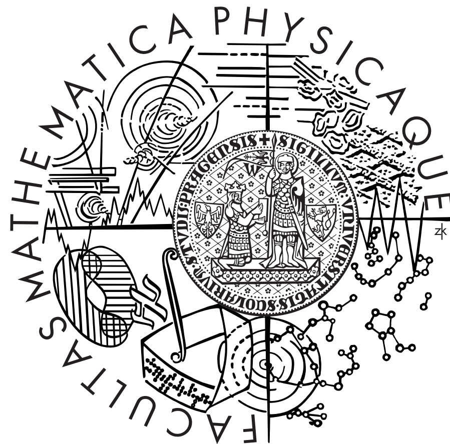
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Tomáš Kasl

Řešení příliš omezených problémů

Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: Doc. RNDr. Roman Barták, Ph.D.
Studijní program: Informatika, Softwarové systémy

Poděkování

Na tomto místě bych chtěl poděkovat svému vedoucímu diplomové práce, Doc. RNDr. Romanu Bartákovi, Ph.D., za nezměrnou trpělivost, cenné rady a pomoc při tvorbě této práce.

Chtěl bych také poděkovat RNDr. Petru Vilímovi, za nápady a pomoc s *gdb* a LaTeXem. Mé díky patří také Janě a Mgr. Jiřině Běželovým za korekturu češtiny.

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

V Praze dne 18. ledna 2006

Tomáš Kasl

Obsah

1 Úvod **1**

2 Hierarchie podmínek **3**

2.1 Omezující podmínky . . . 3
2.2 Hierarchie . . . 5
2.3 Řešení hierarchie . . . 6
2.4 Komparátory . . . 7
2.4.1 Chybová funkce . . . 7
2.4.2 Lokální versus globální . . . 8
2.4.3 Konkrétní komparátory . . . 10
2.5 Cykly . . . 11
2.6 Rodina modrých algoritmů . . . 12
2.7 Deltablue . . . 12
2.7.1 Řešící graf . . . 12
2.7.2 Průchozí preference . . . 14
2.7.3 Hledání řešení . . . 15
2.7.4 Vlastnosti . . . 16
2.8 Indigo . . . 16
2.8.1 Slabá a silná verze . . . 17
2.8.2 Použitý komparátor . . . 17
2.8.3 Vlastnosti řešených hierarchií . . . 18
2.8.4 Slabá implementace . . . 19
2.8.5 Neúplnost . . . 23
2.8.6 Žádné řešení . . . 24
2.8.7 Složitost . . . 24
2.8.8 Použití *locally-predicate-better* . . . 25

3 Alternativní teorie **26**
3.1 Komparátory 26
3.2 Lokální a globální komparátory 27
3.3 Řešení hierarchie 28
3.4 Posloupnosti buněk 29
3.5 Věty o korektnosti a úplnosti 30
3.6 Buňky a sítě 31
3.7 Existující algoritmy 34
3.7.1 Straton 34
3.7.2 Straton 2 35
3.7.3 BlueOne 35

4 Nový algoritmus - Straton 3 **37**
4.1 Koncepce 38
4.1.1 Budování sítě 38
4.1.2 Hledání řešení 39
4.2 Použitá omezení 40
4.2.1 Metody s jedním výstupem 40
4.2.2 Read-only proměnné 41
4.2.3 Neostré nerovnosti 42
4.2.4 Podmínky stay 43
4.3 Zobrazení sítě podmínek 44
4.4 Domény 47
4.5 Cykly 50
4.6 Metafunkční buňky 56
4.7 Fáze budování sítě 61
4.7.1 Algoritmus 62
4.7.2 Stay podmínky a buňky 64
4.7.3 Přidávání podmínky 65
4.7.4 enforce_6 68
4.7.5 enforce_7 73
4.7.6 Složitost budování sítě 79
4.7.7 Korektnost budování sítě 80
4.8 Propagační fáze 83
4.8.1 Reprezentace domén 84
4.8.2 Globální komparátory 85
4.8.3 Implementovaný komparátor 86
4.8.4 Složitost propagační fáze 86

4.8.5 Korektnost 87
4.8.6 (Ne)úplnost 87
4.8.7 Omezené hierarchie 89
4.8.8 Další řešení 90
4.9 Implementace 91
4.9.1 Experimenty 91

5 Závěr **96**

A Implementace a struktura dat **97**

B Obsah CD **103**

Název práce : Řešení příliš omezených problémů
Autor : Tomáš Kasl
Katedra : Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce : Doc. RNDr. Roman Barták, Ph.D.
e-mail vedoucího : bartak@kti.mff.cuni.cz
Abstrakt : Hierarchie omezujících podmínek jsou jednou z metod řešení příliš omezených problémů, což jsou problémy splňování podmínek, ve kterých není možné splnit všechny podmínky. Hierarchie vzniknou rozšířením "plochých" omezujících podmínek o preference, které vyjadřují, jak důležité je splnění konkrétní podmínky.Většina existujících řešičů hierarchií se dá rozdělit mezi algoritmy lokální propagace, které jsou většinou limitovány na podmínky typu rovnost, a zjemňující algoritmy, které jsou neinkrementální a málo efektivní. Obecná teorie sítí podmínek spojuje oba přístupy, ale dosud neexistuje algoritmus , který by plně využil výhod této teorie. Tato práce navrhuje nový algoritmus,který by tuto teorii využil.
Klíčová slova : příliš omezené problémy, hierarchie podmínek, buňka podmínek, síť podmínek

Title : Finding solution of over-constrained problems
Author : Tomáš Kasl
Department : Department of Theoretical Computer Science and Mathematical Logic
Supervisor : Doc. RNDr. Roman Barták, Ph.D.
Supervisor's e-mail address : bartak@kti.mff.cuni.cz
Abstract : Constraint hierarchies belong to techniques for solving over-constrained problems, that is, constraint satisfaction problems where it is not possible to satisfy all the constraints. The idea of constraint hierarchy is to label each constraint by a preference level describing how much the constraint should be satisfied. Currently, there exist two classes of constraint hierarchy solvers: local propagation solvers that can handle more or less equality constraints only and refining solvers that are a bit cumbersome and non-incremental. The theoretical framework of constraint hierarchy solvers combines advantages of both above mentioned classes but no algorithm exploiting the power of this framework has been proposed so far. The thesis describes a new algorithm for solving constraint hierarchies based on this framework.
Keywords : over-constrained systems, constraint hierarchy, constraint cell, hierarchy network

Kapitola 1

Úvod

Omezující podmínky jsou relace, popisující vztah mezi proměnnými. Pro vyřešení nějakého problému je zapotřebí tento problém popsat pomocí omezujících podímnek a mít k dispozici nějaký systém, který popis problému převede z implicitní podoby (seznam podmínek) do podoby explicitní(ohodnocení proměnných), což je to, co si obvykle představíme pod pojmem "řešení".

Programování s omezujícími podmínkami (Constraint Programming) může narazit na situaci, kdy řešení systému podmínek neexistuje, protože podmínky jsou ve vzájemném konfliktu. Ohlásit v takovém případě chybu a požadovat změnu zadání nemusí být vždy přijatelné řešení. Jednou z cest, jak tento problém překonat, jsou hierarchie podmínek. Jejich přístup je založen na pozorování, že ne všechny podmínky jsou stejně "důležité". Je tedy možné rozšířit zadání problému tak, aby zahrnovalo "důležitost splnění" jednotlivých podmínek.

Příkladem ze života je provoz letiště. Letadla jsou odbavována pomocí jednotlivých portů na terminálech. Podmínka, která říká, že v jednu chvíli může být u jednoho portu pouze jedno letadlo, musí být splněna vždy, pokud nemá dojít ke katastrofě. Naopak nesplnění podmínky, že každé letadlo má být obslouženo u portu nejblíže k odbavovací hale, nezpůsobí žádnou pohromu. Pomocí takové podmínky lze určit, které řešení "nutných" podmínek je lepší. Dalším zjemněním pak může být požadavek pilotů, aby byli navedeni k terminálu, kde je bar pro posádky. Tahle podmínka ovlivní výběr řešení pouze tehdy, když více řešení "nutných" a "důležitějších" podmínek bude "stejně dobrých".

Obsahem této práce je návrh nového algoritmu Straton 3 pro řešení hierarchií omezujících podmínek. Tento algoritmus vychází z "Obecného systému pro řešení hierarchií omezujících podmínek", což je celý název teorie popisující sítě podmínek v práci [1].

Síť podmínek je graf, který zachycuje vztahy mezi jednotlivými podmínkami. Tento graf pak definuje částečné uspořádání podmínek určující, v jakém pořadí podmínky splňovat.

Existují již dva algoritmy pro stavbu sítě podmínek, Straton 1 a 2, ale jimi postavené sítě jsou z hlediska "efektivního hledání" řešení dost nevýhodné. Naopak, algoritmus popsaný v této práci staví "dobrou" síť. Cena, kterou za to platí, je neinkrementalita, takže při změně podmínek je třeba postavit celou síť znovu. Výhodou je naopak nezávislost "stavby sítě" a "hledání řešení". V systémech, kde se nemění "struktura hierarchie", ale dochází pouze k opakované změně některých "hodnot" je možné pomocí dříve postavené sítě opakovaně hledat řešení.

Členění práce

Jelikož nejsou od čtenáře očekávány znalosti této problematiky, je v následující kapitole shrnuto dnes již "klasické" zavedení hierarchií omezujících podmínek. Na jejím konci jsou popsány dva algoritmy, na jejichž myšlenkách Straton 3 staví.

Ve třetí kapitole je popsáno "alternativní" zavedení hierarchií podmínek a jejich řešení, tak jak je učiněno v [1]. V této práci je popsána teorie, na které je postaven algoritmus Straton 3 . Proto jsou zde přepsány nejdůležitější definice a věty, na které se odvolává popis algoritmu. Na konci kapitoly jsou popsány již existující algoritmy pro stavbu sítě podmínek.

Kapitola čtyři popisuje samotný algoritmus Straton 3 a je novým přínosem práce. Jsou zde rozebírány požadavky jak na samotné podmínky, tak na celou hierarchii. Mluví se zde o typech podmínek, doménách proměnných, cyklech podmínek. Oproti teorii popsané v předchozí kapitole je zde zavedena jedna malá změna - *metafunkční* buňky. Následuje popis samotného algoritmu, který má dvě fáze. První z nich je vybudování *sítě podmínek* ze zadané hierarchie. Druhá popisuje, jak lze pomocí této sítě najít řešení hierarchie. V závěru se hovoří o implementaci, která je obsahem přiloženého CD.

Kapitola 2

Hierarchie podmínek

Základem pro programování s omezujícími podmínkami je deklarativní popis problému. Problém se popíše pomocí **proměnných** a **podmínek**, které tyto proměnné svazují. Podmínky říkají, **jaké vlastnosti má mít řešení** , nikoli postup, jakým řešení **získat**, jak je tomu u imperativního programování. Každá podmínka vymezuje, kterých hodnot ze svých **domén** mohou její proměnné nabývat, aby byla splněna.

Může se stát, že si zadání protiřečí – tedy že nastane konflikt mezi podmínkami a neexistuje řešení, při kterém by byly splněny všechny podmínky. Takový problém nazýváme **příliš omezený**

Jednou z technik, jak tyto situace řešit, jsou **hierarchie podmínek**. Zadání problému se rozšíří o **preferenci** každé podmínky, kterou je vyjádřena důležitost splnění této podmínky. Kvalitativně pak budou dva druhy podmínek. Podmínky **nutné**, které splněny být musí, a **měkké** podmínky. Měkká podmínka nemusí být splněna, pokud její nesplnění umožní lépe splnit jinou podmínku se silnější preferencí, případně pokud toto nesplnění umožní lépe splnit nějakou skupinu podmínek se stejnou preferencí. Systém pro řešení hierarchií se pak snaží splnit co nejlépe podmínky podle jejich preference.

2.1 Omezující podmínky

Zavedení hierarchií v této kategorii vychází z [3] a hlavně [1], odkud pochází veškerá česká terminologie i znění následujících definic.

Omezující podmínka (*constraint***) je relace nad doménou D**, která omezuje (svazuje) nějaké proměnné. Příkladem domény mohou být reálná

čísla a omezující podmínkou nad touto doménou je třeba relace "$\leq$". *Spolu s doménou D je dána množina predikátových symbolů Π_D pro omezující podmínky taková, že obsahuje symbol "=". Omezující podmínka je potom výraz ve tvaru $p(t_1, t_2, \ldots, t_k)$, kde $p \in \Pi_D$ a t_i jsou termy v obvyklém významu.* Pro doménu reálných čísel a tři proměnné a,b a c je relace $a + b = c$ omezující podmínkou.

Některé proměnné mohou být v rámci podmínky označené jako *read-only* , tedy pouze ke čtení. Obvykle se v zápisu označují otazníkem. Příkladem je vazba mezi pohybem *myši* a pohybem *kurzoru* svázaná podmínkou $mouse.X\,? = cursor.X$. Zatímco systém pro řešení podmínek může na základě změny $mouse.X$ změnit $cursor.X$, tj. změnit pozici kurzoru na obrazovce, změna opačným směrem možná není. I kdyby jiný proces pohnul kurzorem po obrazovce, myš se pohnout nemůže.

Stejně tak se u proměnných můžeme setkat s označením *write-only*, i když to je mnohem méně obvyklé.

Metoda podmínky je funkce, kterou je jednoznačně určena hodnota jedné proměnné na základě hodnot ostatních proměnných tak, aby byla podmínka splněna. Podmínka $A + B = C$, která nemá žádnou read-only proměnnou , bude mít tři metody pro určení jednotlivých proměnných.

- $A \Leftarrow C - B$
- $B \Leftarrow C - A$
- $C \Leftarrow A + B$

Pro každou metodu jsou proměnné rozděleny na **vstupní** a **výstupní**. Výstupní proměnné jsou ty, jejíž hodnoty metoda určuje. Vstupní proměnné jsou pak ty, které jsou použity k určení výstupních proměnných.

Ne pro každou podmínku je možné metody definovat. Například pro podmínku $A > B$ to možné není. Na základě A není možné jednoznačně určit hodnotu B.

Podmínky je podle toho možné rozdělit na **funkcionální** a **nefunkcionální**. Funkcionální jsou takové podmínky, které mají pro každou proměnnou (která není read-only) definovanou metodu, která vždy vrací *právě jedno* řešení. Nechť má podmínka n proměnných a jsou dány hodnoty $n - 1$ z nich. Pak existuje právě jedna hodnota pro zbývající proměnnou taková, aby byla podmínka splněna.

Výše uvedená podmínka $A + B = C$ je podle tohoto rozdělení funkcionální, zatímco $A > B$ je podmínka nefunkcionální.

Podmínky mohou mít i více výstupů. Typickým zástupcem vícevýstupové podmínky je převod souřadnic kartézských na polární a naopak. Metody pak mohou vypadat například takto

- $[r, \omega] \Leftarrow [x, y]$
- $[x, y] \Leftarrow [r, \omega]$

U takových podmínek pak pro každou metodu může být více výstupních proměnných. Funkcionalita podmínky pak závisí na tom, jestli jsou pro každou metodu všechny výstupní proměnné jednoznačně určeny na základě vstupních proměnných.

Označená omezující podmínka(*labeled constraint*) je omezující podmínka doplněná **preferencí**(*strength*). Obvykle se používá zápis lc nebo $c@l$, kde c je omezující podmínka a l preference.

Množina preferencí je obvykle lineárně uspořádaná a musí být konečná. Podle [3] však není nezbytně nutné, aby byly preference úplně uspořádané. Stačí, když je preference "nutných" podmínek odlišena od ostatních. Musí být ostře silnější než všechny ostatní preference, které už mohou být jen v částečném uspořádání. Při hledání řešení se zkoumají všechna konzistentní zúplnění uspořádání preferencí. Je však neobvyklé pracovat s neúplným uspořádáním preferencí. V této práci, stejně jako ve všech algoritmech zmiňovaných na konci této kapitoly, je uspořádání preferencí lineární.

Bývá zvykem pojmenovat preference symbolickými jmény. Při lineární uspořádání preferencí mohou být tato jména mapována na přirozená čísla $0, 1, \ldots, n$, kde n je počet "měkkých" preferenčních úrovní. Číslo 0 je přitom vždy vyhrazeno pro nutné podmínky.

V tomto textu se používá *required* pro nutnou preferenci a *strong*, *medium*, *weak*, *weakest* pro ostatní, postupně oslabující preference.

2.2 Hierarchie

Hierarchie omezujících podmínek je konečná (multi)množina označených podmínek. Tuto množinu lze rozložit do jednotlivých **úrovní** H_0, H_1, $\ldots H_n$, kde H_0 označuje všechny nutné (*required*) podmínky z hierarchie H s odstraněným symbolem preference, H_1 podmínky na nejvyšší preferenční úrovni atd. až k H_n obsahující nejméně preferované podmínky z hierarchie H. Pro $k > n$, kde n je počet preferenčních úrovní hierarchie H , definujeme

množiny H_k jako prázdné. Poznamenejme také, že , je-li $i < j$, potom jsou v H_i silnější, tedy preferovanější podmínky než v H_j.

2.3 Řešení hierarchie

Ohodnocení pro množinu omezujících podmínek je funkce mapující volné proměnné v omezujících podmínkách na prvky domény D.

Řešením hierarchie omezujících podmínek je potom taková množina ohodnocení volných proměnných v H, že každé ohodnocení z řešení splňuje všechny nutné podmínky z H_0, a zároveň splňuje "měkké" podmínky, tj. podmínky z $H_1 \ldots H_n$, přinejmenším tak dobře, jako je splňují ostatní ohodnocení z řešení. Jinými slovy, žádné ohodnocení z řešení splňující nutné podmínky není "lepší" než jiné ohodnocení z řešení.

Když c je podmínka a θ je ohodnocení, pak $c\,\theta$ označuje aplikaci tohoto ohodnocení na tuto podmínku, tedy nahrazení všech volných proměnných z c jejich hodnotami z θ. Formálně nejprve zavedeme množinu ohodnocení S_0 takových, která splňují všechny *required* podmínky hierarchie H:

$$S_0 = \{\theta \mid \forall c \in H_0 \; c\,\theta \, \text{platí}\}$$

Z těchto ohodnocení, která představují "možné" hodnoty volných proměnných(tj. ty hodnoty, které splňují všechny podmínky *required*),je potřeba vybrat ty nejlepší. Nejlepší ohodnocení je takové, pro které neexistuje žádné lepší. Takové porovnání provedeme pomocí *komparátoru.* Zatím řekneme, že komparátor *better* je relace nad množinou ohodnocení. Řešením hierarchie H je množina S taková, že

$$S = \{\theta \mid \theta \in S_0 \wedge \forall \sigma \in S_0 \neg better(\sigma, \theta, H)\}$$

Uspořádání ohodnocení pomocí komparátoru *better*nemusí být úplné. Z toho plyne, že takovýchto "nejlepších" řešení může být více, přičemž tato řešení jsou nesrovnatelná komparátorem *better.* Prvky množiny S budou nadále nazývány *řešeními hierarchie omezujících podmínek.*

2.4 Komparátory

V předchozím zavedení množiny řešení hierarchie byl bez bližší specifikace použit komparátor *better*. Komparátor musí být ireflexivní a tranzitivní, tedy

$$\forall\theta\ \forall H\ :\ \neg better(\theta,\theta,H)$$
$$\forall\theta,\sigma,\tau\ \forall H\ :\ better(\theta,\sigma,H)\wedge better(\sigma,\tau,H)\Rightarrow better(\theta,\tau,H)$$

Složením těchto dvou vlastností nám navíc vychází další vlastnost, antisymetrie.

$$\forall\theta,\tau\ \forall H : \neg(better(\theta,\tau,H)\ \wedge better(\tau,\theta,H))$$

Jak už bylo zmíněno výše, relace nad doménou ohodnocení, označená jako komparátor *better*, nemusí být úplná. Tedy pro dvě ohodnocení θ a σ může nastat situace, že θ není lepší než σ a ani σ není lepší než θ . Jde o nesrovnatelná (a tedy stejně dobrá) ohodnocení. Formálně

$$\neg better(\theta,\sigma,H)\wedge\neg better(\sigma,\theta,H)$$

.

Aby byl komparátor užitečný pro hledání smysluplného řešení hierarchií, je potřeba, aby *respektoval hierarchii*. Vlastnost respektování hierarchie znamená, že pokud nějaké ohodnocení z S_0 splňuje všechny podmínky až do úrovně k(včetně), pak každé ohodnocení z S musí také splňovat všechny podmínky až do úrovně k. Formálně zapsáno

$$\exists\theta\in S_0\ \wedge\exists k>0(\forall i\in 1\ldots k\ \forall c\in H_i : c\,\theta\text{ platí})\Rightarrow$$
$$\Rightarrow\forall\sigma\in S\ \forall i\in 1\ldots k\ \forall c\in H_i : c\,\sigma\text{ platí}$$

2.4.1 Chybová funkce

Při srovnávání toho, jak dobře jednotlivá ohodnocení splňují hierarchii, je potřeba začít od nejmenšího stavebního kamene hierarchie. Tím je omezující podmínka. Proto je potřeba zavést **chybovou funkci**. Hodnota této funkce $e(c\,\theta)$ bude ukazovat, jak dobře ohodnocení θ splňuje podmínku c. Oborem hodnot této funkce jsou nezáporná reálná čísla. Nutnou vlastností je, že chybová funkce nabývá hodnoty nula právě tehdy, je-li podmínka splněná. Formální zápis

$$e(c\theta)=0\Leftrightarrow c\,\theta\text{ platí}$$

Čím je hodnota funkce blíže nule, tím lépe ohodnocení splňuje podmínku.

Jedna z nejčastěji používaných chybových funkcí je funkce *triviální* Ta nabývá jen dvou hodnot: 0 pokud je podmínka při daném ohodnocení splněna a 1 ve všech ostatních případech. Pokud komparátor používá triviální chybovou funkci, obvykle obsahuje v názvu výraz *–predicate–*. Je zřejmé, že při použití této chybové funkce, respektive při použití komparátoru postaveného na takové funkci, jsou rozlišovány pro dané ohodnocení pouze dva případy – zda je podmínka splněná či ne. Už není podstatné, "jak moc nesplněná" podmínka vlastně je. Všechna ohodnocení, pro které podmínka není splněna, jsou stejně špatná.

Pokud je doménou proměnných metrický prostor, je možné použít netriviální chybovou funkci. Například pro predikát = , tedy pro podmínky typu $X = Y$, to může být $dist(Y, X)$, kde $dist$ je funkce vyjadřující vzdálenost dvou bodů v daném metrickém prostoru, například $|Y - X|$. Komparátory s touto netriviální funkcí mívají v názvu *–metric–*, pokud je chybová funkce metrickou vzdáleností, případně *–error–* pro (jinou/libovolnou) netriviální chybovou funkci.

Netriviální chybová funkce se může v algoritmech pro řešení hierarchií projevovat zajímavým způsobem. Podmínky, i když nejsou splněny, mohou ovlivňovat výsledné řešení. Je totiž nutné minimalizovat chybu i na nesplnitelných podmínkách. Pro triviální chybové funkce nemá něco takového smysl, protože pokud podmínka splněná není, je hodnota chybové funkce konstantní 1 a není co zlepšovat.

2.4.2 Lokální versus globální

Při porovnávání ohodnocení jsou různé způsoby, jak vzít v úvahu strukturu hierarchie, tj. rozdělení podmínek do úrovní hierarchie. Rozdíl je v tom, jakým způsobem je porovnáváno splnění podmínek v jedné úrovni. **Lokální komparátory** porovnávají splnění každé podmínky v jedné úrovni zvlášť. Na druhé straně **globální komparátory** se zabývají vždy celou úrovní najednou, tj. zajímají se o "celkovou" chybu na dané úrovni.

Ještě existuje typ relace nad množinou ohodnocení, který se nazývá **regionální komparátor**, i když podle zde uvedené definice to komparátor není, protože nesplňuje podmínku tranzitivity. Na druhou stranu tento typ (ne)komparátoru má oproti lokálním komparátorům výhodu, že jím definované uspořádání ohodnocení je "úplnější", tedy nevyskytuje se zde tolik neporovnatelných dvojic ohodnocení.

Lokální komparátor

Ohodnocení σ je **lokálně lepší** než ohodnocení θ , pokud je hodnota chybové funkce pro ohodnocení σ na všech podmínkách až po nějakou úroveň k stejná jako pro ohodnocení θ a na této úrovni je stejná nebo menší pro všechny podmínky a ostře menší alespoň pro jednu podmínku. Formálně

$$
\begin{aligned}
\text{locally-better}(\theta, \sigma, H) \equiv \exists k \in \mathbf{N} \; : \; & \forall i < k \; \forall c \in H_i \; : \; e(c\,\sigma) = e(c\,\theta) \\
& \wedge \forall c_j \in H_k \; : \; e(c_j\,\sigma) \leq e(c_j\,\theta) \\
& \wedge \exists c_l \in H_k \; : \; e(c_l\,\sigma) < e(c_l\,\theta)
\end{aligned}
$$

Jak už bylo výše zmíněno, podle použité chybové funkce se obvykle mění prostřední část jména komparátoru. Pro triviální funkci se pak bude takovýto komparátor jmenovat *locally-predicate-better*. Pro netriviální chybovou funkci to může být *locally-metric-better* nebo *locally-error-better*.

Globální komparátor

Pro zavedení globálního komparátoru je potřeba nejdříve zavést **kombinační funkci** g. Jde o funkci, která pro každou úroveň hierarchie jako celek řekne, jak dobře je splněna pro dané ohodnocení. Pro lokální (a regionální) komparátory naopak taková funkce nemá význam, protože se posuzují jednotlivé podmínky zvlášť.

Podobně jako při zavedení chybové funkce, je potřeba, aby kombinační funkce nabývala hodnotu 0 právě tehdy, pokud jsou splněny všechny podmínky z dané úrovně :

$$g(\theta, H_k) = 0 \;\Leftrightarrow\; \forall c \in H_k \; c\,\theta \,\text{platí}$$

Ohodnocení σ je **globálně lepší** než ohodnocení θ , pokud je výsledek kombinační funkce g pro ohodnocení σ až do nějaké úrovně $k-1$ stejný jako pro ohodnocení θ a na k-té úrovni je ostře menší. Formálně zapsáno

$$
\begin{aligned}
\text{globally-better}(\sigma, \theta, H, g) \equiv \exists k > 0 \; \forall\, i \in \{1, \ldots, k-1\} \; & g(\sigma, H_i) = g(\theta, H_i) \\
& \wedge \; g(\sigma, H_k) < g(\theta, H_k)
\end{aligned}
$$

V souvislosti s globálními komparátory je třeba ještě zmínit pojem *váhy podmínek*. Je to do jisté míry podobný nástroj pro jemné vylaďování hierarchií jako jsou samotné preference. Ke každé podmínce c je kromě preference

přiřazena ještě váha w_c, což bývá reálné číslo z intervalu $(0, 1\rangle$. Čím je váha vyšší, tím důležitější je splnit danou podmínku v rámci její úrovně preference. Zatímco preference mají při splňování podmínek "právo veta", váhy je nemají. Za cenu splnění jedné podmínky silnější preference má cenu nesplnit libovolné množství podmínek slabší preference. Naopak, pokud má v rámci jedné úrovně preference daná podmínka sebesilnější váhu, nebude splněna, pokud je výsledek kombinační funkce nad váhami podmínek, které by nemohly být splněny, lepší než váha této podmínky.

Pokud zvolený globální komparátor váhy vyžaduje, ale nejsou součástí zadání, stačí položit $\forall c \in H \, : \, w_c = 1$

2.4.3 Konkrétní komparátory

locally-predicate-better – lokální komparátor, který používá triviální chybovou funkci

locally-error-better nebo *locally-metric-better* – lokální komparátor, který používá netriviální chybovou funkci, například metrickou vzdálenost. Pro podmínku $a = b$ by to mohlo být třeba $|b - a|$

weighted-sum-better – globální komparátor, který jako kombinační funkci používá součet výsledků chybové funkce krát váha podmínky, tedy

$$g(\sigma, H_i) \equiv \sum_{c \in H_i} w_c * e(c\,\sigma)$$

least-squares-better – globální komparátor, podobný předchozímu, v kombinační funkci se používá druhá mocnina chyby.

$$g(\sigma, H_i) \equiv \sum_{c \in H_i} w_c * e^2(c\,\sigma)$$

worst-case-better –globální komparátor, jehož kombinační funkce hledá pro každou úroveň nejhůře splněnou podmínku modifikovanou váhou této podmínky.

$$g(\sigma, H_i) \equiv \max_{c \in H_i}\{w_c * e(c\,\sigma)\}$$

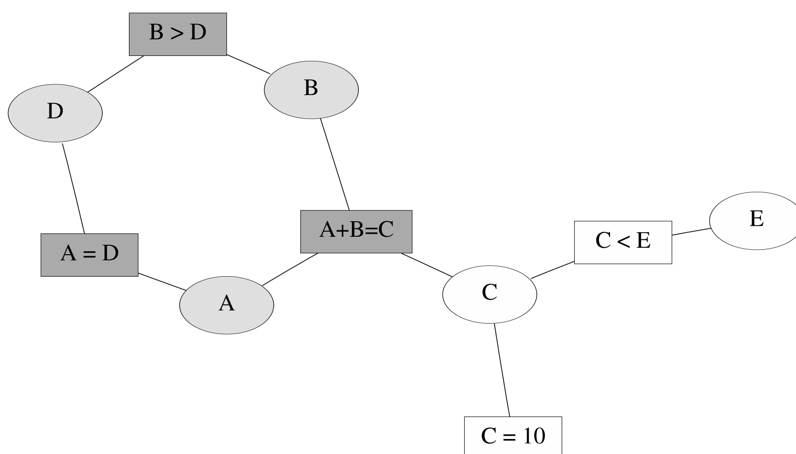
unsatisfied-count-better –globální komparátor nepoužívající váhy podmínek. Špatnost na dané úrovni měříme počtem nesplněných podmínek. Jde vlastně o komparátor predikátového typu.

$$g(\sigma, H_i) \equiv |\{c \,|\, c \in H_i \; c\,\sigma \, \text{neplatí}\}| \equiv |\{c \,|\, c \in H_i \; e(c\,\sigma) > 0\}|$$

2.5 Cykly

Mnoho algoritmů pro řešení hierarchií má omezení na acykličnost hierarchie. Bude proto užitečné zavést tento pojem už tady, v obecné části.

