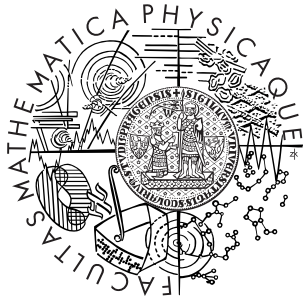
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jaroslav Mlejnek

Globální podmínky

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Doc. RNDr. Roman Barták, Ph. D.
Studijní program: Informatika, programování.

2008

Chtěl bych touto cestou poděkovat vedoucímu práce doc. Romanu Bartákovi za věnovaný čas, konzultace, cenné rady a připomínky a celkové směrování správným směrem.

Dále bych chtěl poděkovat prof. Peterovi Vojtášovi za laskavé zapůjčení knihy od K. R. Apta Principles of Constraint Programming [1], která dala základ první části práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 8. srpna 2008 Jaroslav Mlejnek

Název práce: Globální podmínky
Autor: Jaroslav Mlejnek
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí práce: Doc. RNDr. Roman Barták, Ph. D.
E-mail vedoucího: roman.bartak@mff.cuni.cz

Abstrakt. Globální omezující podmínky hrají klíčovou roli při řešení reálných kombinatorických problémů a to díky zabalení speciálních řešících technik pro konkrétní podproblém a následnou integraci těchto technik v rámci obecných technik splňování podmínek. Dnes existuje velké množství různých globálních podmínek vytvořených pro řešení konkrétních problémů nebo navržených na základě stávajících řešících technik. Tato práce podává snadno použitelný přehled existujících globálních podmínek s důrazem na jejich praktickou využitelnost.

Klíčová slova: programování s omezujícími podmínkami, globální podmínky, CSP

Title: Global constraints
Author: Jaroslav Mlejnek
Department: Department of Theoretical Computer Science
and Mathematical Logic
Supervisor: Doc. RNDr. Roman Barták, Ph. D.
Supervisor's e-mail address: roman.bartak@mff.cuni.cz

Abstract. Global constraints play a crucial role in solving real-life combinatorial problems thanks to encapsulation of dedicated solving techniques for particular sub-problems and integration of these techniques via general constraint satisfaction technology. There exist many global constraints developed for particular problems as well as derived from existing solving techniques. This work focuses on providing an easy-to-use survey of existing global constraints with particular emphasis on practical applicability.

Keywords: constraint programming, global constraints, constraint satisfaction problem

Obsah

Úvod

V každodenním životě se setkáváme s různými omezujícími podmínkami, ať už jsou k našemu prospěchu nebo naopak. Většinou řešíme problémy jakým způsobem skloubit práci s koníčky, jak jíst zdravě a přesto chutně, ale také jak vhodně rozplánovat výrobu v továrnách. Obecně tedy řešíme problém jak zvládat navzájem protichůdné požadavky.

Právě tím se zabývá disciplína teoretické informatiky *programování s omezujícími podmínkami*. Jejím cílem je efektivně získávat řešení takových úloh, které jsou klasickými algoritmy většinou neřešitelné a nebo velmi obtížně řešitelné, kde prohledávaný prostor je natolik rozsáhlý, že užitím těchto algoritmů by výpočet trval velice dlouho.

Navíc díky koncepci omezujících podmínek nabízí informatikovi možnost lépe se zaměřit na samotný problém a jeho výsledek, než na postup, který k němu vede. Umožňuje tak přemýšlet v úplně jiné rovině, než při klasickém programování.

Tento způsob programování byl úspěšně použit v několika oborech, jako jsou interaktivní grafické systémy, rozvrhování a plánování výroby, návrh a testování elektronických obvodů, zpracování přirozených jazyků, výpočet či zjednodušování algebraických rovnic nad různými tělesy.

Tato práce se bude zabývat v úlohách běžně se vyskytujícími omezujícími podmínkami, které nazýváme *globální podmínky*. Představí jejich kategorizovaný katalog a o každé z nich podá základní informace včetně příkladů, nastínění vnitřní struktury filtračních algoritmů a implementaci v dostupných CSP řešičích.

Text je organizován do kapitol. Kapitola 1 podává definice základních pojmů, se kterými programování s omezujícími podmínkami zachází, v odstavci 1.3 popisuje základní výpočetní framework a postupně se snaží dovést čtenáře k problematice globálních podmínek. Úvod do globálních podmínek a katalog vybraných globálních podmínek, které by měl každý informatik pracující v této oblasti znát, je k nalezení v kapitole 2.

Kapitola 1

Programování s omezujícími podmínkami

Jak již bylo naznačeno v úvodu, programování s omezujícími podmínkami se zabývá řešením problémů popsaných podmínkami. V této kapitole v odstavci 1.1 formálně zadefinujeme, co to vůbec podmínka je a co omezuje. V odstavci 1.2 popíšeme problém splňování omezujících podmínek a v odstavcích 1.3 a 1.4 nastíníme jeden z možných přístupů jeho řešení.

Odstavec 1.5 se zabývá tzv. propagací podmínek a propojuje tak obecné informace o programování s omezujícími podmínkami s problematikou globálních podmínek.

Text z této kapitoly je převážně převzat z knihy od K. Apta Principles of Constraint Programming [1] a odstavec o lokální konzistenci z knihy R. Dechterové Constraint Processing [12].

1.1 Podmínky

Ústředím pojmem programování s omezujícími podmínkami je *podmínka*.

Neformálně můžeme podmínku na několika proměnných definovat jako vztah mezi nimi. Říká nám, které kombinace hodnot proměnných jsou přípustné – vyhovují podmínce – a které ne.

Zaveďme nejprve pojem *proměnná* a *doména proměnné*:

Definice 1.1 (proměnná)**.** Objekt, který může nabývat nějaké hodnoty, nazveme *proměnná* a obvykle jej budeme značit malými písmeneny z konce abecedy, tj. x, y a z.

Definice 1.2 (doména proměnné)**.** *Doména D proměnné x* je neprázdná množina hodnot, kterých může proměnná x nabývat.

Skutečnost, že proměnná x nabývá nějaké hodnoty z domény D, budeme značit $x \in D$.

Definice 1.3 (podmínka)**.** Mějme konečnou neprázdnou množinu proměnných $Y = \{y_1, \ldots, y_k\}$ s doménami $D_1, \ldots, D_k$ pro $k \in \mathbb{N}$. Tedy každá proměnná y_i nabývá nějaké hodnoty z množiny D_i. Potom *podmínkou C na Y* rozumíme podmnožinu kartézského součinu $D_1 \times \ldots \times D_k$.

Podmínka tedy není nic jiného, než relace.

Pokud je $k = 1$, potom o podmínce říkáme, že je *unární*. Je-li $k = 2$, potom podmínku nazýváme *binární* atp.

1.2 Constraint Satisfaction Problem

Přidáváním podmínek postupně popíšeme celou úlohu – definujeme tzv. *Constraint Satisfaction Problem*[1], zkráceně CSP. Formálně:

Definice 1.4 (CSP)**.** Nechť $X = \{x_1, \ldots, x_n\}$, $n \in \mathbb{N}$, je konečná neprázdná množina proměnných s doménami $D_1, \ldots, D_n$. Potom *Constraint Satisfaction Problem*, zkráceně *CSP*, definujeme jako konečnou neprázdnou množinu podmínek $\mathcal{C}$, kde každá podmínka je na nějaké podmnožině X.

CSP zapisujeme jako $\langle \mathcal{C}; \mathcal{DE} \rangle$, kde $\mathcal{DE} := x_1 \in D_1,\ x_2 \in D_2,\ \ldots,\ x_n \in D_n$.

K CSP definujme ještě jeho řešení:

Definice 1.5 (řešení CSP)**.** Nechť $\langle \mathcal{C}; \mathcal{DE} \rangle$, kde $\mathcal{DE} := x_1 \in D_1, \ldots, x_n \in D_n$, je CSP. Řekneme, že n-tice $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ *splňuje* podmínku $C \in \mathcal{C}$ na proměnných $x_{i_1}, \ldots, x_{i_m}$, pokud $(d_{i_1}, \ldots, d_{i_m}) \in C$.

Řekneme, že n-tice $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ je *řešením* CSP, pokud

$$\forall C \in \mathcal{C} :\ (d_1, \ldots, d_n) \text{ splňuje } C.$$

Příkladem CSP může být například úloha najít taková $x \in \{1, 2, 3\}$ a $y \in \{2, 3\}$, pro která platí nerovnost $x < y$.

Tuto úlohu na CSP převedeme jednoduše, a to:

$$\langle x < y;\ x \in \{1, 2, 3\},\ y \in \{2, 3\} \rangle \text{, kde } n = 2$$

Řešením CSP jsou potom dvojice $(1, 2)$, $(1, 3)$ a $(2, 3)$.

[1] V české literatuře se Constraint Satisfaction Problem překládá jako problém splňování podmínek.

V uvedeném příkladě jsme nalezli všechna řešení, které CSP měl. To není vždy pravidlem, neboť někdy nás zajímá pouze jedno řešení, jindy jestli daná úloha vůbec nějaké řešení má. V optimalizačních úlohách se zase snažíme najít „nejlepší“ řešení, či optimální řešení na základě nějaké míry kvality řešení.

Proces, kdy ze zadané úlohy vytváříme CSP, nazýváme *modelování*. Je zřejmé, že k jedné reálné úloze existuje více CSP reprezentací. Jak vybrat tu správnou? Začněme od průběhu řešení CSP. Pro vyřešení zvolené reprezentace můžeme použít buď

- doménově specifických metod[2] nebo
- obecných metod[3],

popř. kombinaci obou dvou.

Doménově specifické metody jsou obvykle speciální algoritmy pro známé problémy. Typickými příklady jsou: algoritmus řešení soustavy lineárních rovnic, řešič lineárního programování, implementace unifikačních algoritmů, automatických dokazovačů vět atp. Na druhé straně obecné metody spočívají v redukci prohledávaného prostoru řešení úlohy (pomocí tzv. propagačních algoritmů) spolu se speciálními prohledávacími technikami.

Pokud tedy máme více CSP reprezentací dané úlohy, pak se snažíme vybrat takovou, která by šla řešit nějakou doménově specifickou metodou než metodami obecnými. Doménově specifické metody jsou sice omezeny na řešení nějakého konkrétního podproblému, ale díky znalosti přesného algoritmu, který přímočaře vede k řešení, neriskujeme ve většině případů dlouhý běh prohledávání prostoru obecnými metodami. Typickým příkladem může být právě nalezení řešení soustavy lineárních rovnic. Pro tuto úlohu existují vhodné algoritmy lineární algebry a není tedy důvod proč použít metod obecných.

Ve skutečnosti se v programování s omezujícími podmínkami snažíme hledat účinné doménově specifické metody, které by mohly být použity místo obecných metod, a snažíme se je transparentně včlenit do základního výpočetního frameworku. Tento framework obvykle podporuje

- doménově specifické metody jako specializované moduly,
- obecné metody jako různé vestavěné algoritmy pro propagování podmínek a různé prohledávací techniky.

V dalším textu je pro vysvětlení základních principů řešení CSP používán pojem *ekvivalence dvou CS problémů*. Zadefinujme si jej:

[2] angl. domain specific methods
[3] angl. general methods

Definice 1.6 (ekvivalence dvou CS problémů)**.** Nechť $\mathcal{P}_1$ a $\mathcal{P}_2$ jsou dva CS problémy nad stejnou množinou proměnných. Řekneme, že $\mathcal{P}_1$ a $\mathcal{P}_2$ jsou *ekvivalentní*, pokud jejich množiny řešení jsou si rovny.

Příklad dvou ekvivalentních CS problémů:

$$\langle 3x - 5y = 4;\ x \in \{0, \ldots, 9\}, y \in \{1, \ldots, 8\}\rangle$$

a

$$\langle 3x - 5y = 4;\ x \in \{3, \ldots, 8\}, y \in \{1, \ldots, 4\}\rangle\,.$$

Oba CS problémy mají stejná řešení, a to $(3, 1)$ a $(8, 4)$. Tato řešení jsou zároveň jedinými řešeními každého CSP.

1.3 Základní framework – kostra

Máme zadanou úlohu. Z ní ve fázi modelování vytvoříme CSP a nyní se budeme snažit CSP vyřešit.

Řešení CSP je ukázáno na rekurzivní proceduře `Solve` (viz zdrojový kód 1.1). Procedura na vstupu dostane CSP s nímž provádí následující operace:

1. v proceduře `Preprocess` CSP předzpracuje do vhodné syntaktické formy pro další výpočet,
2. pomocí algoritmů propagace podmínek se pokusí v proceduře `Constraint Propagation` „zjednodušit“ předzpracovaný CSP,
3. pokud ještě nejsme spokojeni s výsledkem (funkce `Happy` není „šťastná“) a následující operace jsou proveditelné (funkce `Atomic` neoznačila CSP za dále nerozložitelný), je „zjednodušený“ CSP rozdělen na menší části procedurou `Split` a pro každou část (tedy nově vzniklý CSP) je znovu zavolána procedura `Solve` z procedury `Proceed by Cases`.

Procedura `Proceed by Cases` tedy vede k rekurzivnímu volání procedury `Solve` pro každý nově formovaný CSP procedurou `Split`.

Proměnná *continue* je v proceduře `Solve` lokální a je tedy jedinečná pro každé zavolání procedury.

Každé proceduře a funkci se nyní budeme věnovat odděleně v následujících odstavcích a kapitolách.

```
procedure Solve;
var continue : boolean;
continue := TRUE;
while continue and not Happy do
    Preprocess;
    Constraint Propagation;
    if not Happy then
        if Atomic then
            continue := FALSE;
        else
            Split;
            Proceed by Cases;
        end if
    end if
end loop
```

Zdroj. kód 1.1: základní framework.

1.3.1 Předzpracování CSP

Předzpracování CSP zajišťuje procedura `Preprocess`. Předzpracováním máme na mysli převod CSP do požadované syntaktické formy. Výsledný CSP by potom měl být ekvivalentní s původním CSP.

Co to tedy předzpracování je? Například budeme chtít, aby v každé podmínce vystupovala každá proměnná nejvýše jednou. V tomto případě procedura `Preprocess` prochází jednotlivé podmínky a převádí je do požadovaného tvaru. Pro ilustraci mějme podmínku

$$ax^7 + bx^5y + cy^{10} = 0.$$

Proměnná x zde vystupuje dvakrát (v prvním a v druhém členu). Procedura `Preprocess` proto v tomto případě zavede pomocnou proměnnou z a výše uvedenou podmínku nahradí dvěma novými:

$$ax^7 + z + cy^{10} = 0$$

$$bx^5y = z.$$

Obvykle je procedura `Preprocess` volána pouze jednou na začátku procedury `Solve`. V našem algoritmu je ale vložena na začátek každého cyklu, tedy ihned po zavolání funkce `Split`[4]. Je to z toho důvodu, že funkce `Split` může občas vygenerovat podmínky, které nemají požadovaný tvar.

[4]Pokud zavolání procedury `Preprocess` ihned po proceduře `Split` není zřejmé: funkce `Split` rozdělí CSP na „menší“ CS problémy. Procedura `Proceed by Cases` na každý „menší“ CSP zavolá proceduru `Solve` a prvním příkazem procedury `Solve` je procedura `Prepocess`.

1.3.2 Atomicita CSP

Funkce `Atomic` testuje, zda aktuální CSP je atomický, tedy dále nerozložitelný na jiné „menší“ ekvivalentní CS problémy.

Obvykle „být atomický“ znamená, že domény proměnných jsou buď jednoprvkové nebo prázdné množiny. CSP může být atomický také v případě, že další hledání „dále po cestě“ není potřeba. Toto nastává například ve chvíli, kdy už máme řešení a nebo pokud hledáme optimální řešení a odtud jej lze dopočítat přímo.

1.3.3 Vyřešení CSP – dosažení cíle

Funkce `Happy` nás informuje o tom, zda bylo dosaženo cíle. Co znamená cíl, záleží na úloze. Těmi nejběžnějšími jsou:

- bylo nalezeno řešení CSP,
- byla nalezena všechna řešení CSP,
- bylo nalezeno takové „částečné řešení“, že všechna další řešení lze odtud vygenerovat přímo. Tato možnost je užitečná v případech, kdy existuje nekonečně mnoho řešení.
- Byla nalezena *nekonzistence*, tj. takové místo na cestě, odkud víme, že řešení dále na této cestě neleží,
- na základě objektivní funkce (míry kvality řešení) bylo nalezeno optimální řešení,
- na základě objektivní funkce byla nalezena všechna optimální řešení,
- v případě podmínek na reálných číslech byly všechny domény proměnných zredukovány na intervaly menší než nějaké pevné ϵ.

1.4 Prohledávací technika

Průběh výpočtu se díky proceduře `Split` větví. Kořenem stromu je CSP, který je na počátku namodelován programátorem; uzly ve stromě jsou procedurou `Split` dělené CS problémy a listy jsou nalezená řešení či nekonzistence. Jak se bude strom větvit není pro žádnou úlohu pevně dáno a závisí na mnoha faktorech: pro jaký CSP řešič jsme se rozhodli a jak je implementován, máme-li úlohu s mnoha triviálními podmínkami a malými doménami proměnných nebo naopak atp.

Na tvaru stromu výpočtu a jeho hloubce se nejvíce projevuje zvolená prohledávací technika a jakého chceme dosáhnout cíle. V případě, že chceme znát

všechna řešení a použijeme algoritmus *backtracking*, bude strom výpočtu úplný. Na druhou stranu, v optimalizačních úlohách při zvolené prohledávací technice *branch and bound* dochází k postupnému úplnému větvení stromu pouze do nalezení prvního řešení a dále díky získaným údajům už mohou být některé větve „zahazovány".

Více k prohledávacím technikám backtracking a branch and bound je k nalezení v odstavci 1.4.2.

1.4.1 Dělení na podproblémy

Jakmile skončí algoritmus propagace podmínek, nebylo ještě dosaženo cíle a CSP není atomický, přichází na řadu rozdělení aktuálního CSP na dva nebo více CS problémy, jejichž sjednocení je ekvivalentní s aktuálním CSP. Toto dělení zajišťuje procedura `Split`. Aktuální CSP hraje roli posledního uzlu v prozatím postaveném stromě výpočtu (tj. pro tuto chvíli roli listu) a dělení představuje jeho syny.

Dělení je možné provést dvěma způsoby: rozdělením domény proměnné (proměnných) nebo rozdělením podmínky (podmínek). Aktuální CSP je poté nahrazen dvěma nebo více CS problémy, které se od aktuálního liší pouze novými doménami proměnných, resp. novými podmínkami.

V příštích odstavcích bude znak „|" znamenat dělení, údaje nad vodorovnou čarou původní stav a údaje pod vodorovnou čarou stav po dělení.

Doménové dělení výčtem prvků

Nechť doména D je konečná a obsahuje alespoň dva prvky. Potom můžeme použít následující pravidlo:

$$\frac{x \in D}{x \in \{a\} \mid x \in D - \{a\}},$$

kde $a \in D$.

V prvním případě uvažujeme, že proměnná x nabývá hodnoty a a v druhém případě, že hodnota proměnné x leží v doméně bez prvku a. Toto pravidlo tedy převádí CSP

$$\langle \mathcal{C};\ \mathcal{DE}, x \in D \rangle$$

na dva CS problémy:

$$\langle \mathcal{C}_1;\ \mathcal{DE}, x \in \{a\} \rangle$$
$$\text{a } \langle \mathcal{C}_2;\ \mathcal{DE}, x \in D - \{a\} \rangle,$$

kde $\mathcal{C}_1$ obsahuje podmínky z $\mathcal{C}$ s nově upravenými doménami (doména x se stala jednoprvkovou množinou $\{a\}$) a podobně $\mathcal{C}_2$. Znovupoužití tohoto pravidla postupně vede k binarizaci stromu výpočtu.

Doménové dělení označováním

Doménu budeme dělit stejným způsobem jako v předchozím příkladě, jen tentokrát na $|D|$ částí:

$$\frac{x \in \{a_1, \ldots, a_k\}}{x \in \{a_1\} \mid \ldots \mid x \in \{a_k\}}.$$

Každé aplikování tohoto pravidla vede k nahrazení aktuálního CSP k CS problémy, kde k je velikost domény proměnné x. Opakované použití tohoto pravidla tedy postupně vede k libovolně konečně rozvětveným stromům.

Doménové dělení půlením

Doménové dělení půlením lze aplikovat na neprázdný reálný interval, označme jej $\langle a, b\rangle$, a to pomocí následujícího pravidla:

$$\frac{x \in \langle a, b\rangle}{x \in \left\langle a, \frac{a+b}{2}\right\rangle \mid x \in \left\langle \frac{a+b}{2}, b\right\rangle}.$$

Neprázdnosti domén je docíleno překrýváním obou intervalů. Triviálně lze toto pravidlo také rozšířit na neprázdné celočíselné intervaly.

Dělení disjunktních podmínek

Předpokládejme, že podmínka je Booleovsky disjunktní. Základem této disjunkce mohou být libovolné „podpodmínky“, ty poté nazýváme *disjunktní podmínky*.

Příklad této disjunkce může být podmínka

$$\texttt{Zacatek}[udalost_1] + \texttt{DobaTrvani}[udalost_1] \leq \texttt{Zacatek}[udalost_2] \vee$$

$$\vee\ \texttt{Zacatek}[udalost_2] + \texttt{DobaTrvani}[udalost_2] \leq \texttt{Zacatek}[udalost_1],$$

která se typicky objevuje v rozvrhovacích problémech. Znamená, zda $udalost_1$ je naplánovaná před $udalost_2$ nebo naopak. Pravidlo pro dělení je tedy přímočaré:

$$\frac{C_1 \vee C_2}{C_1 \mid C_2}.$$

S každou disjunkcí zacházíme odděleně, ostatní podmínky zůstávají nezměněné.

Podmínky C_1 a C_2 by měly být exkluzivní, aby nedocházelo ke zbytečnému redundantnímu prohledávání.

