0.5, 0.2, 0.9) = 0.2$
    projekce $(\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_{\{x\}}$ dává na $(a) : \max(0.8, 0.2) = 0.8$

Výpočet SAC

- Základní algoritmus pro **výpočet SAC:**
  - pro každé x a y změnit definici c_x tak, aby korespondovala s $(\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_{\{x\}}$
  - iterace až do dosažení stability
- $\times$ idempotentní (fuzzy CSP)
  - zajištěna ekvivalence, tj. původní i nový (po dosažení SAC) problém mají stejné řešení
  - zajištěno ukončení algoritmu
- $\times$ není idempotentní (CSP s váhami)
  - pro každé $u, v \in \mathcal{A}$ existuje $w \in \mathcal{A}$ taková, že $v \times w = u$
    - w se nazývá **rozdíl** mezi u a v, **maximální rozdíl** se značí $u - v$
    - nutno požadovat novou vlastnost pro systém (a rozšířit projekci a kombinaci)
  - s novou vlastností zajištěna ekvivalence, při striktní monotonii zajištěno i ukončení
    - tato vlastnost platí pro CSP s váhami

Řešení COP

- **Cíl:** nalezení úplného řešení s optimální hodnotou cenové funkce
- **Prohledávání**
  - lokální prohledávání
    - přímé zahrnutí optimalizačního kriteria do evaluace (hodnota obj. funkce)
  - stromové prohledávání
    - metoda větví a mezí + její rozšíření
- CSP s omezeními: **minimalizace součtu vah omezení** $\min \sum_{c \in C} c(\vec{d})$
  - velmi častá optimalizační úloha
  - váha (**cena**) omezení: 0 = úplné splnění, $(0, \infty)$ částečné nesplnění, ∞ úplné nesplnění
  - hodnota cenové funkce: **cena přiřazení/řešení**
  - maximalizace je duální problém
  - algoritmy pro fuzzy CSP na podobných principech

COP jako série CSP problémů

- Triviální metoda řešení
- Řešení COP jako CSP a nalezení iniciální hodnoty cenové funkce W^1
- **Opakované řešení CSP ($i = 1 \ldots$)**
  s přidáním omezení $\sum_{c \in C} c(\vec{d}) < W^i$
- Když řešení nového CSP neexistuje, pak poslední W^i dává optimum
- Výpočetně zbytečně náročné
- Efektivní rozšíření obtížné

Metoda větví a mezí (*branch&bound*) BB

- Prohledávání stromu do hloubky
  - přiřazené=**minulé proměnné** P, nepřiřazené=**budoucí proměnné** F
  - omezení pouze na minulých proměnných C_P, omezení na minulých i budoucích proměnných C_{PF}, omezení pouze na budoucích proměnných C_F
- Výpočet **mezí**
  - **horní mez** UB: cena nejlepšího dosud nalezeného řešení
  - **dolní mez** LB: dolní odhad minimální ceny pro současné částečné přiřazení
- **Ořezávání**: $LB \geq UB$ (cíl: minimalizace)
  - víme, že rozšíření současného částečného řešení už bude mít horší (vyšší) cenu LB než dosud nalezené řešení UB
  - lze proto ořezat tuto část prohledávacího prostoru
  - příklad: pokud nalezneme řešení s cenou 10 odřízneme všechny větve, které mají cenu vyšší než 9

Metoda větví a mezí: výběr hodnoty

- **Algorimus metody větví a mezí**
  - generický algoritmus rozšiřitelný jako implementace backtrackingu
  - možná rozšíření zejména o: pohled dopředu, výpočet dolní meze
- $LB(\vec{d})$ vrací dolní odhad ceny pro každé částečné přiřazení $\vec{d}$
  - použití při rozšíření o jednu proměnnou $LB(\vec{a}_{i-1}, a)$
- **Optimistický výběr hodnoty**

```
procedure Select-Value-BB
while D'_i is not empty
      vyber a smaž libovolný a ∈ D'_i takový, že min LB(a_{i-1}, a)
      if Consistent(a_{i-1}, x_i = a) a LB(a_{i-1}, a) < UB
         return a                          (jinak ořezej a)
return null                                (konzistentní hodnota neexistuje)
```

