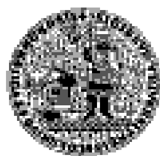
Matematicko-fyzikální fakulta Univerzity Karlovy v Praze
Katedra teoretické informatiky



Expertní systémy založené na omezujících podmínkách

Mgr. Roman Barták

Disertační práce zpracovaná pod vedením
Doc. RNDr. Petra Štěpánka, DrSc.
na
Katedře teoretické informatiky
Matematicko-fyzikální fakulty UK
Malostranské náměstí 2/25
Praha, Česká republika

Leden 1997

Obsah

Úvod ... 1

Část první
Uvedení do hierarchií omezujících podmínek

1.1 Programování s omezujícími podmínkami ... 4
1.1.1 CSP (Constraint Satisfaction Problems) ... 4
1.1.2 CLP (Constraint Logic Programming) ... 7

1.2 Hierarchie omezujících podmínek ... 9
1.2.1 Základní definice k hierarchiím ... 10
1.2.2 Komparátory ... 11
1.2.3 Existence řešení hierachie ... 20
1.2.4 HCLP-logické programování s hierarchiemi omezujících podmínek ... 21
1.2.5 Mezi-hierarchické porovnání ... 23
1.2.6 Vlastnosti komparátorů ... 25
1.2.7 Rozšíření klasické teorie hierarchií omezujících podmínek ... 27

1.3 Alternativní přístupy ... 28
1.3.1 Podmínky vyšších řádů ... 28
1.3.2 Před-řešení a před-míry ... 28
1.3.3 Kompozicionální teorie ... 30

1.4 Omezující podmínky a imperativní programování ... 33

1.5 Systémy řešení hierarchií omezujících podmínek ... 34
1.5.1 Jednoduchý interpret pro HCLP ... 34
1.5.2 DeltaStar ... 35
1.5.3 DeltaBlue ... 37
1.5.4 SkyBlue ... 40
1.5.5 QuickPlan ... 41
1.5.6 Indigo ... 43
1.5.7 Ultraviolet ... 45
1.5.8 Houria ... 45
1.5.9 IHCS ... 47

Část druhá
Expertní systémy a hierarchie omezujících podmínek

2.1 Expertní systémy 51

2.2 Expertní systémy založené na omezujících podmínkách 52

2.3 Obecný systém pro řešení hierarchií omezujících podmínek 54

2.3.1 Teorie řešení hierarchií podmínek 55

2.3.2 Buňky a sítě podmínek 68

2.3.3 Tvorba sítí podmínek (plánovací fáze) 74

2.3.4 Procházení sítí podmínek (prováděcí fáze) 78

2.4 Algoritmus pro mezi-hierarchické porovnání 80

Závěr 84

Literatura 85

Přílohy

Příloha A – CSP (Constraint Satisfaction Problems) 92

Příloha B – Základní HCLP interpret 95

Příloha C – Plánovací algoritmus zjemňovací metody 99

Příloha D – Tvorba sítě podmínek sofistikovaným plánovacím algoritmem 100

Příloha E – Schéma algoritmu pro HCLP s mezi-hierarchickým porovnáním 101

Poděkování

Rád bych na tomto místě poděkoval svému školiteli Doc. RNDr. Petru Štěpánkovi, DrSc. za jeho cenné rady, připomínky, náměty a užitečné diskuse při přípravě této práce stejně jako za pečlivé přečtení jejího rukopisu.

Chtěl bych také poděkovat svým rodičům a sestře za trpělivost, kterou se mnou při sepisování této práce měli.

V Praze dne 31. ledna 1997 Mgr. Roman Barták

„Were you to ask me which programming paradigm is likely to gain most in commercial significance over the next 5 years I'd have to pick Constraint Logic Programming (CLP), even though it's perhaps currently one of the least known and understood.“

Dick Pountain, BYTE, February 1995

Úvod

Programování s omezujícími podmínkami zatím patří k méně známých softwarovým technologiím, kterým se ovšem předpovídá dobrá budoucnost. Jak napovídá motto z úvodu, mohly by se omezující podmínky stát hybnou silou ve vývoji pokročilého softwaru příštích let. Ostatně již v roce 1995 představoval trh se softwarem založeným na omezujících podmínkách kolem 100 miliónů dolarů.