Podmínky ve „smíšené“ formě

Ideou dělení podmínek ve „smíšené“ formě je jejich nahrazení syntakticky jednodušími podmínkami, se kterými už lze pracovat přímo. Mějme například podínku $|p(\bar{x})| = a$, kde $p(\bar{x})$ je reálný polynom a a je nezáporné reálné číslo.

Dělení můžeme provést podle náslujícího pravidla:

$$\frac{|p(\bar{x})| = a}{p(\bar{x}) = a \mid p(\bar{x}) = -a}$$

Tímto jsme vlastně původní podmínku přepsali jako disjunktní podmínku

$$p(\bar{x}) = a \ \vee \ p(\bar{x}) = -a$$

a použili pravidlo pro dělení disjunktních podmínek.

Podmínky, které mohou být přepsány do disjunktní formy, ale není třeba dělit dle pravidel pro disjunktní podmínky. Například podmínka typu $|x - y| = a$ může být převedena na podmínku $x - y = z$, kde z je nová proměnná s doménou $\{-a, a\}$.

—

Procedura `Split` vybírá, jaká pravidla pro dělení domén proměnných a podmínek a v jakém pořadí se budou používat. Pokud máme zadaný CSP, není vždy jednoznačné, jaký postup dělení zvolit. Z tohoto důvodu zavádíme tzv. *heuristiky* – nápovědy pro proceduru `Split`, jakou proměnou, jakou hodnotu proměnné či jakou podmínku zvolit pro dělení.

Typickými heuristikami jsou

- výběr proměnné, která se objevuje v nejvíce podmínkách (tzv. most constrained variable) nebo
- z domény proměnné reprezentované celočíselným intervalem vybrat prostřední hodnotu.

1.4.2 Zpracovávání jednotlivých větví výpočtu

Procedura `Split` rozdělí CSP na dva nebo více nových CS problémů. Tyto potom vstupují do procedury `Proceed by Cases`. Pořadí, v jakém se nové CS problémy zpracovávají, záleží na *prohledávací technice*.

Jak již bylo zmíněno, opakovaným použitím procedury `Split` dochází ke stavbě stromu výpočtu. Účelem procedury `Proceed by Cases` je procházet tímto stromem v předem definovaném pořadí a pokud je to zapotřebí (např. v případě hledání optimálního řešení), nastavovat řídící proměnné podle nově nashromážděných informací.

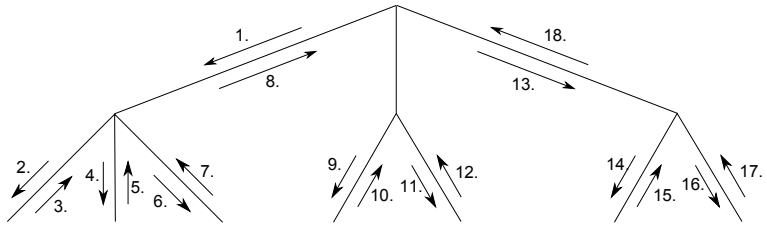
Mezi nejznámější prohledávací techniky patří *backtracking* a pro hledání optimálního řešení *branch and bound*. Obě si teď neformálně popíšeme.

Backtracking

Mějme konečně rozvětvený strom, algoritmus backtracking začíná v kořeni tohoto stromu a přejde na jeho prvního potomka.

Nyní je v novém uzlu a opět přejde na prvního potomka tohoto uzlu. Celý proces se opakuje, dokud novým uzlem není list.

Je-li nyní v listu, vrací se zpět k otci a pokud je k dispozici další (ještě neprozkoumaný) potomek, přejde do něj a pokračuje v prohledávání. Pokud už není k dispozici žádný neprozkoumaný potomek, přejde k otci.



Obr. 1.2: průchod stromem algoritmem backtracking.

Algoritmus končí, pokud se řízení vrátí zpět do kořene a všechny uzly ve stromě byly navštíveny.

Uzly stromu jsou CS problémy a listy pak CS problémy, které jsou buď vyřešeny nebo o nich víme, že řešení nemají. Obvykle CS problémy v listech mají domény proměnných jednoprvkové, takže list představuje pouze jedno řešení. O tom, co za CSP je list a nebo není, rozhodují funkce `Atomic` a `Happy`.

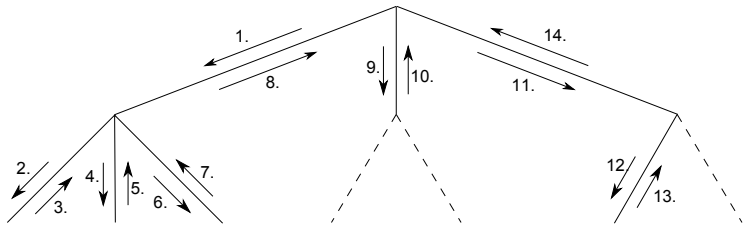
Pokud chceme nalézt pouze jedno řešení CSP, algoritmus backtracking zastaví stavbu stromu hned, jakmile je vygenerován první listový vyřešený CSP. Pokud chceme znát všechna řešení, algoritmus neskončí dříve, než vygeneruje všechny listy. Chceme-li naopak vědět, zda CSP je neřešitelný, algoritmus zastaví buď

hned, co nalezne první vyřešený CSP (tj. nalezne protipříklad), nebo zastaví opět, až vygeneruje všechny listy.

Branch and bound

Algoritmus branch and bound je modifikací backtrackingu, která bere v úvahu hodnotu tzv. *objektivní funkce*.

Chtějme najít řešení s maximální hodnotou objektivní funkce, potom během prohledávání si pamatujme prozatímní nejvyšší dosaženou hodnotu objektivní funkce v proměnné *bound*. Tato proměnná je inicializována na $-\infty$ a její hodnota je upravována vždy, když je vygenerován vyřešený CSP (lépe, každý CSP funkcí `Atomic` označený za atomický) s vyšší hodnotou objektivní funkce. Tento CSP je zároveň také uložen jako (nové) výsledné řešení.



Obr. 1.3: průchod stromem algoritmem branch and bound.

Obvykle je objektivní funkce používána spolu s tzv. *heuristickou funkcí*, která přiřazuje reálné číslo každému (tedy nejen vyřešenému) CSP a je tak možné při stavbě stromu zamezit prohledávání některých větví.

Pro heuristickou funkci platí následující dvě podmínky:

1. Pokud CSP β je přímým potomkem CSP α ve stromě výpočtu, potom pro heuristickou funkci h platí $h(\beta) \leq h(\alpha)$.
2. Pokud je CSP α funkcí `Atomic` označen za atomický, potom pro hodnotu heuristické funkce h a objektivní funkce *obj* platí $obj(\alpha) \leq h(\alpha)$.

První podmínka říká, že čím jsme níž ve stromě výpočtu, tím se heuristická funkce „zpřesňuje“. Druhá podmínka po heuristické funkci požaduje „přeceňování“ objektivní funkce.

Hodnota heuristické funkce h je během výpočtu neustále kontrolována s hodnotou proměnné *bound* a pokud se dostaneme do uzlu, pro který platí $h < bound$, už další jeho potomky nemusíme prohledávat, neboť v nich bychom lepší řešení stejně nenašli.

1.5 Propagace podmínek

Procedura `Constraint Propagation` se snaží nahradit daný CSP ekvivalentním „jednodušším“ CSP. Ideou je, aby nahrazení bylo efektivní, výhodné a další prohledávání probíhalo v menším prostoru. Co znamená „jednodušší“ CSP záleží na konkrétní aplikaci. Ve většině případů jde o zmenšení domén proměnných či zjednodušení podmínek.

Propagace podmínek probíhá opakovanou redukcí domén proměnných a/nebo podmínek, přičemž je stále zachovávána ekvivalence CSP. Jak probíhá doménová redukce si ukážeme na následujícím příkladě.

Mějme podmínku C, vyberme z ní proměnou x. Na doméně D proměnné x proveďme následující operaci:

z domény D proměnné x odstraňme všechny hodnoty, které nemohou být řešením podmínky C.

Smyslem tedy je odebrat z domény proměnné takové hodnoty, o kterých už je v tuto chvíli jasné, že jich v žádném řešení CSP proměnná nenabývá. Tuto operaci nazýváme *projekcí C na x*.

Uveďme příklad. Představme si křížovku znázorněnou na obrázku 1.4, kterou bychom měli vyplnit slovy z tohoto seznamu:

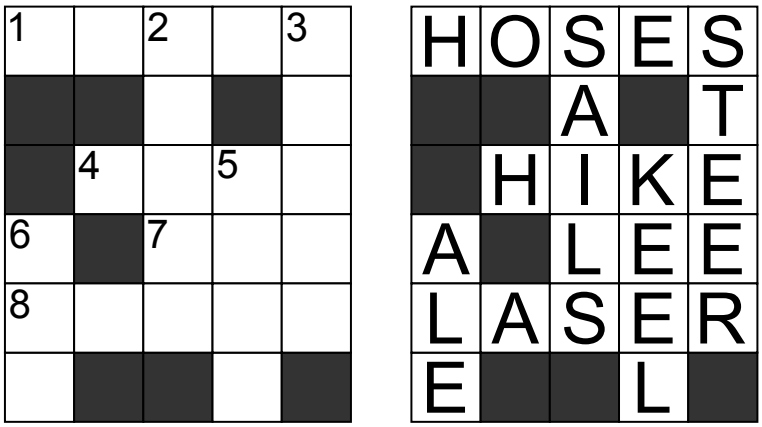
- HOSES, LASER, SAILS, SHEET, STEER,
- HEEL, HIKE, KEEL, KNOT, LINE,
- AFT, ALE, EEL, LEE, TIE.

CSP můžeme namodelovat např. takto: mějme $i \in \{1, \ldots, 8\}$ pozic a ke každé pozici bude proměnná x_i představovat jedno doplněné slovo z výše uvedeného seznamu. Doména každé proměnné x_i se sestává ze slov, která mají správnou délku, tedy proměnná x_6 nabývá hodnot z domény $\{\text{AFT}, \text{ALE}, \text{EEL}, \text{LEE}, \text{TIE}\}$, neboť u pozice 6 směrem dolů bude v křížovce zřejmě trojpísmenné slovo.

Aby křížovka byla opravdu křížovkou, definujme ještě podmínky, že slova mezi sebou sdílí určitá písmena. Například podmínka pro přípustná slova na pozicích 1 a 2, která sdílí třetí písmeno a první písmeno, vypadá takto:

$$C_{1,2} = \{(\text{HOSES}, \text{SAILS}), (\text{HOSES}, \text{SHEET}), (\text{HOSES}, \text{STEER}),$$
$$(\text{LASER}, \text{SAILS}), (\text{LASER}, \text{SHEET}), (\text{LASER}, \text{STEER})\}.$$

Ostatních jedenáct podmínek ($C_{1,3}$, $C_{2,4}$, $C_{2,7}$, $C_{2,8}$, $C_{3,4}$, $C_{3,7}$, $C_{3,8}$, $C_{4,5}$, $C_{5,7}$, $C_{5,8}$ a $C_{6,8}$) zapíšeme zcela analogicky.



Obr. 1.4: nevyplněná křížovka s označením pozic a vyplněná křížovka.

Ukázku propagace doménovou redukcí proveďme na doméně proměnné $x_1 \in \{\text{HOSES}, \text{LASER}, \text{SAILS}, \text{SHEET}, \text{STEER}\}$, která prozatím obsahuje všechna pětipísmenná slova. Z podmínky $C_{1,2}$ vidíme, že díky křížení můžeme vypustit hodnoty SAILS, SHEET a STEER. Zredukovali jsme tedy doménu proměnné x_1 na množinu $\{\text{HOSES}, \text{LASER}\}$.

Stejným způsobem se můžeme podívat i na ostatní podmínky a tak např. díky podmínce $C_{1,3}$ omezit doménu proměnné x_3 z množiny všech pětipísmenných slov na tříprvkovou množinu $\{\text{SAILS}, \text{SHEET}, \text{STEER}\}$.

Správné řešení křížovky je k nalezení na obrázku 1.4.

1.5.1 Lokální konzistence

Na příkladě s křížovkou byl vidět postup propagačního algoritmu. Čeho se ale snažil dosáhnout, kdy má skončit? Propagační algoritmy se snaží o tzv. *lokální konzistenci*.

Lokální konzistence existuje několik druhů, nejprve však neformálně definujme několik pojmů. Prvním z nich je *binarizace podmínek*, kdy z n-árních podmínek vytvoříme podmínky binární, např. zavedením pomocných proměnných – více o binarizaci podmínek se lze dočíst v [4].

Binarizované podmínky můžeme znázornit grafem a to tak, že vrcholy představují jednotlivé proměnné, které v podmínkách vystupují, a hrany mezi vrcholy představují binární podmínky mezi příslušnými proměnnými. Z tohoto grafového modelu mj. vychází názvy jednotlivých konzistenčních technik.

V příkladě s křížovkou jsme ukázali redukci CSP redukcí domén proměnných, což bylo definováno takto:

pro každou podmínku C a každou proměnnou x vystupující v C každá hodnota z domény D proměnné x se vyskytuje v nějakém prvku z C.

Této vlastnosti se říká *hyper-hranová konzistence*[5]. Formálněji: mějme podmínku C na proměnných $x_1, \ldots, x_k$ s příslušnými doménami $D_1, \ldots, D_k$ a tedy $C \subseteq D_1 \times \ldots \times D_k$. Podmínka C je *hyper-hranově konzistentní*, pokud $\forall i \in \{1, \ldots, k\}$ a $\forall a \in D_i \;\; \exists d = (d_1, \ldots, d_k) \in C$ takové, že $a = d_i$.

V případě binárních podmínek se jedná o *hranovou konzistenci*[6]. Vedle hranové konzistence ještě existuje jednodušší *vrcholová konzistence*[7] na unárních podmínkách.

CSP je hranově (hyper-hranově, vrcholově, ...) konzistentní, pokud všechny podmínky jsou hranově (hyper-hranově, vrcholově, ...) konzistentní.

Jako příklad hranově konzistentního CSP uveďme

$$\langle x < y, y < z, x < z;\; x \in \{0, \ldots, 5\}, y \in \{1, \ldots, 7\}, z \in \{3, \ldots, 8\} \rangle .$$

Uvedený CSP zřejmě splňuje definici hranové konzistence, např. u podmínky $x < y$ proměnná y a hodnota 1 z domény y leží v uspořádané dvojici $(0, 1)$ vyhovující podmínce $x < y$.

Na druhou stanu CSP

$$\langle x < y, y < z, x < z;\; x \in \{0, \ldots, 5\}, y \in \{0, \ldots, 7\}, z \in \{3, \ldots, 8\} \rangle$$

není hranově konzistentní, neboť pro hodnotu 0 proměnné y neexistuje žádná hodnota x taková, aby platila podmínka $x < y$.

Je také důležité si uvědomit, že obecně lokálně konzistentní CSP nemusí být (globálně) konzistentní (mít řešení). Například CSP

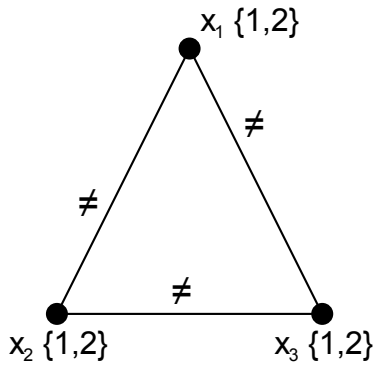
$$\langle x \neq y, y \neq z, x \neq z;\; x \in \{1, 2\}, y \in \{1, 2\}, z \in \{1, 2\} \rangle$$

znázorněný grafem na obrázku 1.5 je zřejmě lokálně konzistentní, ale není konzistentní.

1.5.2 Hranová konzistence a filtrace

V následujícím odstavci se podívejme blíže na hranovou konzistenci[8].

[5] angl. hyper-arc consistency
[6] angl. arc consistency
[7] angl. node consistency
[8] angl. arc-consistency



Obr. 1.5: hranově konzistentní CSP, ale nekonzistentní CSP (nemá řešení).

Definice 1.7 (hranová konzistence)**.** Nechť $\langle \mathcal{C}; \mathcal{DE} \rangle$ je CSP a každá podmínka $C \in \mathcal{C}$ je binární. Potom proměnná x_i je *hranově konzistentní* s proměnnou x_j právě tehdy, když pro každou hodnotu $a_i \in D_i$ z domény proměnné x_i existuje hodnota $a_j \in D_j$ z domény proměnné x_j taková, že $(a_i, a_j) \in C_{i,j}$, kde $C_{i,j} \in \mathcal{C}$ je binární omezující podmínka na proměnných x_i a x_j.

CSP $\langle \mathcal{C}; \mathcal{DE} \rangle$ je hranově konzistentní, pokud všechny proměnné jsou se všemi proměnnými ve všech podmínkách hranově konzistentní.

V současné době je v CSP řešičích nejčastěji implementován právě algoritmus hranové konzistence, konkrétně pak jeho varianta s názvem AC-3. Zdrojový kód algoritmu viz 1.6.

```
procedure AC-3;
var queue : set of pairs (x_i, x_j);
for each (x_i, x_j) in C_{i,j} ∈ C
    queue := queue ∪ {(x_i, x_j), (x_j, x_i)}
end for
while queue ≠ ∅
    Select and Delete (x_i, x_j) from queue
    Revise (x_i, x_j)
    if Revise (x_i, x_j) causes a change in D_i then
        queue := queue ∪ {(x_k, x_i) | k ≠ i, k ≠ j}
    end if
end while
```

Zdroj. kód 1.6: algoritmus hranové konzistence AC-3.

Vysvětleme si jednotlivé kroky algoritmu. Algoritmus na vstupu dostává binarizované CSP $\langle \mathcal{C}; \mathcal{DE} \rangle$ a vrací $\langle \mathcal{C}; \mathcal{DE}' \rangle$, které je s $\langle \mathcal{C}; \mathcal{DE} \rangle$ ekvivalentní, ale domény proměnných mohou mít méně prvků, než v $\langle \mathcal{C}; \mathcal{DE} \rangle$.

V algoritmu vystupuje pomocná proměnná *queue*, která představuje frontu hran v grafu binarizovaného CSP. První `for` cyklus tuto frontu naplní všemi

hranami, které se v grafu vyskytují. Vztah mezi proměnnými nemusí být vždy symetrický, proto do fronty přidáváme obě dvě varianty (x_i, x_j) a (x_j, x_i).

`While` cyklus probíhá dokud máme ve frontě nějakou neprobranou hranu. Procedura `Select and Delete` vybere první dvojici ve frontě a odstraní ji ze začátku fronty. Následuje *filtrační* procedura `Revise`, která se pokusí z domény D_i proměnné x_i odstranit všechny nekonzistentní hodnoty.

Pokud se podařilo některé hodnoty z domény D_i eliminovat (došlo ke změně domény D_i), jsou všechny hrany, ve kterých vystupuje proměnná x_i, znovu přidány na konec fronty – samozřejmě bez aktuálně probrané hrany. Lze si všimnout, že proměnná x_i je přesunuta ve dvojici na „pasivní“ místo. Toto nastává z toho důvodu, že nová doména proměnné může v ostatních podmínkách způsobit nekonzistenci a je proto třeba ostatní proměnné a podmínky znovu zkontrolovat.

Jak již bylo popsáno výše, algoritmus končí, je-li fronta hran prázdná; jinými slovy, pokud už nejde zredukovat žádnou z domén proměnných.

```
procedure Revise (x_i, x_j);
for each a_i ∈ D_i
    if there is no a_j ∈ D_j such that (a_i, a_j) ∈ C_{i,j} then
        delete a_i from D_i
    end if
end for
```

Zdroj. kód 1.7: procedura `Revise`.

Stěžejní částí celé propagace podmínek je právě procedura `Revise`, jejíž obsah v obecné formě je k nalezení ve zdrojovém kódu 1.7. Prochází postupně všechny hodnoty a_i z domény D_i proměnné x_i a dívá se zda pro každou z nich existuje alespoň jedna hodnota a_j z domény D_j proměnné x_j taková, aby platila podmínka $C_{i,j}$ na těchto proměnných. Pokud taková hodnota a_j existuje, nic se neděje; pokud jsme ale žádnou nenašli, tak hodnota a_i může být z domény D_i proměnné x_i odstraněna.

Zde se poprvé setkáváme s globálními podmínkami, neboť právě obsah procedury `Revise` je přizpůsoben potřebám dané globální podmínky. Kód procedury `Revise`, tedy filtrační algoritmus, je také jedním z charakteristik každé globální podmínky.

V globálních podmínkách se často budeme setkávat spíše s hyper-hranovou konzistencí, která k výše uvedeným algoritmům přidává funkčnost pro nebinární podmínky. Hyper-hranová konzistence se také někdy nazývá *zobecněná hranová konzistence*[9], zkráceně *GAC*.

[9] angl. generalized arc-consistency

1.5.3 Konzistence okrajů

Při řešení některých úloh se může stát, že hranová konzistence bude mít příliš velkou režii. To nastává hlavně v případech, kdy doménami proměnných jsou rozsáhlé celočíselné intervaly. V takovou chvíli slevíme z poměrně striktních požadavků hranové konzistence a místo vyžadování hranové konzistence pro každou hodnotu a_i z domény D_i budeme požadovat hranovou konzistenci pouze pro horní a dolní okraj intervalu domény proměnné. Této modifikaci říkáme *konzistence okrajů*[10].

Toto ulehčení sice nedokáže zredukovat domény proměnných stejně dobře jako plnohodnotná hranová konzistence (díváme se pouze na hraniční body intervalů), avšak z hlediska rychlosti a např. paměťové náročnosti řešiče jde o efektivní zásah.