Pro hierarchii nechť je sestrojen graf. Vrcholy tohoto grafu budou dvou typů. Všechny podmínky hierarchie budou reprezentovány vrcholy. Stejně tak všechny proměnné. Mezi vrcholem podmínky a vrcholem proměnné pak bude neorientovaná hrana právě tehdy, když tato podmínka váže tuto proměnnou. **Hierarchie obsahuje cyklus právě tehdy, když tento graf obsahuje cyklus.** Jinou formulací je, že hierarchie obsahuje **cyklus podmínek** Pokud graf cyklus neobsahuje, říkáme, že hierarchie je acyklická, případně že hierarchie cyklus neobsahuje.



Obrázek 2.1: Příklad cyklu podmínek

Na obrázku 2.1 je znázorněn graf . Vrcholy podmínek a proměnných, ležící na cyklu, jsou zobrazeny šedě. Naopak bíle buňky na cyklu nejsou. K vrcholu, představujícímu proměnnou, může být připojeno libovolné množství unárních podmínek, ale ty cyklus nikdy vytvořit nemohou, i když jsou ve vzájemném konfliktu.

2.6 Rodina modrých algoritmů

Algoritmus *Deltablue*, popsaný v práci [6], je jeden z prvních algoritmů z University of Washington patřících do rodiny *modrých* algoritmů pro řešení hierarchií podmínek. Mezi další modré algoritmy patří *Skyblue* popsaný v [5], který je přímým nástupcem Deltablue. Je vylepšený o zacházení s cykly (když najde cyklus tak nehlásí chybu, ale předá ho speciálnímu řešiči) a umožňuje pracovat s podmínkami s více výstupy. Do rodiny také patří *Indigo*, popsané v sekci 2.8, které na rozdíl od předchozích algoritmů dokáže řešit nefunkcionální podmínky. Zajímavý je také *Ultaviolet*, což je řešící systém, který rozdělí hierarchii na samostatné oblasti podle typu podmínek a ty předá jednotlivým algoritmům. Jedná se o *Blue* pro lineární rovnice, již zmíněné Indigo pro nerovnice, *Purple* a *Deep Purple* pro cykly lineárních rovnic a nerovnic.

2.7 Deltablue

Algoritmus Deltablue je typickým zástupcem algoritmů lokální propagace, se věemi výhodami i nevýhodami. Je rychlý, ale omezený pouze na funkcionální prodmínky.

Je rozdělen na dvě části. V té první je připraven *graf řešení*, což je jistá struktura, zachycující vztahy mezi podmínkami a proměnnými. Pro každou podmínku je zde zvolena metoda, pomocí níž má být podmínka splněna.

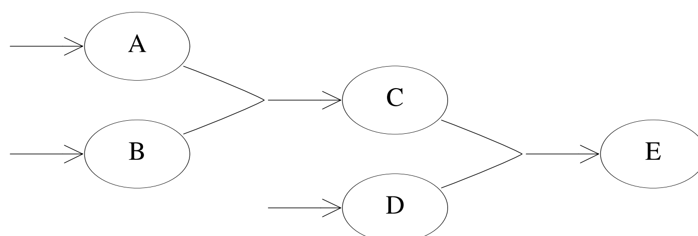
Druhá část pak v pořadí daném řešícícm grafem určuje hodnoty jednotlivých proměnných. Tím postupně získá ohodnocení všech proměnných, které je řešením zadané hierarchie.

Algoritmus Deltablue je inkrementální. Při přidání nebo ubrání podmínky není nutné zpracovat celou hierarchii znovu, ale pouze se upraví stávající řešení. Tato vlastnost je výhodná v interaktivních systémech, kde je důležitá rychlost zpracování.

2.7.1 Řešící graf

Algoritmus Deltablue udržuje podmínky v *řešícím grafu*. Jde o reprezentaci hierarchie, kde vrcholy představují volné proměnné. Podmínky hierarchie jsou pak reprezentovány speciálními hranami. Jde o hrany, které spojují všechny vrcholy, které váže původní podmínka. Tyto hrany jsou oriento-

vané. Orientace reprezentuje zvolenou *metodu*, tedy určuje, která proměnná se změní, když je podmínka odpovídající této hraně splňována. Preference hrany je definována jako preference podmínky, kterou reprezentuje. Ukázka takového grafu je na obrázku 2.2.



Obrázek 2.2: Deltablue : řešící graf

Graf musí být acyklický a do každého vrcholu smí vést pouze jedna hrana. Acykličnost je nutná, aby bylo možné graf topologicky setřídit pro druhou fázi algoritmu, ve které se hledá řešení. Kdyby do některého vrcholu vedlo více hran, nastává konflikt, protože každá z podmínek odpovídajících hranám může vyžadovat nastavení hodnoty proměnné, která odpovídá vrcholu, na jinou hodnotu. Jenže proměnná může mít hodnotu jen jednu.

Při přidávání nové hrany je nutné určit její orientaci, tedy proměnnou, kterou bude *určovat*. Pokud už jsou všechny proměnné, které by mohly být určované touto podmínkou "zablokovány" již existujícími podmínkami, je třeba provést určité změny grafu. Je možné, že "proti proudu" od některé z vázaných proměnných, je hrana s nižší preferencí než právě přidávaná podmínka. V tom případě je možné hranu, reprezentující tuto slabší podmínku,z grafu odstranit a u všech "mezilehlých" hran otočit orientaci. Díky tomu se uvolní jeden z vrcholů, kam může směřovat nová hrana a přidání bude možné.

Pokud není možné nějakou podmínku do grafu vůbec zařadit, tedy pokud nelze určit korektní orientaci její hrany ani po jakékoli změně grafu, je podmínka označená jako *nesplnitelná.* Toto označení je mírně zavádějící. Není totiž pravda, že podmínka *nemůže* být splněna, ale že se jí algoritmus nebude pokoušet splnit, protože jiné, silnější nebo stejně silné podmínky mají přednost. Přesto se může stát, že při řešení ostatních podmínek bude vybráno takové ohodnocení, při kterém je tato podmínka splněna.

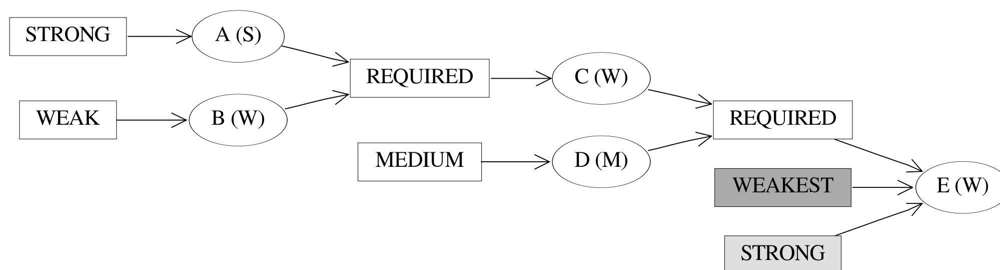
2.7.2 Průchozí preference

Při přidávání nové podmínky do grafu je potřeba určit orientaci její hrany. Běžně může nastat situace, že jsou všechny proměnné, vázané novou podmínku, už určovány jinými podmínkami v grafu. V tom případě je třeba zjistit, jestli se "proti proudu" od některé z těchto proměnných nenachází podmínka se slabší preferencí, než má ta nově přidávaná. Takovou podmínku je pak možno odstranit, změnit orientaci mezilehlých hran a přidat novou podmínku.

Přitom není efektivní pokaždé prohledávat celý graf. Aby bylo možné zajistit rychlé přidání podmínky, je v každém vrcholu udržována *průchozí preference.* Je to preference nejslabší hrany, která musí být z grafu odstraněna, aby bylo možné otočením dalších hran dosáhnout uvolnění daného vrcholu.

Nejlépe je vše demonstrovat na jednoduchém obrázku 2.3. Elipsy představují vrcholy, tedy proměnné, označené písmeny, v závorce je uvedena zkratka průchozí preference. Obdélníky představují podmínky, tedy hrany v řešícím grafu, aby bylo možné reprezentovat hrany spojující více vrcholů a navíc zobrazit jejich preferenci. Jelikož jsou hrany řešícího grafu reprezentovány na obrázku "jiným typem vrcholů", bude raději nadále hovořeno jen o podmínkách, které mají být hranami reprezentované.

Tmavě šedá podmínka reprezentuje *nesplnitelnou* podmínku s preferencí WEAKEST. Ilustruje nám konflikt mezi ní a hranou REQUIRED mezi C, D a E. Tím, že je podmínka nesplnitelná, není její orientace do vrcholu E aktivní a nepředstavuje tedy problém. Světle šedá podmínka pak představuje nově přidávanou podmínku s preferencí STRONG.



Obrázek 2.3: Deltablue : průchozí preference

Je vidět, že průchozí preference ve vrcholech B, C a E indikují přítomnost podmínky WEAK, která váže proměnnou B. Díky tomu je možné při

přidávání nové STRONG podmínky rychle zjistit, že ji přidat lze. Přitom bude nutné "dočasně" odstranit REQUIRED podmínku mezi vrcholy C,D a E. Po přidání nové podmínky STRONG bude potřeba opět do grafu vrátit odstraněnou podmínku. To bude umožněno dočasným odstraněním druhé REQUIRED podmínky – té mezi A, B a C. Nyní bude první REQUIRED podmínka opět přidána, ovšem s orientací "do" vrcholu C. Nakonec bude odstraněna WEAK podmínka a do grafu opět vrácena i druhá REQUIRED podmínka, s orientací "do" vrcholu B.

Současně se změnami v grafu se budou přepočítávat i průchozí preference. Pro všechny vrcholy kromě A a E to bude MEDIUM, protože do buňky D by se daly přesměrovat obě required podmínky v případě přidání nové podmínky. Buňky A a E budou mít průchozí preferenci STRONG, protože jsou vázané unárními podmínkami, které nemají "kam jinam" přesměrovat své hrany. Takže by musely být z grafu nenávratně odstraněny – stejně jako byla už od začátku podmínka WEAKEST a nyní i WEAK určující dosud proměnnou B.

2.7.3 Hledání řešení

Při přidávání a ubírání podmínek se mění řešící graf. Hlavním cílem algoritmu ale není údržba řešícího grafu, tedy nějaké vnitřní struktury, ale hledání ohodnocení proměnných.

Algoritmus hledá *locally-predicate-better* řešení. Jak již bylo řečeno, každou proměnnou smí určovat nejvýše jedna podmínka. Aby problém nebyl příliš volný, vyžaduje Deltablue, aby pro každou proměnnou existovala jedna speciální podmínka. Jde o podmínku, která jednoznačně určí hodnotu proměnné, pokud tak neučiní jiná "běžná" podmínka. Toho lze docílit snadno - tyto speciální podmínky mají vyhrazenu vlastní, nejslabší, preferenci, kterou "běžné" podmínky mít nemohou.

Tím je zaručeno, že každou proměnnou určuje právě jedna podmínka. Druhým požadavkem, který Deltablue klade, je, že v grafu nesmí vzniknout cyklus. Pokud se tak stane, algoritmus ohlásí chybu a skončí. Z tohoto důvodu je přesnější, když říkáme, že algoritmus hledá locally-*graph*-better řešení. Pokud je v řešícím grafu cyklus, může existovat *locally-predicate-better* řešení, ale algoritmus Deltablue ho nenajde.

Pokud se podaří úspěšně postavit korektní řešící graf, stačí ho nyní topologicky setřídit. V tomto pořadí se pak budou splňovat jednotlivé podmínky – a tedy určovat hodnoty proměnných. Díky tomu, že každou proměnnou

určuje právě jedna podmínka, máme zaručeno, že každá proměnná bude mít přiřazenu právě jednu hodnotu. Současně díky topologickému uspořádání se budou podmínky splňovat, až když budou mít všechny vstupní proměnné už své hodnoty nastaveny.

2.7.4 Vlastnosti

Jak už bylo řečeno, tento algoritmus je inkrementální. Je proto důležitý čas přidání a ubrání podmínky do/z existujícího grafu. Autoři složitost diskutují na straně 11 v práci [6].

Složitost přidání či ubrání podmínky je $O(NM)$, kde N je celkový počet podmínek v grafu a M je maximální počet metod na jednu podmínku, což je typicky velmi malé číslo $\ll 10$, takže obvykle je složitost $O(N)$. Stejnou složitost má i fáze hledání ohodnocení – každou podmínku v grafu zpracujeme jednou.

Kromě acykličnosti grafu (což je slabší požadavek než acykličnost hierarchie) je třeba zmínit, že Deltablue pracuje pouze s takzvanými "funkcionálními" podmínkami. Podmínka musí pro zadané vstupní proměnné jednoznačně určit hodnotu výstupní proměnné.

2.8 Indigo

Omezení algoritmů lokální propagace, jako je Deltablue, na použití funkcionálních podmínek je dost svazující. Například v grafických aplikacích je potřeba vyjádřit, že jeden objekt je vpravo od jiného. To lze přirozeně vyjádřit nefunkcionální podmínkou $objectA.x > objectB.x$.

Snaha překonat toto omezení byla důvodem pro vytvoření algoritmu Indigo. Kromě použití nefunkcionálních podmínek se Indigo výrazně odlišuje ještě v jedné věci - v reprezentaci hodnot proměnných. V algoritmu Deltablue byla hodnota proměnné určena v kroku, kdy byla zpracována podmínka, která ji určovala a tato hodnota byla "definitivní". Naopak u Indiga má každá proměnná doménu hodnot, ze které jsou v průběhu výpočtu odstraňovány nekonzistentní hodnoty, až do okamžiku kdy v doméně zbude poslední hodnota.

Tento algoritmus byl důležitou inspirací pro teorii popsanou v další kapitole, na které stojí nově navrhovaný algoritmus Straton 3 . Navíc má mnoho

společného s propagační fázi algoritmu Straton 3 . Proto zde bude popsán podrobněji.

2.8.1 Slabá a silná verze

Algoritmus Indigo je teoreticky popsán v [4], kde je i podán důkaz o tom, že algoritmus je korektní a úplný. To platí pro teoretický popis Indiga. Zároveň je tam popsaná implementace, která je zjednodušením teorie. Tato implementace je korektní, ale neúplná a je označená jako *slabá verze* algoritmu. Autoři v závěru práce navrhují, jak tuto verzi upravit, aby vznikla *silná verze*.

Rozdíl mezi těmito verzemi je v reprezentaci domén jednotlivých proměnných. Slabá verze reprezentuje aktuální doménu proměnné pomocí horní a dolní meze, tedy jednoduchého intervalu. Běžně podmínky "ořezávají" interval ze stran ($A > 10$ a podobě) Problém nastane, pokud jsou nekonzistentní hodnoty "uprostřed" intervalu. Interval se nemůže zmenšit , protože na "okrajích" zůstávají konzistentní hodnoty. Když je později proměnná znovu omezována jinou podmínkou, může se stát, že tato podmínka omezí interval právě na ty nekonzistentní hodnoty "uprostřed". Následkem toho se stane už zpracovaná, silnější podmínka nesplnitelnou. Algoritmus je tento stav schopen detekovat a ohlásit chybu. Příklad je popsán v sekci 2.8.5.

Autoři navrhují jako řešení tohoto problému – tedy jako silnou verzi – použít **divisions**, neboli sjednocení uspořádaných disjunktních intervalů.

2.8.2 Použitý komparátor

Algoritmus Indigo na rozdíl od algoritmů lokální propagace používá metrický komparátor locally-error-better . Pokud nějakou podmínku nemůžeme bezezbytku splnit, neznamená to, že není důležitá. I nesplněná podmínka ovlivňuje řešení. Více v sekci 2.4.3 na stránce 10.

Tento komparátor se velmi dobře hodí například pro interaktivní grafické aplikace, protože pokud podmínka nemůže být zcela splněná, tak je snahou alespoň minimalizovat chybu, tedy míru toho jak moc je nesplněna. Příkladem může být třeba situace, kdy má nějaký pohyblivý objekt sledoval kurzor myši (podmínka *strong*, aby byly souřadnice objektu a kurzoru shodné), ale pouze ve vymezené oblasti(podmínka *required*, aby byly souřadnice objektu ve vymezeném rozsahu). Pokud je použit *locally-predicate-better* komparátor a kurzor opustí vymezenou oblast, pohyb objektu se zastaví na hranici

oblasti do té doby, dokud se kurzor nevrátí zase zpět. Naopak při použití *locally-error-better* se bude objekt pohybovat po hranici vymezené oblasti tak, aby byl pořád co nejblíže kurzoru, který se nachází za hranicí. Snaží se totiž minimalizovat chybu, tedy zmenšit vzdálenost objekt–kurzor na nezbytně nutné minimum.

2.8.3 Vlastnosti řešených hierarchií

Stejně jako předchozí algoritmy, Indigo neumí řešit hierarchie, které obsahují cykly (podmínek). Definice cyklů je v sekci 2.5 na straně 11. Prezentovaná implementace se cykly ani nesnaží detekovat.

Jak už bylo řečeno, v průběhu řešení nejsou proměnným přiřazovány jednoznačné hodnoty, ale jejich domény se postupně zmenšují. Pokud v doméně zbude jediná hodnota, je tato hodnota teprve považována za "definitivní" hodnotu této proměnné v hledaném *locally-error-better* řešení.

Mohlo by se stát, že by hledání řešení skončilo ve stavu, kdy některé proměnné nemají ještě přiřazenu konkrétní hodnotu, tj. jejich domény obsahují více než jednu hodnotu. Ve slabé implementaci to znamená, že dolní mez je menší než horní. V takovém případě tedy není ještě nalezeno konkrétní ohodnocení. Při použití Indiga v interaktivní aplikaci je potřeba konkrétní objekt na obrazovce umístit na konkrétní souřadnice a nestačí vědět, že $70 \leq objectA.x \leq 100$, tedy že objekt má být umístěn "někde tady". Takové "volné" řešení navíc neznamená, že je možno vybrat libovolnou hodnotu z každé netriviální domény - protože proměnné mohou být mezi sebou vázány podmínkami, které ještě neměly možnost být vynuceny - viz příklad:

P1	$1 \leq a \leq 10$	@	required
P2	$1 \leq b \leq 10$	@	required
P3	$a + 2 = b$	@	strong

Pokud bychom takovou hierarchii vyřešili tak jak je, vyšlo by nám, že $a \in \langle 1, 8 \rangle$ a $b \in \langle 3, 10 \rangle$. To ale neznamená, že si pro každou proměnnou můžeme zvolit jakoukoli hodnotu – kdybychom chtěli třeba $a = b = 5$ tak nemáme splněnou třetí podmínku hierarchie.

Řešení je stejné stejné jako u Deltablue. Je nutné, aby každou proměnnou vázala speciální podmínka *stay*. To je podmínka s nejnižší váhou, která vyžaduje, aby proměnná nabývala nějaké konkrétní hodnoty, pokud silnější (tedy všechny ostatní, "běžné") podmínky nevyžadují něco jiného.

Po zpracování všech "běžných" podmínek přichází na řadu podmínky *stay*, protože mají (z definice) nejslabší preferenci.

Je-li při splňování podmínky *stay* doména proměnné netriviální interval, v němž leží hodnota, ke které *stay* podmínka váže "svoji" proměnnou, zmenší se doména proměnné právě na tuto hodnotu. Naopak, pokud tato hodnota v tomto intervalu neleží, bude díky minimalizaci chyby na této podmínce interval omezen na svou dolní čí horní mez a tím bude zaručeno, že doména proměnné bude obsahovat jedinou hodnotu.

Konkrétní hodnota, ke které podmínka *stay* váže svou proměnnou, může být její předchozí hodnota, tj. hodnota před spuštěním algoritmu nebo z předchozí iterace, pokud jde o interaktivní použití s kompilovaným plánem.

Tyto *stay* podmínky plní dvě funkce. Zajišťují nalezení jednoznačného řešení – tj. situace kdy pro každou proměnnou její doména obsahuje právě jednu hodnotu. Pro slabou implementaci to znamená, že pro každou proměnnou je její horní a dolní mez totožná. Další funkcí je zajištění stability, tedy aby se mezi jednotlivými iteracemi (při opakovaném zpracování) neměnily hodnoty těch proměnných, u kterých to není nezbytně nutné.

2.8.4 Slabá implementace

Jak už bylo řečeno výše, slabá verze algoritmu Indigo používá k reprezentaci domén proměnných horní a dolní mez, které reprezentují uzavřený[1] interval. Je třeba zavést běžné aritmetické operace,vyskytující se v podmínkách, které má Indigo řešit, tak, aby pracovaly nad intervaly. Například $\langle l_1, u_1\rangle + \langle l_2, u_2\rangle$ nechť je interval všech součtů libovolného čísla z $\langle l_1, u_1\rangle$ s libovolným číslem z $\langle l_2, u_2\rangle$, což bude $\langle l_1 + l_2, u_1 + u_2\rangle$ kromě případů kdy je jedna z hodnot l_1, l_2, u_1, u_2 buď $-\infty$ nebo ∞. Pseudokód pro operace $+$, $-$, $\times$, $\div$ je uveden v [4]. Pro snazší práci jsou také implementovány operace, kdy je jedním z argumentů interval a druhým číslo, tedy například $\langle l_1, u_1\rangle + n \equiv \langle l_1, u_1\rangle + \langle n, n\rangle$.

Základem splňování podmínek je operace *tighten*, pracující s intervaly. Její úlohou je omezit doménu proměnné v tak, aby se "vešla" do zadaného intervalu i. Pokud není doména proměnné s tímto intervalem disjunktní, je výsledkem jejich průnik. V opačném případě se doména zmenší na triviální interval tím, že se horní mez posune na dolní nebo naopak. Toto reprezentuje chování *locally-error-better* komparátoru, kdy nám jde o minimalizaci chyby

[1] otevřené pro dosud neomezenou dolní a horní mez, tj $-\infty$ a ∞

podmínky, pokud nemůže být splněna. Toto chování popisuje algoritmus 2.1.

V klasických algoritmech pro lokální propagaci má každá podmínka soubor metod. Když je metoda vybrána a aplikována, nastaví hodnotu jedné proměnné na základě hodnot všech ostatních proměnných tak, aby byla podmínka splněna. Například pro podmínku $a + b = c$ jsou k dispozici metody $c := a + b$, $a := c - b$, $b := c - a$. V případě Indiga má každá podmínka k dispozici sadu metod propagujících změny mezí, tedy pro podmínku $a+b = c$ to jsou

- $a.tighten(c.bounds - b.bounds)$
- $b.tighten(c.bounds - a.bounds)$
- $c.tighten(a.bounds + b.bounds)$

Jestliže došlo ke změně mezí na proměnné c, budeme takovou změnu propagovat přes tuto podmínku jak na proměnnou a, tak b, a tudíž zavoláme druhou i třetí metodu. To je výrazný rozdíl oproti algoritmům, jako je *Deltablue*, kde při splňování podmínky dochází vždy ke změně u *právě jedné* proměnné.

Algoritmus 2.1: operace tighten, která omezí doménu proměnné v na interval i

```
1  tighten(variable v, interval i)
2  if intersect(v.bounds, i) is not empty then
3      v.bounds := intersect (v.bounds, i);
4  else
5      if v.upperBound < i.lowerBound then
6          v.lowerBound := v.upperBound;
7      else
8          v.upperBound := v.lowerBound;
9     end if;
10 end if;
```

Pseudokód 2.2 popisuje hlavní část algoritmu Indigo. Ten postupně zpracovává všechny podmínky seřazené podle preference od nejsilnějších k nejslabším. Každou podmínku *cn* se snaží co nejlépe splnit omezením domén jejích proměnných. Omezení domény jedné proměnné může vyvolat potřebu omezit domény dalších proměnných vázaných už zpracovanými podmínkami,

tedy vyvolat řetězovou reakci. Toto je ošetřeno pomoci fronty *queue*, do které je na začátku vložena právě zpracovávaná podmínka *cn*. Z této fronty jsou odebírány a zpracovávány podmínky, dokud není prázdná. Jakmile je omezena doména nějaké proměnné, do fronty *queue* jsou vloženy všechny podmínky nad touto proměnnou, které má ještě smysl zpracovávat.

Algoritmus 2.2: Hlavní část algoritmu Indigo

```
1  Indigo (all_constraints , all_variables)
2  all_constraints podmínky, setříděné sestupně podle preference
3  all_variables seznam všech proměnných
4  active_constraints := empty_set;
5  for v in all_variables do
6      initialize v.bounds to unbounded;
7  end for;
8
9  for current_constraint in all_constraints do
10     tight_variables := empty_set;
11     queue := empty_queue;
12     queue.add(current_constraint);
13     while queue not empty do
14         cn:=queue.front;
15         tighten_bounds(cn,queue,tight_variables ,
16                        active_constraints);
17         check_constraint(cn,active_constraints);
18         queue.dequeue;
19     end while;
20 end for;
```

Množina *active_constraints* udržuje všechny podmínky, které již byly alespoň jednou zpracovány. Jsou odtud ale vyřazeny ty podmínky, které už znovu neovlivní žádnou proměnnou. To se týká unárních podmínek a podmínek, jejichž všechny proměnné jsou už omezené na jedinou, a tedy konečnou hodnotu. Tato situace je reprezentovaná stavem, kdy je dolní a horní mez domény proměnné stejná.

V *tight_variables* je množina proměnných, kterým byla doména v rámci zpracování aktuální podmínky *cn* už změněna. V [4] autoři dokázali, že žádná

proměnná nebude v jedné vlně omezena více než jednou a tato množina pomáhá hlídat, které proměnné už nebudou znovu měněny.

Algoritmus 2.3 představuje proceduru *tighten_bounds*, která se stará o zpracování jedné podmínky, tedy omezení domén jejích proměnných. Dále vkládá do fronty *queue* ty podmínky, jejichž proměnná se právě změnila a jsou v množině *active_constraints*, určené k opakovanému zpracování. Metoda *tight_bounds_for* se postará o to, aby byly omezeny domény proměnné, která je parametrem, na základě aktuálních domén ostatních proměnných této podmínky a vrací *true* pokud došlo ke změně domény a bude tedy nutné tuto změnu propagovat dál.

Algoritmus 2.3: Procedura tighten_bounds

```
1  procedure tighten_bounds (cn,queue,tight_variables,
2                              active_constraints)
3  for v in cn.variables and not in tight_variables do
4    tighten_flag := cn.tighten_bounds_for(v);
5    tight_variables.add(v);
6    if tighten_flag then
7      for c in v.constraints do
8        if c in active_constraints and c not in queue then
9          queue.add(c);
10       end if;
11     end for;
12   end if;
13 end for;
```

Procedura *check_constraint* kontroluje stav řešení hierarchie. Je zodpovědná za zastavení hledání řešení v případě, kdy zjistí, že není možné splnit *required* podmínku, a hierarchie tedy nemá řešení. Dále je schopna detekovat situaci, kdy sice může existovat *locally-error-better* řešení, ale tato slabá, neúplná implementace to nedokáže. Navíc se tato procedura stará o přidání podmíky do *active_constraints* v situaci, kdy je podmínka zpracována, ale některé její proměnné ještě mají netriviální domény (tedy bude nutné tuto podmínku ještě znovu splňovat). Také z této množiny odstraňuje kontrolovanou podmínku ve chvíli, kdy mají všechny její proměnné přiřazenu definivní hodnotu, tedy v okamžiku, kdy je dolní mez domény proměnné rovná horní mezi.

Algoritmus 2.4: Procedura check_constraint

```
1  procedure check_constraint (cn, active_constraints)
2    if cn is unary then
3      if cn is required and cn is not satisfiable then
4        exception(required_constraint_not_satisfied);
5      end if;
6      return;
7    end if;
8
9    if not all of cn's variables have unique values then
10     active_constraints.add(cn);
11     return;
12   end if;
13
14   if cn is satisfied then
15     active_constraints.delete(cn);
16   else
17     if cn is required then
18       exception(required_constraint_not_satisfied);
19     else
20       exception(constraints_too_difficult);
21     end if;
22   end if;
23 end check_constraint;
```

2.8.5 Neúplnost

Problém neúplnosti algoritmu je autory označen za okrajový, nicméně může nastat. Jako příklad může posloužit následující hierarchie

P1	$-1 \leq a \leq 1$	@	required
P2	$-1 \leq b \leq 1$	@	required
P3	$a * b = 2$	@	strong
P4	$a = 0.5$	@	weak

Nejprve zpracujeme nutné podmínky P1 a P2, tedy omezíme domény proměnných a i b na intervaly $\langle -1, 1 \rangle$. Když se pokusíme zpracovat podmínku P3, zjistíme, že to v tuto chvíli nemá žádný efekt - obě její proměnné

nejsou ještě fixovány na konkrétní hodnotu a nezmění se ani jedna z nich. Navíc ani nemůžeme zmenšit jejich domény, protože obě krajní hodnoty, tj. -1 i 1 mohou být řešením této podmínky. Proto je tato podmínka pouze uložena do množiny *active_constraints* a bude znovu zpracovávaná, až se zmenší doména některé z jejích proměnných. Nyní se tedy bude řešit podmínka P4. Jelikož je hodnota 0.5 v doméně proměnné a, může být tato podmínka bez problému splněna. To tedy znamená, že se doména proměnné a zmenší na interval $\langle 0.5, 0.5\rangle$. Nyní se tato změna bude přes podmínku P3 propagovat na proměnnou b. Jelikož tuto podmínku splnit nemůžeme, musíme se pokusit aspoň minimalizovat chybu. To znamená, že se doména proměnné b zmenší na $\langle 1, 1\rangle$. V tuto chvíli už jsou pevně fixovány všechny proměnné podmínky P3. Problém je v tom, že není splněná. Proto je vyvolána vyjímka *constraints_too_difficult*.

Je zjevné, že tato hierarchie má řešení *locally-error-better* $a = 1$, $b = 1$, které slabá verze algoritmu nenašla. Nicméně tento stav dokázala signalizovat.