Algoritmus metody větví a mezí

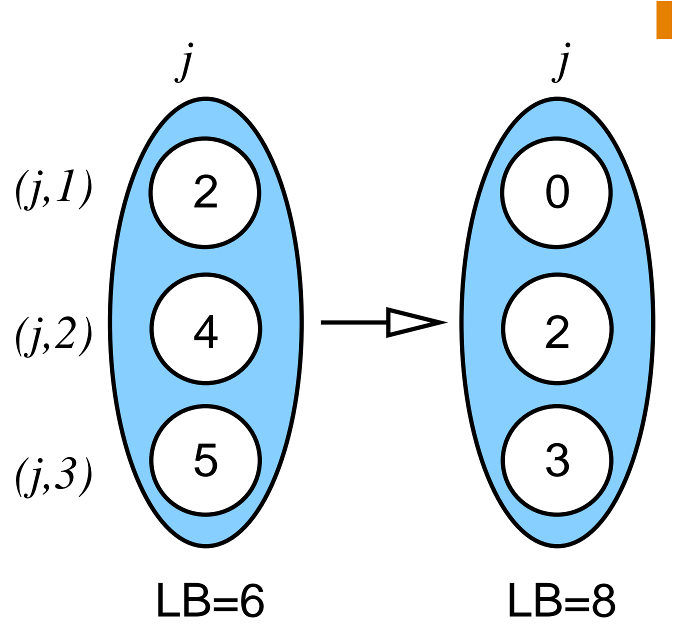
```
procedure Branch-Bound((X,D,C),UB,f)                 rozdíly od backtrackingu
i := 1, D'_i := D_i        (inicializace čítače proměnných, kopírování domény)
Reseni := null                (řešení dosud nenalezeno)
repeat while 1 ≤ i ≤ n
            přiřazení x_i := Select-Value-BB
            if x_i is null          (žádná hodnota nebyla vrácena)
               i := i − 1           (zpětná fáze)
            else i := i + 1         (dopředná fáze)
                 D'_i := D_i
      if i = 0  if Reseni is not null
                  return UB, Reseni
                else return „nekonzistentní''
      else Reseni := x_1,...,x_n    (uložení hodnot dosud nejlepšího přiřazení)
           spočítej cenu současného přiřazení W = Σ_{c∈C} c(x⃗), UB:=min{W,UB}
           i := 1, nastav D'_k na D_k pro k = 1...n
until true
end Branch-Bound
```

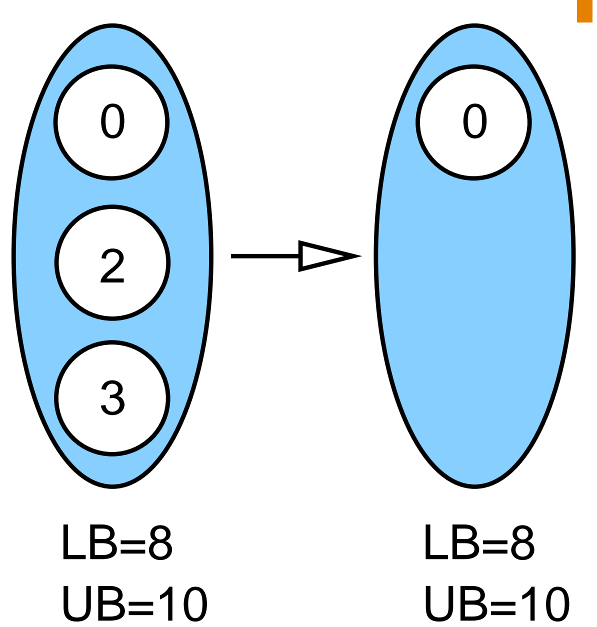
Dolní mez

- **Kvalita dolní meze**: velmi důležitá pro efektivitu BB
- Dolní mez lze ovlivnit pomocí
  - **ceny minulých proměnných**
    - vzdálenost (součet vah omezení na minulých proměnných)
  - **lokální ceny budoucích proměnných vzhledem k minulým proměnným**
    - NC*
  - **lokální ceny budoucích proměnných**
    - AC*
  - **globální ceny budoucích proměnných**
    - prohledávání ruská panenka