[10] angl. bounds-consistency

Kapitola 2

Globální podmínky

Pojem *globální podmínka* označuje nějakou komplexnější, komplikovanější a zároveň v úlohách často se vyskytující podmínku. Formální a všeobecně přijatá definice globální podmínky neexistuje, ale o podmínce lze říci, že je globální, pokud:

- jde o podmínku na proměnlivém počtu proměnných,
- lze ji vyjádřit konjunkcí jiných jednodušších podmínek,
- existuje podmínku efektivně propagující filtrační algoritmus.

Typickým příkladem globální podmínky je podmínka `all different` na proměnných $x_1, \ldots, x_n$, která platí, pokud všechny proměnné $x_1, \ldots, x_n$ mají různou hodnotu. Tuto podmínku můžeme ekvivalentně popsat $\frac{1}{2}\ n\ (n-1)$ nerovnostmi[1]. Příklad pro tři proměnné x, y a z:

$$\texttt{all\_different}(x, y, z) \ \Leftrightarrow\ (x \neq y\ \&\ x \neq z\ \&\ y \neq z)\,.$$

Globální podmínky ve smyslu zkratek za často se vyskytující úlohy mj. zjednodušují modelování a CSP řešičům poskytují lepší pohled na řešený problém.

Hlavním důvodem existence globálních podmínek je jejich mnohem efektivnější propagování. Takováto podmínka totiž přesně vymezuje daný problém a je proto možné přímo na míru vytvořit silný filtrační algoritmus. Výhody silného filtračního algoritmu jsou zřejmé: vyšší efektivita redukce, prohledávání menšího prostoru a celkové zrychlení výpočtu.

Algoritmy filtrace jsou dvojího typu: filtrační algoritmy provádějící *úplnou filtraci*[2] a *částečnou filtraci*[3]. Algoritmy provádějící úplnou filtraci odstraní ze všech

[1] Výraz $\frac{1}{2}\, n\, (n-1)$ vyjadřuje počet dvojic na n prvkové množině.
[2] angl. complete filtering
[3] angl. partial filtering

domén všech proměnných vystupujících v podmínce všechny pro řešení nepoužitelné hodnoty. Algoritmy pro částečnou filtraci odstraní pouze některé pro řešení nepoužitelné hodnoty.

Výhodu existence speciálních filtračních algoritmů si nyní ukážeme na podmínce `all different`.

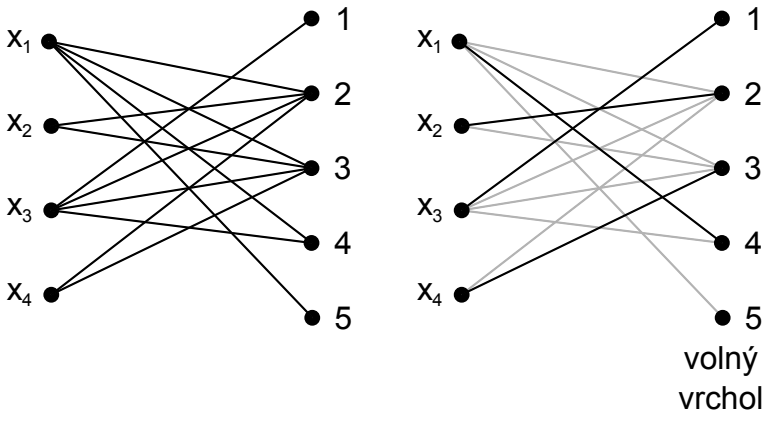
Pracujme s následujícími proměnnými x_1, x_2, x_3, x_4 a jejich doménami $D_1 = \{2,3,4,5\}$, $D_2 = \{2,3\}$, $D_3 = \{1,2,3,4\}$ a $D_4 \in \{2,3\}$. Chceme pro podmínku `all_different`(x_1, x_2, x_3, x_4) docílit hranové konzistence.

Algoritmus je založen na párování v grafech a hledání komponent souvislosti. V následujících odstavcích bude krok po kroku přesně rozebrán a projev každého kroku bude ukázán na výše uvedeném příkladě.

Vstupem do algoritmu je n-tice proměnných $x_1 \ldots x_n$, v našem případě čtveřice x_1, x_2, x_3, x_4.

1. krok – sestavení bipartitního grafu a nalezení maximálního párování

Sestavme neorientovaný bipartitní graf $G = (V_1 \cup V_2, E)$, kde vrcholy jedné partity jsou proměnné, tj. $V_1 = \{x_1, x_2, x_3, x_4\}$, a vrcholy druhé partity jsou prvky ze sjednocení domén proměnných, tj. $V_2 = D_1 \cup D_2 \cup D_3 \cup D_4 = \{1,2,3,4,5\}$. Hrany grafu spojují proměnné s hodnotami z jejich domén, tj. $(x_i, a_j) \in E \Leftrightarrow a_j \in D_i$. Bipartitní graf našeho příkladu je zobrazen na obrázku 2.1 vlevo.



Obr. 2.1: bipartitní graf pro proměnné x_1, x_2, x_3, x_4 a maximální párování.

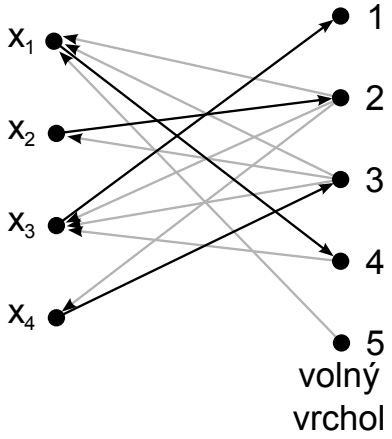
Máme-li hotový neorientovaný bipartitní graf, pokusíme se v něm nalézt *maximální párovaní*. Párování v grafu $G = (V, E)$ je podmnožina hran $M \subseteq E$ taková, že žádné dvě hrany nesdílí jeden vrchol. Nalézt maximální párování znamená maximalizovat velikost množiny $|M|$. Vhodným algoritmem může být např. algoritmus Hopcroft-Karp [25]. K párování dodefinujme ještě tzv. *volné vrcholy*. Volnými vrcholy nazvěme všechny vrcholy z V_2, které nejsou použity žádnou hranou z M. V našem případě jde o vrchol 5.

Příklad nalezeného maximálního párování je na obrázku 2.1 vpravo.

2. krok – zavedení orientace, označení „použitých“ hran

Nacházíme se ve fázi, kdy máme zkonstruován neorientovaný bipartitní graf a v něm nalezené maximální párování.

Zaveďme orientaci v grafu G následujícím způsobem: hrany patřící do párování M nechť směřují od partity proměnných do partity hodnot, ostatní hrany nechť směřují od partity hodnot do partity proměnných. Formálně tedy definujeme nový orientovaný graf $G' = (V_1 \cup V_2, E')$, kde $E' = \{(x_i, a_j) \mid (x_i, a_j) \in M\} \cup \{(a_j, x_i) \mid (x_i, a_j) \in E \backslash M\}$. Orientovaný bipartitní graf pro náš příklad je zobrazen na obrázku 2.2.



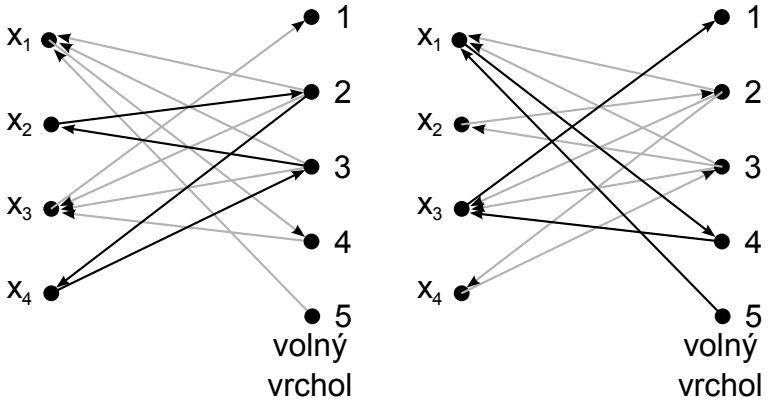
Obr. 2.2: orientovaný bipartitní graf pro proměnné x_1, x_2, x_3, x_4.

Nyní v orientovaném grafu G' spusťme algoritmus pro nalezení silně souvislých komponent. Silně souvislá komponenta grafu je taková komponenta, ve které z každého vrcholu v_i vede cesta do každého vrcholu v_j. Algoritmus pro nalezení silně souvislých komponent je popsán např. v článku [39].

Čeho jsme vlastně dosáhli? Podařilo se nám v grafu G' nalézt všechny kružnice, které navíc mají sudý počet hran. Hrany, které jsou v těchto kružnicích obsaženy označme příznakem „použitá“. V našem příkladu se vyskytuje právě jedna kružnice (viz obr. 2.3 vlevo): $\{(x_4, 3), (3, x_2), (x_2, 2), (2, x_4)\}$.

Dále prohledáváním do šířky nalezněme všechny cesty se sudým počtem hran a začínající ve volných vrcholech. Takovou cestu máme v našem grafu také právě jednu (viz obr. 2.3 vpravo): $\{(5, x_1), (x_1, 4), (4, x_3), (x_3, 1)\}$. Opět všechny hrany ležící na všech nalezených sudých cestách označme příznakem „použitá“.

Pokud se v grafu vyskytují ještě nějaké hrany, které patří do párování M a nejsou obsaženy v žádné kružnici ani v žádné sudé cestě, označme je také příznakem „použitá“.



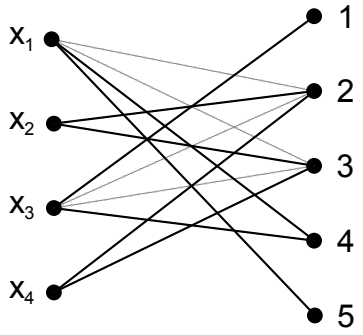
Obr. 2.3: kružnice a cesta v bipartním grafu x_1, x_2, x_3, x_4.

3. krok – redukce domén

Máme graf G' s hranami označenými příznakem „použitá“ a s hranami bez označení.

Nyní procházejme v grafu G' všechny hrany (x_i, a_j) resp. (a_j, x_i), které nemají nastaven příznak „použitá“, a odstraňujme z domény D_i proměnné x_i hodnoty a_j ($D_i := D_i \backslash \{a_j\}$).

Tím jsme docílili hranové konzistence pro podmínku `all different` a filtrační algoritmus končí. Proměnné x_1, x_2, x_3, x_4 nyní nabývají nových domén $D_1 = \{4, 5\}$, $D_2 = \{2, 3\}$, $D_3 = \{1, 4\}$ a $D_4 = \{2, 3\}$. Výsledný graf po filtraci je k vidění na obrázku 2.4.



Obr. 2.4: graf pro proměnné x_1, x_2, x_3, x_4 po filtraci. Tečkované hrany jsou hrany bez příznaku „použitá“.

Zlepšení – zapamatování filtrace do „příště“

Během řešení CSP nedochází k redukci domén pouze u této jedné podmínky, ale u všech podmínek. Pokud dojde ke změně domény filtračním algoritmem jiné

podmínky, algoritmus propagace podmínek z odstavce 1.5.2 opět zavolá filtrační algoritmus této podmínky.

Zlepšení filtračního algoritmu podmínky `all different` spočívá v zapamatování si sestavených grafů a nalezeného párování, které v dalších voláních filtračního algoritmu pouze upravujeme.

Pseudokód procedury `Revise`

Nyní zakončeme popis filtračního algoritmu podmínky `all different` pseudokódem procedury `Revise` (bez využití pamatování do „příště“, zdroj. kód 2.5):

```
procedure Revise (x_1, ..., x_n);
var G : graph;
    M : set of edges;
build bipartite graph G from x_1, ..., x_n and D_1 ∪ ... ∪ D_n;
find maximum matching M;
make G directed according to M;
SCC and BFS;
for each non-used edge (x_i, a_j) or (a_j, x_i) in G
    delete a_j from D_i
end for
```

Zdroj. kód 2.5: procedura `Revise` pro podmínku `all different`.

—

Jak bylo na filtračním algoritmu podmínky `all different` vidět, de-facto se jedná o malý program. Využívali jsme hlavně grafových algoritmů, ale není výjimkou postavení filtračních algoritmů na lineárním programování, dynamickém programování či konečných automatech.

Předvedený filtrační algoritmus měl polynomiální asymptotickou složitost. To také nemusí být pravidlem, například filtrační algoritmus pro ověření existence Hamiltonovské kružnice v grafu je NP-úplný.

Představili jsme si definici globální podmínky, smysl, proč je studujeme a ukázali jsme si práci filtračního algoritmu. Nyní přistupme k samotnému katalogu globálních podmínek.

Úvod této kapitoly byl převzat z knihy [34] z kapitoly 7 sepsané autory Willemem Janem V. Hoeve a Irit Katrielovou.

2.1 Organizace katalogu

Katalog globálních podmínek je uveden několika základními podmínkami, které jsou běžně využívané při modelování CSP a dostupné téměř ve všech CSP řešičích. Jsou zařazeny v kategorii „obecné podmínky“, viz odstavec 2.2.

V další části katalog obsahuje necelých čtyřicet globální podmínek rozdělených do kategorií inspirovaných článkem [10].

Ke každé globální podmínce je uveden její název, odkaz či odkazy na články s úplným zněním podmínky, neformální slovní popis a jednoznačný formálnější popis. Dále jsou ke každé podmínce prezentovány dva příklady, kdy podmínka platí, kdy podmínka neplatí a motivační příklad či oblast použití, má-li to pro danou podmínku smysl. Též bylo snahou ke každé podmínce najít a ve zkratce popsat její filtrační algoritmus a zmapovat dostupnost v existujících CSP řešičích.

CSP řešiče, ve kterých byla zjišťována dostupnost globálních podmínek, jsou tyto: B-Prolog[4] [43], Comet[5] [22], The CSP Library[6] [26], Disolver[7] [20], Ecl^{i}ps^e [8] [11], Gecode[9] [18], GNU Prolog[10] [13], Choco[11] [27], Koalog[12] [29], Mozart[13] [15] a SICStus Prolog[14] [36].

Protože podmínky mají proměnlivý počet argumentů, abychom předešli nedorozumění, bududeme spolu logicky související posloupnosti hodnot nebo proměnných uzavírat do závorek $\langle$ a $\rangle$.

Přejděme k samotnému katalogu.

[4] `http://www.probp.com/`
[5] `http://www.comet-online.org/`
[6] `http://www.hulubei.net/tudor/csp/`
[7] `http://research.microsoft.com/ youssefh/DisolverWeb/Disolver.html`
[8] `http://www.eclipse-clp.org/`
[9] `htpp://www.gecode.org/`
[10] `http://www.gprolog.org/`
[11] `http://choco.sourceforge.net/`
[12] `http://www.koalog.com/php/jcs.php`
[13] `http://www.mozart-oz.org/`
[14] `http://www.sics.se/isl/sicstuswww/site/index.html`

2.2 Obecné podmínky

V tomto odstavci jsou zmíněné podmínky, které stojí na pomezí mezi (primitivními) podmínkami a globálními podmínkami. Mají proměnlivý počet parametrů, ale nejsou pro ně psány žádné výjimečné filtrační algoritmy a jsou dostupné v téměř každém řešiči CSP.

První podmínkou zařazenou do této kategorie je podmínka `domain` (2.2.1), která omezuje rozsah hodnot, kterých proměnné mohou nabývat. Dále je představena podmínka `element` (2.2.2), která se ptá na hodnotu jedné konkrétní proměnné v n-tici proměnných – představuje takový „index do pole“.

Kapitolu uzavírá použitím velmi podobná trojice podmínek `at least` (2.2.3), `at most` (2.2.4) a `exactly` (2.2.5), které omezují počet výskytů jedné hodnoty v n-tici proměnných.

2.2.1 `domain`

Zdroj: [6].

Podmínka `domain` určuje rozsah hodnot proměnných. Jejími parametry jsou n-tice proměnných $x_1, \ldots, x_n$ a čísla l a u, přičemž platí $l \leq u$. Podmínka je splněna, pokud všechny hodnoty proměnných $x_1, \ldots, x_n$ jsou větší nebo rovny l a menší nebo rovny h.

$$\texttt{domain}(x_1, \ldots, x_n, l, u) \;\Leftrightarrow\; \forall i \in \{1, \ldots, n\} : l \leq x_i \leq u.$$

Příklad:

Nechť $n = 4$ a proměnné $x_1, \ldots, x_4$ nabývají hodnot $2, 4, 2, 5$. Nechť dolní mez je rovna $l = 1$ a horní mez je rovna $u = 5$. Potom podmínka

$$\texttt{domain}(\langle 2, 4, 2, 5\rangle, 1, 5)$$

platí, neboť každá hodnota x_i leží v rozmezí $1 \leq x_i \leq 5$.

Implementováno v CSP řešičích:

B-Prolog `domain`, Gecode `dom`, SICStus Prolog `domain`.

2.2.2 `element`

Zdroj: [6].

Mějme n-tici proměnných $x_1, \ldots, x_n$, index z množiny $i \in \{1, \ldots, n\}$ a hodnotu h. Podmínka `element` platí, pokud v n-tici $x_1, \ldots, x_n$ je na i-tém místě hodnota h, resp. hodnota proměnné x_i je h.

$$\texttt{element}(x_1, \ldots, x_n, i, h) \;\Leftrightarrow\; x_i = h.$$

Příklad:

Mějme čtveřici proměnných s těmito hodnotami: $x_1 = 3$, $x_2 = 2$, $x_3 = 4$ a $x_4 = 1$. Je-li index $i = 3$, potom podmínka

$$\texttt{element}(\langle 3, 2, 4, 1 \rangle, 3, h)$$

platí pro hodnotu $h = 4$.

Implementováno v CSP řešičích:

B-Prolog `element`, Disolver `Element`, Gecode `element`, GNU Prolog `fd_element`, Choco `nth`, Koalog `Element`, Mozart `element`, SICStus Prolog `element`.

2.2.3 `at least`

Zdroj: [6].

Mějme proměnné $x_1, \ldots, x_n$, počet proměnných $k \leq n$ a hodnotu h. Podmínka `at least` platí, pokud alespoň k proměnných má hodnotu h.

$$\texttt{at\_least}(x_1, \ldots, x_n, k, h) \;\Leftrightarrow\; k \leq |\{i \in \{1, \ldots, n\} : x_i = h\}|.$$

Příklad:

Nechť počet proměnných je $n = 4$ a nabývají hodnot $x_1 = 7$, $x_2 = 18$, $x_3 = 10$ a $x_4 = 2$. Nechť $k = 2$ a hodnota $h = 3$. Potom podmínka

$$\texttt{at\_least}(\langle 7, 18, 10, 2\rangle, 2, 3)$$

platí, protože počet hodnot, které jsou větší nebo rovny 3 je více jak 2. Konkrétně $x_1 = 7 > 3 = h$, $x_2 = 18 > 3 = h$ a $x_3 = 10 > 3 = h$ a tedy $k = 2 \leq 3$.

Implementováno v CSP řešičích:

B-Prolog `atleast`, Comet `Atleast`, Disolver `AtLeast`, Gecode `atleast`, GNU Prolog `fd_atleast`, Koalog `Atleast`, Mozart `atLeast`.

2.2.4 `at most`

Zdroj: [6].

Podmínka `at most` je „opakem“ podmínky `at least`, neboť platí, pokud při zadání n proměnných $x_1, \ldots, x_n$, počtu proměnných $k \leq n$ a hodnoty h, nejvýše k proměnných má hodnotu h.

$$\texttt{at\_most}(x_1, \ldots, x_n, k, h) \Leftrightarrow |\{i \in \{1, \ldots, n\} : x_i = h\}| \leq k.$$

Příklad:

Nechť počet proměnných je $n = 4$ a nabývají hodnot $x_1 = 7$, $x_2 = 18$, $x_3 = 10$ a $x_4 = 2$. Nechť $k = 2$ a hodnota $h = 3$. Potom podmínka

$$\texttt{at\_most}(\langle 7, 18, 10, 2\rangle, 2, 3)$$

platí, protože hodnota, která je menší nebo rovna 3, je pouze jedna ($x_4 = 2 \leq 3 = h$) a tedy je počet menší než $k = 2$.

Implementováno v CSP řešičích:

B-Prolog `atmost`, Comet `Atmost`, Disolver `AtMost`, Gecode `atmost`, GNU Prolog `fd_atmost`, Koalog `Atmost`, Mozart `atMost`.

2.2.5 `exactly`

Zdroj: [6].

Podmínka `exactly` patří do rodiny podmínek `at most` a `at least`. Tentokrát se ptáme, zda se mezi proměnnými $x_1, \ldots, x_n$ vyskytuje právě k proměnných, které mají hodnotu h.

$$\texttt{exactly}(x_1, \ldots, x_n, k, h) \;\Leftrightarrow\; |\{i \in \{1, \ldots, n\} : x_i = h\}| = k.$$

Příklad:

Mějme tuto čtveřici proměnných: $x_1 = 3$, $x_2 = 7$, $x_3 = 3$ a $x_4 = 5$. Nechť $h = 3$ a $k = 2$, potom podmínka

$$\texttt{exactly}(\langle 3, 7, 3, 5\rangle, 2, 3)$$

zřejmě platí. Proměnné nabývající hodnoty 3 se vyskytují ve čtveřici právě dvě (x_1 a x_3).

Implementováno v CSP řešičích:

B-Prolog `exactly`, Comet `Exactly`, Gecode `exactly`, GNU Prolog `fd_exactly`, Koalog `Exactly`, Mozart `exactly`.

2.3 Podmínky omezující počet výskytů hodnot

S podmínkami, které omezují počet výskytů hodnot, se při modelování CSP setkáváme velmi často a mnohdy hrají klíčovou roli.

Kategorie je uvozena podmínkou `count` (2.3.1), která je z hlediska použitelnosti velmi přímočará, neboť počet výskytů hodnot přímo porovnává se zadaným parametrem. Druhou v pořadí je podmínka `among` (2.3.1), která počítá pouze některé hodnoty proměnných, a za ní následuje podmínka `between min max` (2.3.3) započítávající hodnoty, které leží v daném intervalu.