2.8.6 Žádné řešení

Pokud není splněna některá z *required*podmínek, algoritmus signalizuje vyjímku *required_constraint_not_satisfied*. Intuitivně bychom se tedy mohli domnívat, že pokud jsou všechny nutné podmínky splněny, *locally-error-better* řešení už musí existovat. Není tomu tak, jako protipříklad můžeme uvést tuto jednoduchou hierarchii, pro kterou hledáme *locally-error-better* řešení.

P1	$a > 0$	@	required
P2	a=0	@	medium

Nutná podmínka může být zjevně splněna pro jakékoli kladné x. Problém je, že pro $a = \frac{x}{2}$ je chyba druhé podmínky poloviční než pro $a = x$, tudíž řešení x není *locally-error-better* řešením hierarchie pro jakékoli x.

V představené implementaci tento problém nastat nemůže, protože zde nejsou implementovány ostré nerovnosti. Autoři tvrdí, že by neměl být problém ostré nerovnosti doimplementovat.

2.8.7 Složitost

Dikuzi složitosti slabé verze Indiga provedli autoři v [4] na straně 11.

Nechť má hierarchie N podmínek a M proměnných. Při zpracování podmínky je klíčové, že je doména každé proměnné omezena nejvýše jednou, což je zaručeno acykličností grafu. Zpracování jedné podmínky tedy znamená složitost $O(M)$ v nejhorším případě. Celková složitost tedy bude $O(NM)$.

Ve většině případů se bude algoritmus chovat výrazně lépe, ale samozřejmě se dají najít příklady, kdy bude této složitosti dosaženo.

2.8.8 Použití *locally-predicate-better*

Algoritmus Indigo byl napsán pro použití *locally-error-better* komparátoru, ale není těžké ho upravit pro použití *locally-predicate-better* komparátoru. Je třeba upravit chování ve chvíli, pokud nelze nějakou podmínku bezezbytku splnit. Pokud to možné není, nebude taková podmínka zpracována – neovlivní hodnoty svých proměnných. Nemá smysl minimalizovat chybovou funkci, která je v případě *locally-predicate-better* triviální, a tedy *stejně špatná* pro všechna ohodnocení, kde není tato podmínka splněná. Naopak, tímto "nepovinným" omezováním by z domén mohly být odstraněny hodnoty, které jsou součástí nějakého *locally-predicate-better* řešení této hierarchie.

Kapitola 3

Alternativní teorie

Algoritmus Straton 3 , jehož popis je hlavním obsahem této práce, je postavený na základě teorie sítí podmínek, popsané v práci [1]. V této kapitole bude uveden stručný popis důležitých definic a vět z této práce.

Stejně jako v minulé kapitole jsou použity základní pojmy jako je *omezující podmínka, označená omezující podmínka, hierarchie, úroveň hierarchie, ohodnocení proměnných a chybová funkce.*

3.1 Komparátory

Zavedení hierarchií je zde opačné než v předchozí kapitole, tedy směrem od komparátorů k definici řešení.

Začíná se od komparátorů nad určitou množinou podmínek. Ač to není explicitně řečeno, pro snadnější pochopení je dobré zdůraznit, že jde o komparátor nad množinou podmínek se stejnou preferencí.

Definice 1 *Máme dvě množiny omezujících podmínek C a D, dále θ , σ a π jsou ohodnocení proměnných a e je chybová funkce, pak je relace $\leq^C$* **úrovňový komparátor**, *pokud jsou splněna následující pravidla*

1. $\sigma \leq^C \theta \wedge \theta \leq^C \pi \Rightarrow \sigma \leq^C \pi$ *(tranzitivnost)*
2. $\forall c \in C : e(c\sigma) \leq e(c\theta) \Rightarrow \sigma \leq^C \theta$
3. $\theta \leq^{C \cup D} \sigma \wedge \sigma \leq^C \theta \Rightarrow \theta \leq^D \sigma$
4. $\sigma \leq^C \theta \wedge \sigma \leq^D \theta \Rightarrow \sigma \leq^{C \cup D} \theta$

Pomocí neostré relace $\leq^C$ je definována ekvivalence $\sim^C$ a ostrá nerovnost tak, jak je obvyklé. Stejně tak jsou definovány relace 'opačným směrem' jako například $\geq^C$.

Definice 2 *Nechť je relace $<^H$ na množině ohodnocení definována následujícím způsobem (H je hierarchie podmínek, H_l jsou její příslušné úrovně a σ a θ jsou ohodnocení proměnných) :*

$$\sigma <^H \theta \equiv \exists k > 0 \forall l \in \{1, \ldots, k-1\} \quad \sigma \sim^{H_l} \theta \wedge \sigma <^{H_k} \theta$$

Potom říkáme, že relace $<^H$ je **hierarchický komparátor**. *Podobně jako pro úrovňový komparátor jsou definovány: neostrá relace, ekvivalence a relace 'opačným směrem'.*

Zde je potřeba zdůraznit, že takováto definice hierarchického komparátoru v sobě implicitně obsahuje ideu respektování hierarchie podmínek, tj. že silnější podmínky mají přednost před těmi slabšími a mají v tomto vztahu právo veta.

3.2 Lokální a globální komparátory

Při specifickém zavedení úrovňových komparátorů, dostaneme jejich použitím v hierarchickém komparátoru ekvivalenty lokálních a globálních komparátorů z předchozí kapitoly.

Pro regionální komparátory zde neexistuje ekvivalent, protože tyto komparátory by nemohly splňovat podmínku tranzitivity úrovňového kompará-

toru.

Definice 3 *Říkáme, že komparátor* $\leq^C$ *je* **lokální**, *pokud je jeho relace definována takto:*

$$\sigma \leq^C \theta \equiv \forall c \in C \quad e(c\sigma) \leq e(c\theta)$$

Hierarchický komparátor může použít k porovnání chyby na jednotlivých úrovních *lokální* úrovňový komparátor. V tom případě se takový hierarchický komparátor bude přesně shodovat s lokálním komparátorem z předchozí kapitoly.

Definice 4 *Říkáme, že komparátor* $\leq^C$ *je* **globální**, *pokud je definován takto:*

$$\sigma \leq^C \theta \equiv g(\sigma, C) \leq g(\theta, C)$$

kde $g(\sigma, C)$ *je (kombinační) funkce přiřazující danému ohodnocení* σ *proměnných a dané množině* C *nezáporné reálné číslo tak, že platí:*

1. $g(\sigma, C) = 0 \Leftrightarrow \forall c \in C \quad c\sigma$ *platí*
2. $\forall c \in C e(c\sigma) \leq e(c\theta) \Rightarrow g(\sigma, C) \leq g(\theta, C)$
3. $g(\theta, C \cup D) \leq g(\sigma, C \cup D) \wedge g(\sigma, D) \leq g(\theta, D) \Rightarrow g(\theta, C) \leq g(\sigma, C)$
4. $g(\sigma, C) \leq g(\theta, C) \wedge g(\sigma, D) \leq g(\theta, D) \Rightarrow g(\sigma, C \cup D) \leq g(\theta, C \cup D)$

Vlastnost 1 není pro tuto teorii nutná, ale zajišťuje, že všechny globální hierarchické komparátory jsou současně globálními komparátory podle definic z předchozí kapitoly. Mezi globální hierarchické komparátory patří například *weighted-sum-better* nebo *least-squares-better*. Naopak *worst-case better* není globálním hierarchickým komparátorem, neboť nesplňuje bod 3 této definice.

3.3 Řešení hierarchie

Definice 5 *Říkáme, že funkce* S, *přiřazující dané množině ohodnocení* Θ *a dané hierarchii podmínek* H *nějakou množinu ohodnocení, je* **operátorem splňování hierarchie podmínek**, *pokud platí :*

$$S(\Theta, H) = \{\sigma \in \Theta \mid \neg\exists\theta \in \Theta \quad \theta <^H \sigma\}$$

Definice 6 *Řešením dané hierarchie podmínek H je taková množina ohodnocení $S(H)$, že platí*

$$S(H) = S(\Theta, H),$$

kde Θ je množina všech ohodnocení proměnných z H splňujících všechny nutné podmínky z hierarchie H, tj. všechny podmínky z úrovně H_0.

Za povšimnutí stojí, že tato definice určuje řešení hierarchie stejným způsobem, jakým to bylo zavedeno v předchozí kapitole, pouze s tím rozdílem, že zde jsou definice komparátorů restriktivnější. To umožňuje dokázat zajímavá tvrzení o řešení hierarchií.

3.4 Posloupnosti buněk

Obsahem této teorie je rozdělení hierarchie do co nejmenších částí, které pak budou řešeny postupným aplikováním operátoru splňování hierarchie podmínek. Nejprve tedy definice takového rozdělení hierarchie

Definice 7 *Hierarchii omezujících podmínek H je možné rozložit* **do posloupnosti buněk** $B^1, \ldots, B^n$. *Takovýto rozklad musí splňovat následující podmínky:*

- $H = B^1 \cup B^2 \cup \ldots \cup B^n$
- $\forall i, j \quad i \neq j : B^i \cap B^j = \emptyset$

Takové rozdělení však nemůže být zcela libovolné. Nesmí se stát, aby slabší podmínky zabránily splnění podmínkám silnějším, které by byly v buňkách zpracovávaných později. Jelikož toto rozdělení má fungovat i pro hledání řešení pomocí globálních komparátorů, nesmí k takovému omezení dojít ani pro stejně silné podmínky v různých buňkách.

Pokud v průběhu řešení nebude možné v některé buňce splnit nějakou podmínku "bezezbytku", je potřeba zaručit, že toto nesplnění není zapříčiněno stejně silnou nebo dokonce slabší buňkou. Formální zápis tohoto požadavku obsahuje následující definice.

Definice 8 *Říkáme, že posloupnost buněk splňuje* **vlastnost postupného oslabování**, *pokud* $\forall i \in \{1, 2, \dots, n-1\}$ *platí následující implikace:*

$$\exists \sigma \in S(\dots S(\Theta, B^1)\dots), B^n) \cup S(\Theta, B^1 \cup B^2 \dots \cup B^n) \exists c@l \in B^{i+1}\ e(c\sigma) > 0 \Rightarrow$$
$$\Rightarrow\ \forall j \leq i\ \forall c'@l' \in B^j \quad \forall c''@l'' \in B^{i+1} : l' < l''$$

Na posloupnost buněk, splňující vlastnost postupného oslabování, je možné nahlížet také naopak. Buňky obsahující slabší podmínky mohou být pouze před buňkami, které obsahují (stejně silné nebo silnější) "vždy splnitelné" podmínky.

3.5 Věty o korektnosti a úplnosti

Věta 1 (O korektnosti) *Nechť posloupnost buněk* $B^1, \dots B^n$ *splňuje vlastnost postupného oslabování, potom platí* $S(\dots S(\Theta, B^1)\dots, B^n) \subseteq S(\Theta, B^1 \cup \dots B^n)$

Důkaz indukcí podle počtu buněk v posloupnosti je na stranách 63 a 64 v [1]. Pokud posloupnost buněk splňuje vlastnost postupného oslabování a postupným aplikováním operátoru splňování hierarchie na tuto posloupnost je nalezeno nějaké ohodnocení, pak toto ohodnocení padne do množiny řešení celé hierarchie.

Definice 9 *Říkáme, že úrovňový komparátor* $\leq^C$ *je lineární, pokud splňuje následující podmínku :* $\forall \sigma, \theta \quad \sigma \leq^C \theta \vee \theta \leq^C \sigma$ *Pokud je hierarchický komparátor definovaný pomocí lineárního úrovňového komparátoru, hovoříme o* **lineárním hierarchickém komparátoru**.

Lineárnost komparátoru zaručuje, že libovolná dvě ohodnocení jsou porovnatelná.Všechny globální hierarchické komparátory tuto podmínku triviálně splňují díky úplnosti uspořádání $\leq$ na reálných číslech. Naopak lokální komparátory tuto podmínku nesplňují.

Při využití vlastnosti linearity, což je klíčové, platí tvrzení:

Lemma 1 *Nechť* B^1, B^2 *je rozklad hierarchie* H *do posloupnosti buněk, která splňuje vlastnost postupného oslabování. Nechť je použitý úrovňový komparátor lineární a* $S(S(\Theta, B^1), B^2) \neq \emptyset$. *Pak platí :* $S(S(\Theta, B^1), B^2) = S(\Theta, B^1 \cup B^2)$

Z tohoto tvrzení už lze jednoduše pomocí indukce dokázat klíčovou větu celé teorie, tedy :

Věta 2 (O úplnosti) *Nechť posloupnost buněk $B^1, B^2, \ldots, B^n$ splňuje vlastnost postupného oslabování a $S(\ldots S(\Theta, B^1) \ldots, B^n) \neq \emptyset$ a nechť je použitý úrovňový komparátor lineární. Pak platí*

$$S(\ldots S(\Theta, B^1) \ldots, B^n) = S(\Theta, B^1 \cup B^2 \ldots \cup B^n)$$

Tato věta říká, že pokud opakované aplikování operátoru splňování hierarchie (pro lineární komparátor) na posloupnost buněk, splňující vlastnost postupného oslabování, najde alespoň nějaké řešení, pak je tímto způsobem získána právě množina všech řešení celé hierarchie.

Nyní je třeba pohovořit o možných rozkladech hierarchie do posloupnosti buněk. Triviálním rozkladem je, když z celé hierarchie uděláme jedinou super-buňku. Pro tuto buňku je samozřejmě splněna vlastnost postupného oslabování, takže se na tento 'rozklad' dají aplikovat věty o korektnosti i úplnosti. Takový rozklad nepřináší žádný užitek, protože neposkytuje jakýkoli návod k nalezení řešení hierarchie.

Další neméně jednoduchou možností je rozklad do jednotlivých úrovní hierarchie. Postupná aplikace operátoru splňování hierarchie potom odpovídá zjemňování – *refining metodě*, která hledá řešení od nejsilnějších úrovní k nejslabším. Tato metoda již představuje jistý způsob hledání řešení, je ovšem velmi neefektivní.

3.6 Buňky a sítě

Celá teorie směřuje k zavedení pojmu buňka a síť buněk. Síť buněk by měla být zobecněním grafu podmínek, tak jak se vyskytuje v algoritmu Deltablue, který byl popsán v kapitole 2.7.

Místo samostatných vrcholů, představujících proměnné, a hran, představujících podmínky, jsou v sítích buňky. Buňka představuje skupinu podmínek, které by měly být řešeny dohromady. Zároveň reprezentuje i všechny proměnné, které jsou výstupními proměnnými těchto podmínek.

Definice 10 (Buňka) *Nechť C je konečná neprázdná množina označených podmínek se stejnou preferencí a V množina všech proměnných z těchto podmínek. Pro libovolné množiny proměnných $In, Out \subseteq V$ takových, že*

$In \cup Out = V$ a $In \cap Out = \emptyset$ definujeme **buňku podmínek** *jako trojici (C,In,Out).*

Pro každou proměnnou v dále definujeme buňku podmínek ($\{\}$,$\{\}$,$\{v\}$), obsahující pouze výstupní proměnnou v.

Množiny In,Out z buňky (C,In,Out) nazýváme **vstupní** *respektive* **výstupní** *proměnné. Říkáme také, že buňka (C,In,Out)* **určuje** *(determinuje) každou proměnnou z množiny Out*

Definice 11 (Typy buněk) *Buňky podmínek rozdělujeme na následující typy*

- **volné proměnné** *($\{\}$,$\{\}$,$\{v\}$)*
- **funkční buňka** *je taková buňka podmínek ($\{c@l\}$,In,Out), že $Out \neq \emptyset$ a pro libovolné ohodnocení θ vstupních proměnných z In existuje právě jedno ohodnocení σ výstupních proměnných z Out takové, že $c\,\theta\,\sigma$ platí. Poznamenejme, že pokud $In = \emptyset$, pak chceme, aby existovalo právě jedno ohodnocení σ výstupních proměnných z Out takové, že $c\,\sigma$ platí.*
- **generativní buňka** *je taková buňka, že $C \neq \emptyset$, $Out \neq \emptyset$ a nejde o funkční buňku. To znamená, že pro nějaké ohodnocení vstupních proměnných existuje více než jedno a nebo naopak žádné ohodnocení výstupních proměnných tak, aby buňka platila.*
- **testovací buňka** *(C,In,$\emptyset$) je buňka, jejíž množina výstupních podmínek je prázdná.*
- **nerozhodnutelná buňka** *je generativní nebo testovací buňka*

Funkční buňka typicky obsahuje jednu funkcionální podmínku a mívá jednu výstupní proměnnou. Nefunkcionální podmínka se ve funkční buňce nemůže nacházet, protože nesplňuje požadavek na jednoznačnost splňujícího ohodnocení pro výstupní proměnné. Musí být tedy v generativní či testovací buňce. Záležet bude na tom, jestli alespoň jedna proměnná celé buňky bude výstupní.

Volné proměnné a funkční buňky jsou známé z klasických grafů podmínek, tak jak je zavádí Deltablue v [6]. Naopak nerozhodnutelné buňky jsou novinkou. Nerozhodnutelné jsou označeny proto, že v době stavby sítě není možné rozhodnout, jestli budou podmínky v nich splněné. Na druhou stranu, podmínky z funkčních buněk budou splněné vždy.

Ještě je dobré poznamenat, že všechny podmínky v jedné buňce mají stejnou preferenci a tuto preferenci můžeme tedy svázat s celou buňkou a nazývat ji **vnitřní síla** (preference) buňky. Pro volné proměnné je zavedena preference "free", která je slabší než jakákoli jiná preference.

Definice 12 (Síť podmínek) *Nechť* H *je hierarchie podmínek, tj. konečná množina označených podmínek, a V je množina všech proměnných z podmínek v* H*. Potom dvojici (CC,E) nazýváme* **sítí podmínek***, pokud jsou splněny následující podmínky:*

1. *(CC,E) je orientovaný acyklický graf s uzly CC a hranami E*

2. *CC je konečná množina buněk obsahujícím pouze podmínky z* H*, tj.* $\forall Cell \in CC\ Cell = (C, In, Out) \land C \subseteq H$

3. *každá podmínka z* H *je umístěna v právě jedné buňce z CC, tj.* $\forall c \in H\ \exists! Cell \in CC\ Cell = (C, In, Out) \land c \in C$

4. *každá proměnná z V je určena právě jednou buňkou z CC, tj.*

   $\forall v \in V\ \exists! Cell \in CC\ Cell = (C, In, Out) \land v \in Out$

5. *pro každou buňku* Cell *existují hrany v E vedoucí od buněk určujících vstupní proměnné z buňky* Cell *do buňky* Cell*, tj.* $\forall Cell, Cell' \in CC\ :\ Cell = (C, In, Out) \land Cell' = (C', In', Out')$ $\land In \cap Out' \neq \emptyset \Rightarrow (Cell', Cell) \in E$

6. *pro každou nerozhodnutelnou buňku* Cell *neexistuje buňka se stejnou nebo slabší vnitřní silou, která by se nacházela* proti proudu *od* Cell*, tj.* $\forall Cell \in CC$ *Cell je nerozhodnutelná* $\Rightarrow \forall Cell' \in CC$ *tak že* $Cell'$ *v grafu existuje orientovaná cesta z* $Cell'$ *do* $Cell$*,* $Cell'$ *má silnější vnitřní sílu než* $Cell$

7. *v grafu neexistuje žádné větvení* po proudu *v nerozhodnutelné buňce vedoucí do jiných nerozhodnutelných buněk, které nejsou spojeny orientovanou cestou ,tj.*
   $Cell$ *a* $Cell'$ *jsou nerozhodnutelné*
   $\land$ *neexistuje orientovaná cesta z* $Cell$ *do* $Cell'$ *ani z* $Cell'$ *do* $Cell$
   $\Rightarrow \forall Cell'' \in CC$ *tak, že* $Cell''$ *je proti proudu jak od* $Cell$ *tak od* $Cell'$*,*
   pak $Cell''$ *není nerozhodnutelná, tj. je to funkční buňka*

Sit podmínek má následující vlastnost. Pokud je z podmínek hierarchie postavena síť podle definice 12, a pak je topologicky uspořádaná, vznikne posloupnost buněk. Když jsou tyto buňky zjednodušeny pouze na množiny podmínek v nich obsažené, výsledkem bude rozklad původní hierarchie na buňky podle definice 7.

Tato posloupnost nemusí nutně splňovat vlastnost postupného oslabování. Na stranách 71 až 73 práce [1] je detailně diskutováno, že takováto posloupnost buněk může přesto být použita k řešení hierarchie metodou postupného aplikování operátoru splňování hierarchie, protože "nesprávně zařazené" buňky přicházejí z "paralelních větví" výpočtu, a tudíž nemají žádné společné proměnné. Díky tomu nemohou ovlivňovat splnitelnost problematických buněk,tedy těch, ve kterých se vyskytují potenciálně nesplněné podmínky. Proto je možné takto vzniklou posloupnost použít k hledání řešení původní hierarchie metodou postupného aplikování operátoru splňování hierarchie.

3.7 Existující algoritmy

V práci [1] je detailně rozpracovaná právě popsaná teorie, ale algoritmus, který by jí využil je zde pouze naznačen. Dva takové algoritmy, Straton a Straton 2, jsou navžené v [2].

3.7.1 Straton

Algoritmus Straton, staví nejjednodušší možnou síť, tedy takovou, ve které jsou všechny podmínky stejné preference v jedné buňce.

Algoritmus je inkrementální, takže k síti je možné kdykoli přidat další podmínku. Pro zadanou hierarchii je celá síť je postavena postupným přidáváním podmínek v libovolném pořadí.

Algoritmus při přidávání podmínky zjišťuje, zda už existuje buňka se stejnou preferencí jako přidávaná podmínka. Pokud ano, je podmínka přidána do této buňky. Pokud ne, je vytvořená nová buňka a spojena hranami s buňkami s nejbližší vyšší a nejbližší nižší preferencí.

Následně jsou zpracovány všechny proměnné, které nová podmínka váže. Toto zpracování zahrnuje zařazení proměnné mezi vstupní nebo výstupní proměnné, podle toho jestli už je proměnná vázaná silnější buňkou či ne.

Pokud je proměnná přidána mezi výstupní a dosud ji vázala jiná, slabší buňka, je nutné v síti udělat změny. V této slabší buňce je proměnná přesunuta mezi vstupní proměnné. Hrany, které z této slabší buňky vedly do dalších (ještě slabších) buněk, s touto proměnnou jako výstupní, je třeba přesměrovat tak, aby vedly z nové buňky. Ještě je potřeba přidat hranu z nové buňky do buňky, která měla dosud tuto proměnnou mezi svými výstupními proměnnými. Tím je zajištěno splnění pravidla 5 zmiňované definice.

Nastala-li naopak situace, že proměnnou dosud vázala silnější buňka, je přidána hrana z této silnější buňky do buňky s novou podmínkou. Pravidla 6 a 7 definice 12 jsou triviálně splněna díky tomu, že mezi každými dvěma buňkami, které mají "sousední" preference, vede hrana od silnější do slabší.

Jak autor sám přiznává, takto postavená síť vůbec nevyužívá potenciálu teorie, neboť buňky jsou tak velké, jak jen mohou být, což je špatné z hlediska efektivity. Navíc hrany mezi buňkami "sousedních" preferencí jsou v síti vždy, aniž by tam nutně být musely pro splnění definice 12.

3.7.2 Straton 2

Ve stejném textu je popsán i druhý algoritmus, Straton 2. Co se týká velikosti buněk, je stejně "špatný" jako Straton. Všechny podmínky stejné preference jsou v jedné buňce.

Výhodou oproti jeho předchůdci je, že výsledná síť obsahuje méně hran, v nejlepším případě pouze ty, které jsou nutné pro splnění definice 12.

Nevýhodou je pak "zhoršení" inkrementality. Algoritmus přidání podmínky se rozpadl na dvě části. Nejprve se libovolný počet podmínek přidá do sítě a vznikne "částečná" síť. To znamená síť, ve které jsou hrany podle pravidel 5 a 6 definice 12, ale pravidlo 7 nemusí být nutně splněno.

Po této inkrementální části se jednorázově projde celá síť a doplní se hrany tak, aby bylo splněno zbývající pravidlo.

Kromě přidávání podmínek umí tento algoritmus podmínky i ubírat. Stejně jako po přidávání podmínek musí následovat jednorázová oprava sítě.

3.7.3 BlueOne

Na závěr práce [2] je v hrubých rysech nastíněn "lepší" algoritmus. Jeho hlavní výhodou by měla být lepší struktura sítě, tedy buňky menší než "celá úroveň hierarchie". Je pojmenován jako *BlueOne*. Jméno má evokovat jeho souvislost s algoritmem Deltablue, kterému se podobá. Podobnost má být

v inkrementalitě, které má být stejně jako v Deltablue dosahováno pomocí *průchozí síly* ve vrcholech představujících proměnné.

Cíl nového algoritmu Straton 3 je stejný jako cíl BlueOne, postavit "lepší" než stratifikovanou síť. Přistup je ale odlišný. Straton 3 neumí "otáčet hrany" tak jako DeltaBlue a BlueOne. Problém řeší setříděním podmínek, které minimalizuje změny v již postavené síti.

Kapitola 4

Nový algoritmus - Straton 3

Při řešení hierarchií pomocí sítí podmínek, popsaných v předchozí kapitole, lze nalézt řešení pomocí "lepších" i "horších" sítí. Významně se však může lišit efektivita hledání. Kvalita sítě závisí na její granularitě, tedy na velikosti a počtu buněk. Čím je buněk více a čím jsou buňky menší, tím vyšší je granularita a síť je "lepší".

Algoritmy Straton a Straton 2, uvedené na konci předchozí kapitoly stavěly sítě, které zle označit za "nejhorší". Byly stratifikované, tj. pro každou úroveň hierarchie měly jedinou buňku.

Cílem celé této kapitoly je návrh algoritmu, který dokáže postavit "dobrou" síť a pomocí ní najít řešení hierarchie.

Algoritmus, popsaný v této práci, se jmenuje Straton 3. K rozhodnutí použít jméno Straton 3 vedla podobnost "stylu práce" se Stratonem 2. Naopak, návaznost na jméno "BlueOne", použitý pro nástin "lepšího" algoritmu z konce předchozí kapitoly se moc nehodí. Straton 3 se vydal jinou cestou, nepodobá se tolik Deltablue.

V následující sekci bude Straton 3 ve zkratce představen. Další část popisuje omezení algoritmu, tedy typy řešených podmínek, domény proměnných a acykličnost hierarchií. Dále je vysvětlena změna teorie, která umožňuje stavbu "lepších" sítí. Poté je detailně popsána první fáze algoritmu - stavba sítě ze zadané hierarchie podmínek. Následuje popis druhé fáze - získání řešení hierarchie propagací skrze postavenou síť. Na konci kapitoly jsou popsány výsledky hierarchie.

4.1 Koncepce

Algoritmus Straton 3 je postaven na teorii popsané v předchozí kapitole. Kdykoli se tedy hovoří o síti podmínek, je tím myšlena síť podle definice 12.

Když se hovoří o pravidlech (většinou o pravidlech 6 a 7), jsou tím myšlena právě pravidla této definice.

Algoritmus Straton 3 má dvě fáze. V první se staví z podmínek zadané hierarchie síť podmínek. Ve druhé se pomocí této sítě hledá řešení hierarchie.

Tyto fáze jsou na sobě nezávislé. V interaktivním systému, kde se budou měnit *hodnoty* zatímco struktura podmínek zůstane zachována, je možné samostatně postavit síť předem. Pak se pouze na základě změn *hodnot* (třeba poloha myši) hledá řešení pomocí dříve postavené sítě.

Algoritmus 4.1 ukazuje strukturu Stratonu 3, tedy rozdělení do dvou fází, které budou stručně představeny na následujících stránkách.

Algoritmus 4.1: Straton 3

```
1 Straton3(hierarchy H)
2     network N;
3     solution S;
4 begin
5     N := create_network(H);
6     S := find_solution(N);
7
8     return S;
9 end Straton3;
```

4.1.1 Budování sítě

Vstupem této fáze je hierarchie omezujících podmínek. Výstupem má být *co nejlepší* síť podmínek, postavená z této hierarchie podle definice 12.

Algoritmus Straton 3 je neinkrementální, takže se síť pokaždé staví od začátku, není možné modifikovat nějakou předchozí síť.

Pro zajištění platnosti pravidla 4 (každá proměnná je výstupní pro právě jednu buňku) se nejprve do sítě přidají speciální *stay* buňky (více v 4.2.4). Tyto buňky mají speciální "nejslabší" preferenci.

Do této sítě budou postupně vkládány podmínky v pořadí podle své preference od nejsilnějších. Toto pořadí minimalizuje počet změn v již postavené části sítě. Díky uspořádání vkládaných podmínek jsou v tu chvíli v síti pouze buňky silnější, stejně silné jako nově vkládaná podmínka a *stay* buňky.

Pro každou podmínku je nejprve přidána nové buňka, do které je tato podmínka vložena. Při přidávání se proměnné přidávané podmínky rozdělí mezi vstupní a výstupní proměnné nové buňky. Přitom může dojít k odstranění nějakých *stay* buněk.

Následně je vynucováno splnění pravidla 6 pro právě přidanou buňku. Toto pravidlo zakazuje slabší nebo stejně silné buňky proti proudu od nerozhodnutelné buňky. Tato operace může znamenat spojování s jinými buňkami (stejné síly) nebo odstranění slabších buněk z grafu (právě jen *stay* buňky jsou v síti v tu chvíli slabší).

Stejným způsobem se nakonec zařadí i všechny vyřazené *stay* buňky. Jelikož v síti nejsou žádné buňky slabší než *stay* buňky, nedojde při tom už k žádnému odstraňování ze sítě.

Nakonec, po zpracování všech podmínek, dojde k vynucení pravidla 7 pro celou síť. Toto pravidlo zakazuje určité "větvení" v nerozhodnutelných buňkách. Vynucování tohoto pravidla obnáší přidávání hran nebo spojování některých stejně silných buněk.

Celá fáze je detailněji popsána v sekci 4.7.

4.1.2 Hledání řešení

Cílem této fáze je najít ohodnocení všech proměnných v hierarchii H pomocí sítě podmínek, vzniklé v první fázi. Tato fáze se velmi podobá algoritmu Indigo, jak byl popsán v sekci 2.8. Rozdíl je v pořadí, v jakém se zpracovávají jednotlivé podmínky. Toto pořadí je dáno strukturou sítě.