NC* algoritmus

- Projekce ceny hodnot $(j,*)$ pro každou proměnnou j do dolní hranice LB ceny řešení
- Smazání hodnot (j,a) převyšující (nebo rovné) horní hranici UB





- Hodnotu a proměnné j značíme (j,a)
  - obrázek: proměnná j má nejprve tři hodnoty $(j,1),(j,2),(j,3)$, jejichž cena je 2,4 a 5
- Všechny hodnoty proměnné j značíme $(j,*)$

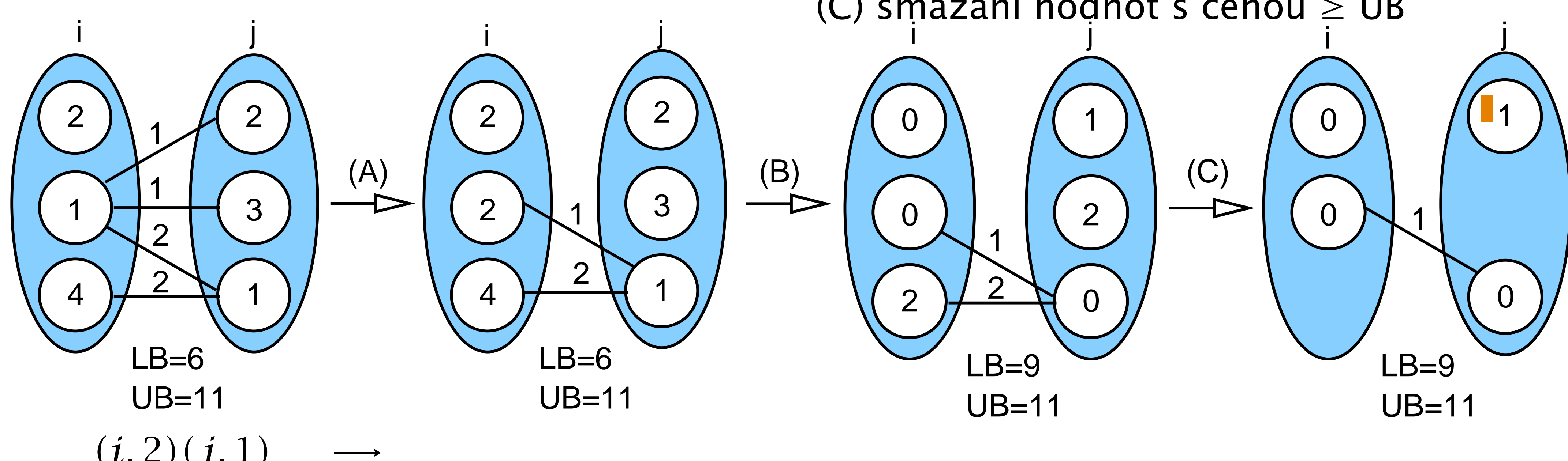
AC* algoritmus

(A) Projekce ceny hrany $(i, a)(j, b)$ do ceny hodnoty (i, a), pokud je tato cena zahrnuta ve všech hranách $(i, a)(j, *)$

- NC* algoritmus

(B) projekce ceny hodnot pro každdou proměnnou

(C) smazání hodnot s cenou $\geq$ UB



$(i, 2)(j, 1) \longrightarrow$

$(i, 2)(j, 2) \longrightarrow \quad (i, 2)$

$(i, 2)(j, 3) \longrightarrow$

Prohledávání ruská panenka (*Russion doll search*)