Podmínka `balance` (2.3.4) se od výše uvedených odlišuje, neboť omezuje rozdíl minimálního a maximálního počtu výskytu jedné hodnoty.

V této kategorii nesmí chybět Global Cardinality Constraint, zkráceně `gcc` (2.3.5), která omezuje počty výskytů velmi obecně a informatikovi tak při modelování nabízí značnou flexibilitu.

Další velmi zajímavou globální podmínkou je podmínka `inflexion` (2.3.6), která se dívá do posloupnosti proměnných, hledá v nich tzv. hory a údolí a omezuje až jejich výskyty.

Kategorie je zakončena podmínkou `connect points` (2.3.7), která nabízí třídimenzionální pohled na posloupnost proměnných a jejich výskyty hodnot.

2.3.1 `count`

Zdroj: [6].

Mějme proměnné $x_1, \ldots, x_n$, hodnotu h, relaci $\rho \in \{=, \neq, <, \leq, >, \geq\}$ a číslo $k \leq n$. Podmínka `count` platí, pokud počet proměnných, které nabývají hodnoty h, je v relaci s číslem k.

$$\texttt{count}(x_1, \ldots, x_n, h, k\rho) \;\Leftrightarrow\; \rho(|\{i \in \{1, \ldots, n\} : x_i = h\}|, k)$$

Použití:

Podmínka `count` je svým použitím velmi jednoduchá. Můžeme jí například využít v případě, kdy nemáme k dispozici vhodné globální podmínky pro alokaci zdrojů a potřebujeme nějakým způsobem omezit kapacitu zdrojů.

Příklad:

Mějme čtveřici proměnných $x_1, \ldots, x_4$ představující čtveřici úkolů, které se musí za směnu provést. Mějme k dispozici 3 stroje, každý o výkonu maximálně $k = 2$ úkoly za směnu. Přiřazení úkolu i ke stroji představuje hodnota proměnné x_i, která tak nabývá hodnoty z množiny $\{1, 2, 3\}$.

Nechť proměnné $x_1, \ldots, x_4$ nabývají po řadě hodnot $1, 1, 3, 2$. Omezme kapacitu prvního stroje, tj. $h = 1$ a $\rho = \texttt{'} \leq \texttt{'}$, podmínkou

$$\texttt{count}(\langle 1, 1, 3, 2\rangle, 1, 2, \leq),$$

která v tomto případě bude platit, neboť hodnoty $h = 1$ nabývají dvě proměné x_1 a x_2. Maximální výrobní kapacita prvního stroje tedy nebude překročena.

Příklad:

Opusťme stroje a přiřazování úkolů. Nechť proměnné $x_1, \ldots, x_4$ po řadě nabývají hodnot $1, 1, 3, 2$. Ptejme se na přesný počet proměnných, které se rovnají hodnotě 3. Tj. $h = 3$ a $\rho = \texttt{'} = \texttt{'}$. Podmínka

$$\texttt{count}(\langle 1, 1, 3, 2\rangle, 3, 2, =)$$

s parametrem $k = 2$ platit nebude, neboť hodnoty 3 nabývá pouze jediná proměnná, a to x_3.

Implementováno v CSP řešičích:

B-Prolog `count`, Gecode `count`, Koalog `Count`.

2.3.2 among

Zdroj: [9].

Podmínka among má tři parametry, a to proměnné $x_1, \ldots, x_n$, počet proměnných $k \leq n$ a množinu hodnot D. Podmínka platí, pokud právě k z n proměnných má hodnotu ležící v D.

$$\texttt{among}(x_1, \ldots, x_n, k, D) \;\Leftrightarrow\; |\{i \in \{1, \ldots, n\} : x_i \in D\}| = k.$$

Motivace:

Vhodným motivačním příkladem pro podmínku among může být problém rozvrhování směn sester v nemocnici, kdy každá sestra by během jednoho týdne (tedy sedmi dní) měla mít nejvýše dvě noční směny. Podmínka among potom vyjadřuje vztah „právě 2 z 7“.

Příklad:

Nechť $k = 2$ je maximální počet nočních směn a počet proměnných je $n = 7$ (počet dní v týdnu). Proměnné mohou nabývat hodnot z množiny $\{0, 4, 8, 12, 16, 20\}$, která představuje časy začátku osmihodinových směn v hodinách. Nechť množina D obsahuje ty hodiny, kdy pracovní směna zasahuje do noci, tj. $D = \{16, 20, 0, 4\}$.

Nechť zdravotní sestra má prozatím přidělené tyto směny: $x_1 = 8$, $x_2 = 8$, $x_3 = 20$, $x_4 = 12$, $x_5 = 16$, $x_6 = 8$ a $x_7 = 12$. Potom podmínka

$$\texttt{among}(\langle 8, 8, 20, 12, 16, 8, 12\rangle, 2, \{16, 20, 0, 4\})$$

platí, neboť $20 \in \{16, 20, 0, 4\}$ a $16 \in \{16, 20, 0, 4\}$, což jsou dvě hodnoty a tedy $2 = k$.

Příklad:

Uvažujme stejný model jako v předchozím příkladě. A nechť má zdravotní sestra prozatím přidělené tyto směny: $x_1 = 8$, $x_2 = 12$, $x_3 = 8$, $x_4 = 12$, $x_5 = 20$, $x_6 = 8$ a $x_7 = 8$. Potom podmínka

$$\texttt{among}(\langle 8, 12, 8, 12, 20, 8, 8\rangle, 2, \{16, 20, 0, 4\})$$

neplatí, neboť z hodnot 8, 12, 8, 12, 20, 8 a 8 pouze $20 \in \{16, 20, 0, 4\}$ a tedy počet hodnot $= 1 < 2 = k$.

Filtrační algoritmus:

Zadaný vektor proměnných $x_1, \ldots, x_n$ je převeden na binární vektor a využit algoritmus filtrace pro sumu hodnot tohoto vektoru. Polynomiální filtrační algoritmus dociluje hranové konzistence. Úplné změní algoritmu je k nalezení v [9].

2.3.3 `between min max`

Zdroj: [6].

Podmínka `between min max` má jako své parametry proměnné $x_1, \ldots, x_n$ a proměnnou x_0. Platí, pokud mezi proměnnými $x_1, \ldots, x_n$ je alespoň jedna s hodnotou menší nebo rovnou x_0 a alespoň jedna s hodnotou větší nebo rovnou x_0.

$$\texttt{between\_min\_max}(x_0, x_1, \ldots, x_n) \;\Leftrightarrow\; \exists i, j \in \{1, \ldots, n\} : \; x_i \leq x_0 \leq x_j.$$

Příklad:

Nechť $n = 4$, $x_1 = 3$, $x_2 = 4$, $x_3 = 6$, $x_4 = 2$ a $x_0 = 5$. Potom podmínka

$$\texttt{between\_min\_max}(5, \langle 3, 4, 6, 2 \rangle)$$

platí, neboť v množině hodnot $\{3, 4, 6, 2\}$ se nachází např. hodnota 2, která je menší než $x_0 = 5$; a zároveň se v této množině nachází hodnota 6, která větší než $x_0 = 5$.

Příklad:

Nechť posloupnost proměnných zůstává stejná jako v přechozím příkladě. Ptejme se nyní, zda podmínka `between min max` platí pro hodnotu $x_0 = 7$? Podmínka

$$\texttt{between\_min\_max}(7, \langle 3, 4, 6, 2 \rangle)$$

neplatí, protože všechny hodnoty proměnných $x_1, \ldots, x_4$ jsou menší než hodnota proměnné $x_0 = 7$.

2.3.4 `balance`

Zdroj: [6].

Podmínka `balance` na vstupu dostává proměnné $x_1, \ldots, x_n$ a číslo $k \leq n$. Při průchodu proměnnými $x_1, \ldots, x_n$ zjišťuje, kolik jich má stejnou hodnotu a pamatuje si jejich maximální a minimální počet výskytů. Podmínka `balance` platí, pokud rozdíl mezi minimálním a maximálním počtem výskytů jedné hodnoty je roven parametru k.

Definujme funkci $occurs : \{x_1, \ldots, x_n\} \longrightarrow \mathbb{N}$ předpisem

$$occurs(x) = |\{i \in \{1, \ldots, n\} : x = x_i\}|,$$

která vyjadřuje počet proměnných v n-tici $x_1, \ldots, x_n$, které nabývají stejné hodnoty jako proměnná x.

Dále definujme čísla $min = \min\{occurs(x_i) \mid i \in \{1, \ldots, n\}\}$ jako minimum z množiny počtu výskytů shodných hodnot a $max = \max\{occurs(x_i) \mid i \in \{1, \ldots, n\}\}$ jako maximum této množiny. Potom

$$\texttt{balance}(x_1, \ldots, x_n, k) \Leftrightarrow |max - min| = k.$$

Motivace:

Motivací pro vytvoření a používání podmínky `balance` je, jak už její název napovídá, vyvažování. Lze si všimnout, že nezáleží na počtu různých hodnot, které proměnné nabývají. Podmínka `balance` si všímá pouze několikanásobných výskytů.

Příklad:

Nechť pětice proměnných $x_1, \ldots, x_5$ nabývá po řadě těchto hodnot $3, 1, 7, 1, 1$. Potom podmínka

$$\texttt{balance}(\langle 3, 1, 7, 1, 1 \rangle, 2)$$

pro parametr $k = 2$ platí. Proč? Počítejme počet výskytů jednotlivých hodnot mezi proměnnými $x_1, \ldots, x_5$. Hodnoty 1 nabývají proměnné x_2, x_4 a x_5, tzn. minimální a maximální počet výskytů hodnot je v tuto chvíli shodně roven 3.

Počítejme počet výskytů hodnoty 3 mezi proměnnými $x_1, \ldots, x_5$. Této hodnoty nabývá jediná proměnná, a to x_1. Minimální počet výskytů je tedy roven 1, maximální zůstává 3.

Spočtěme počet výskytů pro poslední hodnotu 7. I ta se mezi proměnnými $x_1, \ldots, x_5$ objevuje pouze jednou, a to u proměnné x_3. Zůstává tedy minimální počet výskytů roven 1, maximální počet výskytů 3 a rozdíl je roven $|3-1| = 2 = k$.

Příklad:

Nechť pětice proměnných $x_1, \ldots, x_5$ nabývá po řadě těchto hodnot $3, 1, 3, 1, 1$. Potom podmínka

$$\texttt{balance}(\langle 3, 1, 3, 1, 1 \rangle, 2)$$

pro parametr $k = 2$ neplatí. Obdobným postupem jako v předchozím příkladě zjistíme, že minimální počet výskytů jedné hodnoty je 2 a maximální počet výskytů jedné hodnoty mezi proměnnými $x_1, \ldots, x_5$ je 3. Rozdíl těchto čísel dává výsledek 1, což není rovno parametru $k = 2$.

2.3.5 gcc

Zdroj: [6, 34].

Mějme n-tici proměnných $x_1, \ldots, x_n$, které nabývají hodnot z domény D velikosti $m = |D|$, a m-tici četností $c_1, \ldots, c_m$. Podmínka gcc (global cardinality constraint) platí, pokud každá hodnota $d_i \in D$ je v $x_1, \ldots, x_n$ použita právě c_i-krát.

$$\texttt{gcc}(x_1, \ldots, x_n, c_1, \ldots, c_m) \Leftrightarrow \forall i \in \{1, \ldots, m\} : |\{j \in \{1, \ldots, n\} : x_j = d_i\}| = c_i$$

Motivace:

V knize [21] je k nalezení Magic Series Problem, který lze řešit za použití podmínky gcc. Mějme uspořádanou $(n+1)$-tici proměnných $X = (x_0, \ldots, x_n)$. Tuto uspořádanou $(n+1)$-tici nazveme *magickou*, pokud pro každý index i se v uspořádané $(n+1)$-tici $x_0, \ldots, x_n$ vyskytuje právě x_i proměnných mající hodnotu i.

Příkladem takové $(n+1)$-tice může být $(3, 2, 1, 1, 0, 0, 0)$, která obsahuje 3-krát hodnotu 0, 2-krát hodnotu 1, 1-krát hodnotu 2, 1-krát hodnotu 3 a žádnou hodnotu 4, 5 a 6.

Příklad:

Mějme čtyři proměnné x_1, x_2, x_3 a x_4 s hodnotami 1, 2, 4 a 2. Nechť množina D obsahuje hodnoty $\{d_1 = 1, d_2 = 2, d_3 = 3, d_4 = 4\}$ a četnosti výskytů hodnot jsou $c_1 = 1, c_2 = 2, c_3 = 0, c_4 = 1$. Potom podmínka

$$\texttt{gcc}(\langle 1, 2, 4, 2\rangle, \langle 1, 2, 0, 1\rangle)$$

platí, neboť hodnota 1 se mezi proměnnými vyskytuje pouze jednou, a to v proměnné x_1. Hodnoty 2 nabývají 2 proměnné x_2 a x_4. Hodnota 3 nebyla použita žádnou proměnnou a hodnota 4 byla použita pouze proměnou x_3.

Příklad:

Nechť proměnné $x_1, \ldots, x_4$ nyní nabývají hodnot 1, 2, 4 a 4. Zůstanou-li ostatní parametry stejné jako v předchozím příkladě, podmínka

$$\texttt{gcc}(\langle 1, 2, 4, 4\rangle, \langle 1, 2, 0, 1\rangle)$$

nebude platit, protože hodnota 4 by se měla mezi proměnnými $x_1, \ldots, x_4$ vyskytovat právě jednou, ale nabývají ji dvě proměnné x_3 a x_4.

Filtrační algoritmus:

Pro podmínku `gcc` máme několik filtračních algoritmů. V článku [28] je např. popsán algoritmus, který dociluje konzistence okrajů.

Klasický algoritmus pro filtraci podmínky `gcc` zajišťuje hranovou konzistenci a je založen na hledání maximálního toku v síti a silně souvislých komponent grafu. Úplný popis tohoto algoritmu je možno nalézt v [34]. Jeho složitost je $O(n^{3/2}m)$, což je dáno hlavně díky použití Ford-Fulkersonova algoritmu pro hledání maximálního toku v síti.

Implementováno v CSP řešičích:

B-Prolog `global_cardinality`, Disolver podobná podmínka `Occur`, Choco `globalCardinality`, Koalog `GCC`, SICStus Prolog `global_cardinality`.

2.3.6 `inflexion`

Zdroj: [6].

Pro popis podmínky `inflexion` nejprve na n-tici (posloupnosti) proměnných $x_1, \ldots, x_n$ definujme pojem hora. *Hora* je taková podposloupnost $x_i, \ldots, x_{j+1}$ posloupnosti $x_1, \ldots, x_n$, pro kterou platí:

1. $x_i < x_{i+1}$ – „závěrečné stoupání k vrcholu hory“,
2. $x_{i+1} = x_{i+2} = \ldots = x_{j-1} = x_j$ – „vrchol hory (náhorní plošina)“,
3. $x_j > x_{j+1}$ – „začátek sestupu z hory“.

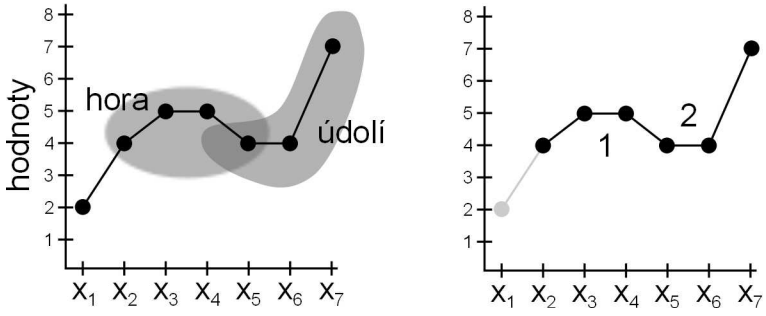
Hora je tedy definována pouze svým „vrcholkem“ a samotná výška hory (přes kolik proměnných bychom měli „stoupat“ či „sestupovat“) nás nezajímá.

Obdobně jako horu, definujeme i pojem údolí. *Údolí* je taková podposloupnost $x_i, \ldots, x_{j+1}$ posloupnosti $x_1, \ldots, x_n$, pro kteru platí:

1. $x_i > x_{i+1}$ – „příchod shora do údolí“,
2. $x_{i+1} = x_{i+2} = \ldots = x_{j-1} = x_j$ – „údolí“,
3. $x_j < x_{j+1}$ – „stoupání z údolí“.

Nyní přejděme k popisu samotné podmínky `inflexion`. Podmínka `inflexion` na vstupu dostává uspořádanou n-tici proměnných $x_1, \ldots, x_n$ a číslo k a počítá výskyty hor a údolí v posloupnosti $x_1, \ldots, x_n$. Platí, pokud celkový počet výskytů hor i údolí dohromady je roven číslu k.

`inflexion`$(k, x_1, \ldots, x_n)$ $\Leftrightarrow$ *celkový počet hor a údolí v* $x_1, \ldots, x_n$ *je roven* k.



Obr. 2.6: k definici a příkladu hor a údolí.

Příklad:

Mějme sedmici proměnných $x_1, \ldots, x_7$ nabývající po řadě hodnoty 2, 4, 5, 5, 4, 3 a 7. Situace je znázorněna na obrázku 2.6 vlevo. Nastavme parametr k na hodnotu 2. Potom podmínka

$$\texttt{inflexion}(\langle 2, 4, 5, 5, 4, 3, 7 \rangle, 2)$$

platí, neboť, jak je zobrazeno na obrázku 2.6 vpravo, posloupnost obsahuje právě jedno údolí a právě jednu horu, tj. počet výskytů je roven $2 = k$.

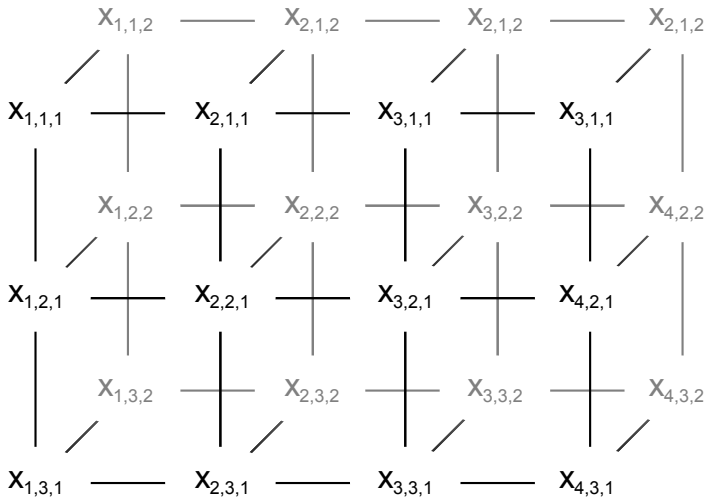
Varianty:

K podmínce `inflexion` ještě existuje podmínka `peak`, která v posloupnosti počítá pouze výskyty hor; a podmínka `valley`, která v posloupnosti počítá pouze výskyty údolí.

2.3.7 `connect points`

Zdroj: [6, 44].

Podmínka `connect points` je svým náhledem na úlohu jedinečná, neboť proměnné jsou tentokrát uspořádány do třírozměrné mřížky, namísto běžné jednorozměrné posloupnosti. Tuto třírozměrnou mřížku si lze představit jako ze sítě vytvořený kvádr, kde v každém vrcholu (i vnitřním) je umístěna jedna proměnná – viz obrázek 2.7.



Obr. 2.7: uspořádání proměnných do třírozměrné mřížky.

Vedle mřížky má tato podmínka ještě parametr počet skupin k. Skupinu tvoří vždy mřížkou propojené proměnné mající stejnou hodnotu. Podmínka `connect points` platí, pokud se v mřížce vyskytuje právě k skupin. Hodnota 0 má neutrální funkci a do výpočtu se skupiny tvořené nulami nezapočítávají.

$$\texttt{connect points}(x_{1,1,1}, \ldots, x_{m,n,r}, k) \Leftrightarrow$$

$\Leftrightarrow$ *počet skupin proměnných se stejnou hodnotou různou od 0 je* k.

Motivace:

Motivací pro tento na „první pohled“ netradiční náhled na problém je problematika návrhu nepřekrývajících se obvodů [44]. Úlohu si můžeme opravdu představit jako návrh funkčního integrovaného obvodu na co nejmenší ploše.

Příklad:

Uveďme příklad popsaný v [6]. Dimenze prostoru bude $m = 4$, $n = 8$ a $r = 2$. Zanesme do mřížky následující údaje a chtějme 2 nezávisle propojené „podsítě“.

1. vrstva	$x_{m,1,1}$	$x_{m,2,1}$	$x_{m,3,1}$	$x_{m,4,1}$	$x_{m,5,1}$	$x_{m,6,1}$	$x_{m,7,1}$	$x_{m,8,1}$
$x_{1,n,1}$			**1**	**1**		*2*		
$x_{2,n,1}$				**1**		*2*		
$x_{3,n,1}$				**1**	**1**	**1**	**1**	**1**
$x_{4,n,1}$		*2*		**1**		*2*		

2. vrstva	$x_{m,1,2}$	$x_{m,2,2}$	$x_{m,3,2}$	$x_{m,4,2}$	$x_{m,5,2}$	$x_{m,6,2}$	$x_{m,7,2}$	$x_{m,8,2}$
$x_{1,n,2}$								
$x_{2,n,2}$						*2*		
$x_{3,n,2}$		*2*	*2*	*2*	*2*	*2*		
$x_{4,n,2}$		*2*				*2*		

Na těchto datech podmínka `connect points` platí, neboť máme dvě spojité „podsítě“ označené **1** a *2*. Spojitost „podsítě“ označené `1` je ihned zřejmá. U „podsítě“ označené *2* musíme přihlédnout i k druhé vrstvě, která propojuje zdánlivě nespojené bloky ve vrstvě první.