V každém kroku je zpracována jedna podmínka. Toto zpracování znamená zmenšení domén proměnných vázaných touto podmínkou. Pokud existuje nějaké kombinace hodnot z domén jednotlivých proměnných taková, že je podmínku možné splnit, ponechají se v doménách právě ty hodnoty, které se účastní nějaké takové splňující kombinace a ostatní se odstraní. Není-li možné podmínku splnit pro žádnou kombinaci hodnot, zůstanou v doménách jen ty hodnoty, pro které je chyba na podmínce nejmenší.

Po té, co jsou takto zmenšeny domény proměnných vázaných právě zpracovanou podmínkou, je potřeba tyto změny propagovat přes další už zpracované podmínky.

Detailní popis této fáze je v sekci 4.8.

4.2 Použitá omezení

Omezující podmínky mohou být velmi různorodé. Z hlediska algoritmu Straton 3 je zajímavé, kolik proměnných váží a jestli jsou funkcionální či nikoli.

Co se týká počtu vázaných podmínek, omezil jsem se v implementaci na podmínky vážící jednu, dvě a tři proměnné. Pro účely demonstrace fungování to bohatě postačuje. V praxi bude samozřejmě užitečné implementovat podmínky konkrétně podle potřeb zadání. Kdybych se omezil na podmínky pro dvě proměnné, nemohlo by existovat větvení a řešené problémy by byly naprosto nezajímavé.

4.2.1 Metody s jedním výstupem

Dalším omezením pro jednodušší prezentaci algoritmu je použití podmínek, jejichž metody mají jen jeden výstup. Pokud bychom chtěli metody s více výstupy používat, mírně by se zkomplikovala první fáze algoritmu, tedy tvorba sítě podmínek. Propagační fáze by v takovém případě zůstala beze změn.

Dobrým příkladem podmínky s více výstupy je převod souřadnic kartézských na polární a naopak, jak bylo popsáno na straně 5. Pro připomenutí, taková podmínka má dvě metody, převádějící vždy obě souřadnice z jednoho systému na obě souřadnice druhého systému, což lze zaznamenat takto:

- $[r, \omega] \Leftarrow [x, y]$
- $[x, y] \Leftarrow [r, \omega]$

V této podobě je podmínka funkcionální - pro r a ω existuje právě jedna kombinace x a y tak, že podmínka platí. Stejně tak jednoznačný je převod i opačným směrem.

Je samozřejmě možné použít tuto podmínku i jako podmínku s jedním výstupem, kdy na základě třech proměnných určujeme čtvrtou, například $r \Leftarrow [x, y, \omega]$. Pak ale podmínka nebude funkcionální, protože nelze zaručit že bude existovat řešení. Spíše naopak, pokud ω nebude odpovídat hodnotě vypočtené na základě x a y, bude podmínka nesplněná pro libovolné r.

Jak je na tomto příkladě vidět, ke každé podmínce s více výstupy je potřeba mít seznam "povolených" kombinací vstupních a výstupních proměnných. Ostatní kombinace vstupních a výstupních proměnných nejsou "zakázané", jen v tomto případě nebude podmínka funkcionální.

Při přidávání podmínky je pak potřeba dávat pozor na tyto kombinace. Pokud je možné přidat podmínku do buňky tak, že výstupními proměnnými bude právě některá "povolená" kombinace proměnných, může být buňka funkční.

Naopak, pokud silnější buňky v grafu neumožňují vybrat žádnou takovou kombinaci, půjde o buňku testovací (bez jediné výstupní proměnné) či o buňku generativní, pokud bude mít buňka jednu nebo více výstupních proměnných (ale ty k sobě "nebudou dobře pasovat", jako třeba x a ω)

Snaha vyhnout se těmto problémům vedla k použití výhradně podmínek s jedním výstupem.

4.2.2 Read-only proměnné

Read-only proměnné jsou často způsobem, jak do hierarchie promítnout nějaká měnící se vstupní data. Často uváděným příkladem je pohyb myši. Na tento pohyb mají nějakým způsobem, definovaným právě hierarchií podmínek, reagovat svým pohybem objekty na obrazovce.

Takovéto read-only proměnné lze snadno reprezentovat pomocí *required* podmínky $a = x$, kde a je jméno proměnné a x je hodnota, které má nabývat. Jelikož preference podmínky je *required*, bude vždy splněna (kromě případů, kdy samotná *required* úroveň preference představuje *příliš omezený problém* a hierarchie tedy nemá *žádné* řešení.)

Pro případ že požadavek bude, aby nějaká proměnná a byla rovna nějaké z venku měněné hodnotě x "jen pokud je to možné" v závislosti na ostatních podmínkách, tedy s preferencí *pref*, Dá se to vyřešit dvojicí podmínek. První *required* podobně jako v předchozím případě nastaví "pomocnou proměnnou" $a' = x$. Druhá podmínka $a = a'$ se zadanou preferencí *pref* pak zajistí rovnost "v rámci hierarchie".

4.2.3 Neostré nerovnosti

Při implementaci podmínek jsem narazil na otázku, jakým způsobem zavést nefunkcionální podmínky[1]. Kdybych zavedl ostré i neostré nerovnosti, bylo by to samozřejmě obecnější. Rozhodl jsem se ale ostré nerovnosti vůbec neimplementovat a navíc je neřešit ani na obecné úrovni.

Algoritmus Straton 3 , stejně jako Indigo jsou dostatečně obecné pro řešení hierarchií s použitím *locally-error-better* komparátoru. Oba tyto algoritmy umí řešit nefunkcionální podmínky, použít *locally-predicate-better* by byla škoda a ubírala by jim na síle. Samozřejmě, pokud je *locally-predicate-better* vyžadován, lze oba tyto algoritmy "okleštit" tak, aby hledaly *locally-predicate-better* řešení. Šlo by o odstranění toho kusu kódu, který se v případě nesplnitelnosti podmínky snaží minimalizovat chybu na této podmínce.

Při použití *locally-error-better* komparátoru nastává problém při řešení podmínek s ostrou nerovností. Jako příklad vezmu hierarchii, která pro *locally-error-better* komparátor řešení nemá, zatímco pro *locally-predicate-better* ano.

$$\begin{array}{llll} P1 & a > 0 & @ & strong \\ P2 & a = 0 & @ & medium \end{array}$$

Když bude mít proměnná a doménu $\mathbf{R}$, pak je pro *locally-error-better* komparátor řešením interval $(0, \infty)$. Tento interval splňuje podmínku P1 bezezbytku. Podmínka P2 nemůže být na tomto intervalu splněna, takže tato podmínka žádným způsobem nemůže ovlivnit výběr ohodnocení pro celou hierarchii.

Pokud budeme hledat řešení pro *locally-error-better* komparátor, žádné řešení nenajdeme. Silnější podmínka P1 nám omezuje možnou doménu a na interval $(0, \infty)$. Když nyní pro proměnnou a vezmeme libovolnou hodnotu $x \in (0, \infty)$, pak pro ohodnocení kde $a = x/2$ bude chyba na podmínce P2 menší než pro $a = x$, takže x nemůže být řešením hierarchie. Z toho je patrné, že pro tento komparátor nemá tato hierarchie žádné řešení.

Když používáme uzavřené intervaly, víme, že pokud doména proměnné obsahuje alespoň jeden interval, obsahuje alespoň jednu přípustnou hodnotu. Pokud bychom používali i otevřené intervaly, měli bychom s tím problém. Zvláště problematická by se stala otázka důkazu existence řešení - sekce 4.8.7 na straně 89. Je tam využíváno faktu, že v doméně každé proměnné

[1] pro funkcionální je "rovnost" ideálním zástupcem

neustále zbývá alespoň jedna přípustná hodnota. Pro otevřené intervaly by se tento požadavek musel nějak speciálně vynucovat či kontrolovat a nelze vyloučit, že by pak algoritmus nemusel nalézt žádné řešení. Stačí se podívat na příklad diskutovaný v této sekci. Aby bylo možné se vyhnout všem těmto komplikacím, budou implementovány pouze neostré nerovnosti.

4.2.4 Podmínky stay

Hierarchie byly zavedeny, aby si poradily s příliš omezenými problémy. Problémem by se ale naopak mohlo stát příliš volné zadání. Takovému stavu je dobré se vyhnout a po vzoru Indiga je třeba do zadání přidat novou vrstvu podmínek.

Když propagační fáze projde všechny buňky, není vyloučeno, že doména některé proměnné bude obsahovat více než jednu hodnotu. Tento stav není vítaný - potřebujeme najít jedno řešení a pro takové "příliš volné" proměnné nějakým jednoznačným způsobem vybrat hodnotu.

Zvoleným řešením je požadovat, aby v zadání hierarchie byla pro každou volnou proměnnou právě jedna speciální podmínka. Tyto podmínky budeme po vzoru Indiga nazývat *stay* podmínky. Budou mít speciální preferenci, která je slabší než preference všech ostatních "běžných podmínek". Tyto podmínky budou funkcionální a budou se podobat unární rovnosti (proměnné a konstanty).

Všechny podmínky hierarchie, které nejsou *stay* podmínkami, budou nadále v textu označované jako "běžné", pokud to bude potřeba pro odlišení od *stay* podmínek.

Podmínky *stay* jsou zavedeny proto, aby po jejich zpracování zůstala v doméně právě jedna hodnota. Pokud je proměnná $a \in \langle 10, 20\rangle \cup \langle 40, 50\rangle$ propagována přes "obyčejnou unární rovnost" $a = 30$, není jednoznačnosti a dosaženo. Hodnoty 20 i 40 mají stejnou chybu, takže po propagaci přes tuto podmínku bude $a \in \langle 20, 20\rangle \cup \langle 40, 40\rangle$.

Proto má *stay* podmínka speciální chování různé od unární rovnosti. Pokud existuje více možných hodnot, vybere jen jednu, tu nejmenší z nich.

Nedá se říct, že by *stay* podmínky "nebyly" součástí zadání. Dají se s výhodou použít k zajištění "stability" systému, který používá řešení hierarchie jen jako jednu svou komponentu. Tyto podmínky pak budou vynucovat, aby proměnné nabývaly "svých" hodnot z předchozích iterací. Tím bude zaručeno, že se změní hodnoty jen těch proměnných, u kterých to bude vyžadováno ostatními "běžnými" (a tedy silnějšími) podmínkami.

Kromě podmínek typu *stay* se v textu bude hovořit i o *stay* buňkách. Jsou to buňky obsahující *stay* podmínku pro některou proměnnou, případně více *stay* podmínek sloučených do jedné buňky. Jelikož *stay* podmínky mají vyhrazenu speciální preferenci, nemůže být v jedné buňce *stay* podmínka společně s jinou "obyčejnou" podmínkou.

Pokud to konkrétní příklady hierarchií uvedené dále v textu nepotřebují, nebudou tam *stay* podmínky uvedeny. Díky tomu nebudou příklady zbytečně velké. Na druhou stranu, pro získání řešení hierarchie v propagační fázi jsou *stay* podmínky nezbytné a nesmí proto chybět v zadání.

4.3 Zobrazení sítě podmínek

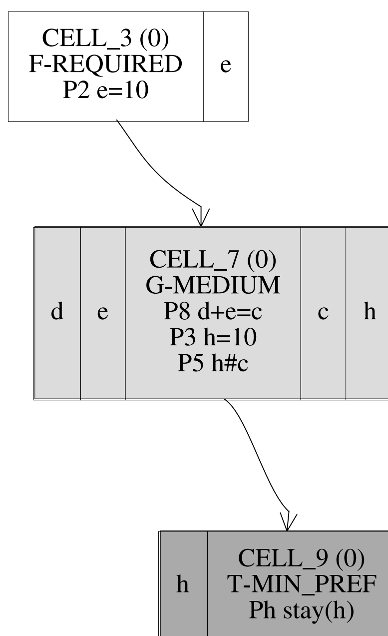
Ze zadané hierarchie omezujících podmínek se v první fázi buduje síť podmínek, ve které se v druhé fázi hledá řešení. Je příjemné, pokud si takovou síť můžeme prohlédnout. Vizualizace sítě byla nezbytná při ladění budování sítě. Implementaci jsem psal v jazyce C, přičemž výstupem byl mimo jiné textový soubor popisující síť podmínek. Pro vizualizaci jsem následně používal program *dotty* (součást balíku *graphviz*[2])

Toto řešení je výhodné z hlediska pracnosti - příprava vstupu v textovém formátu není složitá a výsledkem je víceméně hezký graf. Jsou tu ovšem i nevýhody. Utilitka dotty není bezchybná - občas se stane, že nějakou hranu v grafu nezobrazí viditelně (zobrazí ji 'pod' jinou hranu). Buňky sítě jsou zobrazeny jako netriviální objekty, jak je popsáno dále. Šipky zobrazující hrany někdy vedou pod samotnou cílovou buňkou. Výsledkem je, že hrana končí na okraji buňky v oblasti jiné části, než do které má správně mířit.

Omezující podmínky, kterými se budeme zabývat, mají několik základních atributů. V první řadě jde o seznam proměnných, které daná podmínka váže. Stejně podstatným atributem je preference. Dále potřebujeme vědět, zda z dané podmínky může vzniknout funkční buňka. Tedy jestli pro libovolné ohodnocení vstupních proměnných, existuje právě jedno ohodnocení proměnných výstupních. Zjednodušeně můžeme říci, že z nerovnice funkční buňka vzniknout nemůže, z rovnice naopak ano. Třetím atributem je jméno podmínky. To není podstatné pro řešení hierarchie ani budování sítě. Naopak při vizualizaci sítě se bez něj neobejdeme. Je nutné poznamenat, že *jméno* podmínky nemusí nijak souviset se jmény proměnných, které omezuje, ani

[2]http://www.graphviz.org

typem podmínky. Na druhou stranu je dobré, aby tyto údaje korespondovaly, jinak nám taková vizualizace příliš nepomůže.



Obrázek 4.1: Část vizualizované sítě podmínek

Podívejme se tedy na část sítě [3] vizualizovanou pomocí *dotty* na obrázku 4.1. Vidíme zde tři buňky a hrany mezi nimi. Buňky jsou z hlediska formátu vstupního souboru pro *dotty* vrcholy grafu typu *record*. Každá buňka má tři hlavní části. Vstupní proměnné vlevo, jádro buňky uprostřed a výstupní proměnné vpravo.

Vstupní a výstupní proměnné jsou realizovány pomocí *portů* - samostatných sekcí pro každou proměnnou. Naopak, každá proměnná vázaná nějakou podmínkou v buňce, musí být buď vstupní nebo výstupní proměnná (z definice sítě podmínek), a tudíž bude ve vstupní nebo výstupní části reprezentována. Pro každou proměnnou je její port pojmenován jejím jménem. Při práci na stavbě sítě jsem používal jako označení malá písmena ze začátku abecedy. Hierarchie generované pro testovací účely pojmenovávají proměnné jako 'v'+číslo.

[3] Jde jen o výřez sítě. Chybí zde například buňka, která by vázala proměnnou d.

Jádro buňky nese většinu informací o buňce. Předně je to v prvním řádku její interní označení ('CELL_'+pořadové číslo buňky). Buňky dostávají svá čísla v okamžiku vzniku a tato čísla se nerecyklují. Čísla se berou ze vzestupné řady a jsou tedy unikátní. Číslování ale nemusí být "husté", některá čísla se v řadě nemusí vyskytovat. Mezery v číselné řadě jsou způsobené zániky buněk, k čemuž může dojít například při sloučení s jinou buňkou. Číslo v závorce za identifikací buňky je použitelné pro zobrazování libovolného numerického atributu buňky. Například bude užitečné, když budeme chtít ukázat v jakém pořadí se budou buňky procházet, v té které části algoritmu.

Druhý řádek nese informaci o typu buňky a její preferenci. Písmeno na začátku je F pro funkční buňky, G pro generativní a T pro testovací. Za pomlčkou se nachází slovní alias preference (interně jsou preference reprezentovány proměnnou typu *int*).

Na dalších řádcích následuje seznam podmínek v této buňce, každá podmínka má vlastní řádek. Pro pojmenování podmínek používám konvenci *'P'+číslo/písmeno+mezera+formule_podmínky*. Pro běžné podmínky používám čísla. Jen pro nejslabší podmínky, určené k jednoznačnému zafixování hodnot jednotlivých proměnných, používám písmeno shodné se jménem fixované podmínky.

Co se týká formulí podmínek, jde o značné zjednodušení. Při budování sítě nás totiž nezajímá konkrétní podoba podmínky, ale jen proměnné, které váže a její případná funkcionalita. Pro funkcionální podmínky se ve formuli objeví znak '=', pro ostatní '#' , který bude v naší implementaci zastupovat podmínky obsahující neostrou nerovnost na jednu či druhou stranu. Třetím typem podmínek jsou již zmiňované podmínky *stay* diskutované v sekci 4.2.4. Již zmiňovanou součástí konvence je použití jmen všech proměnných, které se v podmínce vyskytují.

Orientované hrany v síti jsou reprezentovány orientovanými hranami ve vizualizaci. V textovém souboru pro *dotty* je specifikováno, že šipka má vést z 'jádra' rodiče do 'jádra' potomka. Jak už jsem zmínil, ne vždy je to dobře zobrazeno. Podle definice vede povinně hrana mezi dvěma buňkami, které mají neprázdný průnik výstupních proměnných první buňky a vstupních proměnných druhé. Taková hrana by měla být jen jedna, i pokud mají buňky takový průnik o více prvcích. Na druhou stranu nadbytečné hrany v grafu nejsou zakázané. Zde bohužel narážíme na omezení *dotty*. Při interaktivním používání dokáže pro hrany použít různé styly, které se dají použít k odlišení různých typů hran (hrany povinné podle definice, hrany vynucené pravidlem 7, redundantní hrany závislosti proměnných). Bohužel, při exportu do post-

formule	typ	popis
a=10	F	unární podmínka vážící 'a' a konstantu
a#10	G/T	unární nefunkcionální podmínka, např. $a \leq 10$
a=b	F	funkcionální podmínka vážící dvě proměnné
a#b	G/T	podmínka vážící dvě proměnné, např. $a \leq b$
a+b=c	F	podmínka vážící tři proměnné
a+b#c	G/T	podmínka vážící tři proměnné, např. $a - b \leq c$
stay(a)	F	unární podmínka typu *stay* na proměnné 'a'

Tabulka 4.1: Přehled formulí pro reprezentaci podmínek při vizualizaci sítě

scriptu je tato možnost degenerována do té míry, že je naprosto nepoužitelná pro tištěný dokument.

4.4 Domény

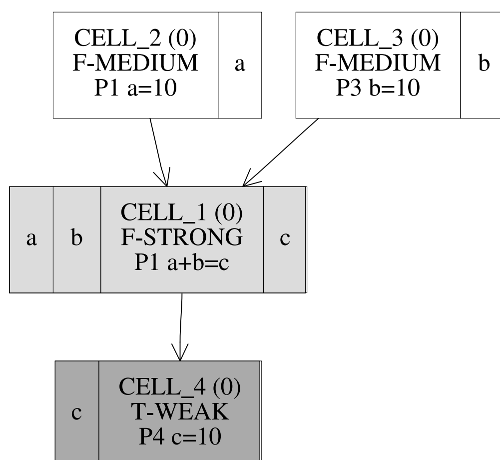
Podkladová teorie neklade žádné explicitní požadavky na domény jednotlivých proměnných. Pokoušel jsem se rozpracovat variantu algoritmu Straton 3 , která by pracovala s konečnými doménami jednotlivých proměnných. Tato varianta se zdála být lákavou hned z několika důvodů. Jelikož podkladová teorie umožňuje najít všechna řešení dané hierarchie, bylo by pak možné najít *všechna* řešení dané hierarchie. To je něco, co Indigo neumí - to hledá vždy pouze jediné řešení. Dále konečné domény zabraňují nekonečným smyčkám. Při procházení sítě podmínek pro nekonečné domény může vzniknout cyklus neustálého zmenšování domény některé proměnné. Pokud bychom problém řešili na konečné doméně, po konečném počtu kroků bychom tuto doménu vyčerpali. Další zajímavou možností by bylo pokusit se prozkoumat možnosti lepší lokální propagace, při použití konzistenčních technik.

Bohužel, konečné domény jsou nevhodné pro zpracování pomocí sítí podmínek. Problém vyvstane při detailnějším pohledu na nutnou vlastnost funkčních buněk. V definici 11 je vyžadováno, aby *... pro libovolné ohodnoceni θ vstupních proměnných z In existuje právě jedno ohodnocení σ výstupních proměnných z Out takové, že $c\,\theta\,\sigma$ platí* Když budeme mít domény volných proměnných definované jako (shora i zdola omezené) intervaly celých čísel, ani nejjednodušší matematické operace nebudou na těchto doménách uzavřené. Tím, že výsledek operace bude ležet mimo aktuálně platnou do-

ménu (podmnožinu původní domény proměnné), nebude možné podmínku beze zbytku splnit. To tedy znamená, že nebude platit uvedená vlastnost funkčních podmínek.

Tuto situaci můžeme ilustrovat na jednoduchém příkladu. Mějme tři volné proměnné a,b,c. Jako doménu pro všechny tři proměnné zvolíme interval celých čísel $\{1, 2, \ldots, 10\}$. Hierarchie bude obsahovat pouze podmínky funkcionálního typu (tj. takové, že kterých mohou za příznivých okolností vzniknout funkční buňky). Hierarchie bude:

$$\begin{array}{llll} P1 & a+b=c & @ & strong \\ P2 & a=10 & @ & medium \\ P3 & b=10 & @ & medium \\ P4 & c=10 & @ & weak \end{array}$$



Obrázek 4.2: Omezené domény

Síť na obrázku 4.2 odpovídá všem pravidlům, je tedy korektní. Buňka obsahující podmínku P1 je funkční, takže před ní mohou být libovolné, tedy i slabší buňky obsahující podmínky P2 a P3. Jedinou buňkou, která není funkční ale testovací je ta, která obsahuje podmínku P4. Všichni její předchůdci ale mají silnější preferenci, takže vše je v pořádku.

Pokud ale nyní začneme takto postavenou síť řešit, musíme nejdříve zpracovat buňky obsahující unární podmínky P2 a P3. To znamená, že aktuální domény proměnných 'a' a 'b' se zmenší na jedinou hodnotu 10. Ovšem, když

budeme nyní chtít tyto hodnoty propagovat přes buňku obsahující P1, narazíme na problém. Doména proměnné 'c' je omezená na interval $\{1, 2, \ldots, 10\}$. To znamená, že i když proměnná 'c' nabude hodnoty 10, podmínka P1 bude nesplněná - s chybou 10 použijeme-li obvyklý metrický komparátor. Přitom chybu na podmínce P1, a tedy na úrovni *strong* byla zapříčiněná splněním podmínek na úrovni *medium*.

Toto je důsledek nesplnění nutné vlastnosti funkčních buněk při použití konečných domén. Tímto směrem tedy cesta nevede.

Dá se namítnout, že omezení domén proměnných na interval $\{1, 2, \ldots, 10\}$ bylo provedeno jaksi 'mimo mechanizmus hierarchie'. Že se takovéto tvrdé omezení musí realizovat v rámci zadání hierarchie a ne na úrovni řešícího systému, jak jsme to udělali v tomto příkladě.

Co to tedy znamená, omezit domény proměnných nativními prostředky hierarchií podmínek? Pro každou proměnnou 'a' budeme do hierarchie muset přidat novou podmínku *P'a' Dom(a)*, kde Dom platí právě tehdy, když je hodnota 'a' z požadované domény. Takováto podmínka zjevně nemůže být základem funkční buňky pro jakoukoliv doménu, která má více než jeden prvek (a takováto triviální doména nám k ničemu nepomůže). Bude z ní tedy generativní nebo testovací buňka.

Při přidání takových podmínek do hierarchie je potřeba rozhodnout, jak s nimi zacházet. Konkrétně jde o to, "kdy" budou tyto podmínky zpracovány. Máme v podstatě dvě možnosti.

První možností je vynutit zpracování těchto nových podmínek ještě před zpracováním jakékoli "původní" podmínky. To můžeme zajistit jedině tak, že každá podmínka bude mít slabší preferenci.

Dá se ale očekávat že původní hierarchie měla nějaké *required* podmínky. Z definice jsou to nejsilnější/nejvíce preferované podmínky. V tom případě budeme muset nějakým "trikem" dostat 'Dom' podmínky ještě před ty *required*. V implementaci se to provede mapováním 'Dom' úrovně na ještě menší hodnotu, než na jakou je mapovaná úroveň *required*.

Jak jsme si již řekli, půjde v případě netriviálních domén o nefunkční, a tedy nerozhodnutelné buňky. Tím pádem díky pravidlu 6 víme, že proti proudu od 'Dom' podmínky nemohou být žádné stejně silné nebo slabší buňky. Tedy žádné buňky původní hierarchie. Tím máme zaručeno zpracování 'Dom' podmínky pro každou proměnnou předtím, než ji začne zpracovávat libovolná 'původní' podmínka.

Toto řešení má ale jednu zásadní nevýhodu. Každá proměnná je z definice výstupní proměnnou *právě jedné* buňky. Všechny proměnné jsou ovšem vý-

stupními podmínkami v 'Dom' vrstvě. Tím pádem budou všechny původní podmínky pouze v testovacích buňkách, protože budou mít prázdnou množinu výstupních podmínek. Vynucování pravidel 6 a 7 ovšem způsobí, že se nám graf uspořádá do jakéhosi hroznu. V nejvyšší úrovni bude pro každou proměnnou jedna 'Dom' buňka. Naopak na nejnižší úrovni 'Stay' podmínek bude jedna jediná superbuňka[4]. Na úrovních odshora dolů bude klesat počet buněk a růst jejich velikost[5] podle toho, jak se bude postupně zvyšovat četnost 'společných předků' pro různé podmínky.

Následné řešení takovéhoto grafu nebude vůbec efektivní - čím větší buňky, tím hůře. V podstatě se přiblížíme stratifikované síti, kde jsou všechny podmínky na dané úrovni sloučeny do jedné buňky a místo levné lokální propagace musíme 'draze' řešit vztahy v buňce.

Druhou možností zavedení 'Dom' podmínek, je jejich umístění na nějakou úroveň slabší než nejsilnější 'původní' podmínky. V takovém případě dojde k řešení některých (případně všech) původních podmínek před ořezáním domén na konečné množiny. To znamená, že před tím, než bychom mohli začít využívat výhody konečných domén (např žádné nekonečné cykly), může se stát, že do nějakého takového nekonečného cyklu spadneme a k omezujícím 'Dom' podmínkám se už nikdy nedostaneme. V podstatě bychom takovým přidáním 'Dom' podmínek nezískali vůbec nic a jen by se nám zhoršila síť[6].

Z tohoto důvodu jsem opustil další pokusy v oblasti konečných domén (snadno a hezky reprezentované malými množinami přirozených čísel) a nadále budeme pracovat s reálnými čísly - v rámci běžné přesnosti. Tato práce je zaměřena na algoritmy pro řešení hierarchií a ne na numerické metody.

4.5 Cykly

Problémem, se kterým si spousta algoritmů pro řešení hierarchií neumí poradit, jsou cykly podmínek. To se týká mimo jiné i Indiga. Indigo spoléhá na acykličnost, když propaguje změnu hodnoty proměnné. Očekává, že propagační vlna se nikdy nedostane k proměnným, které už byly jednou změ-

[4]Důsledek pravidla 7 - za G/T buňkou nesmí být dvě různé G/T buňky nespojené cestou a pravidlo 6 zase ze dvou G/T buněk stejné preference na orientované cestě vyžaduje spojení těchto buněk do jedné. O grafu můžeme předpokládat, že je spojitý, a tudíž mají libovolné dvě *stay* podmínky společného předka

[5]za předpokladu rovnoměrného rozdělení podmínek mezi úrovně preference

[6]přidání G/T podmínky si často vynutí slévání buněk nebo nové hrany

něny.

Cykly podmínek jsou definovány v sekci 2.5 na straně 11. Příklady v této kapitole budou obsahovat nejmenší možné cykly, tedy dvě podmínky a dvě proměnné. Nicméně popisované principy platí pro libovolně velké cykly. Navíc se samozřejmě může vyskytnout několik cyklů současně, a to nejen disjunktních, ale i se společnými vrcholy v grafu - tedy společnými proměnnými či podmínkami.

Podkladová teorie pro sítě podmínek cykly mezi podmínkami nevylučuje. Problém nastává, když chceme od teorie přejít k implementaci samotného algoritmu. Jelikož domény proměnných nemohou být konečné (viz sekce 4.4 na straně 47) ,nemůže být konečný ani počet ohodnocení. Z tohoto důvodu je implementace omezena na propagaci aktuálních domén pro *jednotlivé* proměnné. Takto zvolená implementace s sebou ovšem nese komplikaci, týkající se řešení hierarchií obsahujících cykly.

Nejprve je dobré v jednoduchosti přiblížit fungování propagační fáze. Postupně se zpracovávají podmínky a v závislosti na tom se zmenšují domény proměnných. Propagace změn spočívá v tom, že když je změněna doména nějaké proměnné, je potřeba znovu zpracovat podmínky, které tuto proměnnou omezují. Tím že z domény jedné proměnné zmizí nějaké hodnoty, stane se podmínka nesplnitelnou pro nějaké hodnoty jiné proměnné.

To je možné ilustrovat na malém příkladu. Nechť a a b jsou proměnné, jejichž domény po zpracování podmínky $a+10 = b$ jsou $a \in \langle 1, 5\rangle$ a $b \in \langle 11, 15\rangle$. Když je pak zpracována podmínka $a \geq 3$, změní se doména a na $\langle 3, 5\rangle$. Tím se ale u předchozí podmínky stanou hodnoty 11 a 12 nekonzistentními, protože pro ně neexistuje takové a, aby byla podmínka splněna. Proto musí dojít i ke zmenšení domény proměnné b, která se zmenší na interval $\langle 13, 15\rangle$. Pokud by další podmínka vázala proměnnou b, bude nutné znovu zpracovat i ji a případně další podmínky, které by mohly být ovlivněny touto "vlnou změn"

Propagace vlny zmenšování domén je jednou ze základních myšlenek Indiga. Je to dobře vidět v proceduře tighten_bounds 2.3 na straně 22.