- **n po sobě jdoucích BB prohledávání**, každé má navíc jednu proměnnou
  - první podproblém obsahuje pouze n-tou proměnnou — statické uspořádání proměnných

    i-tý podproblém obsahuje posledních i proměnných $((n-i+1)\dots n)$
  - podproblémy řešeny pomocí BB s využitím LB a UB z předchozích běhů
- Při **řešení podproblému** $(n-i+1)$
  - problém zahrnuje proměnné $x_i, x_{i+1}, \dots, x_n$
  - mějme částečné přiřazení pro tento podproblém $(a_i, a_{i+1}, \dots, a_{i+j})$ s nepřiřazenými proměnnými $x_{i+j+1}, \dots, x_n$
  - do dolní meze lze zahrnout optimální cenu $(n-i-j)$ podproblému $optim(x_{i+j+1}, \dots, x_n)$
    - $LB((a_i, a_{i+1}, \dots, a_{i+j})) = dist((a_i, a_{i+1}, \dots, a_{i+j})) + optim(x_{i+j+1}, \dots, x_n)$

      $dist((a_i, a_{i+1}, \dots, a_{i+j}))$ vzdálenost (součet vah omezení na minulých proměnných)
  - optimální řešení přechozích problémů použita pro:

    výběr hodnoty, pro zlepšení iniciální horní meze
- Vnořená prohledávání se vyplatí vzhledem k prořezání stavového prostoru

Opakování

Vzorové písemné práce

- 2 vzorové písemné práce na webu předmětu
- **Struktura písemné práce**: 7 otázek
  1. 1 přehledová otázka
  2. otázky na propagaci a konzistenční algoritmy
  3. otázky na stromové prohledávání
  4. otázky na lokální prohledávání a optimalizace
  5. 1 otázka na řešení problému v OPL
- **Hlavní typy otázek pro 2,3,4**
  - příklady k vypočítání (jako vzorové příklady – někdy kratší verze)
  - pojem (a příklad)
  - kód algoritmu (a příklad)
  - porovnání pojmů/algoritmů (a příklady)

Příklady s řešeními na webu I.

Příklady 1.

- binarizace
- AC-1 vs. AC-3
- konzistence mezí vs. hranová konzistence

Příklady 2.

- strom stavového prostoru: backtracking vs. kontrola dopředu vs. pohled dopředu

Příklady 3.

- omezení s váhami

Příklady 4.

- přehled konzistenčních algoritmů
- algoritmus AC-4
- algoritmus DAC

Příklady s řešeními na webu II.

Příklady 5.

- strom stavového prostoru: backtracking vs. kontrola dopředu
- pohled zpět: Gaschnigův skok zpět

Příklady 6.

- přehled prohledávacích algoritmů
- problém rozvrhování výuky v OPL

Příklady 7.

- binarizace
- podpora hodnoty
- konzistence mezí vs. hranová konzistence
- algoritmus DAC, nalezení řešení bez navracení

Příklady s řešeními na webu III.

Příklady 8.

- strom stavového prostoru: backtracking vs. kontrola dopředu vs. pohled dopředu
- problém plánování projektu v OPL

Příklady 9.

- stromový CSP, nalezení řešení bez navracení
- algoritmus PC-2
- problém rozvrhování výuky jedné místnosti v OPL
- strom stavového prostoru: kontrola dopředu vs. pohled dopředu bez iniciální konzistence
- rozvrhování úkolů pro zaměstnance na směny

Příklady s řešeními na webu IV.

Příklady 10

- nalezení podpory
- algoritmus DAC a nalezení řešení bez navracení
- problém přiřazení poboček k obchodům v OPL
- BBS-DBS, DDS
- rozvrhování na směny

Pojem: směrová konzistence a šířka grafu

Co to je šířka grafu a jaký je její význam? Použité pojmy objasněte.

- Šířka grafu je minimum z šířek všech jeho uspořádaných grafů.
- Šířka uspořádaného grafu je maximum z šířek jeho vrcholů.
- Šířka vrcholu v uspořádaném grafu je počet hran vedoucích z tohoto vrcholu do předchozích vrcholů.
- Uspořádaný graf je graf s lineárním uspořádáním vrcholů.
- Šířka grafu nám říká, jak silnou úroveň směrové i-konzistence potřebujeme, abychom nalezli řešení problému bez navracení.