Omezující podmínky se dnes používají v takových oblastech jako je robotika, počítačová grafika, grafická uživatelská rozhraní nebo řešení rozsáhlých kombinatorických problémů. Výzkum v této oblasti sponzoruje třeba Apple Computer. Původ omezujících podmínek je částečně ve výzkumech složitých kombinatorických problémů z oblasti umělé inteligence a jedním z cílů této práce je navrátit technologii omezujících podmínek tam, odkud vzešla, tedy do lůna umělé inteligence. A co jiného dnes v očích veřejnosti reprezentuje výsledky a úspěchy umělé inteligence lépe a přesvědčivěji než právě expertní systémy, mnohými démonizované, ale dnes již v praxi běžně používané.

Co to vlastně jsou omezující podmínky a proč vzbuzují taková očekávání? Omezující podmínka není nic jiného než relace zachycující vztah mezi proměnnými, které obsahuje. Omezujícími podmínkami jsou například lineární rovnice a nerovnice. Pomocí sady omezujících podmínek lze deklarativně popsat řadu problémů, kombinatorickými problémy počínaje, přes grafická uživatelská rozhraní až po popis pohybu autonomního robota. Výhodou takového popisu problému je právě jeho deklarativní charakter, který se podobá klasickému, matematicky přesnému zadání problému. Pouhá formalizace problému v řeči omezujících podmínek by sama o sobě nestačila, důležitá je také existence efektivních systémů řešení omezujících podmínek. Ty vlastně převádí řešení z implicitní podoby (seznam podmínek) do podoby explicitní (ohodnocení proměnných), která je prezentována uživateli.

Cílem této práce je navrhnout novou softwarovou architekturu vhodnou zvláště pro tvorbu expertních systémů a systémů s „inteligentním“ chováním vůbec. Nová generace „inteligentních“ systémů založených na omezujících podmínkách zde slouží především jako motivace a prvotní cíl výzkumu. Vlastní obsah práce je potom zaměřen spíše na prostředky z oblasti omezujících podmínek nutné pro dosažení tohoto cíle.

V práci navrhovaná architektura expertních systémů je založena na hierarchiích omezujících podmínek s využitím mezi-hierarchického srovnávání. Její jádro tvoří obecný semi-inkrementální dvoufázový algoritmus pro splňování hierarchií podmínek použitím různých druhů komparátorů. Algoritmus má zásuvnou architekturu podporující všechny známé druhy komparátorů včetně komparátorů globálních. To mu dává dostatečnou obecnost. Na druhou stranu díky použití metod lokální propagace zůstává navrhovaný algoritmus dostatečně efektivní. Předkládaný algoritmus nepředstavuje uzavřený systém, ale právě naopak, poskytuje prostor k dalšímu výzkumu, zvláště co se efektivity různých implementací týče.

Disertační práce je organizována následujícím způsobem. První část práce je věnována poměrně podrobnému úvodu do hierarchií omezujících podmínek. Nejprve jsou stručně představeny „ploché“ omezující podmínky, tj. omezující podmínky bez hierarchií. Tyto podmínky hrají v současných softwarových aplikacích velkou roli díky svému deklarativnímu charakteru a řadě efektivních algoritmů pro jejich splňování. Prosazují se především v oblastech jako je plánování a rozvrhování. V této práci je řeč konkrétně o splňování podmínek nad konečnými doménami (CSP-Constraint Satisfaction Problems) a o logickém programování s omezujícími podmínkami (CLP-Constraint Logic Programming).

Větší díl první části je věnován hierarchiím omezujících podmínek, tj. takovým omezujícím podmínkám, které mohou mít přiřazenu jistou preferenci. Jsou zde uvedeny základní definice týkajících se hierarchií omezujících podmínek následované přehledem a srovnáním komparátorů. Komparátory jsou relace uspořádání umožňující vybrat nejlepší řešení dané hierarchie podmínek. Následuje několik tvrzení týkajících se existence řešení hierarchie podmínek a příklad použití hierarchií v logickém programování s hierarchiemi omezujících podmínek (HCLP-Hierarchical Constraint Logic Programming). Dále je pojem řešení hierarchie podmínek rozšířen o možnost porovnání řešení vzešlých z různých hierarchií, tzv. mezi-hierarchické srovnání, a jsou uvedena některá tvrzení týkající se vlastností komparátorů. Zmíněny jsou také alternativní přístupy k hierarchiím omezujících podmínek (podmínky vyšších řádů, před-řešení a před-míry, kompozicionální teorie) stejně jako integrace hierarchií podmínek s imperativním programováním. Zbytek první části je věnován přehledu algoritmů pro řešení hierarchií podmínek s podrobnějším pohledem do jejich útrob.