Podívejme se nyní podrobněji na příklad hierarchie s cyklem. Budeme potřebovat nějakou generativní buňku 'nahoře proti proudu', aby bylo 'kam se vracet' při změně domény. Nechť je to tedy

$$P1 \quad a \geq 0 \ @ \ required$$

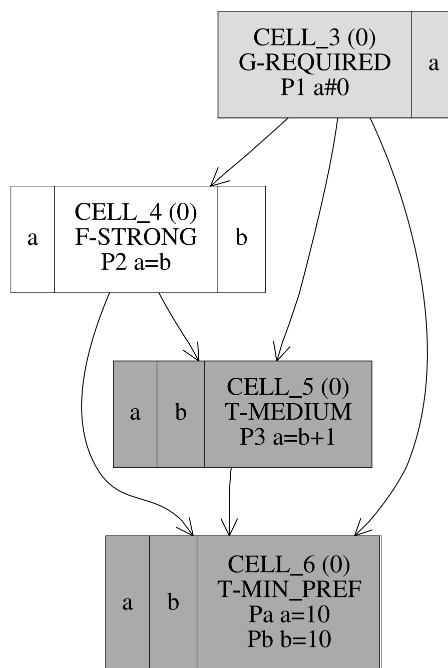
Pak budeme potřebovat dvě podmínky vážící stejné proměnné, ovšem ve vzájemném konfliktu. Budou mít různé preference. Bude to tedy

$$\begin{array}{llcl} P2 & a = b & @ & strong \\ P3 & a = b + 1 & @ & medium \end{array}$$

Nakonec přidáme nejslabší *stay* podmínky, aby problém nebyl příliš volný.

$$\begin{array}{llcl} P4 & a = 10 & @ & min_pref \\ P5 & b = 10 & @ & min_pref \end{array}$$

Tím je zadání hotovo. Tato hierarchie má pro lokální predikátový i metrický komparátor právě jedno řešení $a = b = 10$. Pro toto řešení jsou splněny podmínky P1 a P2. Podmínka P3 splněna není, přitom chyba je 1 pro oba komparátory. Podmínky P4 i P5 jsou splněny. Pro jakékoli jiné ohodnocení, kde $a \neq b$, není splněna podmínka P2 na *strong* úrovni, což je ostře horší než naše řešení. Pro jinou hodnotu x proměnných a i b než 10, pak nebudou splněny podmínky P4 i P5 s chybou $|10 - x|$. Přičemž chyba na vyšších úrovních je stejná nebo horší než pro toto řešení.



Obrázek 4.3: Příklad cyklu

V tabulce 4.2 je vidět, jak cyklus vyřazování hodnot z jednotlivých domén poběží pro jednoduchou propagaci do nekonečna. Přitom existuje právě jedno locally-metric-better řešení.

akce	a	b
START	$(-\infty, \infty)$	$(-\infty, \infty)$
P1	$\langle 0, \infty)$	$(-\infty, \infty)$
P2	$\langle 0, \infty)$	$\langle 0, \infty)$
P3	$\langle 1, \infty)$	$\langle 0, \infty)$
kontrola na P1	$\langle 1, \infty)$	$\langle 0, \infty)$
propagace P2	$\langle 1, \infty)$	$\langle 1, \infty)$
návrat do P3	$\langle 2, \infty)$	$\langle 1, \infty)$
kontrola na P1	$\langle 2, \infty)$	$\langle 1, \infty)$
propagace P2	$\langle 2, \infty)$	$\langle 2, \infty)$
návrat do P3	$\langle 3, \infty)$	$\langle 2, \infty)$
a opět na P1		

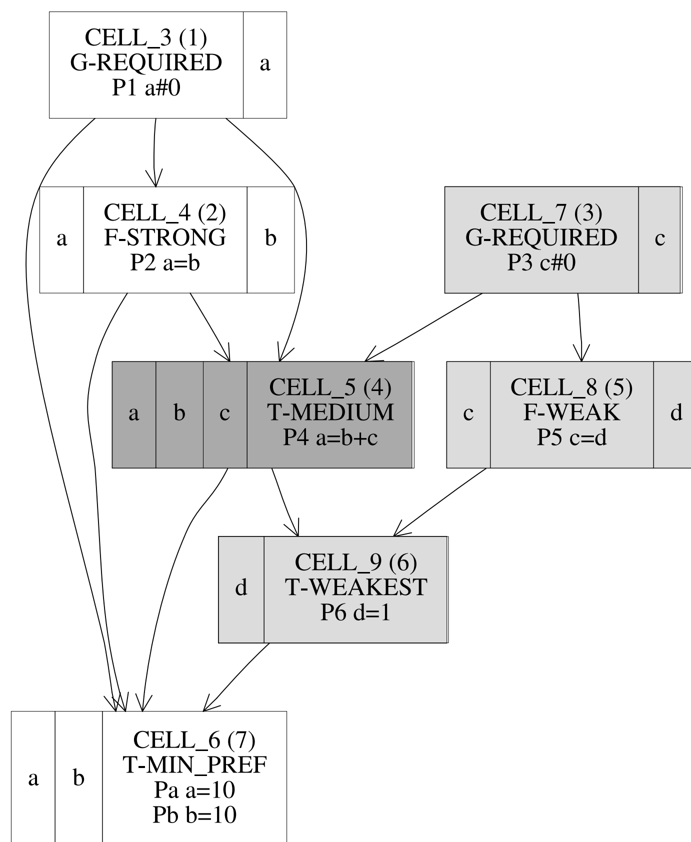
Tabulka 4.2: Nekonečné zmenšování domén proměnných

Aby to nebylo tak jednoduché, můžeme najít příklad, ve kterém podmínkami na cyklu projdeme bez problému a budeme pokračovat dál ve zpracovávání sítě. Až v jiné části sítě se vyskytne testovací podmínka. Ta ovlivní proměnnou na svém vstupu a teprve propagace této změny do buňky, která má tuto proměnnou na svém výstupu a následně propagace směrem dolů nastartuje cyklus 'v jiné části grafu'

Podívejme se na následující příklad na obrázku 4.4, který vychází z předchozího. Přidali jsme světle šedé buňky a tmavě šedá buňka se změnila, zatímco nezměněné buňky jsou bílé. Číslo v závorce za identifikací buňky nyní obsahuje pořadí v jakém by se buňky zpracovávaly, tedy výsledek topologického třídění.

Tmavě šedá buňka předtím obsahovala podmínku $a = b + 1$. Ta způsobovala, že po propagaci touto buňkou se posune dolní mez na proměnné a o jedna nahoru, a tím se roztáčí cyklus změn na proměnných a a b. Nyní je tam podmínka $a = b + c$. Když se zpracovává poprvé tato část sítě, má proměnná c doménu $\langle 0, \infty)$. To znamená, že není potřeba upravovat hodnotu proměnné a, protože pro $c = 0$ platí, že $a = b$. Nyní se algoritmus vydá řešit 'vpravo' od buňky 7, to znamená P5 $c = d$. Poprvé se vyskytující proměnná d bude mít nyní stejnou doménu jako c, protože její doména dosud nebyla vůbec omezená. Jde o funkční buňku, takže nemůže dojít při jejím zpracování ke změně domény vstupní proměnné c.

Pak se bude zpracovávat buňka 9, tj. podmínka P6 $d = 1$. Hodnota 1



Obrázek 4.4: Příklad složitějšího cyklu

je prvkem aktuální domény proměnné d, tj. $\langle 0, \infty)$, a je tedy možné splnit tuto podmínku. Tato buňka není funkční, a proto je možné změnit doménu vstupní proměnné d na $\langle 1, 1\rangle$. Tuto změnu musíme propagovat 'proti proudu' do sítě podmínek.

Proměnnou d určuje (má mezi svými výstupními podmínkami) buňka 8, která obsahuje jedinou podmínku $c = d$. Aktuální doména proměnné c je $\langle 0, \infty)$ a je tedy možné ji omezit na $\langle 1, 1\rangle$. Tuto změnu propagujeme proti proudu do buňky 7, která má proměnnou c jako výstupní. Zde se nestane nic, protože podmínka $c \geq 0$ je splněna. Nyní dochází k distribuci změny domény proměnné c směrem 'po proudu' ke všem buňkám, které mají proměnnou c jako vstupní. V našem případě to obnáší buňku 5.

Nyní jsme v buňce 5 ve stejné situaci jako v předchozím případě, jelikož

proměnná c nabyla konkrétní hodnoty 1, takže podmínka $a = b+c$ je totožná s podmínkou $a = b + 1$ z předchozího příkladu. Aktuální domény proměnných a i b byly teď $\langle 0, \infty)$. Stejně jako v předchozím příkladě se musí zmenšit doména a tak, aby byla podmínka P3 splněna. Začne tedy nekonečný cyklus růstu dolních mezí proměnných a a b vždy o 1.

Na tomto příkladu je vidět několik úskalí cyklů v podmínkách.

- k zacyklení nemusí dojít během prvního průchodu buňkami, které leží na cyklu
- zacyklení může být vyvoláno buňkou, která neleží na cyklu samotném
- zacyklení může být vyvoláno změnou domény proměnné, která se v cyklu vůbec nevyskytuje.

Jak už bylo zmíněno, tyto problémy by se nevyskytly, kdyby se síť řešila podle teorie, tedy kdyby na buňkách docházelo k filtrování celých ohodnocení. V takovém případě by po zpracování buňky 4 zůstaly v množině možných řešení pouze taková ohodnocení, kde je hodnota proměnné a stejná jako hodnota proměnné b. Když taková ohodnocení profiltrujeme přes buňku 5, zbudou jen ta ohodnocení, kde $c = 0$, protože $a = b$ a $a = b + c$.

S cykly lze naložit

- stejně jako Indigo - vůbec se jimi nezabývat
- detekovat zacyklení a ohlásit chybu
- detekovat zacyklení, zastavit ho a zotavit se

V této implementaci je použita první možnost. Jde o to, že detekce zacyklení je dost komplikovaná - viz tři body uvedené výše v textu. Druhým důvodem je problematika zotavení se.

Původní myšlenka hledání řešení hierarchie propagací skrz síť podmínek počítá s lineárním průchodem, kdy se na každé buňce odfiltrují "ne nejlepší" řešení. Po průchodu poslední buňkou pak zbude množina ohodnocení, která představují množinu (všech[7]) řešení hierarchie.

Při použití domén místo celých ohodnocení se to mírně komplikuje. Jeden krok - zpracování buňky - může obnášet jednu vlnu propagace změn domén

[7] pro globální komparátor - věta 2

do již zpracované sítě pro každou podmínku v buňce obsaženou. Nicméně se na to pořád dá dívat jako na lineární cestu vpřed.

Naopak, při detekci cyklu a následném zotavení by muselo dojít k návratu ke stavu domén "před započetím cyklu". Pak by bylo nutné najít nejslabší z podmínek[8], která cyklus způsobila a jejíž nesplnění zabrání zacyklení. Pak by následovala opět propagace "vpřed" s tím, že budeme vědět, na kterou podmínku si dát pozor.

4.6 Metafunkční buňky

Pro efektivní fungování propagační fáze algoritmu Straton 3 je potřeba rozšířit teorii o **metafunkční** buňky. Tento typ buněk vznikne rozšířením typu funkční buňka z definice 11. Pro pochopení bude jednodušší, když bude nejdříve diskutován problematický případ sítě na následujícím obrázku a pak bude teprve popsáno, co je to metafunkční buňka.

Série obrázků 4.5 představuje postup přidání podmínky P6 do sítě vzniklé ze zbývajících pěti podmínek následující hierarchie:

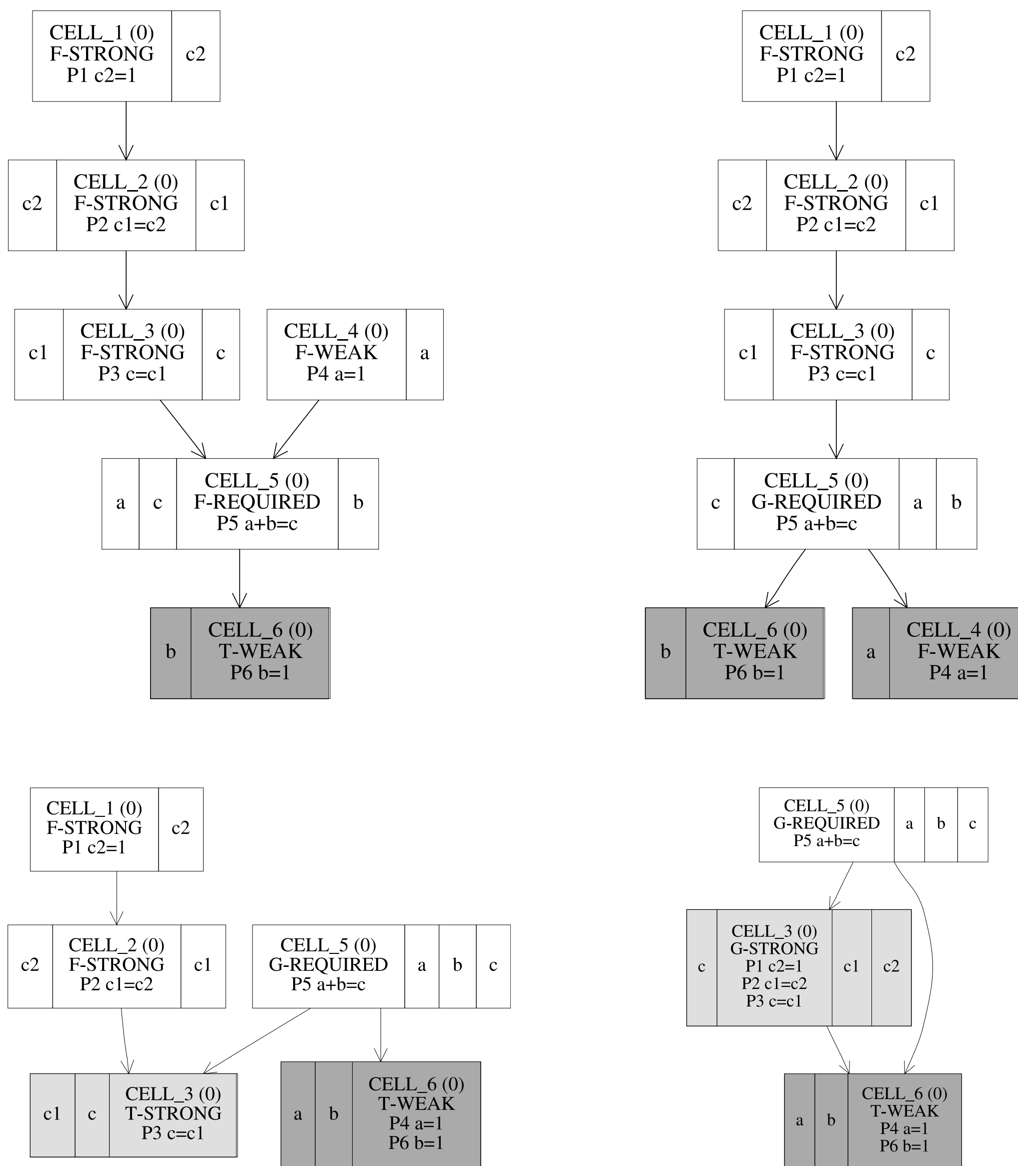
P1	$c2 = 1$	@	strong
P2	$c1 = c2$	@	strong
P3	$c = c1$	@	strong
P4	$a = 1$	@	weak
P5	$a + b = c$	@	required
P6	$b = 1$	@	weak

První obrázek (vlevo nahoře) představuje síť těsně po přidání nové buňky CELL_6, obsahující podmínku P6. Pravidla 1-5 definice 12 jsou splněna tj. jde o acyklický orientovaný graf (pravidlo 1), postavený výhradě s podmínek hierarchie(2), každá podmínka je v právě jedné buňce(3), každá proměnná je výstupem právě jedné buňky (4) a hrany vedou mezi buňkami s průnikem vstupních na jedné a výstupních proměnných na druhé straně(5).

Pravidlo 6 této definice ovšem neplatí. Proti proudu od nerozhodnutelné buňky CELL_6 existuje jiná buňka se stejnou nebo slabší vnitřní silou. Tou buňkou je CELL_4, která má stejnou vnitřní sílu.

V takovém případě je potřeba graf upravit tak, aby bylo toto pravidlo splněno. Jedinou možnou úpravu ukazuje druhý obrázek (vpravo nahoře) Zde

[8]mluvíme o podmínkách, v buňce mohou být jiné, "nezacyklující" podmínky



Obrázek 4.5: Vývoj sítě podmínek

došlo ke změně buňky z funkční na testovací a proměnná a se přesunula mezi vstupní proměnné. Jenže pro zajištění pravidla 4 je potřeba, aby proměnná a byla výstupem právě jedné buňky. Kdyby se tak stalo v CELL_4, byl by to návrat k předchozímu stavu. Proměnnou a musí tedy určovat buňka CELL_5, protože je jediná další, která tuto proměnnou váže.

V této situaci se buňka CELL_5 stává buňkou generativní. Je do důsledek její přílišné "volnosti", pro podmínku $a + b = c$ má vstupní proměnnou c a na jejím základě není *jednoznačně* možné určit hodnoty výstupních proměnných a a b. Zde je třeba poznamenat, že buňka CELL_5 si zachovala "druhou část" vlastnosti funkční buňky, tedy že *pro každé ohodnocení vstupních proměnných existuje alespoň jedno ohodnocení výstupních proměnných tak, aby byla buňka splněna.*

Nicméně podle pravidla 6 je nepřípustné, aby proti proudu od generativní buňky se silou *required* byla buňka CELL_3 se silou *strong*. Nutná úprava je na dalším obrázku (vlevo dole). CELL_3 se přesunula pod *required* buňku CELL_5 a stala se z ní testovací buňka, protože všechny proměnné jsou v tuto chvíli vstupní. Pro zjednodušení obrázku se buňky CELL_4 a CELL_6 spojily do jedné buňky, jak to vyžaduje pravidlo 7. To říká, že po proudu za nerozhodnutelnou buňkou nesmí být dvě nerozhodnutelné buňky nespojené orientovanou cestou. Kdyby se mezi tyto dvě buňky přidala hrana libovolné orientace, bylo by porušeno pravidlo 6, protože obě mají stejnou sílu a obě jsou nerozhodnutelné. Jediným řešením tedy je spojit obě buňky.

Na třetím obrázku je sice splněno pravidlo 6 pro buňku CELL_5, ale přestalo platit pro CELL_3, před kterou je stejně silná CELL_2. Musí tedy následovat připojení CELL_2 "pod" CELL_3, stejně jako to bylo v minulém kroku s CELL_5 a CELL_3. Navíc CELL_5 přesunula proměnnou c mezi své výstupní.

Následně nebude platit pravidlo 6 pro CELL_2, před kterou bude CELL_1 stejné síly. Výsledkem bude přesunutí CELL_1 pod CELL_2. Jenže tím se problém nevyřeší. CELL_1 má jen jednu výstupní proměnnou $c1$. Když se z ní stane vstupní proměnná, buňka CELL_1 se stane testovací.

Tím pádem nebude platit pravidlo 6 pro CELL_1, protože před ní je stejně silná CELL_2, což je opačná situace než v předchozím kroku. Jediným řešením situace je spojit tyto dvě buňky do jedné. Tato buňka bude generativní, protože obsahuje podmínky $c1 = c2$ a $c2 = 1$. Současně je $c1$ vstupní proměnnou, takže splnění obou podmínek nelze zaručit, pokud $c1$ nabude jiné hodnoty než 1.

Bude tedy muset následovat slévání s buňkou CELL_3, která je stejně

silná a je "proti proudu" od nově vzniklé generativní buňky CELL_1_2.

Výsledkem bude jedna velká generativní buňka (označená nyní CELL_3), obsahující všechny tři strong podmínky P1, P2, P3, která bude po proudu od buňky CELL_5.

Pro zajištění platnosti pravidla 7 pak přibude ještě hrana mezi "strong" buňkou a "weak" buňkou (označená CELL_6). Výsledek je posledním obrázku 4.5 vpravo dole.

Výsledná síť splňuje všechny pravidla definice 12. Jenže je dokonale *stratifikovaná*, je tedy "špatná" stejně, jako by ji postavil původní algoritmus Straton (popis v sekci 3.7.1).

Řešením tohoto problému, zvoleným v této práci, je zavedení nového typu buněk - **metafunkčních buněk**. S funkčními buňkami budou mít společné to, že pro libovolné ohodnocení vstupních proměnných existuje **alespoň jedno** ohodnocení výstupních proměnných tak, aby byla podmínka splněna. Díky této vlastnosti nejde o nerozhodnutelné buňky.

Nerozhodnutelné buňky mohou při svém splňování "omezovat domény vstupních proměnných". Jde o to, že ne pro každé ohodnocení vstupních proměnných je buňka splněna. Tím pádem při splňování těchto buněk je třeba odstranit "nekonzistentní" hodnoty nejen z domén výstupních proměnných, ale i z domén těch vstupních.

Funkční buňky toto nedělají, protože pro každé ohodnocení vstupních proměnných lze nalézt odpovídající ohodnocení výstupních proměnných takové, že bude podmínka splněna. Tuto vlastnost mají i metafunkční buňky, takže také neomezují domény vstupních proměnných.

Připustíme, aby metafunkční buňky vznikaly pouze z funkčních buněk, pokud vynuceným odstraněním jiné podmínky vznikne "přebytek" výstupních proměnných a tím pádem přílišná volnost podmínky.

rozšíření definice 11
metafunkční buňka je buňka, která vznikne z *funkční buňky*, pokud se jedna nebo více jejich proměnných přesune z množiny vstupních do množiny výstupních proměnných.

Jak bylo řečeno výše, metafunkční buňky nejsou nerozhodnutelné, takže se na ně pravidlo 6 definice 12 "nevztahuje". Přesněji řečeno vztahuje, ale platí vždy, protože není splněn předpoklad implikace z tohoto pravidla. Toto pravidlo má zabránit tomu, aby nesplnění nějaké silnější buňky (což se může

stát nerozhodnutelným ale nikoli funkčním a metafunkčním buňkám) bylo zapříčiněno splněním stejně silné nebo slabší buňky, která se nachází proti proudu v síti.

Jelikož splnitelnost metafunkční buňky je možné zaručit už ve fázi stavby sítě, mohou proti proudu od ní být libovolné buňky - nesplnění metafunkční buňky zapříčinit nemohou. Metafunkční buňky nejsou nerozhodnutelné, takže se na ně "nevztahuje" pravidlo 6 definice 12.

Metafunkční buňky mohou mít pro jedno ohodnocení vstupních proměnných více splňujících ohodnocení výstupních proměnných. Nerozhodnutelné buňky po proudu mohou při svém splňování některé hodnoty těchto ohodnocení odstranit z domén dotyčných proměnných. Jenže to může zapříčinit nesplnitelnost jiných nerozhodnutelných buněk po proudu od metafunkční buňky. Proto je třeba, aby se na ně vztahovalo pravidlo 7, které musíme patřičně předefinovat.

nové znění pravidla 7 definice 12

V grafu neexistuje žádné větvení po proudu v nerozhodnutelné **ani metafunkční** buňce vedoucí do jiných nerozhodnutelných buněk, které nejsou spojeny orientovanou cestou ,tj.
$Cell$ a $Cell'$ jsou nerozhodnutelné
$\wedge$ neexistuje orientovaná cesta z $Cell$ do $Cell'$ ani z $Cell'$ do $Cell$
$\Rightarrow \forall Cell'' \in CC$ tak, že $Cell''$ je proti proudu jak od $Cell$ tak od $Cell'$, pak $Cell''$ není nerozhodnutelná **ani metafunkční**, tj. je to funkční buňka

Pokud se kdekoli v textu objeví odkaz na pravilo 7 definice 12, jde samozřejmě o toto upravené znění pravidla.

Toto zavedení je velmi opatrné, ale díky tomu se nezmění nic na platnosti rozšíření věty 1 o korektnosti, věty 2 o úplnosti a jejich rozvedení na stranách 34-34.

V diskuzi o tomto rozšíření, provedené na stranách 71-73 práce [1] se mluví o přechodu od sítí podmínek k posloupnosti buněk. V diskuzi o tom , že slabší buňky mohou být před silnějšími je u funkčních buněk využita jen existence "alespoň jednoho splňujícího ohodnocení výstupních proměnných" pro každé vstupní ohodnocení. Jednoznačnost takového ohodnocení není v této části využívaná. Proto je možné, aby se s metafunkčními buňkami v tomto ohledu zacházelo stejně, protože tuto vlastnost mají také.

Naopak, v druhé části tohoto rozvedení (strana 72 v [1]) se rozebírá pravi-

dlo 7 a jeho vliv na řešení hierarchie. Zde je použitou vlastností generativních buněk to, že pro jedno ohodnocení vstupních proměnných může být buňka splněna pro více různých ohodnocení výstupních proměnných. Tuto vlastnost mají metafunkční buňky také, takže je nutné, aby pro ně platilo pravidlo 7, tedy žádné větvení do nerozhodnutelných buněk nespojených cestou.

V našem příkladě se tedy buňka CELL_5 stane metafunkční buňkou. Výsledkem pak bude síť z druhého obrázku (vpravo nahoře) jen s drobnými úpravami. Zaprvé, typ buňky CELL_5 nebude generativní ale metafunkční, takže v označení buňky se změní "G-" na "M-". Zadruhé, buňky CELL_4 a CELL_6 se spojí do jedné buňky, protože pravidlo 7 musí platit i pro metafunkční buňky.

4.7 Fáze budování sítě

Pro každou netriviální hierarchii je možné vytvořit množství různých sítí, která budou splňovat definici 12. Není možné říct, že jen jedna je správná a všechny ostatní jsou špatné. Všechny sítě vzniklé z dané hierarchie splňující definici 12 lze považovat za korektní výsledek plánovací fáze.

Různé sítě pro zadanou hierarchie však mohou být různě "dobré" z hlediska hledání výsledného řešení hierarchie - propagační fáze algoritmu. Jak bylo diskutováno v [1], více *granulované* sítě jsou "lepší".

Granularitou sítě rozumíme velikost jejich buněk, tedy počty podmínek v jednotlivých buňkách. Větší buňky znamenají méně funkčních buněk. Větší buňky znamenají větší pravděpodobnost, že alespoň jedna z podmínek nebude funkcionální a nebo že alespoň dvě z podmínek budou v konfliktu (a výsledek propagace skrz buňku bude záviset na pořadí jejich zpracování).

Nejhorší variantou je *stratifikovaná síť*. Je to síť, kde pro každou preferenci vyskytující se v hierarchii existuje právě jedna buňka. Takovouto síť generují algoritmy Straton a Straton 2.

Naopak, čím jsou sítě jemněji granulované, tím jsou výhodnější.

Přesněji řečeno - při použití globálního komparátoru jsou v propagační fázi problematické nerozhodnutelné buňky a zvláště buňky s větším počtem obsažených podmínek. Takové buňky totiž mohou vyvolávat potřebu přehodnocování už zpracované části sítě. Naopak funkční buňky (které navíc vždy obsahují pouze jednu podmínku) žádné zpětné operace nevyžadují.

Pro lokální komparátor není výhodnost jemně granulované sítě tak výrazná, protože při nesplnitelnosti podmínky není potřeba přehodnotit již

zpracovanou část sítě. Jemněji granulovaná síť má ale přesto jednu výhodu - obsahuje větší podíl funkčních buněk. Tyto buňky, již podle definice, nemění doménu vstupní proměnné. To je velmi výhodné, protože zpracování takové buňky je pak možno považovat za elementární operaci. Naopak pokud nerozhodnutelná buňka způsobí změnu domény vstupní proměnné, dojde k propagaci této změny do již zpracované části sítě, což může znamenat v nejhorším případě změnu domén všech podmínek, které se ve zpracované části sítě vyskytují.

Rozdíl mezi "přehodnocováním" už zpracované části sítě u globálních komparátorů a "propagaci změny domény" u lokálních komparátorů je zásadní. Při propagaci se změní doména každé dosud zpracované proměnné. Naopak přehodnocování už zpracované sítě v nejhorším případě znamená, že se vyzkouší všechna pořadí zpracování jednotlivých podmínek pro všechny buňky proti proudu od problematické buňky. Takové zkoušení je velice neefektivní. Navíc se dá očekávat, že k němu bude opakovaně docházet, protože v obecném případě nelze očekávat bezkonfliktní hierarchii (takovou, kde by všechny podmínky byly splněny).

4.7.1 Algoritmus

Proces vytváření sítě je zachycen v algoritmu 4.2. Vstupem pro tvorbu sítě je hierarchie podmínek. Podmínky této hierarchie je třeba rozdělit na "běžné" a *stay* podmínky.

Se *stay* podmínkami se zachází speciálně, jak bude popsáno v 4.7.2. Jsou z nich vytvořeny *stay* buňky ve funkci *prepare_network*. Tato funkce vytvoří prázdnou síť, do které umístí pro každou proměnnou jednu *stay* buňku, která ji bude určovat. Tím je od počátku zajištěno, že je každá proměnná určována právě jednou podmínkou.

Ostatní podmínky se ve funkci *sort_constraints* setřídí podle preferencí od nejsilnějších po nejslabší. Výsledkem bude fronta Q.

Z této fronty se postupně (tedy v pořadí daném setříděním) odebírají podmínky P, které se vloží do sítě, jak je popsáno v 4.7.3, čímž vznikne nová buňka C. Buňka se nejen "volně" přidá, ale také se připojí potřebnými hranami ke zbytku sítě. Při přidávání se také může stát, že bude nějaká buňka ze sítě odstraněna (vždy to bude *stay* buňka). Proto je parametrem této funkce i fronta Q, do které bude přidána podmínka z případě odstraněné buňky. Jelikož jsou *stay* buňky slabší než všechny ostatní buňky, budou vždy zařazeny na konec fronty, čímž zůstane zachováno uspořádání fronty podle

preference.

Pak je pro novou buňku C vynuceno pravidlo 6, což bude popsáno v sekci 4.7.4. Přitom může opět docházet k odstraňování *stay* buněk, jejichž podmínky se budou znovu přidávat do fronty pro zpracování Q.