Algoritmus: lokální prohledávání

- Napište (nebo alespoň slovně popište) algoritmus prohledávání s tabu seznamem. Je Váš algoritmus kombinován s nějakou další metodou lokálního prohledávání?

```
procedure TSHC(MaxZmen)
θ := náhodné ohodnocení proměnných    % iniciální přiřazení
PocetZmen := 0
while E(θ) > 0 ∧ PocetZmen<MaxZmen do
      vyber ⟨V,a⟩ s nejlepší evaluací tak, že
            není v tabu seznamu a nebo splňuje aspirační kriterium
      přidej ⟨V,c⟩ do tabu seznamu, kde c je současná hodnota V
      smaž nejstarší položku v tabu seznamu
      přiřad’ a do V
      PocetZmen := PocetZmen+1
return θ
```

Algoritmus je kombinován s metodou stoupání/hill climbing.

Příklady stručněji: omezení s váhami

- Jaká je úroveň konzistence pro následující CSP problém s váhami s omezeními c1 a c2?

```
P=({c1,c2},{X,Y})
dom(X)=dom(Y)={r,s}
con(c1)={X,Y}
def(r,r)=1, def(r,s)=2, def(s,r)=4, def(s,s)=6
con(c2)={Y}
def(r)=0, def(s)=1
```

Výsledek: blevel(P)=1. Nutno uvést stručný postup, např.

Spočítáme $(c_1 \otimes c_2) \Downarrow_{\emptyset} () =$

$$= \min(\mu_{c_1 \otimes c_2}(r,r), \mu_{c_1 \otimes c_2}(r,s), \mu_{c_1 \otimes c_2}(s,r), \mu_{c_1 \otimes c_2}(s,s)) =$$

$$= \min(1+0, 2+1, 4+0, 6+1) = 1$$

Příklady stručněji: omezení s váhami II.

- A jaká je úroveň splnění pro problém P1=({c1,c2},{Y})?
  - výsledek: úroveň splnění pro Y=r: 1, pro Y=s: 3, postup:

    Spočítáme

    $(c_1 \otimes c_2) \Downarrow_Y (r) = \min(\mu_{c_1 \otimes c_2}(r,r), \mu_{c_1 \otimes c_2}(s,r)) = \min(1+0, 4+0) = 1$

    $(c_1 \otimes c_2) \Downarrow_Y (s) = \min(\mu_{c_1 \otimes c_2}(r,s), \mu_{c_1 \otimes c_2}(s,s)) = \min(2+1, 6+1) = 3$

```
P=({c1,c2},{X,Y})
dom(X)=dom(Y)={r,s}
con(c1)={X,Y}
def(r,r)=1, def(r,s)=2, def(s,r)=4, def(s,s)=6
con(c2)={Y}
def(r)=0, def(s)=1
```

Doménové proměnné

- Celočíselné proměnné dvar int+
  - Sudoku: hodnota pole `dvar int+ sudoku[1..Size][1..Size] in 1..Size;`

    ```
    forall (i in 1..Size) allDifferent(all (j in 1..Size) sudoku[i][j]);      //ruzne pro kazdy radek
    ```
  - přiřazení pracovníků:pracovník vyrábí produkt

    ```
    dvar int+ W[1..nbWorkers] in 1..nbProducts;
    total == sum (w in 1..nbWorkers) effectivity[w][W[w]];
    ```
- Binární proměnné dvar boolean
  - problém bat'ohu: je předmět v bat'ohu? `dvar boolean take[1..nbItems];`

    ```
    sum (i in Items) (take[i] * weights [i]) <= capacity;
    ```
  - Sudoku: hodnota pole `dvar boolean sudoku[1..Size][1..Size][1..Size];`