Druhá část práce je vlastním novým přínosem do oblasti hierarchií omezujících podmínek a konkrétně do oblasti algoritmů pro efektivní řešení hierarchií podmínek. Úvodní kapitola této části je věnována světu pravidlově orientovaných expertních systémů, který zde posloužil jako základní motivační prvek. V další kapitole je potom navržena architektura expertních systémů založených na hierarchiích omezujících podmínek. Jsou zde především rozebrány požadavky takovýchto systémů na podkladový systém řešení hierarchií omezujících podmínek.

Jádro druhé části tvoří popis obecného semi-inkrementálního algoritmu pro řešení hierarchií omezujících podmínek. Nejprve je navržena alternativní teorie hierarchií podmínek umožňující efektivní řešení hierarchií. Tato teorie je zde srovnána s přístupem prezentovaným v první části práce a jejím vrcholem jsou tvrzení o korektnosti a úplnosti navrhované metody řešení hierarchií. Následuje popis konkrétní instance navržené metody využívající pojmu sítě podmínek. V další dvojici kapitol jsou potom prezentovány algoritmy pro vytváření korektních sítí podmínek (plánování) a pro jejich procházení resp. pro propagaci ohodnocení sítí podmínek (provádění). Plánovací a prováděcí algoritmus dohromady tvoří efektivní systém pro řešení hierarchií omezujících podmínek. Aby bylo dostáno za dost požadavkům, které v práci navrhované expertní systémy kladou na podkladový systém řešení hierarchií, je v předposlední kapitole navržena obecná nadstavba algoritmu pro řešení hierarchií umožňující provádět mezi-hierarchické porovnání.

Práce je zakončena shrnutím dosažených výsledků, závěrečnými poznámkami o možnostech dalšího výzkumu a sadou příloh s popisem implementací některých v práci zmíněných technik.

Část první

Uvedení do hierarchií omezujících podmínek

1.1 Programování s omezujícími podmínkami

Pojem omezující podmínky není v informatice nějakou novinkou. První práce [Sut63], kterou lze považovat za popis systému s omezujícími podmínkami, vznikla již v roce 1963. Popisovala interaktivní grafický systém SKETCHPAD, který uživateli dovoloval vytvářet geometrické objekty z daných jazykových primitiv a jistých omezujících podmínek. Následníkem tohoto systému se stal také známý systém THINGLAB [Bor81], jehož jazyk již měl příchuť objektové orientace. Opět se jednalo o systém interaktivní grafiky. Řada dalších systémů založených na omezujících podmínkách pak následovala a jejich použití se rozšířilo do dalších oblastí jako je návrh elektronických obvodů, plánování a rozvrhování nebo třeba sekvencování DNA. Z dnešních populárních systémů jmenujme namátkou jazyky PROLOG III a IV, pocházející z dílny jednoho z tvůrců PROLOGu, Alaina Colmerauera, jazyk CHIP (Constraint Handling in Prolog) vyvinutý v ECRC nebo úspěšný komerční produkt ILOG-SOLVER, který je knihovnou funkcí pro práci s podmínkami v jazyce C++.

Co je to vlastně omezující podmínka? *Omezující podmínka* (*constrainf*)je relace zachycující vztah mezi proměnnými, které obsahuje. Cílem systémů řešících omezující podmínky je nalézt takové ohodnocení proměnných vyskytujících se v podmínkách, že všechny omezující podmínky jsou splněny. Podobnou problematikou se zabývají i jiné oblasti matematiky a informatiky jako je lineární algebra (soustavy lineárních rovnic) nebo matematické programování (soustavy lineárních nerovnic). Programování s omezujícími podmínkami na výsledky těchto oblastí plně navazuje a dále je rozšiřuje o možnost řešit širší třídu podmínek.