Nakonec, až už budou všechny podmínky hierarchie H zařazeny v síti, se vynutí pravidlo 7, což bude popsáno v 4.7.5.

Tím je stavba sítě hotová.

Algoritmus 4.2: Create_network

```
1  create_network(hierarchy H)
2      queue Q;
3      constraint P;
4      cell C;
5      network N;
6  begin
7      N := prepare_network(H);  prázdná síť + stay podmínky
8      Q := sort_constraints(H);  od nejsilnějších po nejslabší
9
10     while(not_empty(Q))
11         P := queue_get(Q);
12         C := insert_constraint(N,P,Q);
13         enforce_6(C,Q);
14     end while;
15
16     enforce_7(N);
17
18     return N;
19 end create_network;
```

Vstupem pro stavbu sítě je seznam všech podmínek z hierarchie. Ty se setřídí. Volné proměnné z těchto podmínek jsou takové "pevné body", na základě kterých se daná síť staví. Při přidávání nové buňky se nová buňka připojuje ke zbytku sítě přes jednotlivé proměnné. Setříděné, dosud do sítě nepřidané podmínky, budou ve frontě Q.

Z této fronty budou podmínky postupně odebírány a přidávány do budované sítě. Může se stát, že v průběhu přidávání nějaké podmínky bude jiná

podmínka odstraněna ze sítě a přidána na konec fronty nezpracovaných podmínek. Takto mohou být opětovně přidávány pouze *stay* podmínky, navíc každá nejvýše jednou.

Cyklus přidávání podmínek do sítě poběží tak dlouho, dokud nebude fronta nezpracovaných podmínek prázdná. V každém kroku cyklu se nejprve odebere podmínka z čela fronty nezpracovaných podmínek. Pak je přidána do sítě jako nová buňka a připojena na proměnné, což je taková "kostra", na kterou se všechny buňky váží svými vstupními a výstupními podmínkami.

Po vyprázdnění fronty Q následuje vynucování pravidla 7. V průběhu tohoto vynucování už nemůže přestat platit žádné pravidlo, takže po vynucení pravidla 7 pro celou síť, je fáze stavby sítě hotová.

4.7.2 Stay podmínky a buňky

Před tím, než se do sítě začnou vkládat "běžné" podmínky, se pro každou proměnnou vloží *stay* buňka. Jak bylo diskutováno v 4.2.4, hierarchie musí obsahovat pro každou proměnnou právě jednu *stay* podmínku. Z těchto podmínek se vyrobí jednoduché *stay* buňky. Taková buňka obsahuje jedinou podmínku a má jedinou proměnnou. Tato proměnná je výstupní, je to ta proměnná, která je *stay* buňkou vázána.

Toto opaření má tři důvody. Prvním z nich je, aby bylo pořád splněno pravidlo 4, tedy že každá proměnná je výstupní pro právě jednu buňku.

Druhý důvod je technický. Když se přidává nová podmínka, je potřeba rozdělit její proměnné na vstupní a výstupní. Výstupními proměnnými se mohou stát jen ty, které nejsou výstupními proměnnými *silnějších* buněk. Buňky *stay* vlastně "rezervují" proměnné, které se nestaly výstupními proměnnými silnějších buněk, aby se mohly stát výstupními proměnnými pro slabší buňky. Tím je umožněno, aby síť mohla být "nemonotónní", tj. aby uspořádání nebylo dáno pouze vnitřní silou buněk. Toto "rezervování" je popsáno níže v textu.

Třetí důvod souvisí s předchozím. Nová buňka se může dostat proti proudu od silnější buňky, pouze pokud "nahradí" *stay* buňku, která byla proti proudu od té silnější. Proti proudu od nerozhodnutelných buněk nemohou být žádné *stay* buňky, což je důsledek vynucení pravidla 6 pro tu "již zpracovanou" nerozhodnutelnou buňku. Tím je zaručeno, že se před nerozhodnutelnou buňku nemůže dostat *žádná* buňka, a tedy ani slabší, která by narušila pravidlo 6.

4.7.3 Přidávání podmínky

Jak už bylo řečeno, nejdříve se do sítě vloží *stay* buňky pro každou proměnnou, a pak se v pořadí podle preference přidávají "běžné" buňky. Podmínky *stay* ze *stay* buněk, které byly v průběhu přidávání "běžných" buněk odstraněny ze sítě, se nakonec (protože mají nejslabší preferenci) přidají stejným způsobem.

Díky tomu dochází k minimu změn v síti. Při přidávání buňky může být odstraněna jedna nebo více *stay* buněk. Při vynucování pravidla 6 mohou být opět odstraněny *stay* buňky. Ale nikdy už se ze sítě neodstraňují buňky s "běžnými" podmínkami.

Buňkám obsahujícím "běžné" podmínky se tedy nemůže stát, že by byly někdy odstraněny. Jediné, co je může potkat, je spojování s jinou, stejně silnou buňkou při vynucování pravidla 6 či 7. To je také hlavní důvod, proč je Straton 3 neinkrementální. Díky tomuto přístupu nepotřebuje umět odstraňovat ze sítě "běžné" podmínky. Tím se vyhýbá riziku, že po přidání nějaké hodně silné podmínky bude nutné odstranit část sítě nebo dokonce celou síť a stavět ji "znovu" kolem této silné podmínky.

Při přidávání podmínky do sítě mohou nastat následující možnosti:

- *Všechny proměnné nové podmínky jsou už vázané v buňkách stejné či vyšší síly, tj. žádnou z těchto proměnných neváže* stay *buňka.* Pak se ze všech proměnných podmínky stanou vstupní proměnné nové buňky. Díky tomu bude tato nová buňka testovací. Pro každou proměnnou se přidá hrana z buňky, která ji určuje, do nové buňky.

- *Přidávaná podmínka je funkcionální a existuje alespoň jedna její proměnná, která je dosud vázána* stay *podmínkou.* V tom případě se vybere libovolná taková proměnná. Ta se stane výstupní a *stay* buňka, která ji dosud vázala bude ze sítě odstraněna a její *stay* podmínka zařazena na konec fronty podmínek ke znovuzpracování. Pokud z této *stay* buňky vedly hrany do dalších buněk, budou tyto hrany přesměrovány tak, aby vedly z nové buňky.

  Ostatní proměnné přidávané podmínky se stanou vstupními a přidají se hrany, které je spojí s buňkami, jenž tyto proměnné určují. Nová buňka bude funkční.

- *Pokud je přidávaná podmínka nefunkcionální a je-li alespoň jedna její proměnná vázaná* stay *buňkou*, pak se postupuje obdobně jako v před-

chozím bodě. Rozdíl je v tom, že *všechny* proměnné nové podmínky, které jsou vázány *stay* buňkami se stanou výstupy nové buňky. Se všemi *stay* buňkami a hranami z nich vedoucími bude provedeno totéž v předchozím bodě s jednou *stay* buňkou. Výsledkem bude generativní buňka.

Pseudokód procedury pro přidání podmínky do sítě je v algoritmech 4.3 a 4.4. Vstupem je dosud postavená síť N, přidávaná podmínka P a fronta dosud nepřidaných podmínek Q, do které se umísťují *stay* podmínky z odstraňovaných *stay* buněk. Je důležité si uvědomit, že síť N ještě neodpovídá pravidlu 7, takže všechny hrany spojují buňky na základě pravidla 5. To říká, že do buňky C vedou hrany ze všech buněk, které mají jako svou výstupní proměnnou některou ze vstupních proměnných buňky C. Proto jsou přesměrovávány všechny hrany vedoucí z odstraňovaných *stay* buněk.

Algoritmus 4.3: Přidání podmínky do sítě,1. část

```
1  insert_constraint(network N, constraint P, queue Q)
2    cell C,D,E; list L_all, L_stay; variable V, V_out;
3  begin
4    C := empty_cell();
5    list_add (C.constraint_list,P);
6    L_all := P.variables;
7    L_stay := empty_list();
8    for (V in L_all)
9    begin
10     if (V.parent is stay cell) then
11       list_add(L_stay,V)
12     end if
13   end for;
14   if(L_stay is empty) then
15     for (V in L_all) begin
16       list_add(C.input_vars, V);
17       add edge[V.parent,C] into N;
18     end;
19     C.cell_type = TEST;
20   else ...
```

Algoritmus 4.4: Přidání podmínky do sítě,2. část

```
20    else L_stay not empty
21      if (P.cn_type = FUNCT) then
22        V_out := remove_first(L_stay);
23        list_add(C.output_vars,V_out);
24        D := V_out.parent;
25        V_out.parent = C;
26        for (edge[D,E] in N) replace [D,E] with [C,E];
27        add D.constraint_list into Q;
28        remove D from N;
29        for (V in L_all except V_out)
30        begin
31          list_add(C.input_vars, V);
32          add edge[V.parent,C] into N;
33        end;
34        C.cell_type = FUNCT;
35      else L_stay není prázdné a podmínka není funkcionální
36        for (V_out in L_stay)
37        begin
38          list_add(C.output_vars,V_out);
39          D := V_out.parent;
40          V_out.parent = C;
41          for (edge[D,E] in N) replace [D,E] with [C,E];
42          add D.constraint_list into Q;
43          remove D from N;
44        end for;
45        for (V in L_all except L_stay members)
46        begin
47          list_add(C.input_vars, V);
48          add edge[V.parent,C] into N;
49        end;
50        C.cell_type = GENERATIVE;
51      end if; podmínka není funkcionální
52    end if; L_stay není prázdné
53    return C;
54  end insert_constraint
```

4.7.4 enforce_6

Úkolem této procedury, jak už název napovídá, je vynutit pravidlo 6. Jak bylo diskutováno, pravidlo 6 se nevynucuje pro funkční buňky a podle sekce 4.6 ani pro metafunkční buňky.

Pro připomenutí text pravidla 6:
Pro každou nerozhodnutelnou buňku Cell *neexistuje buňka se stejnou nebo slabší vnitřní silou, která by se nacházela* proti proudu *od* Cell*, tj.* $\forall Cell \in CC \quad Cell$ *je nerozhodnutelná* $\Rightarrow \forall Cell' \in CC$ *tak že* $Cell'$ *v grafu existuje orientovaná cesta z* $Cell'$ *do* $Cell$*,* $Cell'$ *má silnější vnitřní sílu než* $Cell$

Algoritmus 4.5: Vynucování pravidla 6

```
1  enforce_6(Cell C, Queue Q,Network N)
2    Queue parents;
3    Cell D;
4  begin
5    if(C.cell_type is FUNCT or METAFUNCT)
6    then
7      return;
8    else nerozhodnutelná buňka
9      parents = sort_not_stronger_parents(C);
10     while(queue_not_empty(parents))
11     begin
12       D = queue_get(parents);
13       if (D is stay cell)
14       then
15         cell_stay_remove(D,N,Q,C);
16       else
17         cell_merge(C,D,N);
18       end if;
19     end while;
20   end if
21 end enforce_6
```

Pokud je v síti nerozhodnutelná buňka C, nelze zaručit, že bude splněna. Kdyby se proti proudu od této buňky vyskytovala slabší buňka, nelze vyloučit, že by právě splňování této slabší buňky mohlo způsobit *nesplnitelnost*

buňky C.

Stejný problém se týká i stejně silných buněk, protože při použití globálního komparátoru by nesplnění jedné buňky za cenu splnění jiné buňky stejné síly nemuselo být v pořádku (například kdyby měly podmínky v těchto buňkách různé váhy).

Pseudokód vynucování pravidla 6 je popsán v algoritmu 4.5. Funkce *sort_not_stronger_parents* najde a setřídí všechny předky dané buňky, kteří jsou stejně silní nebo slabší. Toto třídění bude "topologické proti proudu", tedy takové, jaké by vzniklo při otočení směru všech hran a následném topologickém setřídění.

Tyto předky procházíme ve *while* cyklu, dokud je všechny nezpracujeme. Buňky, které jsou stejně silné jako buňka, pro kterou se pravidlo vynucuje, se spojují pomocí procedury merge. Ta je popsána níže v textu a ilustrována na obrázku 4.6.

Jedinými buňkami, které jsou ostře slabší, mohou být *stay* buňky. Ty ze sítě odstraníme a přidáme na konec fronty *unsatisfied*. Při odstraňování *stay* buňky D je potřeba najít jinou buňku, která by nově určovala proměnnou, kterou dosud určovala buňka D. Označme tuto proměnnou jako V. Nově přidaná buňka C nemůže být "přímým" potomkem D (tj. C nemá V mezi svými vstupními proměnnými), protože taková proměnná by se při *insert_constraint* dostala mezi výstupní proměnné C a buňka D by už byla odstraněna z grafu, protože C se nepovedlo přidat jako funkční. To může nastat, pokud je podmínka nefunkcionální (pak odstraní všechny *stay* buňky na "svých" proměnných) nebo jsou už všechny proměnné vázané "běžnými" buňkami.

Buňka D nemá žádné nerozhodnutelné potomky, kromě nově přidávané buňky, protože je slabší než všechny ostatní "běžné" buňky. Pro buňky, které už v síti jsou, už bylo pravidlo 6 vynucováno, když byly přidávány. To znamená, že všechny buňky, do kterých z D vede hrana a tedy mají proměnnou V mezi svými vstupními proměnnými, jsou funkční nebo metafunkční buňky. Proti proudu od takových buněk mohou být další *stay* buňky a nebo buňky stejně silné, jako nově přidaná buňka C, pro kterou je právě vynucováno pravidlo 6. Aby se tyto slabší nebo stejně slabé buňky nemusely zbytečně dostat do fronty pro vynucování pravidla 6 (tj. spojení s C nebo odstranění z grafu), bude nejlepší, když proměnnou V bude určovat nějaká buňka E, která je už teď proti proudu od C, protože všechny buňky proti proudu od E jsou i proti proudu od C.

Buňka E se stane nutně metafunkční, protože se jí zvětší "volnost" - jedna

proměnná se přesune ze vstupních proměnných do výstupních.

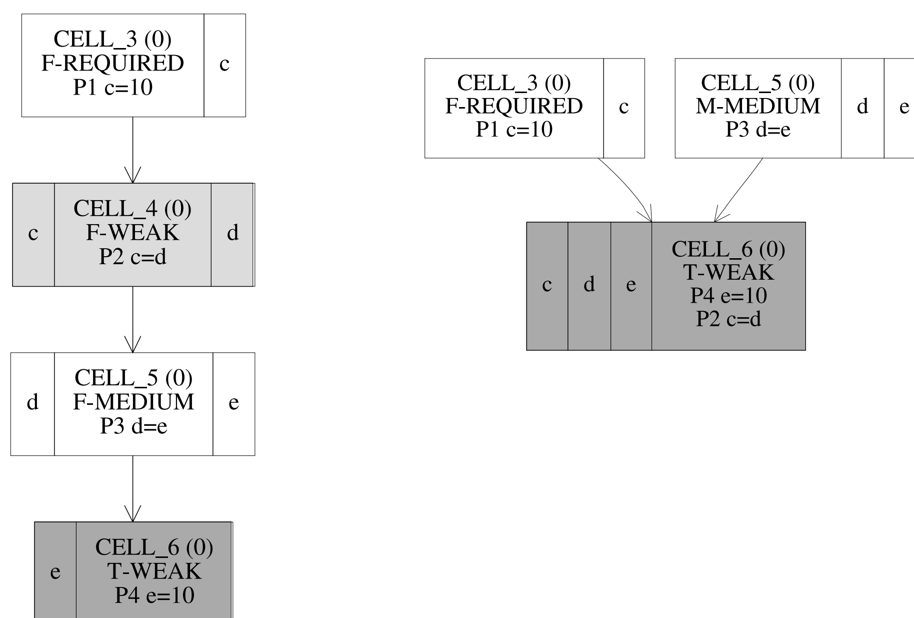
Celý tento postup je popsán algoritmem 4.6. Je dobré si uvědomit, že v každé *stay* buňce je jen jedna podmínka a jedna proměnná. Během přidávání "běžných" podmínek se *stay* buňky jen odstraňují z grafu. Když přijde řada na *stay* podmínky ve frontě k zařazení do grafu, nebude už docházet k odstraňování jiných *stay* buněk. Stejně silné podmínky se při vynucování pravidla 6 slévají. Jenže *stay* buňky se přidávají až poté, co byly přidány všechny "běžné" buňky.

Algoritmus 4.6: Odstraňování *stay* buňky

```
1  cell_stay_remove(cell D,network N, queue Q, cell C);
2    cell E,F;
3  begin
4    find edge[D,E] in N where exists direct_path[E,C] in N;
5    V:=first(D.output_vars); má jedinou proměnnou
6    V.parent = E;
7    list_remove(E.input_vars,V);
8    list_add(E.output_vars,V);
9    E.cell_type = METAFUNCT;
10   for(edge[D,F] in N) replace [D,F] with [E,F];
11   queue_add(Q, D.constraint_list);
12   remove D from N;
13 end cell_remove;
```

Úkolem procedury *merge* je spojit dvě stejně silné buňky a vyřešit rozdělení jejich společných proměnných mezi vstupní a výstupní. Přitom dojde také k doplnění hran do sítě podle nově rozdělených vstupních a výstupních proměnných.

Jak takové spojování buňky C s buňkou D, která je *proti proudu* od C, vypadá ? Výsledek spojování bude obsahovat všechny podmínky z obou původních buněk. Musí vyřešit rozdělení proměnných mezi vstupní a výstupní. Všechny vstupní proměnné C i D budou vstupními proměnnými výsledné buňky. Všechny výstupy C budou výstupy výsledku. Výstupní proměnné D se musí zkontrolovat, jestli by nevznikl cyklus, pokud by se staly výstupem výsledné buňky. Na obrázku 4.6 vlevo se CELL_6 slévá s buňkou CELL_4. Kdyby byla proměnná d výstupem výsledné buňky, vznikl by cyklus $d \rightarrow e \rightarrow d$ s buňkou CELL_5.



Obrázek 4.6: enforce_6 - merge

Stejně jako pro odstraňované *stay* buňky je potřeba najít nové buňky, které budou nově určovat proměnné, které byly výstupem D, ale nemohly se stát výstupem výsledné buňky, aby nedošlo ke vzniku orientovaného cyklu. Pokud pro výstupní proměnnou D nehrozí vznik cyklu, stane se výstupní proměnnou výsledné buňky. Pokud by už byla zařazena mezi vstupní proměnné, protože byla vstupní buňkou C, je odtud odstraněna - nemůže být současně vstupní i výstupní buňkou.

Následně je třeba doplnit hrany tak, aby bylo splněno pravidlo 5. To znamená, že pro všechny vstupní proměnné výsledné buňky se přidají hrany z buněk, které tyto proměnné určují. Naopak, pro výstupní buňky se přidají hrany do buněk, které mají tyto proměnné jako své vstupní.

Je možné, že v průběhu přebírání "určování proměnné" se z nějaké funkční buňky stane metafunkční buňka. Případně se metafunkční buňce přesune další proměnná ze vstupů mezi výstupy. Na druhou stranu se zde nemůže "transformovat" generativní buňka na metafunkční, protože žádná buňka F z řádků 14-15 není generativní. Buňka D je slabší než F a kdyby F byla metafunkční, D tam být nemůže. Kdyby byla buňka F stejně silná jako D, byla by i stejně silná jako C. Jenže buňka F leží na cestě mezi C a D, takže

Algoritmus 4.7: Spojování buněk - merge

```
1  merge(Cell C, Cell D, network N)
2    cell E,F,G;
3    variable V;
4  begin
5    E:=empty_cell();
6    list_add (E.constraint_list ,C.constraint_list);
7    list_add (E.constraint_list ,D.constraint_list);
8    list_merge(E.input_vars ,C.input_vars);
9    list_merge(E.input_vars ,D.input_vars);
10
11   list_merge(E.output_vars ,C.output_vars);
12   for (V in D.output)
13   begin
14     if ( exists edge[D,F] in N and V in F.input_vars
15          and F != C and exists direct_path[F,C] in N)
16     then zabráníme vzniku cyklu
17       list_add(E.input_vars ,V);
18       list_remove(F.input_vars ,V);
19       list_add(F.output_vars ,V);
20       V.parent := F;
21       F.cell_type := METAFUNCT;
22       for (edge[D,G] in N
23            and V in G.input_vars) replace [D,G] with [F,G];
24     else
25       list_add(E.output_vars ,V);
26       V.parent := E;
27       for (edge[D,G] in N
28            and V in G.input_vars) replace [D,G] with [E,G];
29     end if;
30   end for;
31   list_minus(E.input_vars , E.output_vars);
32   for (V in E.input_vars) add edge[V.parent ,E] into N;
33   for (edge[C,G] in N) replace [C,G] with [E,G];
34 end merge;
```

by díky "inverznímu topologickému třídění" (viz algoritmus 4.5, řádek 9) byla zpracována před buňkou D a už by byla spojená s buňkou C. Slabší být nemůže, protože *enforce_6* se volá hned po přidání buňky. V té době jsou v grafu pouze *stay* buňky (které nemají vstupní proměnné) a buňky silnější nebo stejně silné jako právě přidávaná buňka C. Při použití *merge* v *enforce_7* se pak slučují "sousední buňky", mezi kterýma není "žádný prostředník" F a nemůže vzniknout cyklus, takže podmínka z řádků 14-15 není nikdy splněna.

K tomu například dojde, pokud nastane situace z obrázku 4.6 vlevo. Na obrázku je tmavě šedě označená nově přidaná buňka, pro kterou byla procedura *enforce_6* volána a světle šedá je buňka, se kterou se bude spojovat procedurou *merge*. Výsledkem bude buňka obsahující podmínky P2 i P4, se vstupními proměnnými c, d, e. Buňka CELL_5 se stane metafunkční a povede z ní hrana do buňky, která bude výsledkem spojení CELL_4 a CELL_6. Stejně tak do ní povede hrana z CELL_3. Výsledek takového spojování je na obrázku 4.6 vpravo.

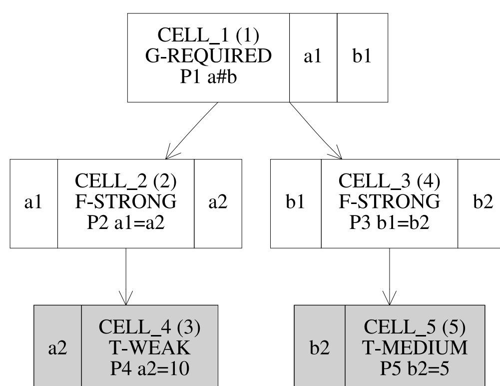
4.7.5 enforce_7

Na již postavené síti je potřeba zajistit, aby platilo pravidlo 7. Pro připomenutí:
V grafu neexistuje žádné větvení po proudu v nerozhodnutelné ani metafunkční buňce vedoucí do jiných nerozhodnutelných buněk, které nejsou spojeny orientovanou cestou ,tj.
Cell a Cell$'$ jsou nerozhodnutelné
$\wedge$ *neexistuje orientovaná cesta z Cell do Cell$'$ ani z Cell$'$ do Cell*
$\Rightarrow \forall Cell'' \in CC$ *tak, že Cell$''$ je proti proudu jak od Cell tak od Cell$'$, pak Cell$''$ není nerozhodnutelná ani metafunkční, tj. je to funkční buňka.*

To znamená, že po proudu za nerozhodnutelnou či metafunkční buňkou nesmí být dvě různé nerozhodnutelné buňky, mezi kterými by nebyla orientovaná cesta ani jedním směrem.

Příklad situace, která je v rozporu s pravidlem 7 a způsobila by nalezení ohodnocení, které není řešení, je na obrázku 4.7. V buňce CELL_1 je podmínka $a1 > b1$. Čísla v závorkách za jménem buňky ukazují pořadí zpracovávání buněk podle nějakého topologického setřídění. Topologických setřídění je pro danou síť více, pořadí na obrázku reprezentuje jedno z těch "problematických".

Po zpracování buněk CELL_1 a CELL_2 mají všechny proměnné ($a1$,$a2$,$b1$) neomezené domény. Při zpracování CELL_4 se změní doména $a2$ na 10, tato



Obrázek 4.7: enforce_7 - pravidlo neplatí

změna se propaguje i do $a1$. Podmínko v CELL_1 tuto změnu transformuje pro $b1$ jako omezení domény na $\langle 10, \infty)$.

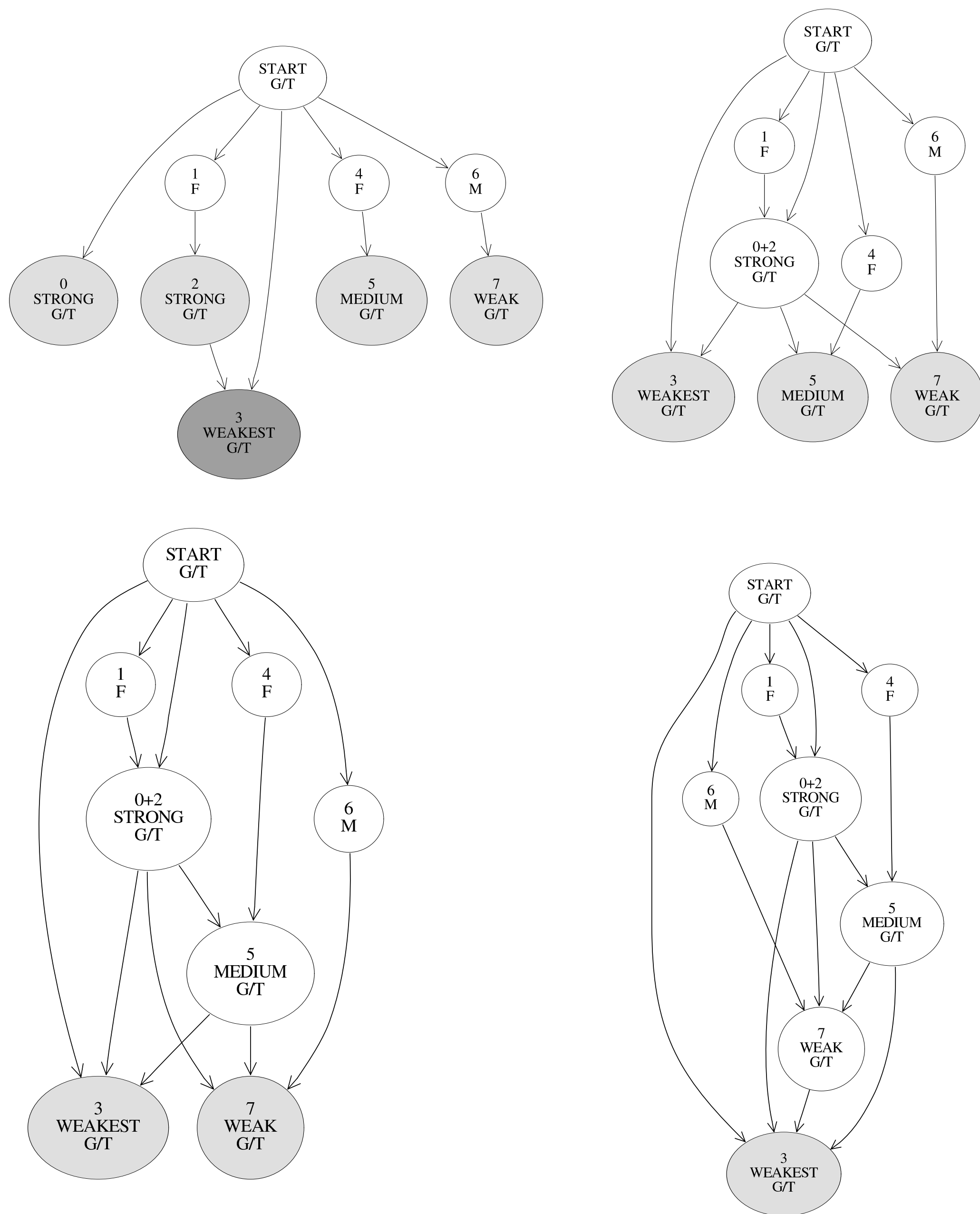
Pak se zpracuje CELL_3 a nastaví doménu $b2$ na $\langle 10, \infty)$. Jenže poté se už nepodaří splnit podmínku P5 $b = 5$. Toto nesplnění je způsobené slabší podmínkou P4.

Když bude pravidlo 7 vynuceno, přibude do sítě hrana z CELL_5 do CELL_4. V důsledku toho žádné topologické třídění nezařadí CELL_4 před CELL_5.

Stojí za povšimnutí, že se v pravidle 7 vůbec nehovoří o "zákazu" větvení ve funkčních buňkách. To znamená, že pravidlo 7 platí pro všechny takové buňky. Navíc, když je kontrolována nějaká nerozhodnutelná buňka C, tak se hledají její **nerozhodnutelní** potomci. Tím jsou myšleni přímí i *nepřímí* potomci. Nepřímými potomky jsou takové nerozhodnutelné buňky, do kterých vede orientovaná cesta z C, i když s C nemají společnou hranu. To znamená, že funkční buňky jsou "ignorovány" při hledání nerozhodnutelných potomků a důležité jsou jen "cesty", které skrze tyto buňky vedou z kontrolované buňky po proudu - jak bylo na předchozím případě vidět na buňkách CELL_2 a CELL_3.

Speciální zacházení vyžadují metafunkční buňky. Pravidlo 7 pro ně musí být také vynucováno, jak bylo diskutováno na straně 60. Na druhou stranu, při vynucování tohoto pravidla je potřeba kontrolovat jen *nerozhodnutelné* buňky. Metafunkčními buňkami se při tomto hledání jen prochází, stejně jako těmi funkčními.