    ```
    forall (i,k in 1..Size)  sum (j in 1..Size) sudoku[i][j][k] == 1;
    ```

    navíc: `forall (i, j in 1..Size) sum(k in 1..Size) sudoku[i][j][k] == 1;`

Rozvrhování: zdroje

- Jeden zdroj
  - jednotková doba trvání úlohy: dvar int, allDifferent(casy)
  - odlišné doby trvání úloh: dvar interval, dvar sequence, noOverlap(casy)
- Více zdrojů
  - nepožadujeme určení zdroje pro úlohu
    - bez doménových proměnných pro určení zdroje úlohy

      cumulFunction, pulse(casy[uloha], zdroju[uloha])
  - požadujeme určení zdroje pro úlohu

    včetně doménových proměnných pro určení zdroje úlohy

    dvar interval prirazeni ... optional

    alternative(casy[uloha], ... prirazeni[uloha][zdroje], zdroju[uloha])

Rozvrhování II.

- Vztahy mezi úlohami a precedence endBeforeStart, endAtStart, ...

```
tuple Zavislost { int Pred; int Po; }
{Zavislost} zavislosti = ...;
forall (zav in zavislosti) endBeforeStart(casy[zav.Pred], casy[zav.Po]);
```

- Volitelné úlohy presenceOf(dvar interval)

```
dvar interval prirazeni[1..Uloh][1..Stroju] optional;
forall (u in 1..Uloh, s not in mozne[u] ) presenceOf( prirazeni[u][s] ) == 0;
forall (u in 1..Uloh, s in prikazane[u]) presenceOf( prirazeni[u][s] ) == 1;
```

- Dovolená: práce lidí se nepřekrývá s jejich dovolenou

  řešení: přidáme dodatečné intervalové proměnné

```
dvar interval casy[u in 1..Prace+Dovolene] size trvani[u];
noOverlap(all (u in 1..Prace+Dovolene) casy[u]);
cumulFunction nastroje = sum (p in 1..Prace)  pulse( casy[p],nastroju[p] );
```

Optimalizace

- Problém baťohu: maximalizace součtu cen předmětů v baťohu

  ```
  maximize sum (i in nbItems) ( take[i] * prices[i] );
  ```

- Minimalizace času dokončení poslední úlohy

  ```
  minimize max (u in 1..Pocet) endOf( casy[u] );
  ```

  zadány poslední operace úlohy:

  ```
  minimize max (u in 1..Pocet) endOf( casy[u][Posledni] );
  ```

- Maximalice (váženého) počtu realizovaných úloh

  ```
  maximize sum (u in 1..Pocet) ( vahy[u] * presenceOf(casy[u]) );
  ```

Určete čas a místnost pro výuku množiny předmětů. Datová instance:

1. rozvrh tvoříte pro 3 vyučovací dny a každý den má 10 hodin
2. máte zadáno 14 předmětů
3. délka výuky předmětu je 4 nebo 5 hodin (určeno pro každý předmět předem)
4. máte k dispozici 3 místnosti
5. máte zadány 3 třídy (skupiny žáků)
6. každá třída má v zadání určeno 4 až 5 předmětů, které bude navštěvovat

Omezující podmínky:

7. výuka předmětu musí probíhat souvisle bez přerušení (tj. nesmí např. začínat jeden den večer a končit druhý den ráno)
8. v každé místnosti je nejvýše jeden předmět v danou dobu
9. pro každou třídu je zadána množina jejích předmětů; každá třída může mít vždy nejvýše jeden z těchto předmětů v danou dobu

Účelová funkce:

10. Jednotlivé předměty by měly být do rozvrhu umístěny tak, aby výuka všech tříd skončila co nejdříve (např. třetí den dopoledne). Tj. minimalizujte součet koncových časů výuky posledních předmětů všech tříd.