2.4 Podmínky rozdílnosti hodnot proměnných

Podmínky rozdílnosti hodnot proměnných, mající na proměnné požadavek různorodosti přidělených hodnot, spolu s podmínkami omezujícími počet výskytů hodnot z předchozí kategorie řadíme mezi základní globální podmínky.

Nejtypičtější podmínkou tohoto typu je podmínka `all different` (2.4.1). Požaduje, aby všechny hodnoty proměnných byly různé. Zjemněním této podmínky je podmínka `some different` (2.4.2), která požadavek rozdílnosti hodnot klade pouze na vybrané dvojice proměnných.

Třetí podmínka `nvalue` (2.4.3) stojí na pomezí této kategorie a předchozí kategorie omezující počet výskytů hodnot. `nvalue` je splněna, pokud počet různých hodnot mezi proměnnými je roven zadanému parametru.

Podmínka `inter-distance` (2.4.4) k rozdílnosti hodnot mezi proměnnými přidává požadavek minimální vzdálenosti mezi nimi a podmínka `golomb` (2.4.5) ještě navíc omezuje i tyto vzdálenosti.

Podmínka `dijoint` (2.4.6) zachází s dvěma kolekcemi proměnných a odlišnost hodnot je kladena na jednu kolekci od druhé.

2.4.1 `all different`

Zdroj: [6, 34].

Podmínka `all different` na vstupu dostává n proměnných $x_1, \ldots, x_n$ a platí, pokud všechny proměnné nabývají různých hodnot, tj.

$$\texttt{all\_different}(x_1, \ldots, x_n) \;\Leftrightarrow\; \forall i, j \in \{1, \ldots, n\},\; i \neq j :\; x_i \neq x_j.$$

Motivace:

Podmínka `all different` se vyskytuje téměř v každém CSP, přesto lze vybrat typický motivační problém, a to n-Queen Problem. Spočívá v umístění n dam na šachovnici o rozměrech $n \times n$ takovým způsobem, aby se žádné dvě dámy navzájem neohrožovaly.

Zvolme takovouto reprezentaci problému: proměnná $x_i \in \{1, \ldots, n\}$ vyjadřuje číslo řádku v i-tém sloupci na šachovnici, kam je umístěna jedna z dam. Tedy každá dáma se nachází ve sloupci i na pozici x_i.

Potřebujeme formulovat celkem čtyři omezení: v každém sloupci může být jen jedna dáma, v každém řádku může být jen jedna dáma, diagonálně jedním směrem může být jen jedna dáma a diagonálně druhým směrem může být jen jedna dáma. První podmínku máme „zadarmo“ díky zvolené reprezentaci, další triviálně namodelujeme podmínkami

- $\texttt{all\_different}(x_1, \ldots, x_n)$ pro řádky,
- $\texttt{all\_different}(x_1, x_2 + 1, \ldots, x_n + n - 1)$ pro diagonály jedním směrem a
- $\texttt{all\_different}(x_1 + n - 1, x_2 + n - 2, \ldots, x_n)$ pro diagonály druhým směrem.

Příklad:

Uveďme jednoduchý příklad bez návaznosti na motivační problém. Nechť počet proměnných je $n = 4$ a proměnné x_1, x_2, x_3 a x_4 nabývají hodnot 1, 22, 14 a 7. Podmínka

$$\texttt{all\_different}(1, 22, 14, 7)$$

s těmito vstupními daty platí, protože $1 \neq 22 \neq 14 \neq 7$.

Příklad:

Nechť počet proměnných je $n = 4$ a proměnné x_1, x_2, x_3 a x_4 nabývají hodnot 7, 22, 14 a 7. Podmínka

$$\texttt{all\_different}(7, 22, 14, 7)$$

s těmito vstupními daty neplatí, neboť hodnota proměnné x_1 je shodná s hodnotou proměnné x_4 ($7 = x_1 = x_4$).

Filtrační algoritmus:

Celý filtrační algoritmus je popsán např. v [34]. Algoritmus je polynomiální, dociluje hranové konzistence, je založen na maximálním párování v bipartitních grafech a nalezení silně souvislých komponent.

Historie vývoje filtračních algoritmů spolu s odkazy na patřičné články je k nalezení v katalogu [6] pod klíčovým slovem *alldifferent*.

Implementováno v CSP řešičích:

B-Prolog `all_different` a `all_disctinct` jako variantu se silnější propagací, Comet `AllDifferent`, CSP Library `AllDiff`, Disolver `AllDiff`, Ecl^{i}ps^e `alldifferent`, Gecode `distinct`, GNU Prolog `fd_all_different`, Choco `allDifferent`, Koalog `AllDifferent`, Mozart `distinct`, SICStus Prolog `all_different`.

2.4.2 `some different`

Zdroj: [33].

Globální podmínka `some different` je zobecněním podmínky `all different` a vyžaduje pouze, aby některé dvojice proměnných nabývaly různých hodnot. Jako parametr dostává proměnné $X = \{x_1, \ldots, x_n\}$ a seznam dvojic indexů $E \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ vyjadřující vztahy mezi proměnnými. Podmínka `some different` platí, pokud všechny dvojice proměnných s indexy z množiny E nabývají různých hodnot, tedy:

$$\texttt{some\_different}(x_1, \ldots, x_n, E) \;\Leftrightarrow\; \forall (i,j) \in E : x_i \neq x_j.$$

Motivace:

Motivační úlohou pro podmínku `some different` je Workforce Management Problem popsaný v [41]. Máme nějaké úkoly, zaměstnance a informaci o tom, jaký zaměstnanec může pracovat na jakém úkolu. Úkoly mohou začínat a končit v různých časech nebo např. vyžadují pouze částečnou účast zaměstnance na jejich plnění. Závěrem tedy, nějaké dvojice úkolů mohou být přiřazeny k jednomu zaměstnanci, jiné ne.

Příklad:

Mějme čtveřici proměnných $x_1, \ldots, x_n$ a přiřaďme jim hodnoty 1, 4, 2, 2. Množinu E naplňme těmito dvojicemi indexů: $E = \{(1,3), (2,3), (2,4)\}$. Potom podmínka

$$\texttt{some\_different}(\langle 1,4,2,2\rangle, \{(1,3),(2,3),(2,4)\})$$

platí, protože pro každou dvojici indexů (1, 3), (2, 3), (2, 4) jsou vždy hodnoty příslušných proměnných různé ($x_1 = 1 \neq 2 = x_3$, $x_2 = 4 \neq 2 = x_3$ a $x_2 = 4 \neq 2 = x_4$).

Příklad:

Nyní naplňme množinu dvojic indexů těmito daty $E = \{(1,3), (2,3), (3,4)\}$. Potom podmínka

$$\texttt{some\_different}(\langle 1,4,2,2\rangle, \{(1,3),(2,3),(3,4)\})$$

neplatí, neboť vezmeme-li dvojici indexů $(3,4)$ a k nim příslušné proměnné x_3 a x_4, zjistíme, že jejich hodnoty se rovnají ($2 = x_2 = x_4$).

Filtrační algoritmus:

Filtrační algoritmus podmínky `some diffent` je popsán v článku [33]. Základní myšlenkou algoritmu je sestavení grafu z proměnných, přiřazení ke každé proměnné množiny barev dle její domény a snaha o obarvení tohoto grafu. Algoritmus dociluje hranové konzistence.

2.4.3 `nvalue`

Zdroj: [6, 7].

Mějme posloupnost proměnných $x_1, \ldots, x_n$ a číslo $k \leq n$. Podmínka `nvalue` platí, pokud počet různých hodnot mezi proměnnými $x_1, \ldots, x_n$ je právě k.

$$\texttt{nvalue}(x_1, \ldots, x_n, k) \;\Leftrightarrow\; |x_1 \cup \ldots \cup x_n| = k.$$

Motivace:

Úloha řešitelná použitím podmínky `nvalue` (number of distinct values) je Dominating Queens Chess Puzzle Problem, který spočívá v umístění několika dam na šachovnici $n \times n$ tak, aby každé pole šachovnice bylo buď dámou obsazeno a nebo nějakou dámou ohrožováno.

Vhodně zvoleným modelem lze výše uvedený problém vyjádřit pouze jednou podmínkou `nvalue`. Jak tento model vypadá, je popsáno v [6] pod heslem *nvalue*.

Příklad:

Nechť proměnné $x_1, \ldots, x_4$ nabývají po řadě hodnot $4, 4, 2, 3$. Potom pro $k = 3$ podmínka

$$\texttt{nvalue}(\langle 4, 4, 2, 3 \rangle, 3)$$

platí, protože proměnné $x_1, \ldots, x_4$ nabývají pouze třech hodnot 2, 3 a 4.

Filtrační algoritmus:

Práce polynomiálního filtračního algoritmu nejen pro podmínku `nvalue` je popsána v [7]. Algoritmus se řídí aktuálním stavem domény D proměnných $x_1, \ldots, x_n$ a volí strategie třídění domény či maximální párování v bipartitních grafech.

2.4.4 `inter-distance`

Zdroj: [2].

Podmínka `inter-distance` na vstupu dostává n proměnných $x_1, \ldots, x_n$ a číslo h. Zajišťuje, aby vzdálenost mezi hodnotami každých dvou proměnných byla alespoň h. Tj.

$$\texttt{inter-distance}(x_1, \ldots, x_n, h) \;\Leftrightarrow\; \forall i, j \in \{1, \ldots, n\},\ i \neq j :\ |x_i - x_j| \geq h.$$

Pokud je číslo $h = 1$, podmínka `inter-distance` se redukuje na podmínku `all different`.

Motivace:

Vznik této podmínky byl motivován řízením letového provozu [3], kdy máme jednu přistávací dráhu a několik letadel ve vzduchu, která chtějí přistát. Jakmile je letadlo připraveno k přistávacímu manévru (například na vyčkávacím okruhu), je mu automaticky přidělen časový úsek (tzv. okno), během kterého musí přistání zvládnout. Úkolem je maximalizovat minimální čas mezi po sobě jdoucími přistáními. Popř. jiná varianta úkolu (rozhodovací problém): minimální čas mezi po sobě jdoucími přistáními je pevně dán a úkolem je rozhodnout, zda jsou pro další přistání k dispozici vhodná časová okna nebo ne.

Příklad:

Nechť $h = 5$, počet proměnných $n = 4$ a proměnné $x_1, \ldots, x_4$ nabývají hodnot 1, 22, 14 a 7. Podmínka

$$\texttt{inter-distance}(5, \langle 1, 22, 14, 7 \rangle)$$

s těmito vstupními daty platí, neboť $|1 - 22| \geq 5$, $|1 - 14| \geq 5$, $|1 - 7| \geq 5$, $|22 - 14| \geq 5$, $|22 - 7| \geq 5$ a $|14 - 7| \geq 5$.

Příklad:

Nechť $h = 5$, počet proměnných $n = 4$ a proměnné $x_1, \ldots, x_4$ nabývají hodnot 3, 22, 14 a 7. Podmínka

$$\texttt{inter-distance}(5, \langle 3, 22, 14, 7 \rangle)$$

s těmito vstupními daty neplatí, neboť $|3 - 7| = 4$, což je menší než zadaný minimální rozdíl 5.

Filtrační algoritmus:

Filtrační algoritmus blíže popsaný v článku [2] využívá převodu podmínky `inter distance` na rozvrhovací podmínku, kde každé proměnné x_i je přiřazena událost začínající v čase x_i a mající délku trvání h. K této rozvrhovací podmínce jsou ještě přidány podmínky disjunkcí, aby události nemohly běžet současně.

2.4.5 `golomb`

Zdroj: [6, 38].

Mějme posloupnost proměnných $x_1, \ldots, x_n$ s ostře rostoucími hodnotami. Potom rozdíly mezi hodnotami všech dvojic proměnných jsou různé. Tedy:

$$\texttt{golomb}(x_1, \ldots, x_n) \;\Leftrightarrow\; |\{|x_i - x_j| \;:\; i, j \in \{1, \ldots, n\}, i \neq j\}| = n.$$

Motivace:

Vznik podmínky byl motivován problémem Golombova pravítka (odtud také název podmínky `golomb`). *Golombovo pravítko* je ostře uspořádaná posloupnost celých čísel taková, že rozdíly každých dvou hodnot jsou navzájem různé. Golombova pravítka jsou kategorizována podle délky a úkolem je většinou pro zadanou délku najít co nejmenší Golombovo pravítko. Více o této problematice lze nalézt v [38].

Příklad:

Uveďme pouze jeden příklad. Mějme uspořádanou čtveřici proměnných $x_1, \ldots, x_4$ s hodnotami po řadě $0, 1, 4, 6$. Potom podmínka

$$\texttt{golomb}(0, 1, 4, 6)$$

platí, neboť rozdíly mezi všemi dvojicemi hodnot jsou různé: $|x_1 - x_2| = 1$, $|x_1 - x_3| = 4$, $|x_1 - x_4| = 6$, $|x_2 - x_3| = 3$, $|x_2 - x_4| = 5$ a $|x_3 - x_4| = 2$.

Filtrační algoritmus:

Ač podmínka vypadá jako `all different` (2.4.1), nemáme k dispozici žádný vhodný filtrační algoritmus. Lze však využít filtračního algoritmu pro podmínku `all different`, což ale zaručí pouze částečné filtrování.

2.4.6 `disjoint`

Zdroj: [9].

Podmínka `disjoint` na vstupu dostává n-tici proměnných $x_1, \ldots, x_n$ a m-tici proměnných $y_1, \ldots, y_m$. Platí, pokud všechny proměnné $x_1, \ldots, x_n$ nabývají různých hodnot, než proměnné $y_1, \ldots, y_m$. V rámci jedné n-tice, resp. m-tice, však stejná hodnota může být použita vícekrát.

Nechť množina $Y = y_1 \cup \ldots \cup y_m$ je množina hodnot, kterých právě nabývají proměnné $y_1, \ldots, y_m$. Potom

$$\texttt{disjoint}(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow \forall i \in \{1, \ldots, n\} : x_i \notin Y.$$

Motivace:

Podmínku `disjoint` můžeme použít např. ve chvíli, kdy chceme, aby pracovní směny přiřazené jedné osobě byly různé od pracovních směn druhé osoby, která k první osobě chová antipatie.

Podmínka `disjoint` je mj. speciálním případem podmínky `common` (viz 2.5.2), kdy $k = l = 0$.

Příklad:

Namodelujme motivační úlohu tímto způsobem: označme si jednotlivé směny nějakým přirozeným číslem. Proměnné $x_1, \ldots, x_n$ budou představovat přiřazení první osoby ke směnám tak, že každá proměnná představuje jedno navštívení směny a hodnota proměnné udává přímo které směny. Význam proměnných $y_1, \ldots, y_m$ je zcela analogický pro druhou osobu.

Nechť přiřazení osob ke směnám je $x_1 = 1$, $x_2 = 3$, $x_3 = 4$, $x_4 = 8$, $y_1 = 2$, $y_2 = 5$ a $y_3 = 6$. Potom podmínka

$$\texttt{disjoint}(\langle 1, 3, 4, 8\rangle, \langle 2, 5, 6\rangle)$$

platí, neboť mezi proměnnými x_i není žádná taková, která by nabývala hodnot z množiny $Y = y_1 \cup y_2 \cup y_3 = \{2\} \cup \{5\} \cup \{6\} = \{2, 5, 6\}$.

Příklad:

Nechť model je stejný jako v předchozím příkladě. Definujme nová data $x_1 = 1$, $x_2 = 3$, $x_3 = 4$, $x_4 = 8$, $y_1 = 1$, $y_2 = 5$ a $y_3 = 3$. Podmínka

$$\texttt{disjoint}(\langle 1, 3, 4, 8\rangle, \langle 1, 5, 3\rangle)$$

neplatí, protože existují proměnné x_i, které nabývají nějaké hodnoty z množiny $Y = y_1 \cup y_2 \cup y_3 = \{1, 3, 5\}$. Konkrétně jsou to proměnné $x_1 = 1 \in Y$ a $x_2 = 3 \in Y$.

Filtrační algoritmus:

Jak se lze dočíst v [9], filtrační algoritmus zajišťující hranovou konzistenci je NP-úplný.

Pokud nám ale postačí slabší propagace, lze podmínku `disjoint` rozložit na nerovnosti mezi každou dvojicí $x_i \neq y_j$. Tato dekompozice je dobře použitelná hlavně pro konzistenci okrajů.

Více o filtračních algoritmech podmínku `dijoint` je též k nalezení v [9].

Implementováno v CSP řešičích:

B-Prolog `fd_disjoint`, Mozart `disjoint`.

2.5 Podmínky shodností hodnot proměnných

Opačnou problematikou k podmínkám z odstavce 2.4 se zabývají podmínky shodností hodnot proměnných. Jak už z názvu kategorie vyplývá, tyto podmínky se snaží, aby hodnoty proměnných si byly k sobě co nejbližší.

První podmínkou v této kategorii je podmínka `same` (2.5.1), která po dvou n-ticích proměnných požaduje jejich naprostou shodnost (až na pořadí prvků). Podmínka `common` (2.5.2) je méně striktní a požaduje pouze, aby některé prvky z n-tic byly shodné.

Poslední podmínka v této kategorii `used by` (2.5.3) je dalším polevením a tentokrát je vyžadována shodnost jen jedním směrem.

2.5.1 `same`

Zdroj: [8].

Mějme dvě n-tice proměnných $x_1, \ldots, x_n$ a $y_1, \ldots, y_n$. Podmínka `same` platí, pokud hodnoty proměnných $x_1, \ldots, x_n$ a $y_1, \ldots, y_n$ jsou shodné a liší se pouze svým umístěním.

$$\texttt{same}(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow$$
$$\Leftrightarrow \exists \textit{ bijekce } \pi : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\} \textit{ taková, že } \forall i \in \{1, \ldots, n\} : x_i = y_{\pi(i)}$$

Motivace:

Motivační úlohou pro podmínku `same` je problém Doctors Without Borders [14]. Máme zadán seznam doktorů a zdravotních sester. Každý z nich pravidelně každý rok dobrovolně nabídne své služby pro jednu záchrannou misi. Přihlášky jsou ve formě možných termínů, kdy se doktor či sestra mohou nějaké záchranné mise zúčastnit. Každé mise se účastní právě jeden doktor a právě jedna sestra. Úkolem je sestavit takový seznam dvojic doktor-sestra, aby doktor i sestra byli v daném termínu k dispozici a každý doktor i každá sestra by vystupovali právě v jedné dvojici (účastnili se právě jedné záchranné mise).

Příklad:

Nechť proměnné $x_1, \ldots, x_4$ po řadě nabývají hodnot 2, 2, 4, 8 a proměnné $y_1, \ldots, y_4$ po řadě nabývají hodnot 4, 2, 8, 2. Potom podmínka

$$\texttt{same}(\langle 2, 2, 4, 8\rangle, \langle 4, 2, 8, 2\rangle)$$

platí, neboť hodnoty v obou dvou posloupnostech jsou stejné, jen se liší pořadím: $x_1 = y_2$, $x_2 = y_4$, $x_3 = y_1$ a $x_4 = x_3$. Pomocné zobrazení π by vypadalo následnovně: $\pi = \{(1, 2), (2, 4), (3, 1), (4, 3)\}$.

Příklad:

Mějme proměnné $x_1, \ldots, x_4$ po řadě nabývající hodnot 2, 2, 4, 8 a proměnné $y_1, \ldots, y_4$ po řadě nabývající hodnot 4, 4, 8, 2. Potom podmínka

$$\texttt{same}(\langle 2, 2, 4, 8\rangle, \langle 4, 4, 8, 2\rangle)$$

neplatí, neboť hodnoty v obou dvou posloupnostech jsou sice stejné, ale počet výskytů každé z nich už ne. Hodnota 4 v y-posloupnosti vystupuje dvakrát, mezitím co v x-posloupnosti se tato hodnota vyskytuje pouze jednou.

Filtrační algoritmus:

Polynomiální filtrační algoritmus zajišťující hranovou konzistenci je založen na sestavení bipartitního grafu z proměnných $x_1, \ldots, x_n$, $y_1, \ldots, y_n$, nalezení maximálního toku v síti a nalezení silně souvislých komponent v grafu.

Existuje též filtrovací algoritmus pro konzistenci okrajů, jehož princip je shodný s výše uvedeným algoritmem.

Přesný popis obou filtrační algoritmů se nachází v [8].

2.5.2 `common`

Zdroj: [9].

Podmínka `common` pracuje s n-ticí proměnných $x_1, \ldots, x_n$ a m-ticí proměnných $y_1, \ldots, y_m$. Dalšími parametry jsou čísla $k \leq n$ a $l \leq m$.

Podmínka `common` platí, pokud k proměnných z $x_1, \ldots, x_n$ nabývá stejných hodnot jako proměnné $y_1, \ldots, y_m$ a l proměnných z posloupnosti $y_1, \ldots, y_m$ nabývá stejných hodnot jako proměnné $x_1, \ldots, x_n$.

Nechť $X = x_1 \cup \ldots x_n$ je množina hodnot, kterých právě nabývají proměnné $x_1, \ldots, x_n$ a $Y = y_1 \cup \ldots y_m$ je množina hodnot, kterých právě nabývají proměnné $y_1, \ldots, y_m$. Potom

$$\texttt{common}(x_1, \ldots, x_n, y_1, \ldots, y_m, k, l) \Leftrightarrow$$
$$\Leftrightarrow |\{i \in \{1, \ldots, n\} : x_i \in Y\}| = k \ \& \ |\{j \in \{1, \ldots, m\} : y_j \in X\}| = l.$$

Motivace:

Podmínka `common` je zobecněním podmínek `among` (viz 2.3.2) a `all different` (viz 2.4.1). Pro ilustraci zobecnění na podmínce `all different` vypadá takto: proměnné $y_1, \ldots, y_n$ nabývají všech hodnot proměnných $x_1, \ldots, x_n$ a $l = n$.