Při vynucování pravidla 7 se nedělá veškerá práce najednou. Když je



Obrázek 4.8: enforce_7 - hledání potomků

Algoritmus 4.8: Vynucování pravidla 7

```
1  enforce_7(Network * N)
2    Cell C, D, E;
3    Queue *childs;
4  begin
5    for (C in N) bráno v topologickém uspořádání
6    begin
7      if(C.cell_type!=FUNCT) pro metafunkční musí být vynucováno
8      then
9        childs=find_GT_childs(C);
10       if(queue_members(childs) > 1)
11       then
12         D=queue_get_strongest(childs);
13         while(not_empty(childs))
14           E=queue_get(childs);
15           if (D->strength == E->strength)
16           then
17             merge(D,E,N);
18           else
19             add edge[D,E] into N;
20           end if;
21         end while;
22       end if;
23     end if;
24   end for;
25 end enforce_7
```

vynucováno pravidlo 7 pro nerozhodnutelnou buňku C, vynutí se pouze pro *první GT vrstvu* nerozhodnutelných buněk po proudu od C a "zbytek" práce se nechá na další buňky. *První GT vrstva* nerozhodnutelných buněk jsou ty nerozhodnutelné buňky, do kterých vede cesta z C a současně do nich *nevede* cesta z jiné nerozhodnutelné buňky, která by byla také po proudu od C.

Pseudokód vynucování pravidla 7 je v algoritmu 4.8 a jeho postupná aplikace je na obrázku 4.8.

Buňky označené G/T znamenají nerozhodnutelné buňky, F jsou funkční a M metafunkční buňky. U nerozhodnutelných buněk je uvedena vnitřní síla.

Na obrázku nahoře vlevo je "první GT vrstva" nerozhodnutelných buněk světle šedá. Buňka 3 je tmavě šedá, protože do ní vede cesta z nerozhodnutelné buňky 2.

Pokud obsahuje první GT vrstva pod C jen jednu buňku, je vynucování C ukončeno a pokračuje se vynucováním další buňky.

Pokud obsahuje první GT vrstva více buněk, je třeba graf upravit, aby tomu tak nebylo.

Z první GT vrstvy vybereme buňku D s nejsilnější preferencí. Pokud jsou ve vrstvě ještě jiné buňky se stejnou preferencí jako D, všechny sloučíme dohromady pomocí funkce *merge* popsané v 4.7.

Zbývají-li ve vrstvě další buňky, jsou přidány hrany vedoucí z D (případně z D sloučené s dalšími buňkami) do všech slabších buněk (E - řádek 19). Kdyby už hrana $[D, E]$ existovala, nemohla by buňka E být v první GT vrstvě a tudíž by nebyla cyklem na řádcích 13 až 21 zpracovaná.

Všechny tyto operace se týkají nerozhodnutelných buněk. Pokud slučujeme dvě stejně silné buňky D a E, pro buňky proti proudu od obou z nich platí pravidlo 6, takže se ani v jedné z "protiproudových" oblastí nevyskytují slabší nebo stejně silné buňky jako D.

Pokud je přidána hrana ze silnější nerozhodnutelné buňky do slabší, pravidlo 6 nemůže přestat platit. Proti proudu od té silnější buňky se nemůže vyskytnout buňka slabší.

Na obrázku 4.8 je vidět stav sítě po vynucení pravidla pro buňku START. Buňky 0 a 2 byly v první GT vrstvě nejsilnější, takže se spojily dohromady a přibyly hrany do buněk 5 a 7. Do buňky 3 už hrana vedla.

Nyní by mělo být vynucováno pravidlo 7 pro buňku 0+2, proto je už označená bíle. Buňka 3 změnila barvu na světle šedou, protože je už v první GT vrstvě vůči buňce 0+2. Výsledek po vynucení je vidět na dalším obrázku - 4.8 vlevo dole. Nejsilnější nyní byla buňka 5, takže přibyly hrany z 5 do 3 a do 7.

Pokračuje se ve vynucování pravidla 7 pro buňku 5, proto je bílá. Nejsilnější buňka z její první GT vrstvy je buňka 7, takže je přidána hrana ze 7 do 3. Výsledek je vidět na posledním obrázku 4.8 vpravo dole. Následuje vynucování pro buňku 7. Ta má v první GT vrstvě pouze jednu buňku, takže pravidlo je "vynuceno" bez změny grafu a pokračuje se dále. Buňka 3 už nemá v první GT vrstvě vůbec žádné potomky a vynucování končí, protože už v grafu ani nejsou další buňky, pro které by bylo možné pravidlo vynucovat.

Tímto postupem je "po vrstvách" vynucena platnost pravidla 7 pro celý graf. Po jednom kroku pro buňku C ještě neplatí - ale všechny nerozhodnu-

telné buňky po proudu od C mají nového společného "předka" D. Buňka D je ostře silnější než ostatní nerozhodnutelné buňky, které jsou po proudu od C (ty stejně silné se právě sloučily, aby vytvořily D). Tímto způsobem se vynucování postupně dostane až na nejslabší úroveň.

Podstatným pozorováním je, že všechny nerozhodnutelné a metafunkční buňky D po proudu od C budou postupně zpracovány, tj. bude pro ně vynucováno "pro první GT úroveň". Žádná taková buňka nemůže "uniknout", tj. nevstoupit někdy do "první GT vrstvy" pro nějakou buňku E. Pokud v tu chvíli není v první GT vrstvě silnější buňka, spojí se D se stejně silnými buňkami ve vrstvě a jsou přidány hrany do ostatních buněk v tu chvíli v první GT vrstvě.

Pokud je naopak v tu chvíli ve vrstvě silnější buňka F, bude do grafu přidána hrana $[F, D]$. Díky topologickému pořadí zpracování buněk v síti je tím zaručeno, že pro buňku F bude pravidlo 7 vynucováno dříve než pro jinou buňku z první GT vrstvy. Při vynucování pravidla 7 pro F bude D opět v první GT vrstvě. To se bude opakovat, dokud nebude D nejsilnější v nějaké první GT vrstvě. Pak bude D sloučena se stejně silnými buňkami z vrstvy a budou přidány hrany do těch slabších.

Ještě je to komplikováno o metafunkční buňky, kterými algoritmus "volně prochází" při hledání "první GT vrstvy". Ty mohou v pořadí vynucování pravidla 7 předběhnout "nejsilnější" buňku právě zpracované GT vrstvy. To ale práci jen urychlí - metafunkční buňka bude zpracována jen jednou a nebude muset "na svou chvíli" čekat v postupně se měnící "první GT vrstvě". Tím, že je vynuceno pravidlo pro metafunkční buňku (která nikdy nebyla v žádné první GT vrstvě) je jen "zlepšen" stav vynucování pravidla 7 v síti. Může se stát, že dojde k přidání hrany mezi dvě nerozhodnutelné buňky, které byly v předchozí první GT vrstvě. Buňka A, *do* které hrana vede, bude odsunuta *pod* buňku B, *ze* které hrana vychází. Tím pádem A nemůže být v první GT vrstvě pro žádné buňky proti proudu od B. Až když bude pravidlo 7 vynucováno pro buňku B, bude buňka A v první GT vrstvě.

Tím je ukázáno, že každá nerozhodnutelná či metafunkční buňka D po proudu od C musí někdy vstoupit do "první GT vrstvy" pro nějakou buňku E po proudu od C a jednou se stane nejsilnější buňkou vrstvy (případně bude spojena se stejně silnými "nejsilnějšími" buňkami).

Jsou-li po proudu od nerozhodnutelné nebo metafunkční buňky C dvě nerozhodnutelné buňky F a G, mezi kterými nevede orientovaná cesta, mohou nastat tyto možnosti:

- obě současně budou v nějaké první GT vrstvě těmi nejsilnějšími. Pak budou spojeny do jedné buňky a pravidlo "nevětvení" pro ně triviálně platí.
- buňka F bude "nejsilnější buňkou první GT vrstvy" dříve než G. V tom případě přibude hrana z F (případně spojené s dalšími buňkami) do všech slabších buněk GT vrstvy. Tím je (díky definici GT vrstvy) zaručeno, že vede orientovaná cesta z F do G
- nebo nastane opačná situace a vznikne cesta z G do F

Tím je vidět, že aplikování vynucování pravidla 7 po vrstvách je korektní, tj. po zpracování celé sítě pravidlo platí.

4.7.6 Složitost budování sítě

Síť vzniká postupným přidáváním celkem N podmínek. Podmínky *stay* jsou do sítě přidány jako první a jako jediné mohou být ze sítě odstraněny a znovu zařazeny ke zpracování (každá nejvýše jednou). Taková podmínka je jedna pro každou volnou proměnnou, kterých je M. Celkově se krok přidávání do sítě provede nejvýše $N + M$ krát. Jak složité je takové přidání jedné nové podmínky?

Nejprve se přidá podmínka jako nová buňka. Při tom se zkontrolují všechny proměnné, které podmínka váže, a hledá se proměnná dosud určovaná nejslabší podmínkou. Je rozumné předpokládat, že jedna podmínka bude vázat pouze malý počet proměnných, řekněme p. Kromě velmi malých problémů bude $p \ll M$, kde M je celkový počet volných proměnných v celé hierarchii.

Po přidání nové buňky následuje vynucení pravidla 6. To obnáší projít všechny buňky proti proudu od nově přidané buňky. Silnější buňky nevyžadují žádné zpracování. Slabší buňky (které mohou obsahovat pouze *stay* podmínky) jsou z grafu vyřazeny. Stejně silné buňky jsou spojovány s novou buňkou. To znamená, že projít je třeba nejvýše tolik buněk, kolik jich už v síti je - pokud je nová buňka po proudu od všech ostatních buněk. Takových buněk bude nejvýše tolik, kolik bude v síti podmínek (nejsou tam žádné buňky bez podmínek, tj buňky typu "volná proměnná") To tedy znamená, že procházet se bude nejvýše N buněk.

V rámci spojování stejně silných buněk a odstraňování slabších buněk bude docházet k přesměrovávání hran v síti. Sice nevíme přesně kolik těch

hran bude, ale víme, že se budou měnit hrany v závislosti na vazbě na proměnnou podle pravidla 5.

Proměnných je M. Při šikovné implementaci je možné přesměrování hran z jedné buňky určující danou proměnnou na jinou buňku, která bude nově určovat tuto proměnnou, považovat za elementární operaci.

Při vynucování pravidla 6 může dojít k tomu, že nějaká buňka se *silnější* preferencí změní svůj typ z funkční na metafunkční buňku. To ovšem nevyžaduje znovuvynucování pravidla 6, protože to je pro metafunkční automaticky splněno stejně jako pro funkční buňky.

Celkem bude tedy složitost přidání jedné podmínky $O(p + N + M)$.

Následuje ještě krok vynucování pravidla 7. Při něm pro každou buňku hledáme přímé nerozhodnutelné potomky. Při tom v nejhorším případě budeme muset prohledat celou síť od dané buňky "po proudu". Můžeme si ale pomoci vhodnými datovými strukturami. V textu [1] jsou navrhovány například indexy "výskytu nerozhodnutelné buňky dále po proudu". Má-li síť C buněk, bude celková složitost $O(C^2)$. Buněk bude nejvýše tolik jako podmínek ($C \leq N$), protože neexistují bunky bez podmínek, takže vynucování pravidla 7 bude mít nejhorší složitost $O(N^2)$

Celá fáze budování sítě tedy znamená $N + M$ přidání buňky do sítě a následné vynucení pravidla 6. Potom následuje jediný běh vynucování pravidla 7. To je dohromady $(N+M)\times(p+N+M)+N\times N$. Pro běžné hierarchie bude platit, že p bude malá konstanta (typicky $\ll 10$), kterou vzhledem k ostatním hodnotám můžeme zanedbat. Výsledná složitost je tedy $O((N + M)^2)$.

4.7.7 Korektnost budování sítě

Fáze budování sítě staví z podmínek zadané hierarchie síť, která má splňovat definici 12, respektive její úpravu po zavedení metafunkčních buněk, jak je popsána na konci sekce 4.6.

Pravidlo 1 říká, že síť je orientovaný acyklický graf. Pravidlo 2 říká, že množina buněk je konečná, a že jsou v nich pouze podmínky z hierarchie H. Obě pravidla jsou zjevně splněna po celou dobu stavby sítě. Zde je dobré poznamenat, že *stay* podmínky *jsou nutnou součástí* zadané hierarchie, algoritmus si je "nevymyslí".

Pravidlo 3 říká, že každá podmínka z hierarchie H je právě v jedné buňce. Toto pravidlo bude platit až po té, co budou skutečně všechny podmínky hierarchie vloženy do sítě, tj. až po té, co budou do sítě zpětně vloženy i *stay* podmínky, které byly v průběhu stavby odstraněny.

Pravidlo 4 říká, že každá proměnná je výstupní pro právě jednu buňku. Na začátku jsou to *stay* buňky. V průběhu stavby sítě se proměnná může stát výstupem buňky s 'běžnou" podmínkou. To se změní jen v případě, že buňka, která ji v tu chvíli určuje, se sloučí s jinou buňkou v proceduře *cell_merge* popsané v algoritmu 4.7, kde je zajištěna korektní změna určující buňky.

Pravidlo 5 říká, že do buňky musí vést hrany z buněk, které určují vstupní proměnné této buňky. To je zajištěno přidáváním a přesměrováváním hran, což se děje prakticky ve všech procedurách.

Pravidlo 6 říká, že proti proudu od nerozhodnutelné buňky C nesmí být jiná buňka, jejíž vnitřní síla je stejně silná nebo slabší než vnitřní síla C. Jelikož není možné předem rozhodnout, bude-li nerozhodnutelná buňka splnitelná či nikoli, musí být před takovou buňkou zpracovány pouze ostře silnější buňky. Tím bude zajištěno, že v případě nesplnění takové buňky je toto nesplnění zapříčiněno silnějšími buňkami, což je korektní a v souladu s libovolným komparátorem.

V algoritmu je toto pravidlo vynucováno pro každou buňkou jen jednou, bezprostředně poté, co je buňka přidána do sítě - řádek 13 algoritmu 4.2.

Buňka C s vnitřní silou $C.strength$ nechť je nerozhodnutelná - jinak by pro ní pravidlo 6 nemohlo přestat platit (není splněn předpoklad implikace). Vynucení pravidla 6 procedurou *enforce_6* je popsáno v algoritmu 4.5. Všechny buňky proti proudu od C jsou tedy silnější než C, což mimo jiné znamená, že tam nejsou žádné *stay* buňky.

Postup stavby sítě z algoritmu 4.2 ukazuje, že po vynucení pravidla 6 pro konkrétní buňku C na řádku 13 bude následovat

- (opakované) přidávání podmínky, tedy vznik další buňky D - řádek 12
- (opakované) vynucování pravidla 6 pro tuto buňku D - řádek 13
- (jednorázové) vynucení pravidla 7 - řádek 16

Nemohou tyto operace způsobit neplatnost pravidla 6 pro buňku C? Možné problémy při třech výše uvedených situacích budou nyní detailně rozebrány.

Přidávání další podmínky

Při přidávání nové podmínky do sítě procedurou *insert_constraint* popsanou v algoritmu 4.3 se vždy vytváří nová buňka - nechť je označená D a má

vnitřní sílu $D.strength$. Tato buňka musí všechny proměnné přidávané podmínky rozdělit mezi své vstupní a výstupní proměnné. Jako výstupy může použít pouze proměnné, které nejsou dosud vázány jinými buňkami se stejnou nebo silnější vnitřní silou. Tedy jen proměnné, které jsou dosud výstupy *stay* buněk, se mohou stát výstupními proměnnými buňky D - řádky 23 a 38 algoritmu 4.3.

Jenže proti proudu od buňky C se nemohou vyskytovat žádné *stay* buňky, protože pravidlo 6 už bylo pro buňku C vynuceno. Zpět před buňku C se už zařadit nemohou - musely by nahradit jinou *stay* buňku a žádná taková před C není.

Z toho ovšem plyne, že nová buňka nemůže být nikdy přidána proti proudu od libovolné jiné nerozhodnutelné buňky. Pravidlo pro buňku C tedy nemůže být porušeno.

Vynucování pravidla 6

Nová podmínka je přidána do sítě - vznikla buňka D. Nyní se pro buňku D začne vynucovat pravidlo 6 pomocí procedury *enforce_6* popsané v algoritmu 4.5. Takové vynucování se bude týkat pouze buněk proti proudu od D (řádek 9). Ovlivňovány budou pouze "slabší" buňky *stay* které budou odstraňovány, a buňky se stejnou silou jako D, které budou spojovány s buňkou D. Pokud je C proti proudu od D a $C.strength = D.strength$, pak se obě buňky spojí a sledování buňky C přestává mít smysl, bylo by třeba začít sledovat D, což je také nerozhodnutelná buňka, pro kterou právě začalo platit pravidlo 6.

Naopak, pokud je $C.strength$ silnější než $D.strength$, pak před C už nemohou být žádné buňky se silou $D.strength$ ani slabší, tedy ani žádné *stay* buňky. Vynucování pravidla 6 proti proudu od C už nemůže změnit strukturu sítě.

Je tedy vidět, že vynucování pravidla 6 pro další buňky buď skončí sloučením buňky C s jinou , stejně silnou buňkou, pro kterou bude pravidlo 6 také platit nebo vůbec nebude ovlivněna síť proti proudu od C, takže platnost pravidla 6 pro buňku C nemůže být narušena.

Vynucování pravidla 7

Vynucování pravidla 7 procedurou *enforce_7*, popsanou v algoritmu 4.8, mění síť dvěma způsoby. Dvě stejně silné nerozhodnutelné buňky se mohou

spojit do jedné (řádek 17) nebo může přidat hranu mezi dvě nestejně silné nerozhodnutelné buňky (směrem od silnější k slabší - řádek 19) .

Nechť je buňka C spojena se stejně silnou nerozhodnutelnou buňkou D, čímž vznikne buňka E pomocí procedury *merge* (algoritmus 4.7). Pro buňku D musí pravidlo 6 platit stejně jako pro "sledovanou" buňku C. Platí, že $C.strength = D.strength = E.strength$. Buňky C a D byly nerozhodnutelné, což bude platit i pro E. Proti proudu od C ani od D nemohly být buňky s vnitřní silou $C.strength$ ani slabší. Proti proudu od buňky E budou právě ty buňky, které byly proti proudu C nebo D. Tím pádem bude pravidlo 6 splněno.

Pro buňky po proudu od C a od E se také nic nemění. Nemohou tam být nerozhodnutelné buňky se silou stejnou nebo větší než $C.strength$. A proti proudu od C a D se vyskytují jen silnější buňky.

Jak je to v druhém případě, tedy přidávání hrany? Pokud je přidávána hrana z buňky C do D, změna se netýká buňky C, protože pravidlo se zajímá pouze o buňky proti proudu. Pro buňku D přibude proti proudu pouze buňka C, která je silnější než D, a buňky proti proudu od C, které jsou silnější než C a tedy i silnější než D. To znamená že nepřibude žádná buňka stejně silná nebo silnější než D a pravidlo 6 zůstane zachováno.

Pravidlo 7 bude vynuceno úplně na konci stavby sítě, takže jeho platnost už nic neohrozí. Správnost vynucování platnosti pravidla 7 byla velmi detailně diskutována na konci kapitoly 4.7.5.

4.8 Propagační fáze

Poté, co je síť podmínek postavena, je možné pomocí ní najít řešení původní hierarchie. Tato fáze se nazývá propagační, protože se v ní postupně jistým způsobem propagují ohodnocení skrze síť. Přesněji řečeno, jde o propagaci domén jednotlivých proměnných skrze posloupnost podmínek, která je určena topologickým setříděním sítě.

Princip propagační fáze je zachycen v algoritmu 4.9. Prvním krokem je setřídění podmínek. To se děje ve funkci *topo_sort* a bude detailněji diskutováno v sekci 4.8.3

Následuje postupné splňování podmínek v pořadí daném tímto setříděním. Tento krok je prakticky totožný s Indigem, které bylo detailně popsáno v 2.8. Podstatou rozdílu propagační fáze od Indiga je právě uspořádání podmínek. Indigo je totiž splňuje v pořadí daném jejich preferencemi.

Funkce *get_solution* už jen vybere řešení ze sítě. Projde všechny buňky, které mají nějaké výstupní proměnné. Díky použití *stay* podmínek bude mít každá proměnná v doméně právě jednu hodnotu. Dvojice proměnná-hodnota pak tvoří výsledné ohodnocení, které je vráceno jako řešení celé hierarchie.

Algoritmus 4.9: Find_solution

```
1  find_solution(network N)
2      queue Q;
3      constraint P;
4      solution S;
5  begin
6      Q:=topo_sort(N);
7      while(not_empty(Q))
8          P:=queue_get(Q);
9          propagate(N,P);
10     end while;
11
12     S:=get_solution(N);
13     return S;
14 end find_solution;
```

4.8.1 Reprezentace domén

Jak už bylo několikrát řečeno, teorie na které je postaven algoritmus Straton 3 , mluví o propagaci celých ohodnocení. Něco takového není možné přímo realizovat (viz 4.4), proto se skrz síť propagují pouze domény jednotlivých proměnných. V článku o Indigu [4] byl tento problém podrobně diskutován. Není těžké najít příklad, kdy použití jednoho intervalu pro reprezentaci aktuální domény proměnné nestačí. Takový příklad je rozebrán na straně 23 v sekci 2.8.5. Problematický je fakt, že v doméně po propagaci skrz podmínku zůstaly hodnoty, pro které nemůže být podmínka splněna. Následně jiná podmínka odstranila ostatní hodnoty, takže zbyly jen ty, které tvoří nesplňující ohodnocení.

V tomtéž článku je navrženo používat místo jednoho intervalu složitější struktury. Navrhováno je použití sjednocení uspořádaných disjunktních in-

tervalů, nazývaných *division* . Tyto struktury jsou použity v implementaci, kde je k jejich reprezentaci použit spojový seznam jednotlivých intervalů.

4.8.2 Globální komparátory

Teorie z kapitoly 3 původně směřovala k použití globálních komparátorů. Je to dobře vidět například na větě 2, kde je jeden z předpokladů linearita komparátorů. Tato vlastnost platí pro všechny globální komparátory (protože reálná čísla jsou úplně uspořádaná), naopak žádný lokální komparátor tuto vlastnost nesplňuje.

Zdálo by se tedy snazší implementovat propagační fázi pro nějaký globální komparátor. Není tomu tak. Problém nastává u nerozhodnutelných buněk, které se nepodaří bezezbytku splnit. Pokud proti proudu od takové nesplněné buňky leží další nerozhodnutelná buňka, musíme ji prozkoumat. Jde o to, že nerozhodnutelná buňka, zvláště pokud obsahuje více podmínek, má tu vlastnost, že může mít více různých řešení. Typickým příkladem nechť je buňka, kde nastává nějaký konflikt mezi podmínkami. Pak záleží na pořadí uspořádání jednotlivých podmínek uvnitř buňky. Může se stát, že pokud podmínky v této buňce uspořádáme jinak a pak výsledek odpropagujeme po proudu do už zpracovaných buněk, podaří se aktuálně nesplněnou buňku splnit lépe a současně se nezhorší stav splňování na ostatních buňkách, ležících po proudu od přeuspořádané buňky.

Takovéto řešení má ještě jednu nevýhodu. Při zpracování nerozhodnutelných buněk dochází ke změně vstupních domén. V případě, že po zpracování kusu sítě narazí algoritmus na nesplněnou podmínku, vrací se. Přitom by měl vrátit zpět všechny změny, které povedl na všech dosud ovlivněných proměnných. Poté, co se v některé nerozhodnutelné buňce změní pořadí podmínek, a tedy i výsledné ohodnocení, budou opět "vracet vrácené" změny. Jenže přitom musí hlídat, aby se jednotlivé "mezilehlé" podmínky nesplnily hůře, než tomu bylo před prvním návratem.

Až když budou vyzkoušeny všechna pořadí uspořádání podmínek ve všech nerozhodnutelných buňkách proti proudu od problematické buňky, bude možné s jistotou říci, že danou podmínku *opravdu* nelze splnit. Pak teprve bude propagace pokračovat za nesplněnou podmínkou. Navíc pokud algoritmus narazí na další nerozhodnutelnou podmínku, kterou se nepodaří rovnou splnit, bude opět následovat pokus o zpřeházení pořadí v nejhorším případě ve všech nerozhodnutelných buňkách proti proudu od nové problematické buňky.

Tento přístup by vyžadoval velmi časté navracení se ve výpočtu. Zřejmě

by se choval přijatelně při implementaci v neprocedurálním jazyku typu Prolog (kde se o navracení stará systém) a pro velmi malé testovací problémy. Pro velký problém s větším podílem nefunkcionálních buněk se takové řešení stává krajně problematické.

4.8.3 Implementovaný komparátor

Algoritmus Indigo používá komparátor označený *locally-error-better* . Tedy lokální komparátor, který používá "jakoukoli netriviální" chybovou funkci. Stejně tak v algoritmu Straton 3 je implementován lokální komparátor s netriviální chybovou funkcí. Jelikož je jako chybová funkce použita metrická vzdálenost, bude zde použitý komparátor nazýván *locally-metric-better* . Jedná se ale v podstatě o totožný komparátor, jen označení *locally-metric-better* je trošku konkrétnější.

Je dobré připomenout, že metrické komparátory, na rozdíl od predikátových, berou ohled i na podmínky, které se nepodaří úplně splnit. Pro takové podmínky se snaží alespoň minimalizovat chybu.

Když má být zpracováno více podmínek stejné preference, nezáleží při použití lokálního komparátoru na jejich pořadí (a tedy na tom, které konkrétně se případně podaří splnit, a které ne). Je to dáno tím, že dvě ohodnocení jsou nesrovnatelné, pokud je první horší pro jednu podmínku a lepší pro jinou. A pokud jsou nesrovnatelná, jsou stejně dobrá nebo stejně špatná, záleží jen na úhlu pohledu.

Propagační fáze potřebuje uspořádat podmínky tak, aby *odpovídaly struktuře sítě*. Co se týká celých buněk, ty stačí topologicky setřídit. V obecném případě existuje více různých setřídění, stačí vybrat libovolné. Jenže některé buňky obsahují více podmínek. V takovém případě, jak bylo výše řečeno, nezáleží na jejich pořadí, protože všechny podmínky z jedné buňky mají stejnou preferenci.

4.8.4 Složitost propagační fáze

Propagační fáze funguje obdobně jako Indigo. Rozdíl je pouze v uspořádání podmínek. To se může výrazně projevit na tom, jak často a jak daleko projde skrze už zpracované podmínky vlna "zmenšování domén" na základě splňování nové, dosud nezpracované podmínky.

Pro připomenutí - díky acykličnosti grafu hierarchie se každá proměnná může v jedné "vlně" změnit nejvýše jednou a proměnných je M. Taková vlna

může odstartovat po *prvním* splňování každé podmínky, kterých je N. Dohromady tedy dojde v hierarchii nejvýše k $O(MN)$ změnám domén, což pro potřeby odhadu časové složitosti můžeme považovat za elementární operaci.

4.8.5 Korektnost

Na tuto implementaci rozšířené teorie (o metafunkční buňky) z [1] se podle diskuze v kapitole 4.6 dá vztáhnou diskuze korektnosti ze stran 71-73 práce [1]. To znamená, že i pro posloupnost buněk, která vznikne topologickým uspořádáním sítě, platí věta 1. Tedy platí, že pokud Straton 3 najde nějaké řešení, bude správné.

4.8.6 (Ne)úplnost

Jeden z předpokladů věty 2 (o úplnosti) je linearita komparátoru. Linearita znamená, že libovolná dvě ohodnocení jsou komparátorem porovnatelná. Všechny globální komparátory jsou lineární. Naopak lokální komparátory obecně nejsou lineární. Rozhodně *locally-metric-better* i *locally-predicate-better* jsou nelineární na jakékoli netriviální hierarchii.

Pro lineární komparátor a posloupnost buněk splňující vlastnost postupného oslabování věta říká, že *pokud* postupné aplikování operátoru splňování hierarchií najde alespoň jedno ohodnocení, *pak* výsledek tohoto aplikování bude roven množině řešení celé hierarchie.

Jenže co s nelineárním komparátorem? Bohužel se dá ukázat opak, tedy že i když algoritmus najde nějaké řešení, nebude pro mnoho hierarchií schopen najít všechny.

Protipříklad Hierarchie H nechť obsahuje podmínky c_1 a c_2 s preferencí *strong*, které jsou ve vzájemném konfliktu, tedy nelze obě splnit současně. Dále nechť hierarchie obsahuje další podmínku c_3 se slabší preferencí, tedy *medium*. Podmínka c_3 nechť se chová "stejně" jako c_1, tedy pro libovolné ohodnocení π ať platí

$$e(c_1\,\pi) = e(c_3\,\pi) \tag{4.1}$$

(Typicky to bude "stejná podmínka", jen s jinou preferencí a jinak pojmenovaná).

Takovou hierarchii rozdělíme do buněk $B^1 = \{c_1, c_2\}$ a $B^2 = \{c_3\}$. Toto rozdělení zjevně splňuje vlastnost postupného oslabování.

Jelikož je množina *required* podmínek prázdná, bude Θ, tedy množina všech řešení *required* podmínek obsahovat všechna možná ohodnocení proměnných v H. Ještě je potřeba, aby první buňka měla dvě různá neporovnatelná řešení, tedy aby $\exists \sigma, \theta \in S(\Theta, B^1) \ : \sigma \nsim \theta$. Potřebujeme netriviální hierarchii, obsahující konflikt, tedy aby každé ohodnocení nesplňovalo alespoň jednu podmínku.

Z definice 5 rozepíšeme operátor splňování

$$S(\Theta, H') = \{\pi \in \Theta \mid \neg\exists\phi \in \Theta \quad \phi <^{H'} \pi\} \tag{4.2}$$

V tomto konkrétním případě $H \sim B^1$. Jelikož jsou obě ohodnocení σ i θ v řešení B^1, nemůže být jedno lepší než to druhé. Takže

$$\sigma \not<^{B^1} \theta \wedge \theta \not<^{B^1} \sigma \tag{4.3}$$

Když vezmeme v úvahu, že B^1 obsahuje pouze dvě podmínky, můžou nastat pouze dvě situace. Buď bude chyba na obou podmínkách pro obě ohodnocení stejná a obě ohodnocení budou stejně dobrá tedy $\sigma \sim^{B^1} \theta$ a nebo budou tato ohodnocení neporovnatelná. První možnost byla ale vyloučena zadáním, takže zbývá, že tato ohodnocení jsou neporovnatelná.

