Globální podmínky

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`

- **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}`

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

- **Konzistence:** musí platit:

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

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

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

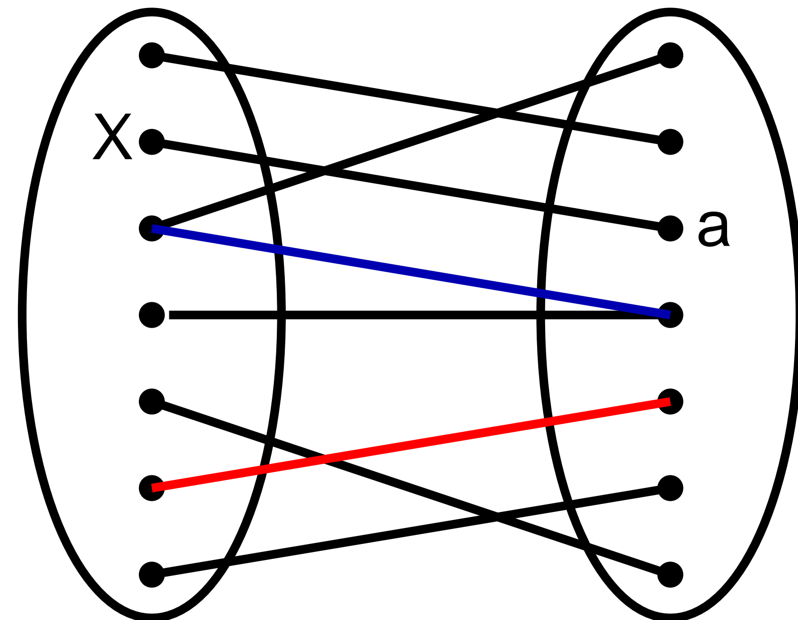
Párování v bipartitních grafech

- 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,

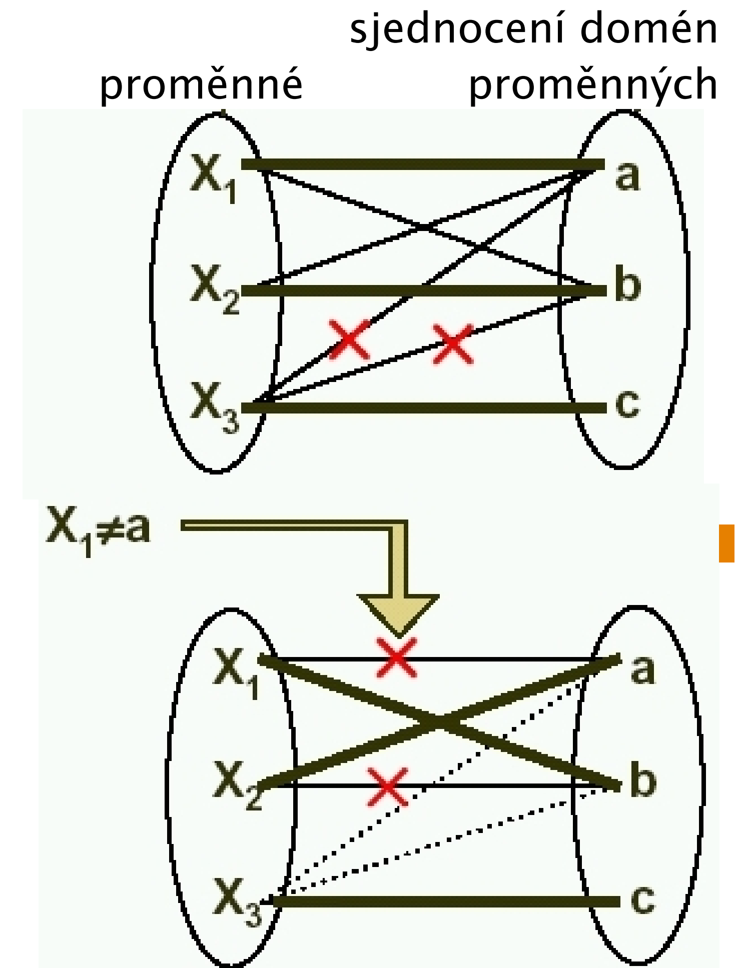
    které by měly společný vrchol
- **Maximální párování**
  - párování, které má maximální počet hran



- 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.