Školní rozvrh: vstupní proměnné

```
int Dny = 3;
int Hodin = 10;
int Predmetu = 14;
int Mistnosti = 3;
int Tridy = 3;

range rTridy = 1..Tridy;
range rMistnosti = 1..Mistnosti;

tuple predmetyTrid{
 key int predmet;
 int trida;
 int trvani;}

// např. trida 1 ma predmety 1,2,3; trida 2 ma predmety 4,5, ...
//       predmety 1,2,3,4,5 maji trvani 4,4,5,5,4
// predmet je klic, tj. napr. <2> odkazuje na cely tuple <2,1,4>
{predmetyTrid} PredmetyTrid = {<1,1,4>,<2,1,4>,<3,1,5>,<4,2,5>,<5,2,4>,...};
```

Školní rozvrh: model

Doménové proměnné

```
// přiřazení času předmětům v maximálním rozmezí Dny*Hodin, zajištění trvání
dvar interval casy[p in PredmetyTrid] in 0 .. Dny*Hodin size p.trvani;

// volitelný interval pro přiřazení předmětů a tříd do místnosti
dvar interval prirazeni[PredmetyTrid][rMistnosti] optional;
```

Každý předmět je vyučován právě v jedné místnosti

```
forall(p in PredmetyTrid)
     alternative(casy[p.predmet], all(m in rMistnosti) prirazeni[p][m]);
```

V každé místnosti je nejvýše jeden předmět v danou dobu

```
// sekvence rozvrhu místností z důvodu nepřekrývání
dvar sequence rozvrhMistnosti[m in rMistnosti] in
     all (p in PredmetyTrid) prirazeni[p][m];

forall(m in rMistnosti) noOverlap(rozvrhMistnosti[m]);
```

Školní rozvrh: model II.

Každý předmět je vyučován pro konkrétní třídu (skupinu žáků),

tj. pro každou třídu je zadána množina jejích předmětů

```
// sekvence rozvrhu tříd z důvodu nepřekrývání
dvar sequence rozvrhTridy[t in rTridy] in
  all (m in rMistnosti, p in PredmetyTrid : p.trida == t) prirazeni[p][m];
```

a tedy každá třída může mít vždy nejvýše jeden předmět v danou dobu

```
forall (t in rTridy)
       noOverlap(rozvrhTridy[t]);
```

Výuka předmětu musí probíhat bez přerušení

```
forall (p in PredmetyTrid)
       startOf(casy[p]) % Hodin <= (Hodin-p.trvani);
```

Školní rozvrh: optimalizace

Jednotlivé předměty by měly být do rozvrhu umístěny tak, aby výuka všech tříd skončila co nejdříve (např. třetí den dopoledne). Tj. **minimalizujeme součet koncových časů výuky posledních předmětů všech tříd**.

```
// čas konce posledního předmětu každé třídy
dexpr int maxTridy[t in rTridy] =
      max(p in PredmetyTrid : p.trida == t) endOf(casy[p]);
// součet koncových časů posledních hodin všech tříd
minimize sum(t in rTridy) maxTridy[t];
```

Rozvrhování vyšetření pacientů v nemocnici

Každý pacient musí absolvovat vysetření zadaného typu, která probíhají na různých pracovištích. Cílem řešení je pro každé vyšetření pacienta určit, ve *kterém čase* bude probíhat a *které pracoviště* bude vyšetření realizovat. Dále:

- Vyšetření jednoho pacienta se nesmí překrývat.
- Vyšetření na jednom pracovišti se nesmí překrývat.
- Každý typ vyšetření má stanovenu délku provádění a pracoviště, na kterých může být prováděno.
- Některý typ vyšetření musí proběhnout před realizací jiného typu vyšetření (pokud je pacient absolvuje oba). Jsou proto zadány dvojice typů vyšetření určující, které vyšetření musí být realizováno před zahájením dalšího vyšetření.

Vyšetření pacientů se závažnějším typem onemocnění by měla být dokončena dříve, aby mohli dříve nastoupit na operaci. Cílem je tedy minimalizovat vážený součet dokončení nejpozdějších vyšetření pacientů.

Jednoduché zadání

Příklad jednoduchého zadání s řešením:

- počet pacientů 2, počet typů vyšetření 2, počet pracovišť 2;
- doby trvání vyšetření daného typu: 1, 2;
- pracoviště pro vyšetření daného typu: 1:1,2, 2:1;
- typy vyšetření pro pacienty: 1:1,2, 2:1;
- priority pacientů: 1,10;
- precedence pro dané typy vyšetření: 1 před 2;
- dvě optimální řešení s kvalitou 13:
  - pacient, vyšetření, čas, pracoviště
  - 1,1,0,1; 1,2,1,2; 2,1,0,2
  - 1,1,0,2; 1,2,1,2; 2,1,0,1.