Komparátor je lokální. To podle definice 3 znamená, že

$$\pi \leq^{C} \phi \equiv \forall c \in C \quad e(c\pi) \leq e(c\phi) \tag{4.4}$$

Pro pouhé dvě podmínky a dvě neporovnatelná ohodnocení musí být jedno ostře lepší na jedné podmínce a druhé ostře lepší na druhé podmínce. Nechť tedy

$$e(c_1\,\sigma) < e(c_1\,\theta) \wedge e(c_2\,\theta) < e(c_2\,\sigma) \tag{4.5}$$

Nyní je vhodná chvíle na druhé aplikování operátoru splňování hierarchie.

$$S(S(\Theta, B^1), B^2) = \{\pi \in S(\Theta, B^1) \mid \neg\exists\phi \in S(\Theta, B^1) \quad \phi <^{B^2} \pi\} \tag{4.6}$$

Z 4.1 víme, že hodnota chybová funkce na c_1 je pro libovolné ohodnocení stejná jako na c_3. Pro jedinou podmínku v B^2 můžeme tedy psát

$$e(c_3\,\sigma) = e(c_1\,\sigma) < e(c_1\,\theta) = e(c_3\,\theta) \tag{4.7}$$

Jelikož je v této buňce c_3 jedinou podmínkou, platí tato nerovnost "pro všechny podmínky v buňce" a tedy $\sigma <^{B^2} \theta$ To ovšem znamená, že

$$\theta \notin S(S(\Theta, B^1), B^2) \tag{4.8}$$

Problém je v tom že $\theta \in S(\Theta, B^1 \cup B^2)$, protože hierarchický komparátor nedokáže ohodnocení σ a θ porovnat na *strong* úrovni a $\sigma, \theta \in S(\Theta, B^1)$, takže neexistuje ohodnocení, které by bylo na *strong* úrovni lepší než θ.

Na tomto příkladě je vidět, že aplikace operátoru splňování hierarchie nemůže najít všechna řešení pro lokální (tedy nelineární) komparátor.

Konkrétním příkladem může být následující hierarchie, pro kterou má být nalezeno *locally-predicate-better* řešení

$$\begin{array}{llll} c_1 & a = 10 & @ & strong \\ c_2 & a = 20 & @ & strong \\ c_3 & a = 10 & @ & medium \end{array}$$

Pro tu bude $S(\Theta, H) = \{\{a = 10\}, \{a = 20\}\}$, protože na úrovni *strong* jsou tyto dvě ohodnocení neporovnatelná, takže na slabší úrovně se ohlede nebere.

Při rozkladu na buňky, jak bylo výše v textu uvedeno, bude

$$S(\Theta, B^1) = S(\Theta, H) = \{\{a = 10\}, \{a = 20\}\}$$

Jenže při aplikaci $B^2 = \{c_3\}$ na $S(\Theta, B^1)$ bude vybráno jen první z těchto dvou řešení.

Tato kapitola ukazuje, že v obecném případě není možné pomocí sítí podmínek najít všechna řešení pro *lokální komparátor*. Navíc je ve větě 2 (o úplnosti) podmínka, že věta platí, pokud se podaří postupným aplikováním operátoru splňování hierarchie najít *alespoň jedno* řešení.

To znamená, že ani pro lineární(globální) komparátory není zaručeno, že by pomocí postupného aplikování operátoru splňování hierarchie bylo možné vždy najít řešení každé hierarchie, která řešení má.

4.8.7 Omezené hierarchie

Na počátku celé této kapitoly byla diskutována dvě omezení, kladená na zpracovávané hierarchie. Hierarchie musí být acyklické a nerovnosti v podmínkách musí být neostré.

Pokud je hierarchie acyklická, skončí propagační fáze po konečném počtu kroků. Důkaz je stejný jako u Indiga (stránky 7 až 11 v [4]). Algoritmus se může kdykoli zastavit z důvodu nesplnění *required* podmínky. Jinak pro každou podmínku hierarchie bude jednou volána procedura *propagate* - řádky 7 až 10 algoritmu 4.9. V rámci této procedury se změní doména každé proměnné nejvýše jednou a není možné, aby tam vznikl nekonečný cyklus "bez změn domén proměnných".

Druhým omezením, kladeným Stratonem 3 na podmínky hierarchií je, že nerovnosti musí být neostré. Navíc jsou uvažovány jen jednoduché podmínky, které nezpůsobí odstranění konkrétní hodnoty (nebo neostrého intervalu)z domény. V důsledku toho jsou jednotlivé intervaly v *divisions*, které reprezentují domény také neostré.

Při splňování podmínky se může zmenšovat doména. Toto zmenšování je v podstatě hledáním průniku mezi aktuální doménou proměnné a doménou, pro kterou by podmínka byla splněna, když ostatní proměnné nabývají hodnot svých domén. Pokud existuje průnik těchto dvou domén, omezí se na něj doména proměnné. Pokud průnik neexistuje, zůstanou v doméně pouze ty hodnoty, ve kterých je chyba na podmínce nejmenší. Příkladem nechť je podmínka $A + B = C$, doména A je $\langle 10, 20\rangle \cup \langle 40, 50\rangle$, doména B je $\langle 0, 5\rangle$ a doména C je $\langle 30, 35\rangle$. Pro proměnná A by měla být omezená na doménu $C - B$, což je $\langle 25, 35\rangle$. Pro hodnoty A 20 a 40 je bude chyba na podmínce minimální (tedy 5), proto bude aktuální doména A omezená na hodnoty $\langle 20, 20\rangle \cup \langle 40, 40\rangle$.

Jak je vidět, po libovolném splňování podmínky zůstává v doméně každé proměnné alespoň jedna hodnota. Díky použití *stay* podmínek nebude v doméně žádné proměnné více než jedna hodnota, takže pokud algoritmus skončí, aniž by našel nesplnitelnou *required* podmínku, bude mít každá proměnná právě jednu hodnotu a tyto hodnoty budou určovat ohodnocení, které bude výstupem algoritmu.

Díky korektnosti v 4.8.5 je pak zřejmé, že toto ohodnocení je skutečně řešením původní hierarchie.

4.8.8 Další řešení

Algoritmus nalezne vždy jen jedno ohodnocení, které je řešením. Pro postavenou síť je možné pokusit se najít jiné řešení opětovným spuštěním jen propagační fáze, ve které by se "jinak" uspořádaly podmínky. Půjde o jiné "vyhovující" uspořádání. Jiných takových uspořádání bude obvykle velmi mnoho, protože topologických setřídění bude také více a podmínky z každé buňky je možné setřídit libovolně.

Bohužel, toto nelze považovat za hledání "alternativního" řešení, protože je přitom vykonána vždy celá práce propagační fáze od začátku. Stejně tak by pak bylo možné tvrdit, že alternativní řešení umí hledat i Indigo - taky je možné jinak setřídit podmínky v rámci jednotlivých preferencí a pak pustit celé Indigo od začátku.

4.9 Implementace

Algoritmus Straton 3 nebyl jen navržen, ale také úspěšně implementován v C-čku. Tato implementace je obsahem přiloženého CD. Detaily této implementace jsou popsány v příloze.

Součástí implementace je nejen algoritmus Straton 3, ale pro porovnávání výsledku i Indigo. Propagační fáze Stratonu 3 se liší od Indiga pouze pořadím splňování podmínek. Jelikož oba algoritmy sdílejí datové struktury, je možné dobré porovnání jejich rychlosti.

Jak bude z naměřených časů vidět, první fáze Stratonu 3, tedy stavba sítě, zabírá hodně času. Na druhou stranu druhá fáze, tedy propagační, je rychlejší (průměrně o 25%) než Indigo. Pokud se hierarchie mění a má být opakovaně řešena, je Straton 3 výrazně pomalejší než Indigo. Naopak, pokud je struktura hierarchie stabilní a mění se jen nějaké "hodnoty", je Straton 3 lepší.

Experimenty byly prováděny na notebooku s procesorem Intel Centrino 1.6GHz, paměť 512 MB RAM, linux Debian 2.6.12.

4.9.1 Experimenty

Algoritmus Straton 3 byl testován na sadě hierarchií. Pro každý "typ" hierarchie bylo vygenerováno deset exemplářů a na nich se měřila doba běhu.

Velikost hierarchie

Prvních pět typů hierarchií se liší v počtu buněk od 500 do 8000. Ostatní vlastnosti jsou stejné

experiment 1	
počet "běžných" podmínek	mění se
podíl funkcionálních podmínek	50%
podíl podmínek tří proměnných	40%
podíl podmínek dvou proměnných	40%
podíl unárních podmínek	20%
počet úrovní hierarchie	64

Výsledky jsou v následující tabulce. První sloupec představuje identifikaci typu hierarchie a druhý měněnou veličinu, v tomto případě počet "běžných podmínek". Třetí sloupec obsahuje počet buněk v postavené síti. Je dobré

nezapomínat, že kromě "běžných" podmínek jsou v hierarchii ještě *stay* podmínky. Těch je stejně jako proměnných, kterých je pro daný podíl podmínek dvou a tří proměnných přesně o 20% více než podmínek.

Následující čísla jsou v sekundách. Jako první pod zkratkou *build* je "postavení sítě" bez vynucení pravidla 7, tj. postupné přidávání podmínek a vynucování pravidla 6. Další sloupec představuje čas běhu procedury *enforce_7*, tedy vynucování pravidla 7. Sloupec *prop.* je doba běhu druhé fáze algoritmu, tedy hledání řešení propagací skrze síť. Sloupec *total* značí celkovou dobu běhu algoritmu.

Sloupec *Indigo* je doba běhu "porovnávacího" Indiga. Poslední sloupec je v procentech a vyjadřuje podíl doby běhu Stratonu 3 a Indiga.

ID	#podm.	#buněk	build	enf7	prop	total	Indigo	%
A	500	501	0.04	0.17	0.04	0.28	0.04	85.71
B	1000	907	0.15	1.51	0.12	1.90	0.16	79.49
C	2000	1817	0.58	7.95	0.43	9.42	0.56	76.83
D	4000	3597	3.36	59.97	1.68	67.50	2.42	69.39
E	8000	7245	25.02	473.93	12.05	542.56	21.91	54.99

Je vidět, že doba běhu s velikostí hierarchie roste opravdu rychle. Zásluhu na tom nemá jen samotný algoritmus, ale hlavně jednoduchá, leč neefektivní implementace s použitím jednoduchých datových typů. Nejzajímavějším pozorováním ale je, že s velikostí hierarchie se zvyšuje náskok Stratonu 3 před Indigem.

Počet úrovní hierarchie

V tomto experimentu jde o to prozkoumat, jaký vliv na síť má počet úrovní hierarchie. Teto experiment je více než jiné ovlivněn náhodou. Na rozdíl od ostatních veličin se podmínky mezi hierarchie rozhazují naprosto náhodně, není kontrolováno, jestli není na některé úrovni moc nebo málo podmínek. Tato nevýhoda by měla být kompenzována zprůměrováním deseti hierarchií, vygenerovaných se stejnými parametry (jak je tomu ostatně i u jiných experimentů)

experiment 2	
počet "běžných" podmínek	2000
podíl funkcionálních podmínek	50%
podíl podmínek tří proměnných	40%
podíl podmínek dvou proměnných	40%
podíl unárních podmínek	20%
počet úrovní hierarchie	mění se

Popis výsledků je obdobný, jen druhý sloupec neobsahuje počet podmínek ale počet úrovní preferencí.

ID	#pref	#buněk	build	enf7	prop	total	Indigo	%
F	4	1347	0.57	2.32	0.39	3.74	0.52	74.19
G	16	1787	0.56	4.90	0.41	6.34	0.54	76.34
H	64	1980	0.57	5.89	0.40	7.34	0.53	76.08
I	256	2074	0.58	8.03	0.40	9.47	0.52	75.95
J	1024	2107	0.58	19.93	0.40	21.38	0.53	75.85

Počet úrovní má kupodivu opačný vliv na dobu stavby sítě, než by se dalo očekávat. Čím více úrovní je, tím déle algoritmus běží. Konkrétně jde o výrazný nárůst doby běhu *enforce_7*. Je to dáno výrazným nárůstem "granularity", což je dobře vidět na počtech buněk. Současně s granularitou roste i "výška sítě", tedy průměrné délky orientovaných cest. To se negativně projevuje právě na proceduře *enforce_7*.

Zajímavé by bylo, kdyby byl dispozici nějaký "propagátor" pro globální komparátor. Zde je totiž nejmarkantnější rozdíl v granularitě sítě, což by se mělo, podle teorie, projevit na složitosti hledání řešení pro globální komparátor. Bohužel, implementace takového propagátoru se zatím jeví jako velmi komplikovaná.

Podíl funkcionálních podmínek

Stejně tak jako počet hierarchií strukturu sítě ovlivňuje podíl funkcionálních proměnných. To totiž ovlivňuje podíl buněk, které nemohou být funkční a budou muset být generativní.

experiment 3	
počet "běžných" podmínek	2000
podíl funkcionálních podmínek	mění se%
podíl podmínek tří proměnných	40%
podíl podmínek dvou proměnných	40%
podíl unárních podmínek	20%
počet úrovní hierarchie	64

Obdobně jako v předchozím případě, mění se význam druhého sloupce. Tady obsahuje procento funkcionálních buněk. Stejným poměrem se týká podmínek tří, dvou i jedné proměnné.

ID	%funct	#buněk	build	enf7	prop	total	Indigo	%
K	0	1623	0.57	14.31	0.46	15.80	0.60	76.21
L	25	1798	0.57	7.57	0.42	9.02	0.55	75.68
M	50	1965	0.57	6.33	0.40	7.77	0.53	76.43
N	75	2174	0.58	3.91	0.39	5.34	0.51	76.13
O	100	2359	0.57	1.94	0.36	3.34	0.47	76.12

Oproti předchozímu experimentu je zde přesně opačná závislost granularity a doby vynucování pravidla 7. Je to dáno tím, že zde více granularizované sítě obsahují hlavně funkční buňky, pro které není nutné pravidlo 7 vynucovat. Naopak, pro hierarchie typu K, neobsahující žádné funkcionální podmínky, musí být *všechny* buňky nerozhodnutelné a proto se pro všechny buňky musí pravidlo 7 vynucovat.

Typy podmínek

Dosud byly všechny hierarchie "podobné". Nyní bude prověřeno, co se děje při změně podílu podmínek dvou a podmínek tří proměnných. Se vzrůstajícím podílem podmínek tří proměnných roste počet proměnných v hierarchii a tím i počet *stay* podmínek, takže tyto výsledky už nejsou tak "dobře srovnatelné, jako u ostatních experimentů. Nicméně dá se očekávat, že velikost problému se bude měřit buď počtem proměnných a nebo právě počtem "běžných" podmínek, protože *stay* podmínky jsou "nezajímavé". Jsou unární, příliš slabé a "pokaždé stejné (pro každou proměnnou jedna podmínka).

experiment 4	
počet "běžných" podmínek	2000
podíl funkcionálních podmínek	50%
podíl podmínek tří proměnných	mění se%
podíl podmínek dvou proměnných	mění se%
podíl unárních podmínek	20%
počet úrovní hierarchie	64

Druhý sloupec vyjadřuje procento podmínek dvou proměnných a procento podmínek tří proměnných. Dohromady vždy dávají 80 procent, zbývajících dvacet procent jsou "běžné" unární podmínky. Přibyl nový třetí sloupec *#prom.*, ukazující, kolik má daná hierarchie proměnných a tedy i *stay* podmínek.

ID	2:3	#prom.	#buněk	build	enf7	prop	total	Indigo	%
P	80:0	1601	1928	0.30	1.62	0.23	2.44	0.31	72.90
Q	60:20	2001	1939	0.39	4.38	0.32	5.47	0.42	76.19
R	40:40	2401	1989	0.57	6.03	0.40	7.46	0.52	76.35
S	20:60	2801	2015	0.69	6.39	0.50	8.15	0.66	75.27
T	0:80	3201	2038	0.82	7.12	0.59	9.22	0.78	75.77

Zajímavé je, že s "komplikovaností" hierarchie v důsledku většího podílu podmínek tří proměnných nevzrůstá časová náročnost. Tento nárůst jde ruku v ruce s nárůstem počtu proměnných a tím pádem i *stay* buněk.

Kapitola 5

Závěr

Cílem této práce bylo navrhnout nový algoritmus, který by dokázal řešit hierarchie podmínek využitím teorie sítí podmínek. To se povedlo, algoritmus Straton 3 dokáže pro zadanou hierarchii postavit "dobrou" síť podmínek a pomocí ní najít řešení. Samotná stavba sítě je časově náročná. Na druhou stranu, pokud je "struktura" hierarchie stabilní, je pomocí předem postavené sítě možné hledat řešení v lepším čase, než to dokáže algoritmus Indigo.

Algoritmus Straton 3 , na rozdíl od svých předchůdců (Straton 1 a 2), staví "nestratifikovanou" síť podmínek. Jako první tedy umožňuje zkoumat strukturu "netriviální" sítě a její závislost na vlastnostech hierarchie, ze které je postavena.

Díky tomu, že je algoritmus složen ze dvou nezávislých částí, je možné každou zlepšovat zvlášť. Cenným přínosem by bylo implementovat inkrementální přidávání podmínky do již existující sítě, což by výrazně zlepšilo použitelnost algoritmu.

Dalším zajímavým tématem by byla implementace globálního komparátoru. Po teoretické stránce jsou k tomu dobré předpoklady, ale implementace se zřejmě bude potýkat s velkými problémy, co se týká časové a prostorové efektivity.

Příloha A

Implementace a struktura dat

Algoritmus Straton 3 je implementován v čistém C-čku a bez úprav jde zkompilovat pod linuxem (Debian 2.6.12) i pod Windows XP.

Cílem implementace bylo demonstrovat funkčnost algoritmu. Proto je podřízena snaze o maximální jednoduchost, nikoli maximální efektivitu. Jsou použity jednoduché datové struktury jako fronta a seznam. implementace těchto struktur je v souborech `structures.c` a `structures.h`.

Reprezentace domén proměnných a splňování podmínek v propagační fázi je realizováno pomocí implementace *divisions*, tedy sjednocení uspořádaných disjunktních intervalů, a operací nad nimi. Ošetřování dosud neomezených okrajů intervalů domén $(-\infty, \infty)$ je realizováno pomocí C-čkové konstanty `HUGE_VAL`, ke které se dají přičítat a odečítat malá čísla, aniž by se měnila. Toto řešení plně postačuje pro potřeby "demonstrace" funkčnosti. Okraje intervalů jsou implementovány s přesností `double`. Toto vše je implementováno v souborech `divisions.c` a `divisions.h`

Samotný kód algoritmu Straton 3 je realizovaný pomocí těchto dvou knihoven v souboru `Straton.c`.

Program načte hierarchii ze souboru, jehož jméno je prvním parametrem volání. Z této hierarchie je postavena síť, která je uložená do souboru se jménem hierarchie plus příponou `.dot`. Jde o formát čitelný programem *dotty*, což je "zobrazovač" orientovaných grafů . Pro hierarchie mající více než 100 proměnných je použitá "úspornější" forma zápisu, kdy jsou buňky reprezentovány jen nejzákladnějšími údaji, aby bylo možné si alespoň prohlédnout "strukturu". I tak zobrazení grafu o více než tisící vrcholech mnoho neřekne, protože bude velice "rozlehlé".

Následně algoritmus použitím této sítě hledá řešení hierarchie, které uloží

do souboru s příponou `.txt`.

Nakonec je puštěno Indigo nad stejnou hierarchií - všechny podmínky jsou řešeny v pořadí podle preferencí. Díky tomu mají oba algoritmy stejné podmínky. Kód propagační fáze Stratonu 3 se od Indiga totiž liší pouze v pořadí zpracovaných podmínek. To zaručuje dobrou porovnatelnost obou algoritmů.

graphviz

Graphviz je balík programů pro vizualizaci grafů. Obrázky v této práci byly generovány pomocí *dotty*. Tento program je specializován na orientované grafy. Kromě něj byl použit ještě program *naeto* pro neorientované grafy.

Celý balík je možné stáhnout z `http://www.graphviz.org/`, kde jsou k dispozici jak zdrojové kódy, tak "balíčky" pro různé distribuce linuxu a také verze pro windows.

Generátor hierarchií

Pro testování při vývoji i pro zkoumání chování implementace je potřeba mít dostatek hierarchií. Proto bylo nutné napsat nějaký generátor, který by podle zadaných parametrů vyráběl testovací data.

Tento generátor je implementovaný v souboru `generator.c`. Dá se spouštět přímo z příkazové řádky s parametry, ale pro pohodlnější generování velkých souborů dat bylo třeba udělat ještě *shell-ovský skript*, který generuje hierarchie "ve velkém".

Parametry generátoru určují vlastnosti výsledné hierarchie.

#	popis parametru
1	počet "běžných" podmínek - tj. "velikost" hierarchie
2	procento funkcionálních podmínek
3	procento podmínek dvou proměnných
4	procento podmínek tří proměnných
5	počet úrovní hierarchie

Kromě "běžných" podmínek se vygenerují ještě *stay* podmínky, jedna pro každou proměnnou. Počet proměnných bude záviset na třetím a čtvrtém parametru. Hierarchie mají být acyklické, takže grafy hierarchií budou stromy nebo lesy. Lesům je potřeba se vyhnout, protože by šlo o "lehčí" problémy

- propagace z jedné proměnné by nikdy nemohla projít celou sítí. Z tohoto důvodu generátor staví hierarchie, které jsou stromy.

Každá proměnná dvou podmínek pak znamená přidání jedné "nové" proměnné. Podmínka o třech proměnných pak představuje přidání dvou proměnných. Je-li N počet "běžných" podmínek (první parametr), pak počet podmínek tří proměnných bude $N_3 = N \times p_3/100$, kde p_3 je čtvrtý parametr, tedy procento podmínek tří proměnných. Stejně tak $N_2 = N \times p_2/100$, kde p_2 značí procento podmínek dvou proměnných. $N_1 = N - N_3 - N_2$. Celkový počet vrcholů pak bude $M = 1 + N_2 + 2 \times N_3$. Stejně tak počet *stay* podmínek bude $N_{stay} = M$

Jako ukázka poslouží malé hierarchie, vyrobené generátorem hierarchii. Obsahuje 16 podmínek, šest vážících tři proměnné, šest vážících dvě proměnné a čtyři unární. Padesát procent podmínek je funkcionálních (rovnosti), zbytek nefunkcionální (nerovnosti) Celkem je v hierarchii 19 volných proměnných označených v0 až v18, pro které jsou na konci uvedeny odpovídající *stay* podmínky. V hierarchii nejsou žádné *required* podmínky.

Soubor je strukturován jako *pipe-delimited* s proměnnou strukturou řádky v závislosti na podmínce. Příklad jednoho řádku:

```
A B     C      D    E   F  G        H
2|310|P00000|1+2=3|v4|v18|v5|v4 + v18 = v5
```

A	preference podmínky, nižší číslo ∼ silnější preference
B	identifikace typu podmínky
C	označení podmínky
D	symbolický typ podmínky
E	první proměnná
F	druhá proměnná
G	třetí proměnná
H	text pro zobrazení podmínky

Zde je dobré rozvést identifikaci typu podmínky. Jde o trojmístné číslo, přičemž první číslice udává, kolik proměnných podmínka váže. Vyjímkou je *stay* podmínka, která má kód `000`. Druhá číslice udává, jestli je v podmínce plus (1) nebo mínus (0). Třetí číslice udává typ relace, tj. rovnost (0),"neostré větší než" (1) nebo "neostré menší než"(2).

Následující sloupec, označení podmínky, popisuje typ podmínky v "lidsky čitelné podobě" Čísla znamenají symbolické číslo proměnné, takže výše uvedené 1+2=3 znamená, první plus druhá proměnná se mají rovnat třetí proměnné. Jména těchto proměnných jsou pak v dalších sloupcích.

Pokud je v symbolickém typu podmínky uvedeno c místo symbolického čísla proměnné, jde o konstantu, která je uvedená na příslušném místě místo proměnné. Zřejmé je to z textu pro zobrazení.

kód	označení podmínky
000	*stay*podmínka
100	$1 = c$
101	$1 < c$
102	$1 > c$
200	$c - 1 = 2$
201	$c - 1 > 2$
...	
311	$1 + 2 > 3$
312	$1 + 2 < 3$

V textu pro zobrazení se vyskytuje "#" místo "<" nebo ">", aby bylo možno síť vizualizovat pomocí *dotty*.

```
#TIME : Wed Jan 18 00:40:11 2006

#generator 16 50 40 40 5
#cons1_num 4
#cons2_num 6
#cons3_num 6
#funct_density 50
#var_num 19
1|300|P00000|1-2=3|v3|v4|v10|v3 - v4 = v10
2|310|P00002|1+2=3|v2|v0|v11|v2 + v0 = v11
...
4|311|P00005|1+2>3|v15|v8|v11|v15 + v8 # v11
5|210|P00006|c-1=2|-290|v2|v7|-290 - v2 = v7
...
3|201|P00011|c+1>2|-128|v4|v9|-128 + v4 # v9
1|100|P00012|1=c|v3|-439|v3 = -439
...
```

```
1|102|P00015|1#c|v11|298|v11 # 298
6|000|P00016|stay(1)|v0|-991|stay(v0) = -991
...
6|000|P00034|stay(1)|v18|147|stay(v18) = 147
#STROMU :1
#TIME : Wed Jan 18 00:40:11 2006
#DURATION   0.00s
```

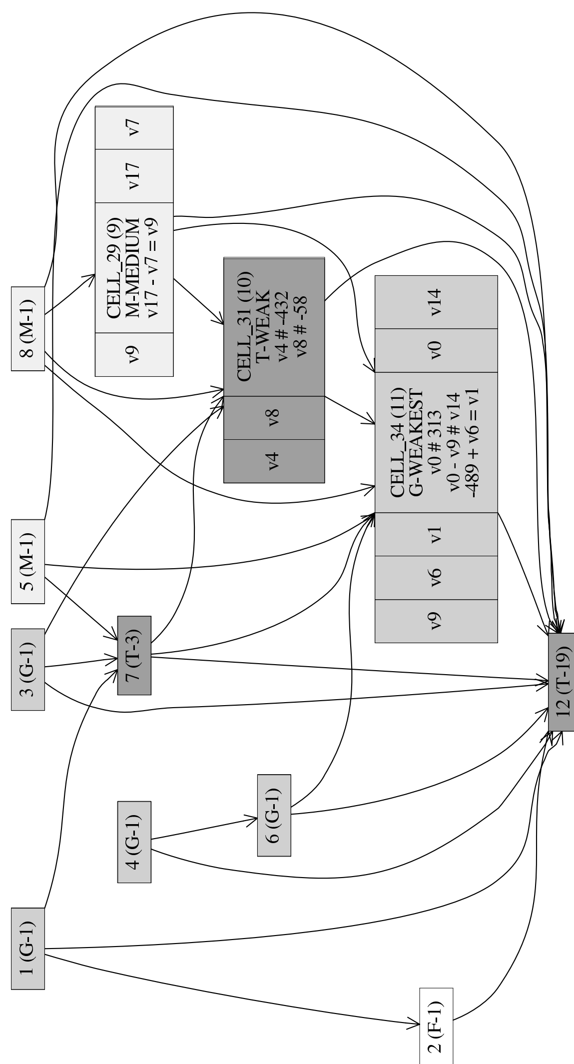
Na základě této hierarchie byla postavena síť. Jelikož je příliš velká na to aby se dala čitelně a přehledně vytisknout, je zde uvedena pouze v kompresované podobě zachycující samotnou strukturu. V buňkách je uvedeno pouze pořadí, které je výsledkem topologického třídění, typ buňky a počet podmínek v buňce.

Aby bylo vidět, jak by vypadala "plná" struktura při nekompresovaném zobrazení, jsou ponechány tři buňky beze změny. Jejich pořadí, tak jak jsou identifikovány kompresované buňky, je uveden v závorce v prvním řádku.

Za povšimnutí stojí buňka 12. Jde o buňku, ve které se v důsledku pravidla 7 spojily *stay* podmínky pro všechny proměnné – obsahuje tedy 19 podmínek.

Výstupem algoritmu je řešení hierarchie. To je uloženo do souboru taktéž v *pipe-delimited* formátu. Má ale pouze dva sloupce,jméno proměnné a její hodnotu. Proměnné nejsou setříděné podle svých jmen, ale podle pořadí, v jakém se vyskytují v podmínkách.

```
v3|          -439.000
v4|          -290.000
v10|          -149.000
...
v5|           155.000
v12|          -227.000
v9|           162.000
```



Obrázek A.1: Kompresovaná síť

Příloha B

Obsah CD

CD obsahuje následující adresáře:

`bin` - spustitelné soubory pro linux - "binárky" stratonu 3 a generátoru zkompilované pomocí gcc pro Debian 2.6.12

`experiments` - data, na základě kterých byly vyrobeny statistiky na konci kapitoly 4.

`graphviz` - instalační soubor pro *graphviz* pro windows. *Neoverena funkcnost* - programy z tohoto baliku byly pouzivany pod linuxem.

`source_code` - zdrojove soubory implementace stratonu 3 a generatoru nahodnyc hierarchi

`text` - zdrojove texty diplomove prace vLaTeXu.

`win` - implementace spustitelna pod windows. Obsahuje soubor `run_me.bat` předvádějící pouštění.

Literatura

[1] Barták, R., *Expertní systémy založené na omezujících podmínkách* (Disertační práce)
Universita Karlova,Praha, duben 1997

[2] Barták, R., *Constraint Hierarchy Networks*
in Proceedings of 3rd ERCIM/Compulog Workshop on Constraints, Amsterdam, září 1998

[3] Borning A., Freeman-Benson B., Wilson M.,*Constraint Hierarchies, Lisp and symbolic computation* in An International Journal, 1992/5,223-270

[4] Borning A.,Anderson R., Freeman-Benson B. *The Indigo Algorithm*
Tech. Report 96-05-01, Departement of Computer Science and Technology ,University of Washington, červenec 1996

[5] Sannella M. *The SkyBlue Constraint Solver*
Tech. Report 92-07-02 Dept. of Computer Science and Engineering, University of Washington, únor 1993

[6] Sannella M., Maloney J., Freeman-Benson B., Borning A. *Multi-way versus One-way Constraints in User Interface : Experience with the DeltaBlue Algorithm*
Tech. Report 97-02-05 ,Departement of Computer Science and Technology, University of Washington, červenec 1992