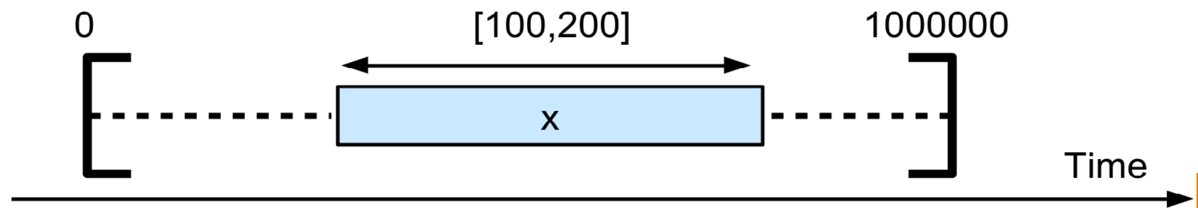
- Inicializace
  - vypočti maximální párování
  - odstraň všechny hrany, které nepatří do žádného maximálního párování
- Propagace zmenšené domény
  - odstraň odpovídající hrany
  - vypočti nové maximální párování
  - odstraň všechny hrany, které nepatří do žádného maximálního párování
- Algoritmus založen na doménové konzistenci
  - každé maximální párování definuje doménovou podporu
  - složitost $O(n^2k^2)$, $n\ldots$ počet proměnných, $k\ldots$ 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!

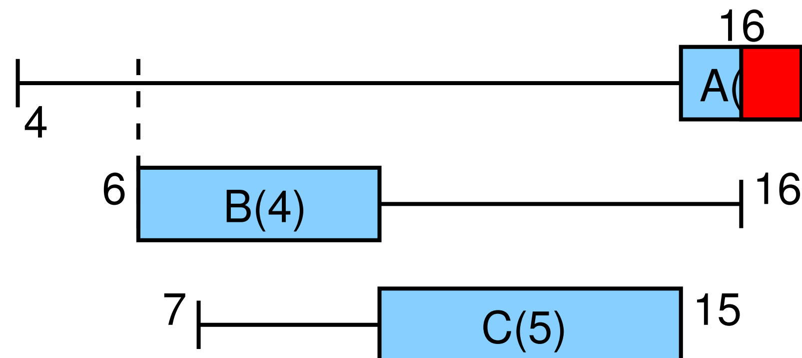
Omezení `noOverlap(p)`

- 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

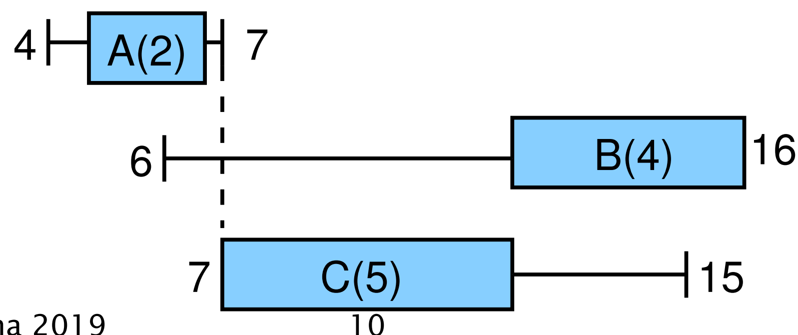
Disjunktivní rozvrhování: princip algoritmu

- 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 $x \neq \bot$

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)
```

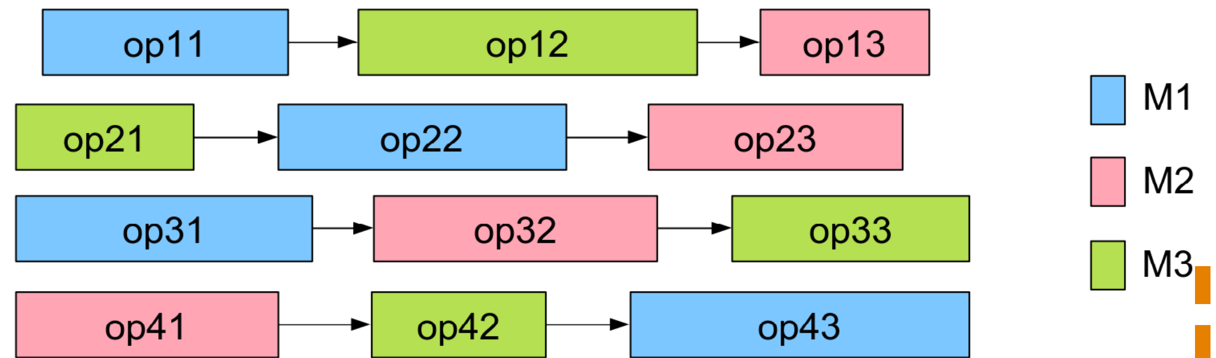
Příklad: minimalizace času dokončení poslední úlohy (tzv. makespanu)