Vstupní data a typy

Konstanty a intervaly

```
int NbPacients = ...;
int NbMedicals = ...;
int NbRooms = ...;
range RPacients = 1..NbPacients;
range RMedicals = 1..NbMedicals;
range RRooms = 1..NbRooms;
```

Vyšetření: typ, místnosti

```
tuple Tmedical{
 int duration; {int} rooms; }
Tmedical Medicals[RMedicals] = ...;      // [<2,{1,3}>, <2,{2,4}>, <3,{1,2,3}>]
```

Pacient: typy vyšetření

```
tuple pMedical{
 int pacient; int medical; }
{pMedical} PMedicals = ...;              // {<1,1>,<1,2>,<2,2>,<2,3>,<3,1>, ...}
```

Vyšetření pacienta

```
tuple Tmedical{
 int duration; {int} rooms; }
Tmedical Medicals[RMedicals] = ...;
tuple pMedical{
 int pacient; int medical; }
{pMedical} PMedicals = ...;
```

Časy vyšetření pacienta

```
dvar interval times[pm in PMedicals] size Medicals[pm.medical].duration;
```

Nepřekrytí vyšetření pacienta

```
dvar sequence seqPacient[p in RPacients] in
                 all(pm in PMedicals: pm.pacient == p) times[pm];
forall (p in RPacients) noOverlap(seqPacient[p]);
```

Vyšetření na jednom pracovišti

```
dvar interval times[pm in PMedicals] size
Medicals[pm.medical].duration;
```

Proměnné pro výběr místnosti pro vyšetření pacienta

```
dvar interval assigned[PMedicals][RRooms] optional;
```

Na každém pracovišti maximálně jedno vyšetření

```
forall (pm in PMedicals)
        alternative (times[pm],
                     all (r in RRooms: r in Medicals[pm.medical].rooms)
                          assigned[pm][r]);

forall (r in RRooms)
        noOverlap(all (pm in PMedicals: r in Medicals[pm.medical].rooms)
                  assigned[pm][r]);
```

Precedence

```
tuple pMedical{
 int pacient; int medical; }
{pMedical} PMedicals = ...;
dvar interval times[pm in PMedicals] size Medicals[pm.medical].duration;
```

Typy a vstupní data

```
tuple Tprec{
 int before;
 int after; }
{Tprec} Prec = ...;
```

Omezení na precedence

```
forall (pr in Prec, pmB in PMedicals, pmA in PMedicals:
            pmB.pacient == pmA.pacient &&
            pmB.medical == pr.before &&
            pmA.medical == pr.after)
          endBeforeStart(times[pmB],times[pmA]);
```

Minimalizace vážených časů posledních vyšetření

Typy a vstupní data

```
int Priorities[RPacients] = ...;
```

Minimalizace

```
dexpr int pacientMax[p in RPacients]
              = max (pm in PMedicals: pm.pacient == p) endOf(times[pm]);

minimize sum (p in RPacients) Priorities[p]*pacientMax[p];
```

Zdroje, ze kterých průsvitky čerpají

V průsvitkách jsou použity obrázky a texty z uvedených zdrojů:

- Barták R., MFF UK, Praha. Průsvitky k přednášce Programování s omezujícími podmínkami http://kti.ms.mff.cuni.cz/~bartak/podminky/prednaska.html
- Apt K., CWI, Holandsko. Průsvitky ke knize Principles of Constraint Programming http://homepages.cwi.nl/~apt/books.html
- Dechter R., University of California, USA. Průsvitky ke knize Constraint processing http://www.ics.uci.edu/~dechter/books/materials.html
- Laborie P., Introduction to CP Optimizer for Scheduling. Tutoriál prezentovaný na 27th International Conference on Automated Planning and Scheduling, 2017. http://icaps17.icaps-conference.org/tutorials/#tut3