Příklad:

Mějme čtveřici proměnných x_1, x_2, x_3, x_4, které nabývají hodnot 1, 2, 5, 2. Dále mějme pětici proměnných y_1, y_2, y_3, y_4, y_5, které nabývají hodnot 4, 4, 2, 1, 5. Nastavme parametry $k = 4$ a $l = 3$. Procházejme nyní definici a ukažme, že podmínka

$$\texttt{common}(\langle 1, 2, 5, 2\rangle, \langle 4, 4, 2, 1, 5\rangle, 4, 3)$$

platí. Spočtěme množiny X a Y: $X = x_1 \cup \ldots \cup x_4 = \{1\} \cup \{2\} \cup \{5\} \cup \{2\} = \{1, 2, 5\}$, $Y = y_1 \cup \ldots \cup y_5 = \{1, 2, 4, 5\}$.

Proměnné x_i, které nabývají nějaké hodnoty z množiny Y, jsou čtyři: x_1, x_2, x_3 i x_4. Tedy je splněna rovnost $4 = k$. Proměnné y_i nabývající nějaké hodnoty z množiny X jso tři: y_3, y_4 a y_5. I zde je splněna rovnost $3 = l$.

Příklad:

Uvažujme stejné proměnné jako v předchozím příkladě i s hodnotami, které nabývají. Záměnou parametru k na hodnotu 2 docílíme zřejmé neplatnosti podmínky `common`.

Filtrační algoritmus:

Pro podmínku `common` existuje v nejhorším případě exponeciální algoritmus splňující hranovou konzistenci. Je však možné vytvořit slabší filtrační algoritmus založený na dekompozici podmínky `common` na dvě podmínky `among`. Důkaz exponenciality a informace k dekompozici je možné najít v [9].

2.5.3 `used by`

Zdroj: [8]..

Mějme n-tici proměnných $x_1, \ldots, x_n$ a m-tici proměnných $y_1, \ldots, y_m$, přičemž pro počty proměnných platí nerovnost $n \geq m$. Podmínka `used by` je splněna, pokud všechny hodnoty proměnných $y_1, \ldots, y_m$ jsou obsaženy mezi hodnotami proměnných $x_1, \ldots, x_n$ (včetně vícenásobného výskytu).

$$\texttt{used\_by}(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow \exists \textit{ prosté zobr. } f : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}$$
$$\textit{takové, že } \forall i \in \{1, \ldots, m\} : y_i = x_{f(i)}.$$

Pokud jsou počty proměnných stejné, tj. $n = m$, podmínka `used by` degeneruje na podmínku `same` (viz 2.5.1).

Motivace:

Podmínka `used by` je stejně jako podmínka `same` (viz 2.5.1) motivována problémem Doctors Without Borders [14], tentokrát v modifikaci, kdy máme k dispozici více zdravotních sester, než lékařů. Každé mise se opět účastní pouze jeden doktor a jedna sestra, ale úkolem je nalezení takového seznamu dvojic doktor-sestra, aby každý lékař vystupoval v právě jedné dvojici a každá sestra vystupovala nejvýše v jedné dvojici. Tedy ne každá sestra bude vybrána jako dobrovolník k zúčastnění se záchranné mise.

Příklad:

Zkusme namodelovat přiřazení zdravotních sester k misím tak, aby na každé misi byla k dispozici alespoň jedna sestra. Proměnné $x_1, \ldots, x_n$ budou představovat zdravotní sestry a hodnota proměnné x_i bude „identifikátor“ mise, na kterou byla sestra i přiřazena. Seznam misí nechť představují proměnné $y_1, \ldots, y_m$, hodnotami jsou pak „identifikátory“ misí.

Nechť jsou k dipozici tyto tři mise $y_1 = 1$, $y_2 = 2$ a $y_3 = 3$. Mějme čtyři sestry s přidělením misí $x_1 = 2$, $x_2 = 3$, $x_3 = 1$ a $x_4 = 3$. Potom podmínka

$$\texttt{used\_by}(\langle 2, 3, 1, 3\rangle, \langle 1, 2, 3\rangle)$$

platí, neboť všechny mise jsou obsazeny, tj. hodnoty y_1 nabývá proměnná x_3, hodnoty y_2 nabývá proměnná x_1 a hodnoty y_3 nabývají proměnné x_2 a x_4.

Příklad:

Do modelu popsaném v předchozím příkladě vložme nová data: jsou k dipozici tři mise $y_1 = 1$, $y_2 = 2$, $y_3 = 3$ s přiřazením sester $x_1 = 0$, $x_2 = 0$, $x_3 = 1$ a $x_4 = 3$. Potom podmínka

$$\texttt{used\_by}(\langle 0, 0, 1, 3\rangle, \langle 1, 2, 3\rangle)$$

neplatí, protože pro pro hodnotu $y_2 = 2$ neexistuje mezi proměnnými $x_1, \ldots, x_4$ proměnná s tutéž hodnotou.

Filtrační algoritmus:

Nejjednodušším způsobem je doplnit proměnné $y_1, \ldots, y_m$ $n - m$ pomocnými proměnnými, převést tak tuto podmínku na podmínku `same` a k filtraci využít její algoritmy.

Je také možné naprogramovat filtrační algoritmy zajišťující hranovou konzistenci nebo konzistenci okrajů využitím hledání maximální toku v síti, ale asymptoticky půjde o stejnou časovou složitost jako při použití výše uvedeného převodu. Při reálné implementaci by filtrační algoritmus přímo na míru podmínky `used by` měl být rychlejší, než převod na `same` a následné použití `same` filtračního algoritmu.

Více k podmíce `same` viz 2.5.1. Filtrační algoritmy je možno nalézt v [8].

2.6 Podmínky s uspořádáním a permutacemi

Podmínky s uspořádáním a permutacemi jsou velmi významné pro eliminaci symetrií v CSP.

Než přikročíme ke krátkému přehledu podmínek z této kategorie, připomeňme si pojem lexikografického uspořádání.

Lexikografické uspořádání zavádíme na posloupnostech (vektorech či n-ticích) hodnot a to následujícím způsobem. Nechť $(a_1, \ldots, a_n) \in D^n$ a $(b_1, \ldots, b_n) \in D^n$ jsou dvě uspořádané n-tice a na D je definováno uspořádání $\leq$. Potom

$$(a_1, \ldots, a_n) \leq_{lex} (b_1, \ldots, b_n) \;\Leftrightarrow\; (a_1 \leq b_1) \vee (a_1 = b_1 \;\&\; a_2 \leq b_2) \vee$$

$$\vee \ldots \vee (a_1 = b_1 \;\&\; \ldots \;\&\; a_{n-1} = b_{n-1} \;\&\; a_n \leq b_n).$$

První podmínka v této kategorii, podmínka `lex` (2.6.1) je přesným přepisem výše uvedené definice pro lexikografické uspořádání. Porovnává dvě n-tice a platí, pokud jsou lexikograficky uspořádané.

Podmínka `all permutations` (2.6.2) v sobě kloubí lexikografické uspořádání a permutace. Podmínka `correspondence` (2.6.3) je podobná podmínce `same` (2.5.1), ale narozdíl od ní navíc sleduje permutaci na prvcích.

Další tři podmínky – `sort` (2.6.4), `increasing` (2.6.5) a `decreasing` (2.6.6) – jsou uvedeny jen pro úplnost a vzhledem k jejich jednoduchosti i bez příkladů.

2.6.1 `lex`

Zdroj: [17].

Mějme dvě uspořádané n-tice proměnných $(x_1, \ldots, x_n)$ a $(y_1, \ldots, y_n)$. Podmínka `lex` platí, pokud $(x_1, \ldots, x_n) \leq_{lex} (y_1, \ldots, y_n)$, tedy $(x_1, \ldots, x_n)$ je lexikograficky menší, než $(y_1, \ldots, y_n)$.

$$\texttt{lex}((x_1, \ldots, x_n), (y_1, \ldots, y_n)) \Leftrightarrow (x_1, \ldots, x_n) \leq_{lex} (y_1, \ldots, y_n).$$

Motivace:

V CSP se občas setkáváme se symetrií, jako například v případě sportovního rozvrhování. Máme matici proměnných představující sportovní zápasy v daném týdnu a v daném kole. Týdny a kola jsou zaměnitelná, a proto můžeme provádět libovolné permutace na řádkách a sloupcích. Pro porušení této symetrie můžeme přidat podmínku, aby řádky i sloupce v matici byly lexikograficky uspořádané.

Příklad:

Mějme tyto dvě uspořádané čtveřice: $(2, 4, 1, 5)$ a $(2, 4, 3, 1)$. Podmínka

$$\texttt{lex}((2, 4, 1, 5), (2, 4, 3, 1))$$

platí protože první uspořádaná n-tice je lexikograficky menší než druhá uspořádaná n-tice. První členy $x_1 = y_1 = 2$ a $x_2 = y_2 = 4$ jsou si rovny, pro třetí člen v uspořádaných n-ticích platí $x_3 = 1 < 3 = y_3$.

Příklad:

Mějme tyto dvě uspořádané čtveřice: $(2, 4, 4, 2)$ a $(2, 4, 3, 1)$. Podmínka

$$\texttt{lex}((2, 4, 4, 2), (2, 4, 3, 1))$$

na těchto datech neplatí, neboť první dva členy v uspořádaných n-ticích jsou si sice rovny ($x_1 = y_1 = 2$ a $x_2 = y_2 = 4$), ale pro třetí člen platí $x_3 = 4 > 3 = y_3$.

Filtrační algoritmus:

Hlavní myšlenkou lineárního filtračního algoritmu podmínky `lex` je posunování ukazatelů po proměnných a jejich hodnotách v uspořádaných n-ticích. Zevrubně je tento algoritmus docilující hranové konzistence popsán v [17].

Implementováno v CSP řešičích:

Ecl^{i}ps^e `lexico_le`, Gecode `lex`, Choco `lex`, SICStus Prolog `lex_chain`.

2.6.2 `all permutations`

Zdroj: [6, 16].

Podmínka `all permutations` na vstupu dostává $m + 1$ vektorů $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_m$ každý délky n, a platí, pokud vektor $\vec{v}_0$ je v lexikografickém uspořádání menší nebo roven než všechny permutace na hodnotách ostatních vektorů $\vec{v}_1, \ldots, \vec{v}_m$.

$$\texttt{all\_permutations}(\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_m) \Leftrightarrow \forall i \in \{1, \ldots, m\}\ \forall \pi(\vec{v}_i) : \vec{v}_0 \leq_{lex} \pi(\vec{v}_i)$$

Motivace:

Při řešení úloh z oblasti rozvrhování, plánování a designu občas narážíme na symetrii: stejné stroje v továrnách dělají shodné úkoly ve stejných časech a se stále stejně přiděleným vyškoleným personálem. Pokud jí ignorujeme, CSP řešič zbytečně tráví čas prohledáváním, přičemž výsledek je stejný.

Podmínka `all permutations` patří mezi jedny z „bořičů“ symetrií a více o této problematice se lze dočíst v [16].

Příklad:

Nechť vektor $\vec{v}_0 = (1, 2, 3)$, $\vec{v}_1 = (3, 1, 2)$ a $\vec{v}_2 = (6, 4, 5)$. Podmínka

$$\texttt{all\_permutations}((1, 2, 3), (3, 1, 2), (6, 4, 5))$$

platí, neboť $(1, 2, 3)$ je lexikograficky menší nebo rovno, než všechny permutace na vektoru $(3, 1, 2)$, tj. lexikograficky menší nebo rovno než vektory $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ a $(3, 2, 1)$. Vektor $(1, 2, 3)$ je zřejmě lexikograficky menší, než všechny permutace na vektoru $(6, 4, 5)$.

Příklad:

Nechť vektor $\vec{v}_0 = (1, 2, 4)$, $\vec{v}_1 = (3, 1, 2)$ a $\vec{v}_2 = (6, 4, 5)$. Podmínka

$$\texttt{all\_permutations}((1, 2, 4), (3, 1, 2), (6, 4, 5))$$

neplatí, protože existuje permutace na vektoru $\vec{v}_1$ lexikograficky menší než $\vec{v}_0$ a to $\pi(\vec{v}_1) = \pi((3, 1, 2)) = (1, 2, 3) \leq_{lex} (1, 2, 4) = \vec{v}_0$.

2.6.3 `correspondence`

Zdroj: [6, 42].

Podmínka `correspondence` pracuje s permutacemi. Parametrem podmínky jsou tři uspořádané n-tice proměnných $x_1, \ldots, x_n$, $p_1, \ldots, p_n$ a $y_1, \ldots, y_n$. Proměnné $x_1, \ldots, x_n$ a $y_1, \ldots, y_n$ nabývají hodnot z nějaké domény D, proměnné $p_1, \ldots, p_2$ nabývají hodnot pouze z množiny $\{1, \ldots, n\}$.

Význam prostřední n-tice $p_1, \ldots, p_n$ je určení, na jakou pozici se hodnota z n-tice $x_1, \ldots, x_n$ „přesune“ do n-tice $y_1, \ldots, y_n$. Tedy podmínka platí, pokud hodnota každé proměnné x_i je rovna hodnotě y_{p_i}.

$$\texttt{correspondence}(x_1, \ldots, x_n, p_1, \ldots, p_n, y_1, \ldots, y_n) \Leftrightarrow$$
$$\Leftrightarrow \forall i \in \{1, \ldots, n\} : x_i = y_{p_i}$$

Motivace:

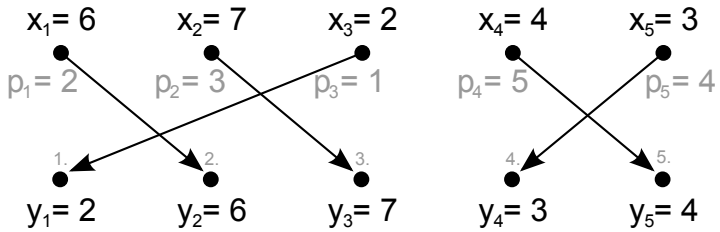
Job Scheduling Problem popsaný v [42] patří mezi dlouho studované problémy, které svojí složitostí spadají do kategorie NP-úplné. Motivací pro vytvoření podmínky `sort permutation` a z ní následného odvození podmínky `correspondence`, bylo zvolit jiný pohled na Job Scheduling Problem.

Příklad:

Mějme pětice proměnných $x_1, \ldots, x_5$, $p_1, \ldots p_5$ a $y_1, \ldots, y_5$. Nechť pětice nabývají následujících hodnot $x = (6, 7, 2, 4, 3)$, $y = (2, 3, 1, 5, 4)$, $z = (2, 6, 7, 3, 4)$. Potom podmínka

$$\texttt{correspondence}(\langle 6, 7, 2, 4, 3\rangle, \langle 2, 3, 1, 5, 4\rangle, \langle 2, 6, 7, 3, 4\rangle)$$

platí, jak je vidět z obrázku 2.8.



Obr. 2.8: k příkladu použití podmínky `correspondence`.

Filtrační algoritmus:

Vhodné algoritmy pro propagaci podmínek typu `correspondence` lze nalézt taktéž v článku [42].

2.6.4 `sort`

Zdroj: [6].

Mějme proměnné $x_1, \ldots, x_n$ a $y_1, \ldots, y_n$. Podmínka `sort` platí, pokud proměnné $y_1, \ldots, y_n$ nabývají stejných hodnot, jako proměnné $x_1, \ldots, x_n$, ale narozdíl od nich jsou seřazeny vzestupně.

$$\texttt{sort}(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow$$
$$\Leftrightarrow \exists \textit{ bijekce } \pi : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\} \textit{ taková, že } \forall\, x_i = y_{\pi(i)}$$
$$a\ \forall j \in \{1, \ldots, n-1\} : y_j \leq y_{j+1}$$

Podmínka je `sort` velmi podobná podmínce `same`, jen je silnější – přidává požadavek uspořádání na proměnných $y_1, \ldots, y_n$.

2.6.5 `increasing`

Zdroj: [6].

Mějme posloupnost proměnných $x_1, \ldots, x_n$. Podmínka `increasing` platí, pokud hodnoty proměnných jsou seřazeny vzestupně. Tedy

$$\texttt{increasing}(x_1, \ldots, x_n) \Leftrightarrow \forall i \in \{1, \ldots, n-1\} : x_i \leq x_{i+1}.$$

Implementováno v CSP řešičích:

Koalog `Increasing`.

2.6.6 `decreasing`

Zdroj: [6].

Podmínka `decreasing` na vstupu dostává n-tici proměnných $x_1, \ldots, x_n$ a platí, pokud jsou hodnoty těchto proměnných v posloupnosti seřazeny sestupně. Tj.

$$\texttt{decreasing}(x_1, \ldots, x_n) \Leftrightarrow \forall i \in \{1, \ldots, n-1\} : x_i \geq x_{i+1}.$$

2.7 Podmínky s alokací zdrojů

Podmínky s alokací zdrojů jsou už od začátku vytvořeny pro plánovací a rozvrhovací úlohy. Pracují nejčastěji s předměty, přihrádkami, událostmi a stroji.

Podmínka `bin-packing` (2.7.1) sleduje váhu předmětů a kapacitu přihrádek. `cummulative` (2.7.2) kontroluje, zda nedošlo k překročení kapacity zdroje v čase.

Podmínka `disjunctive` (2.7.3) kontroluje překrývání událostí v čase, `shift` (2.7.4) uspořádává události do směn a v podmínce `assign and counts` (2.7.5) se vraťme opět k umisťování předmětů do přihrádek tentokrát ve variantě s barvením předmětů.

2.7.1 `bin-packing`

Zdroj: [6, 37].

Mějme m přihrádek, všechny o stejné kapacitě $c \in \mathbb{N}_0$. Nechť $X = \{x_1, \ldots, x_n\}$ je množina předmětů, kde x_i je index z množiny $\{1, \ldots, m\}$ a představuje umístění předmětu i do přihrádky x_i. Funkce $w : X \longrightarrow \mathbb{N}_0$ přiřazuje každému předmětu váhu. Podmínka `bin-packing` platí, pokud součet vah předmětů umístěných v každé přihrádce nepřesahuje kapacitu přihrádky c.

$$\texttt{bin-packing}(x_1, \ldots, x_n, m, c, w) \;\Leftrightarrow\; \forall i \in \{1, \ldots, m\} : \sum_{x_j \text{ tž. } x_j = i} w(x_j) \leq c.$$

Motivace:

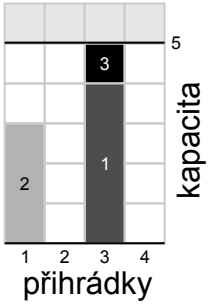
`Bin-packing` má poměrně mnoho aplikací, jakými jsou například snaha předcházet dopravním zácpám, rozvržení televizní reklam během vysílání či samozřejmě opravdové balení věcí.

Příklad:

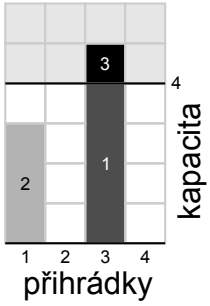
Uveďme velmi jednoduchý příklad, kdy máme k dispozici 3 předměty x_1, x_2, x_3 o váhách $w(x_1) = 4$, $w(x_2) = 3$ a $w(x_3) = 1$. Umístění předmětu do přihrádek je $x_1 = 3$, $x_2 = 1$ a $x_3 = 3$. Nechť celkový počet přihrádek je $m = 4$. Potom při kapacitě $c = 5$ každé přihrádky podmínka

$$\texttt{bin-packing}(\langle 3, 1, 3\rangle, 4, 5, \{(1, 4), (2, 3), (3, 1)\})$$

platí, neboť v první přihrádce je pouze předmět x_2 s vahou $3 \leq c$ a ve třetí přihrádce je předmět x_1 s předmětem x_3 a celkovou vahou $4 + 1 = 5 \leq c$. Znázornění situace viz obrázek 2.9.



Obr. 2.9: rozmístění předmětů do přihrádek o kapacitě 5.



Obr. 2.10: rozmístění předmětů do přihrádek o kapacitě 4.

Příklad:

Změňme nyní kapacitu přihrádek na $c = 4$, ostatní nechme stejné jako v předchozím příkladě. Potom podmínka

$$\texttt{bin-packing}(\langle 3, 1, 3\rangle, 4, 4, \{(1, 4), (2, 3), (3, 1)\})$$

neplatí. Sice v první přihrádce je váha předmětu $x_2 = 3 \leq c$, ale ve třetí přihrádce je součet vah předmětů x_1 a x_3 roven $4 + 1 = 5 > c$. Znázornění této situace viz obrázek 2.10.

Filtrační algoritmus:

Docílit hranové konzistence pro podmínku `bin-packing` je NP-úplný problém, avšak podmínku `bin-packing` lze převést na jednodušší problém batohu (knapsack problem), pro který už máme vhodný algoritmus.

O podmínce `bin-packing` a způsobu její propagace je více informací k dispozici v [37].

Implementováno v CSP řešičích:

Comet implementuje slabší podmínku `Knapsack`.

2.7.2 `cumulative`

Zdroj: [6, 24, 34].

Nechť n-tice uspořádaných trojic $(s_1, d_1, h_1), \ldots, (s_n, d_n, h_n)$ jsou v čase umístěné události, kde s_i je počáteční čas události i, d_i doba trvání události i a h_i celočíselný požadavek potřeby zdroje událostí i. Události se v čase mohou překrývat. Dále mějme zadánu kapacitu zdroje c.

Podmínka `cumulative` platí, pokud v žádném časovém okamžiku nejsou požadavky událostí na zdroj vyšší, než je jeho kapacita, zvláště pak v případech, kdy se události překrývají a potřeby zdroje se kumulují.

$$\texttt{cumulative}((s_1, d_1, h_1), \ldots, (s_n, d_n, h_n), c) \;\Leftrightarrow$$

$$\Leftrightarrow \textit{ v každém časovém okamžiku } t \textit{ platí, že součet požadavků } \sum_{i \in I} h_i \leq c,$$

kde do množiny indexů I patří indexy i takových událostí, že $t \in \langle s_i; s_i + d_i \rangle$.