```
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

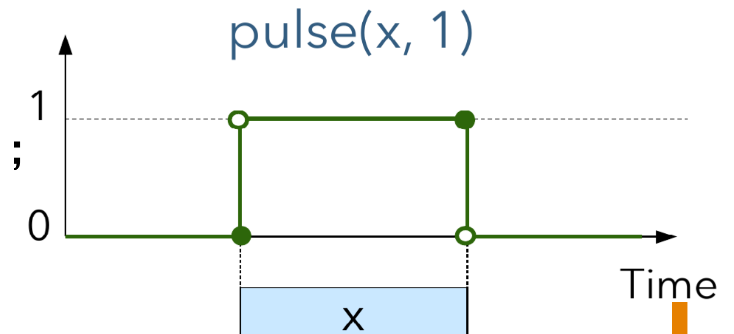
Hodnota **výrazu kumulativní funkce** reprezentuje vývoj kvantity v čase, která může být inkrementálně změněna (snížena nebo navýšena) intervalovými proměnnými.

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]);
```



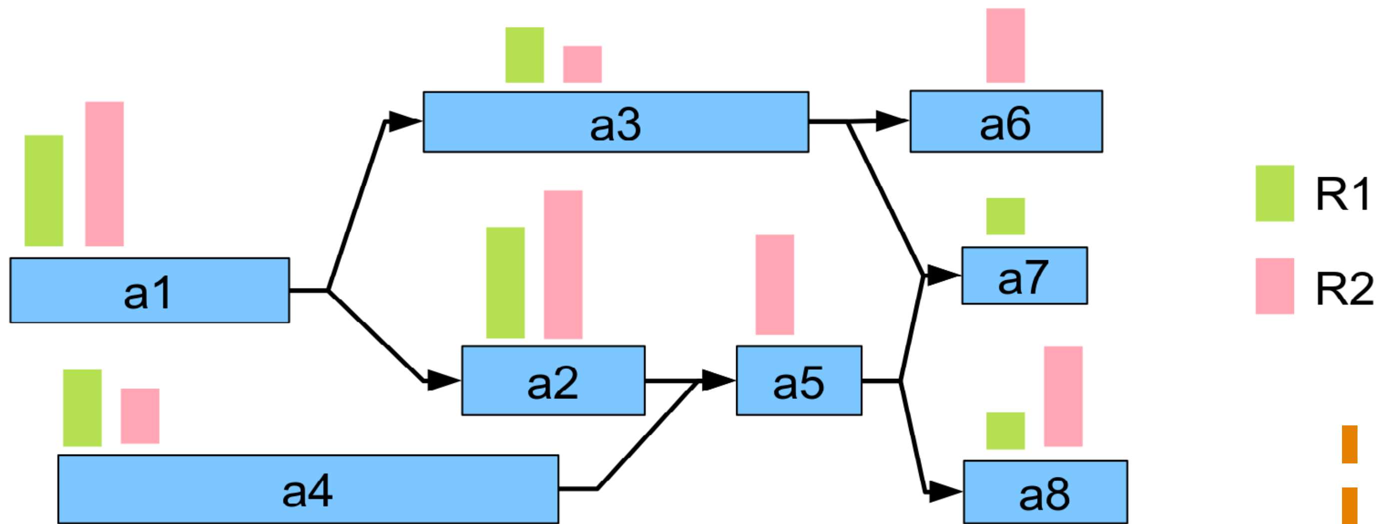
Omezení na výrazech kumulativní funkce

```
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 = ...;
```




```
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]);
```

Obecný konzistenční algoritmus

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 V_j ∈ Q
      for ∀ C takové, že V_j ∈ scope(C) do
          W := revise(V_j, C)
             // 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-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 $event(V_j) \in$ Q

  for $\forall C$ takové, že $V_j \in scope(C)$ 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 $\exists\ V_i \in W$ taková, že $D_i = \emptyset$ then return fail

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

  end Nonbinary-AC-3-with-Events