Oblast splňování podmínek má velkou tradici také v umělé inteligenci, kde byly vyvinuty mnohé sofistikované metody pro efektivní zjišťování konzistence podmínek. Stačí připomenout problematiku analýzy 3D scény založené na ohodnocování hran v 2D obrazech těles [Wal72, Wal75]. Ostatně prohledávací algoritmy tvořící jádro mnohých systémů umělé inteligence jsou předmětem zájmu i u splňování podmínek (viz. kapitola 1.1.1).

Značné popularity se v poslední době dostává logickému programování s omezujícími podmínkami [JaMa94], které díky snadné sémantice a dostatečné efektivitě považují mnozí za velice slibný nástroj pro vývoj komplikovaných aplikací.

1.1.1 CSP (Constraint Satisfaction Problems)

Problematika splňování podmínek (CSP-Constraint Satisfaction Problems) je již delší dobu předmětem výzkumu především v oblastech jako je plánování, rozvrhování a řešení složitějších kombinatorických problémů [Fre78, Gas74, HaEl80, Mac77, Mon74, Smi95, Tsa93, VaH89]. Úkolem CSP je, stručně řečeno, přiřadit hodnoty konečné množině proměnných, z nichž každá může nabývat konečně mnoha hodnot, tak aby byly splněny zadané omezující podmínky.

Příklad 1.1: (jednoduché problémy CSP)

a) Kryptoaritmetika

Najděte řešení rovnice SEND + MORE = MONEY tak, aby každé písmeno odpovídalo jedné cifře a všechna písmena měla přiřazena navzájem různé cifry.

b) Problém osmi dam

Najděte rozmístění osmi dam na šachovnici tak, aby se navzájem neohrožovaly.

Zadání problému v systémech CSP se zpravidla skládá z následující trojice kroků:

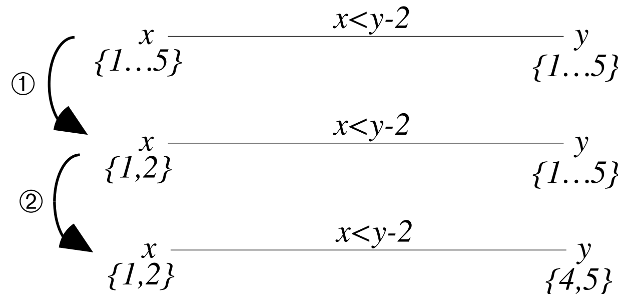
- určení proměnných a jejich domén, tj. množin hodnot, kterých mohou nabývat
- zadání omezujících podmínek, tj. relací nad proměnnými z předchozího kroku
- explicitní vyvolání ohodnocovací procedury, tzv. labelling.

Tento způsob zadání problému do systému CSP se velice podobá přirozenému popisu problému (viz. Příloha A). Proto také CSP a vlastně všechny systémy založené na omezujících podmínkách získávají na popularitě oproti dosud používaným technikám matematického programování, kde je přeformulování problému do formálního tvaru často poměrně pracný úkol vyžadující hlubší znalosti matematického programování.

Již při vstupu podmínky do systému CSP lze provést první omezení domén proměnných tak, aby byly vyřazeny některé nekonzistentní hodnoty. Unární podmínky omezí doménu přípustných hodnot rovnou tak, že všechny zbylé hodnoty podmínce vyhovují. Unární podmínku je proto možné po omezení domény ze systému vypustit. Všechny ostatní podmínky vyšší arity lze zjednodušit na podmínky binární [Smi95], které potom můžeme reprezentovat formou neorientovaného grafu. Vrcholy tohoto grafu jsou ohodnoceny příslušnými proměnnými a jejich doménami, hrana mezi dvěma vrcholy vede právě tehdy, když v systému existuje binární podmínka mezi proměnnými, kterými jsou vrcholy označeny. Hrana je potom ohodnocena touto podmínkou. Vzniklý graf se nyní omezováním domén proměnných převádí na graf *hranově konzistentní* (*arc consistency*). V hranově konzistentním grafu je každá hrana konzistentní, což znamená, že pro každou hodnotu z domény jedné proměnné existuje hodnota z domény druhé proměnné na hraně tak, že je splněna podmínka, kterou je hrana ohodnocena.