Motivace:

Už samotný popis podmínky `cumulative` v sobě skrývá motivační rozvrhovací problém, kdy na stroj kladené požadavky nikdy nesmí překročit jeho výrobní kapacitu.

Příklad:

Uvažujme následujících pět událostí: $(1, 3, 1)$, $(2, 9, 2)$, $(3, 10, 1)$, $(6, 6, 1)$, $(7, 2, 3)$. Stanovme kapacitu zdroje $c = 8$, potom podmínka

$$\texttt{cumulative}(\langle (1, 3, 1), (2, 9, 2), (3, 10, 1), (6, 6, 1), (7, 2, 3) \rangle, 8)$$

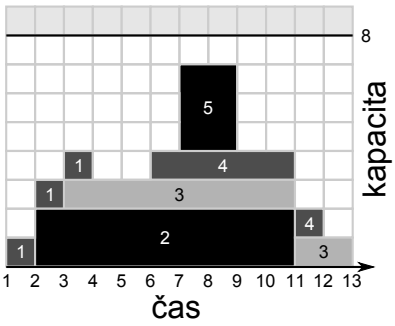
platí, neboť v žádném časovém okamžiku nebyla překročena kapacita zdroje. Rozvržení událostí je zobrazeno na obrázku 2.11.

Příklad:

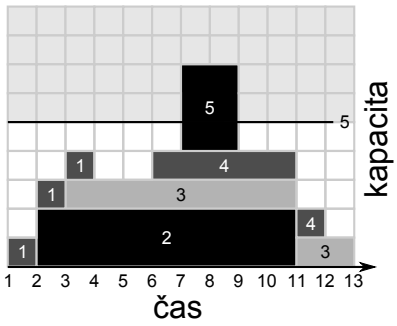
Vezměme události z předchozího příkladu, ale kapacitu zdroje tentokrát nastavme na $c = 5$. Potom podmínka

$$\texttt{cumulative}(\langle (1, 3, 1), (2, 9, 2), (3, 10, 1), (6, 6, 1), (7, 2, 3) \rangle, 5)$$

neplatí, neboť v čase $t = 7$ došlo díky páté události k přetížení zdroje – viz obr. 2.12



Obr. 2.11: zobrazení událostí a kapacity zdroje $c = 8$. Každá událost má své číslo.



Obr. 2.12: zobrazení událostí a kapacity zdroje $c = 5$.

Filtrační algoritmus:

Pro podmínku `cumulative` nemáme vhodný filtrační algoritmus dosahující hranové konzistence nebo konzistence okrajů, který by nebyl NP-úplný. Existuje však metoda, tzv. *constraint relaxing*, kdy podmínku převedeme na jinou („lehčí“), pro kterou už známe efektivní filtrační algoritmus. Pro podmínku `cumulative` je takový postup popsán v [24].

Implementováno v CSP řešičích:

B-Prolog `cumulative`, Ecl^{i}ps^e `cumulative`, Gecode rozšířená varianta pro více zdrojů `cumulatives`, Choco `cumulative`, Koalog `Cumulative`, SICStus Prolog `cumulative` i `cumulatives`.

2.7.3 `disjunctive`

Zdroj: [6].

Mějme n-tici událostí $(s_1, d_1), \ldots, (s_n, d_n)$, kde s_i je počáteční čas události a d_i je délka jejího průběhu. Podmínka `disjunctive` platí, pokud se události s nenulovou délkou průběhu v čase nepřekrývají.

$$\texttt{disjunctive}((s_1, d_1), \ldots, (s_n, d_n)) \;\Leftrightarrow$$

$$\Leftrightarrow\; [\forall i, j \in \{1, \ldots, n\}, i \neq j : (d_i \neq 0 \;\&\; d_j \neq 0) \;\Rightarrow\; (s_i + d_i \leq s_j \vee s_j + d_j \leq s_i)]\,.$$

Příklad:

Uvažujme následující čtveřici událostí: $(1, 3)$, $(2, 0)$, $(7, 2)$ a $(4, 1)$. Potom podmínka

$$\texttt{disjunctive}((1, 3), (2, 0), (7, 2), (4, 1))$$

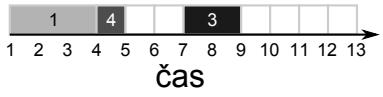
platí, neboť žádná s událostí se nepřekrývá. Sice pro událost 2 je počáteční čas roven 2, tedy čas běhu události 1, ale událost 2 má nulovou délku průběhu a je tak eliminována. Události jsou znázorněny na obrázku 2.13.

Příklad:

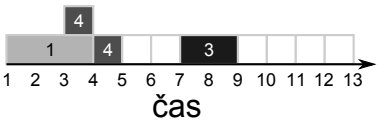
Mějme tyto události: $(1, 3)$, $(7, 2)$ a $(3, 1)$. Potom podmínka

$$\texttt{disjunctive}((1, 3), (7, 2), (3, 1))$$

neplatí, neboť v čase 3 dochází k překryvu událostí 1 a 3, jak je vidět na obrázku 2.14.



Obr. 2.13: nepřekrývání se událostí v čase.



Obr. 2.14: překrývání události 4 s událostí 1.

Filtrační algoritmus:

Pro podmínku `disjunctive` máme čtyři základní metody filtrace. Všechny (i s patřičnými odkazy) jsou popsány v [6].

Implementováno v CSP řešičích:

Ecl^{i}ps^e `disjunctive`, Koalog `Disjunctive`.

2.7.4 `shift`

Zdroj: [6].

Podmínka `shift` na vstupu dostává číselný parametr b, číselný parametr r a n-tici (posloupnost) událostí $(s_1, d_1), \ldots, (s_n, d_n)$, kde s_i je čas začátku události a d_i je doba průběhu události. Dále zaveďme pojem *směna* jako podposloupnost událostí $(s_1, d_1), \ldots, (s_n, d_n)$. Parametr r udává maximální dobu trvání jedné směny a parametr b minimální dobu trvání přestávky mezi dvěma směnami.

Dvě události patří do jedné směny, pokud

- druhá událost začne po skončení první události a to ne později, než v čase b;
- resp. první událost událost začne po skončení první události a to ne později, než v čase b;
- první a druhá událost se v čase překrývají.

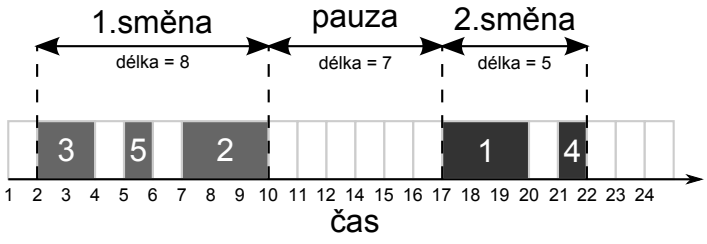
Ještě definujme dobu trvání jedné směny jako rozdíl mezi koncovým časem (tj. časem $s_i + d_i$) poslední události ve směně a počátečním časem první události ve směně.

Podmínka `shift` platí, pokud doba trvání žádné směny nepřekročí čas r.

$$\texttt{shift}(b, r, (s_1, d_1), \ldots, (s_n, d_n)) \;\Leftrightarrow$$
$$\Leftrightarrow\; \forall \textit{ směnu platí, že doba trvání směny } \leq r$$

Použití:

Podmínka `shift` může být použita například v rozvrhování výroby, kdy po určitém počtu vykonaných akcí musí být na stroji provedená kalibrace. V takovém případě číselný parametr r určuje maximální počet využití stroje a parametr b představuje minimální čas potřebný ke kalibraci stroje.



Obr. 2.15: k příkladu podmínky `shift`.

Příklad:

Uvažujme události $(17, 3)$, $(7, 3)$, $(2, 2)$, $(21, 1)$, $(5, 1)$, přičemž maximální délka směny $r = 8$ a minimální délka přestávky je $b = 6$. Potom jsou všechny předpoklady splněny a podmínka

$$\texttt{shift}(6, 8, \langle(17, 3), (7, 3), (2, 2), (21, 1), (5, 1)\rangle)$$

platí. Události rozdělené do směn jsou znázorněny na obrázku 2.15.

2.7.5 `assign and counts`

Zdroj: [6].

Podmínka `assign and counts` má tyto parametry:

1. podmnožinu barev předmětů $C \subseteq \mathcal{C}$, kde $\mathcal{C}$ je množina všech barev předmětů,
2. m přihrádek,
3. n předmětů $x_1, \ldots, x_n$, kde každá proměnná $x_i \in \{1, \ldots, m\}$ představuje umístění předmětu i do přihrádky x_i,
4. funkci $c : \{x_1, \ldots, x_n\} \longrightarrow \mathcal{C}$ přiřazující každému předmětu jeho barvu (barva nemusí být pevně dána, může se v průběhu výpočtu měnit),
5. relaci $\rho \in \{=, \neq, <, \leq, >, \geq\}$ a
6. počet předmětů $k \leq n$.

Máme tedy nějaké obarvené předměty i s jejich umístěním do přihrádek. Podmínka platí, pokud v každé přihrádce je počet předmětů mající nějakou z barev z C v relaci se zadným počtem předmětů k.

Definujme funkci $count : \{1, \ldots, m\} \longrightarrow \mathbb{N}_0$ předpisem

$$count(p) = |\{i \in \{1, \ldots, n\} : x_i = p \ \& \ c(x_i) \in C\}|.$$

Tedy funkce $count(p)$ vrací počet předmětů v přihrádce p, které mají nějakou z barev z C.

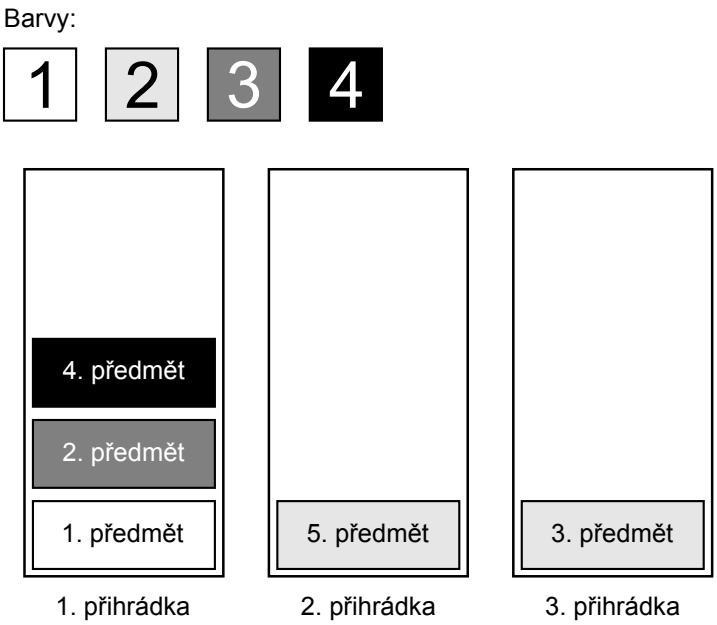
$$\texttt{assing\_and\_counts}(x_1, \ldots, x_n, C, m, c, \rho, k) \ \Leftrightarrow$$
$$\Leftrightarrow \ \forall i \in \{1, \ldots, m\} : \rho(count(i), k)$$

Příklad:

Mějme množinu barev $\mathcal{C} = \{1, 2, 3, 4\}$. Dále mějme $n = 5$ předmětů po řadě obarvených barvami 1, 3, 2, 4 a 2 a umístěných do $m = 3$ přihrádek po řadě 1, 1, 3, 1 a 2. Znázornění situace viz obrázek 2.16.

Nechť $C = \{1, 3\} \subseteq \{1, 2, 3, 4\} = \mathcal{C}$, $\rho = \leq$ a $k = 2$. Potom podmínka

$$\texttt{assign\_and\_counts}(\ \langle 1, 3, 2, 4, 2\rangle\ , \{1, 3\}, 3,$$
$$\langle (x_1, 1), (x_2, 1), (x_3, 3), (x_4, 1), (x_5, 2)\rangle\ , \leq, 2)$$



Obr. 2.16: rozmístění předmětů v přihrádkách a barva předmětů k příkladu funkce `assign and counts`

platí, neboť v první přihrádce jsou dva předměty s barvou z C a to x_1 a x_2. Tedy $2 \leq 2 = k$. V dalších přihrádkách není ani jeden předmět s barvou z C, platí tedy $0 \leq 2 = k$.

Příklad:

Nechť situace je stejná jako v předchozím příkladě, pouze změňme hodnoty v množině: $C = \{1, 3, 4\}$. Potom podmínka

$$\texttt{assign\_and\_counts}(\langle 1, 3, 2, 4, 2\rangle, \{1, 3, 4\}, 3,$$
$$\langle (x_1, 1), (x_2, 1), (x_3, 3), (x_4, 1), (x_5, 2)\rangle, \leq, 2)$$

neplatí, protože v první přihrádce všechny tři předměty mají barvu z množiny C a $3 > 2 = k$.

2.8 Grafové podmínky

Podmínky zařazené v této kategorii sice pracují s grafy, ale není to jediná možná vhodná struktura, nad kterou umí pracovat. Obecně jsou tyto podmínky platné nad libovolnou strukturu typu „bod a jeho následníci“.

V této kategorii je představená zajímavá globální podmínka `nocycle` (2.8.1) a její (skoro) protiklad podmínka `circuit` (2.8.2), kdy prvně jmenovaná platí, pokud graf neobsahuje kružnice a druhá platí, pokud se v grafu nachází kružnice přes všechny vrcholy. Obě dvě podmínky byly využity při modelování problému obchodního cestujícího, což opět ukazuje na pestrost programování s omezujícími podmínkami.

Další podmínky, jako `tree` (2.8.3), `bipartite` (2.8.4) a `symmetric` (2.8.5), jsou uvedeny pouze pro úplnost této kategorie.

2.8.1 `nocycle`

Zdroj: [6, 23].

Zadefinujme si pojem cyklické posloupnosti: mějme uspořádanou posloupnost $S = s_1, \ldots, s_n$, kde $s_i \in \{1, \ldots, n\}$. Definujme množinu C předpisy:

$$\{1\} \in C \text{ a}$$
$$i \in C \Rightarrow s_i \in C.$$

Řekneme, že posloupnost S je *cyklická*, pokud $|C| = n$.

Podmínka `nocycle` na upořádané n-tici proměnných $x_1, \ldots, x_n$ s doménami $x_i \in \{1, \ldots, n\}$ platí, pokud n-tice hodnot $x_1, \ldots, x_n$ není cyklická.

$$\texttt{nocycle}(x_1, \ldots, x_n) \Leftrightarrow x_1, \ldots, x_n \textit{ není cyklická.}$$

Podmínka `nocycle` také platí pro grafy $G = (V, E)$, kde $V = \{x_1, \ldots, x_n\}$ a $E = \{(x_i, x_j) \mid i \in \{1, \ldots, n\}, j \in D_i\}$ a je splněna, pokud přiřazením hodnot do proměnných $x_1, \ldots, x_n$ získáme acyklický graf.

Motivace:

Motivační úlohou pro podmínku `nocyclic` je problém obchodního cestujícího, který chce projít všechna města na mapě tak, aby ušel co nejmenší vzdálenost a žádné město nenavštívil vícekrát.

Příklad:

Nechť proměnné $x_1, \ldots, x_n$ nabývají hodnot $2, 1, 4, 3$. Potom podmínka

$$\texttt{nocycle}(\langle 2, 1, 4, 3 \rangle)$$

platí. Konstruujme množinu C: $\{1\} \in C \Rightarrow x_1 \in C \equiv \{2\} \in C \Rightarrow x_2 \in C \equiv \{1\} \in C$. Tedy $C = \{1, 2\}$ a její mohutnost je $2 < 4 = n$.

Příklad:

Naplňme čtveřici proměnných $x_1, \ldots, x_4$ hodnotami 2,4,1,3. Potom podmínka

$$\texttt{nocycle}(\langle 2, 4, 1, 3 \rangle)$$

neplatí. Konstruujme množinu C: $\{1\} \in C \Rightarrow x_1 \in C \equiv \{2\} \in C \Rightarrow x_2 \in C \equiv \{4\} \in C \Rightarrow x_4 \in C \equiv \{3\} \in C \Rightarrow x_3 \in C \equiv \{1\} \in C$. A tedy $C = \{1, 2, 3, 4\}$ a její mohutnost je $4 = n$.

Filtrační algoritmus:

Filtrační algoritmus docilující hranové konzistence pro podmínku `nocycle` by měl exponciální složitost – de-facto by šlo o nalezení Hamiltonovské cesty v grafu. V [23] byl ale prezentován filtrační algoritmus nedosahující sice takové účinnosti, ale který se snaží z grafu odstranit všechny hrany, které by mohly v budoucnu vytvořit kružnici.

2.8.2 `circuit`

Zdroj: [6].

Podmínka `circuit` má pouze jeden parametr a to orientovaný graf G. Platí, pokud v grafu G existuje Hamiltonovská kružnice, tedy kružnice, která obsahuje všechny vrcholy grafu G.

$$\texttt{circuit}(G) \Leftrightarrow \exists \textit{ Hamiltonovská kružnice} \in G.$$

Motivace:

K Hamiltonovské kružnici se váže kombinatorický problém obchodního cestujícího, který by chtěl navštívit všechna města v okolí tak, aby ušel co nejkratší vzdálenost a v žádném městě se neocitl více jak jednou.

Příklad:

Mějme následující graf na pěti vrcholech $v_1, \ldots, v_5$ s množinou hran $E = (v_1, v_2)$, (v_2, v_3), (v_3, v_4), (v_4, v_2) a (v_4, v_1). Potom podmínka

$$\texttt{circuit}(\{v_1, v_2, v_3, v_4, v_5\}, \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_2), (v_4, v_1)\})$$

platí, neboť v grafu existuje kružnice přes všechny vrcholy, a to (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1).

2.8.3 `tree`

Zdroj: [6].

Mějme libovolný (může být i nesouvislý) orientovaný graf G a celočíselnou proměnnou $n \in \mathbb{N}_0$. Podmínka `tree` platí, pokud graf G dokážeme pokrýt n libovolně rozvětvenými stromy tak, aby každý vrchol grafu G patřil do právě jednoho stromu a orientované hrany měly směr od listů ke kořeni každého stromu.

$$\begin{aligned}\texttt{tree}(n, G) \Leftrightarrow\ & \exists \textit{ pokrytí grafu } G = (V, E) \textit{ } n \textit{ stromy } T_1 = (V_1, E_1), \ldots, \\ & T_n = (V_n, E_n), \textit{ kde } V_i \subseteq V \textit{ a } E_i \subseteq (V_i \times V_i) \cup E, \textit{ takové,} \\ & \textit{že } V_1 \cap \ldots \cap V_n = \emptyset, V_1 \cup \ldots \cup V_n = V \\ & \textit{a } \forall e \in E_i \textit{ má směr ke kořeni stromu } T_i.\end{aligned}$$

2.8.4 `bipartite`

Zdroj: [6].

Podmínka platí, pokud v zadaném orientovaném grafu G existuje symetrický bipartitní podgraf. Tj. takový podgraf, ve kterém lze množinu vrcholů rozdělit na dvě disjunktní podmnožiny V_1 a V_2 takové, že každá hrana spojuje vrchol z V_1 s vrcholem z V_2 a ke každé hraně (v_i, v_j) existuje v podgrafu opačná hrana (v_j, v_i).

$$\texttt{bipartite}(G) \Leftrightarrow \textit{v } G \textit{ existuje symetrický bipartitní podgraf.}$$

2.8.5 `symmetric`

Zdroj: [6, 23].

Podmínka `symmetric` na vstupu dostává orientovaný graf G a platí, pokud v grafu G existuje symetrický podgraf, tj. ke každé orientované hraně (v_i, v_j) v podgrafu existuje opačná hrana (v_j, v_i).

$$\texttt{symmetric}(G) \Leftrightarrow \textit{v } G \textit{ existuje symetrický podgraf.}$$

Filtrační algoritmus:

Filtrační algoritmus asymptotické složitosti $O(n + m)$, kde n je počet vrcholů a m je počet hran grafu G, je možno nalézt v [23]. Z grafu G odstraňuje všechny hrany (v_i, v_j), pro které neexistují opačné hrany (v_j, v_i).

2.9 Podmínky pro automaty a gramatiky

Tato kategorie obsahuje dvě podmínky svým zaměřením zasahující do toerie automatů a gramatik. První z nich, podmínka `regular` (2.9.1), pracuje s konečnými automaty a rozhoduje, zda daná slova patří do jazyka generovaného daným automatem.

Podmínka `grammar` (2.9.2) testuje příslušnosti slov do jazyků generovanými bezkontextovými gramatikami.

Příslušné pojmy z teorie automatů a gramatik budou vysvětleny zvlášť u každé podmínky.

2.9.1 `regular`

Zdroj: [32].

Pro vysvětlení funkce podmínky `regular` si nejprve připomeňme definici konečného automatu. *Konečným automatem* nazýváme pětici $A = (Q, X, \delta, q_0, F)$, kde Q je konečná neprázdná množina stavů, X konečná neprázdná množina symbolů, $\delta : Q \times X \longrightarrow Q$ přechodová funkce, $q_0 \in Q$ počáteční stav a $F \subseteq Q$ množina přijímacích stavů. Symbolem $L(A)$ označme všechna slova přijímaná konečným automatem A.

Více o automatech je možné nalézt v [5].

Podmínka `regular` má za parametry n-tici proměnných $x_1, \ldots, x_n$ s doménami $D_i \subseteq X$ a automat A. Testuje, zda slovo patří do jazyka rozpoznávaného konečným automatem, tj. zda platí $x_1 x_2 \ldots x_n \in L(A)$.

$$\texttt{regular}(x_1, \ldots, x_n, A) \;\Leftrightarrow\; x_1 x_2 \ldots x_n \in L(A).$$

Motivace:

Regulární jazyky jsou dobrým kompromisem mezi výpočetní složitostí a vyjadřovací schopností. Příkladem využití může být například Car Sequencinq Problem [19], kdy máme za úkol rozplánovat výrobu aut pro montážní linku s ohledem na individuální odlišnosti každého vozu (např. barva laku).

Příklad:

Mějme pětici proměnných $x_1, \ldots, x_5$ s doménami $D_1 = \ldots = D_5 = X = \{a, b, c\}$. Dále mějme regulární výraz $aa^*bb^* + c^*$ zobrazený automatem na obrázku 2.17. Potom pro n-tici hodnot a, a, b, b, a podmínka

$$\texttt{regular}(\langle a, a, b, b, a\rangle, aa^*bb^* + c^*)$$

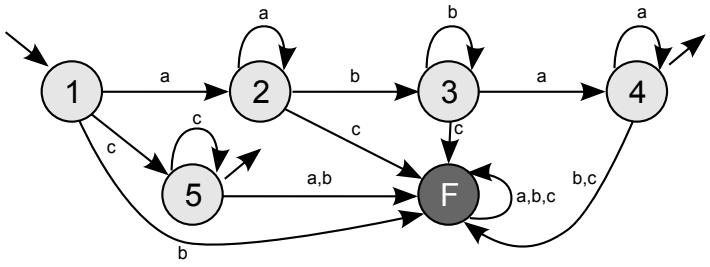
platí, neboť jdeme postupně přes stavy 1, 2, 2, 3, 3 do výstupního stavu 4.

Příklad:

Uvažujme stejný automat jako v předchozím příkladě. Pro n-tici hodnot a, b, b, b, c podmínka

$$\texttt{regular}(\langle a, b, b, b, c\rangle, aa^*bb^* + c^*)$$

neplatí, neboť jdeme-li přes stavy 1, 2, 3, 3 skončíme ve stavu 3, odkud přes znak c se dostáváme do chybového stavu.



Obr. 2.17: konečný automat odpovídající regulárnímu výrazu $aa^*bb^* + c^*$.

Filtrační algoritmus:

Filtrační algoritmus pro podmínku `regular` splňující hranovou konzistenci je možno nalézt v [32].

Základní myšlenkou algoritmu je sestavení n-vrstevnatého orientovaného grafu z automatu A, kde pro každou vrstvu vrcholy grafy jsou stavy automatu. Poté v dopředném směru od počátečního stavu probíhá „výstavba“ orientovaných cest vždy do další vrstvy. Při ukončení stavby cest dojde ve zpětném směru k eliminaci v koncovém stavu neukončených cest a následně odstranění některých hodnot z domén proměnných.

Implementováno v CSP řešičích:

Gecode impementuje podmínku pro práci s konečnými automaty `dfa`.

2.9.2 `grammar`

Zdroj: [35].

Stručně připomeňme pojem gramatiky. *Gramatika* je čtveřice $G = (V_N, V_T, S, P)$, kde V_N je konečná neprázdná množina neterminálních symbolů, V_T konečná neprázdná množina terminálních symbolů, $V_N \cap V_T = \emptyset$, $S \in V_N$ počáteční neterminální symbol a P systém přepisovacích pravidel $u \to v$, kde $u, v \in (V_N \cup V_T)^*$ a u obsahuje alespoň jeden neterminální symbol z V_N. Symbolem $L(G)$ označme jazyk generovaný gramatikou G.

Gramatika G je *bezkontextová*, pokud obsahuje pouze přepisovací pravidla ve tvaru $X \to w$, kde $X \in V_N$ a $w \in (V_N \cup V_T)^*$.

Více o gramatikách lze nalézt např. v [5].

Podmínka `grammar` zjišťuje, zda posloupnost znaků je slovem z jazyka generovaného bezkontextovou gramatikou. Na vstupu dostává bezkontextovou gramatiku G a n-tici proměnných $x_1, \ldots, x_n$ s doménami $D_i \subseteq V_T$. Platí, pokud slovo složené ze znaků $w = x_1 x_2 \ldots x_n$ je slovem z jazyka generovaného gramatikou G, tj. $w \in L(G)$.

$$\texttt{grammar}(x_1, \ldots, x_n, G) \;\Leftrightarrow\; x_1 x_2 \ldots x_n \in L(G).$$

Motivace:

Podmínka `grammar` pro bezkontextové gramatiky může být užitečná v případech, kdy chceme nejít rekurzivní posloupnost nějakých do sebe vnořených objektů. Příkladem může být matematická hádanka, kdy máme sestavit ze dvou znaků 3, ze dvou znaků 8, operací $+$, $-$, $\cdot$, $/$ a závorek $(,)$ takový aritmetický výraz, aby výsledek byl roven číslu 24. Tento problém samozřejmě patří do kategorie NP-úplných, ale podmínka `grammar` nám umožní jednoduše zachytit syntaktickou formu aritmetického výrazu.

Příklad:

Využijme zadání motivační úlohy a mějme gramatiku $G = \{V_N, V_T, S, P)$ takovou, že

$$V_N = \{E, T, F\},$$
$$V_T = \{3, 8, -, /, (,)\},$$
$$S = E,$$
$$P = \{E \to E - T, E \to T, T \to T/F, T \to F, F \to (E), F \to 3, F \to 8\}.$$

Mějme nyní posloupnost jedenácti proměnných $x_1, \ldots, x_{11}$ po řadě nabývající hodnoty $8, /, (, 3, -, (8, /, 3,))$. Potom podmínka

$$\texttt{grammar}(\langle 8, /, (, 3, -, (8, /, 3,))\rangle, G)$$

platí, neboť slovo $8/(3-(8/3))$ je z jazyka generovaného gramatikou G. Shodou okolností je uvedený výraz i řešením hádanky $8/(3-(8/3)) = 24$. Existuje minimálně ještě jedno řešení, které je možno nalézt v [31].

Příklad:

Pro ilustraci neplatnosti podmínky `grammar` zvolme nyní jednoduchou dvojici proměnných $x_1 = ($ a $x_2 =)$. Potom podmínka

$$\texttt{grammar}(\langle (,)\rangle, G)$$

neplatí, neboť slovo $()$ není z jazyka $L(G)$ – neterminál F se „neumí“ přepsat na prázdné slovo λ.

Filtrační algoritmus:

K filtraci a docílení hranové konzistence je využíván algoritmus založený na algoritmu Cocke-Younger-Kasami (CYK) pro zjišťování příslušnosti slova do bezkontextové gramatiky v Chomského normální formálně. Jeho úplné znění je možno nalézt v [35].

2.10 Podmínky bez zařazení

V této kategorii se objevují podmínky, kterou jsou jedinečné a něčím výjimečné, jako například podmínka `discrepancy` (2.10.1), která v „běžném“ prohledávání nasimuluje prohledávání technikou LDS.

Podmínka `global contiguity` (2.10.2) hledá v binárním vektoru ucelenou posloupnost jedniček a podmínka `crossing` (2.10.3) v uvozovkách „přináší spojitost do diskrétního světa“.

2.10.1 `discrepancy`

Zdroj: [23].

Mějme číslo k a posloupnost n proměnných $x_1, \ldots, x_n$ nabývajících hodnot z domény D a posloupnost n podmnožin nepřípustných hodnot $B_1, \ldots, B_n \subseteq D$. Podmínka `discrepancy` platí, pokud počet proměnných x_i, které v tomto okamžiku mají hodnotu z příslušné množiny nepřípustných hodnot B_i je roven k.

$$\texttt{discrepancy}(x_1, \ldots, x_n, B_1, \ldots, B_n, k) \;\Leftrightarrow\; |\{i \in \{1, \ldots, n\} \mid x_i \in B_i\}| = k$$

Motivace:

Vedle v odstavci 1.4 zmíněných prohledávacích technik backtracking a branch-and-bound existuje algoritmus LDS (Limited Discrepancy Search) popsaný v [23]. Podmínka `discrepancy` přidává funkčnost tohoto prohledávacího algoritmu do „běžného“ hledání algoritmem backtracking.

V práci [23] je také navíc provedeno experimentální srovnání použití podmínky `discrepancy` a „běžného“ prohledávacího algoritmu s prohledáváním algoritmem LDS na variantě problému obchodního cestujícího. O něco lépe vychází použití podmínky `discrepancy`.

Příklad:

Nechť proměnné $x_1, \ldots, x_4$ nabývají hodnot 3, 2, 8, 5 a parametr $k = 2$. Stanovme množiny nepřípustných hodnot následovně: $B_1 = \{4, 2, 7\}$, $B_2 = \{1, 2\}$, $B_3 = \{8, 3, 7\}$ a $B_4 = \emptyset$. Potom podmínka

$$\texttt{discrepancy}(\langle 3, 2, 8, 5\rangle, \langle\{4, 2, 7\}, \{1, 2\}, \{8, 3, 7\}, \emptyset\rangle, 2)$$

platí, neboť počet proměnných x_i, které nabývají nějaké hodnoty z B_i je $2 = k$. Jde o proměnné $x_2 = 2 \in \{1, 2\}$ a $x_3 = 8 \in \{8, 3, 7\}$.

Příklad:

Uvažujme proměnné $x_1, \ldots, x_4$. Všechny množiny nepřípustných hodnot nechť jsou prázdné, tj. $B_1 = \ldots = B_4 = \emptyset$. Potom při nastavení parametru k na libovolnou nenulovou hodnotu, např. 1, podmínka

$$\texttt{discrepancy}(\langle 3, 2, 8, 5\rangle, \langle\emptyset, \emptyset, \emptyset, \emptyset\rangle, 1)$$

nebude splněna při žádném ohodnocení proměnných.

2.10.2 `global contiguity`

Zdroj: [6].

Podmínka `global contiguity` zachází s n-ticí proměnných $x_1, \ldots, x_n$ nabývající hodnoty z dvouprvkové domény $D = \{0, 1\}$. Dívá se na všechny výskyty hodnot 1 v n-tici proměnných a platí, pokud se všechny 1 vyskytují v n-tici vedle sebe, aniž by byly přerušeny nějakou 0.

`global_contiguity`$(x_1, \ldots, x_n)$ $\Leftrightarrow$ *existuje-li v posloupnosti* $x_1, \ldots, x_n$ *souvislá podposloupnost* $x_{c_1}, \ldots, x_{c_m}$ *taková, že* $\forall i \in \{1, \ldots, m\} : x_{c_i} = 1$*, pak taková podposloupnost je jen jedna a ostatní proměnné, které do této podposloupnosti nepatří, mají hodnotu 0.*

Příklad:

Mějme zadánu tuto posloupnost 0 a 1: 001100. Potom podmínka

`global_contiguity`$(0, 0, 1, 1, 0, 0)$

platí, neboť v celé posloupnosti existuje pouze jedna jediná souvislá podposloupnost složená ze samých jedniček, a to mezi pozicemi 3 a 4.

Příklad:

Uvažujme posloupnost 0 a 1: 0100110. Potom podmínka

`global_contiguity`$(0, 1, 0, 0, 1, 1, 0)$

neplatí, protože v posloupnosti 0100110 existují dvě souvislé podposloupnosti složené jen z jedniček – jednoprvková na pozici 2 a dvouprvková mezi pozicemi 5 a 6.

Filtrační algoritmus:

Filtrační algoritmy docilující různých forem lokální konzistence je možné nalézt v článku [30].

2.10.3 `crossing`

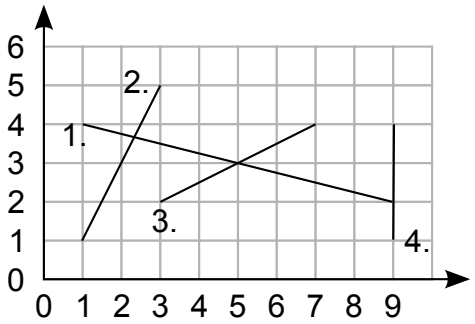
Zdroj: [6].

Mějme ve dvojrozměrném prostoru množinu úseček $P = \{(x_{1,1}, y_{1,1}, x_{1,2}, y_{1,2}), \ldots, (x_{n,1}, y_{n,1}, x_{n,2}, y_{n,2})\}$, kde $x_{i,1}, y_{i,1}$ jsou souřadnice počátečního bodu a $x_{i,2}, y_{i,2}$ jsou souřadnice koncového bodu i-té úsečky. Dále mějme nezáporné celé číslo k. Podmínka `crossing` platí, pokud počet průniků úseček z P je roven k.

$$\texttt{crossing}(k, (x_{1,1}, y_{1,1}, x_{1,2}, y_{1,2}), \ldots, (x_{n,1}, y_{n,1}, x_{n,2}, y_{n,2})) \Leftrightarrow$$
$$\Leftrightarrow |\{(x_{i,1}, y_{i,1}, x_{i,2}, y_{i,2}) \cap (x_{j,1}, y_{j,1}, x_{j,2}, y_{j,2}) \mid i, j \in \{1, \ldots, n\}\}| = k,$$

kde $(x_{i,1}, y_{i,1}, x_{i,2}, y_{i,2}) \cap (x_{j,1}, y_{j,1}, x_{j,2}, y_{j,2})$ znamená průnik úseček.

Podmínka `crossing` byla vybrána z toho důvodu, že i když domény proměnných jsou diskrétní, tak počítá průnik úseček ve spojitém prostoru.



Obr. 2.18: rozvržení přímek z příkladu k podmínce `crossing`.

Příklad:

Mějme k dispozici tyto čtyři úsečky: $(1, 4, 9, 2)$, $(1, 1, 3, 5)$, $(3, 2, 7, 4)$ a $(9, 1, 9, 4)$. Situace je znázorněná na obrázku 2.18.

Podmínka

$$\texttt{crossing}(3, \langle(1, 4, 9, 2), (1, 1, 3, 5), (3, 2, 7, 4), (9, 1, 9, 4)\rangle)$$

při hodnotě parametru $k = 3$ platí, neboť, jak je z obrázku 2.18 patrno, dochází mezi přímkami ke třem křížením (mezi 1. a 2. úsečkou, mezi 1. a 3. úsečkou a mezi 3. a 4. úsečkou).

Závěr

V práci byla představena disciplína teoretické informatiky programování s omezujícími podmínkami. Byly formálně zadefinovány důležité pojmy a ukázána metoda řešení problémů splňování omezujících podmínek užitím algoritmů backtracking a branch-and-bound. V závěru první části této práce byla nastíněna technika propagace podmínek a lokální konzistence, čímž bylo plynule navázáno na problematiku globálních podmínek.

Druhá část práce formou katalogu představila výběr užitečných reálně využitelných globálních podmínek a pokusila se vytvořit rozcestník pro případné zájemce o podrobnější studium globálních podmínek.

Programování s omezujícími podmínkami spolu s využíváním globálních podmínek je silnou metodou pro řešení problémů nejen kombinatorického rázu. Vedle generalizovatelných globálních podmínek, jaké byly např. uvedeny v této práci, vzniká i mnoho globálních podmínek a filtračních algoritmů určených pouze pro jednorázové řešení konkrétních úloh. To jen dokazuje, že koncept globálních podmínek má smysl.

Jak již bylo zmíněno, cílem této práce bylo popsat globální podmínky využitelné v reálných úlohách. Vedle těchto podmínek však ještě existují takové, které byly formulovány, experimentálně implementovány, ale zatím se vyskytují jen v teoretických pracích, kde na své opravdové využití teprve čekají. I v těch se skrývají zajímavé myšlenky, a jejich soupis by mohl být námětem pro další práci.

Index

Literatura

[1] K. R. Apt: *Principles of Constraint Programming*, Cambridge, 2003.

[2] K. Artiouchine, P. Baptiste: *Inter-distance Constraint: An Extension of the All-Different Constraint for Scheduling Equal Length Jobs*
CP 2005, LNCS 3709, str. 62–76, 2005.

[3] K. Artiouchine, P. Baptiste, C. Dürr:
Runway Sequencing with Holding Patterns.
`http://www.lix.polytechnique.fr/Labo/Konstantin.Artiouchine/ejor04.pdf`.

[4] R. Barták: *On-line Guide to Constraint Programming*, 1998,
`http://kti.mff.cuni.cz/~bartak/constraints/`.

[5] R. Barták: *studijní materiály k přednášce Automatiky a gramatiky*, MFF UK, 2008, `http://ktiml.ms.mff.cuni.cz/ bartak/automaty/`.

[6] N. Beldiceanu: *Global Constraint Catalog*
`http://www.emn.fr/x-info/sdemasse/gccat/`, 2008.

[7] N. Beldiceanu:
Pruning for the Minimum Constraint Family and for the Number of Distinct Values Constraint Family,
CP 2001, LNCS 2239, str. 211-224, 2001.

[8] N. Beldiceanu, I. Katriel, S. Thiel:
Filtering Algorithms for the Same and UsedBy Constraints,
Technical report 2004/1/001 MPI 2004,
`http://domino.mpi-sb.mpg.de/internet/reports.nsf/NumberView/2004-1-001`.

[9] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, T. Walsh:
Among, common and disjoint Constraints,
CSCLP 2005, str. 223-235, 2005.

[10] C. Bessière, E. Hebrard, B. Hnich, T. Walsh:
Disjoint, Partition and Intersection Constraints for Set and Multiset Variables,
CP 2004, LNCS 3258, str. 138–152, 2004.

[11] P. Brisset a kol.: *ECL^{i}PS^e Constraint Library Manual*, 2008,
`http://www.eclipse-clp.org/`.

[12] R. Dechter: *Constraint Processing*, Morgan Kaufmann Publishers, 2003.

[13] D. Diaz: *GNU Prolog*, 2007,
`http://www.gprolog.org/manual/gprolog.pdf`.

[14] Doctors Without Borders,
`http://www.doctorswithoutborders.org`

[15] D. Duchier, L. Kornstaedt, M. Homik, T. Müller, Ch. Schulte, P. V. Roy:
Mozart Documentation, 2008,
`http://www.mozart-oz.org/documentation/`.

[16] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh:
Breaking Row and Column Symmetries in Matrix Models
CP 2002, LNCS 2470, str. 462–477, 2002.

[17] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel1, T. Walsh:
Global Constraints for Lexicographic Orderings,
CP 2002, LNCS 2470, str. 93–108, 2002.

[18] Gecode Reference Documentation, 2008,
`http://www.gecode.org/gecode-doc-latest/index.html`

[19] I. P. Gent: *Two Results on Car-sequencing Problems*,
Report APES-02-1998, duben 1998.

[20] Y. Hamadi: *Disolver 3.0: the Distributed Constraint Solver*,
`http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf`.

[21] P. V. Hentenryck: *Constraint Satisfaction in Logic Programming*,
MIT Press, 1989.

[22] P. V. Hentenryck, Laurent Michel: *Comet API Documentation*, 2008,
`http://www.cs.brown.edu/people/pvh/CometDoc/www/api/api.html`.

[23] W.-J. van Hoeve: *Operations Research Techniques in Constraint Programming*,
Ph. D. Thesis University of Amsterdam, CWI 2005.

[24] J. N. Hooker, H. Yan: *A Relaxation of the Cumulative Constraint*
CP 2002, LNCS 2470, str. 686–691, 2002.

[25] J. E. Hopcroft, R. M. Karp:
An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs,
SIAM Journal on Computing, 2(4):225–231, 1973.

[26] T. Hulubei: *The CSP Library*, 2003,
`http://www.hulubei.net/tudor/csp/doc/index.html`.

[27] Choco User Guide, 2008,
`http://choco-solver.net/index.php?title=User_guide`.

[28] I. Katriel, S. Thiel: *Fast Bound Consistency for the Global Cardinality Constraint*
CP 2003, LNCS 2833, str. 437–451, 2003.

[29] Koalog Constraint Solver™ API documentation, 2007,
`http://www.koalog.com/resources/doc/jcs-javadoc.jar`.

[30] M. J. Maher: *Analysis of a Global Contiguity Contraint*
In Workshop on Rule-Based Constraint Reasoning and Programming, 2002.

[31] Mathisfun: *24 from 8,8,3,3 - Solution*, 2006,
`http://www.mathsisfun.com/puzzles/24-from-8-8-3-3-solution.html`

[32] G. Pesant: *A Regular Language Membership Constraint for Finite Sequences of Variables*
CP 2004, LNCS 3258, str. 482–495, 2004.

[33] Y. Richter, A. Freund, Y. Naveh: *Generalizing AllDifferent: The SomeDifferent Constraint*
CP 2006, LNCS 4204, str. 468–483, 2006.

[34] F. Rossi, P. V. Beek, T. Walsh: *Handbook of Constraint Programming*,
Elsevier, 2006.

[35] M. Sellmann: *The Theory of Grammar Constraints*,
CP 2006, LNCS 4204, str. 530–544, 2006.

[36] SICStus Prolog Documentation and Manuals, 2008,
`http://www.sics.se/isl/sicstuswww/site/documentation.html`.

[37] P. Shaw: *A Constraint for Bin Packing*,
CP 2004, LNCS 3258, str. 648–662, 2004.

[38] J. B. Shearer: `Golomb rulers`,
`http://www.research.ibm.com/people/s/shearer/grule.html`

[39] R. Tarjan: *Depth-first search and linear graph algorithms.*
SIAM Journal on Computing, 1:146–160, 1972.

[40] Wikipedia: *Lexikografické uspořádání*,
`http://cs.wikipedia.org/wiki/Lexikografické_uspořádání`.

[41] R. Yang: *Solving a workforce managementr problem with constraint programming*,
The 2^{nd} International Conference on The Practical Application of Constraint Technology, str. 373-387, 1996.

[42] J. Zhou: *A Permutation-Based Approach for Solving the Job-Shop Problem* Constraints: An International Journal, 2, str. 185–213, 1997.

[43] Neng-Fa Zhou: *B-Prolog Users's Manual*, 2008,
`http://www.probp.com/download/manual.pdf`.

[44] Neng-Fa Zhou: *Channel Routing with Constraint Logic Programming and Delay*, Gordon and Breach Science Publishers, str. 217-231, 1996.