Příklad 1.2: [Smi95] (hranová konzistence)

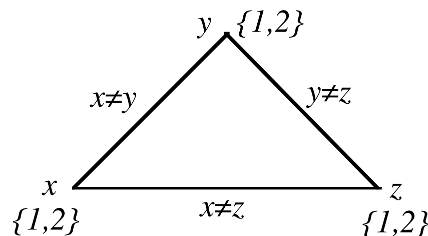
Nechť máme dány dvě proměnné x, y, obě s počáteční doménou $\{1\ldots5\}$ a nechť proměnné svazuje podmínka $x<y-2$. Hranové konzistence mezi proměnnými x a y dosáhneme ve dvou krocích. Nejprve se omezí doména proměnné x, tj. vyřadí se z ní ty hodnoty, pro které neexistuje hodnota v doméně pro y tak, aby byla splněna podmínka. Ve druhém kroku se stejným způsobem omezí doména proměnné y.



Testování hranové konzistence lze snadno efektivně implementovat, často je ale možné dalšími dedukcemi odstranit další nekonzistentní hodnoty. Přirozeným krokem po hranové konzistenci je testování konzistence tří a více proměnných. Hovoříme potom o testování *konzistence na cestě* (*path consistency*).

Příklad 1.3: (konzistence na cestě)

Nechť máme tři proměnné x, y a z, každou s počáteční doménou $\{1,2\}$, a nechť jsou tyto proměnné svázány podmínkami $x\neq y$, $y\neq z$ a $x\neq z$. Po testování hranové konzistence nejsou domény proměnných nijak omezeny a dostáváme následující hranově konzistentní graf:



Pokusíme-li se nyní tento graf udělat konzistentní po cestách délky 3, tj. ke každé hodnotě z domény jedné proměnné musí existovat hodnoty v doménách dalších

proměnných tak, aby byly splněny podmínky mezi testovanými proměnnými, zjistíme, že již při omezování domény proměnné *x* dostaneme prázdnou množinu. Jinými slovy to znamená, že zadaný problém CSP nemá řešení, což bychom při testování hranové konzistence neodhalili.

Testování konzistence na cestách delších než tři se z důvodů výpočtové náročnosti používá v praxi jen zřídka. Většina systémů zůstává ve fázi předzpracování podmínek pouze u testování hranové konzistence.

Po přezpracování podmínek, kdy se prvotně omezí domény proměnných, pokračuje většina algoritmů pro řešení CSP tzv. *labellingem*, tj. systematickým procházením všech možných ohodnocení proměnných a hledáním takového ohodnocení, které vyhovuje všem podmínkám. Tímto způsobem je zaručeno nalezení řešení, pokud nějaké existuje, případně dokázání, že problém je neřešitelný. Běh těchto algoritmů ale může být „dosti“ dlouhý.

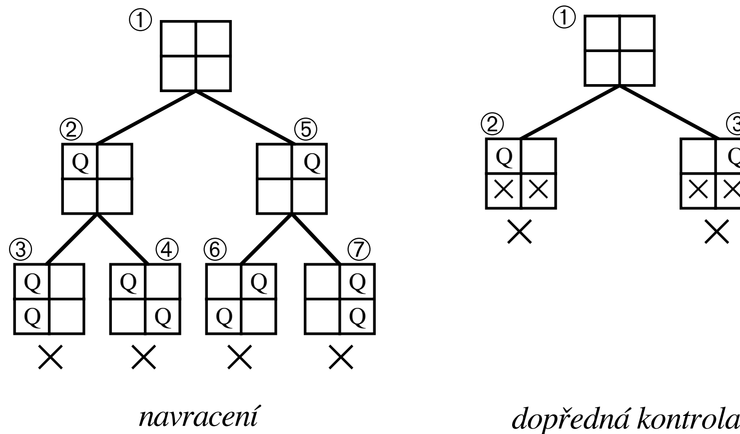
Mezi nejrozšířenější prohledávací algoritmy patří jednoduché prohledávání s navracením (backtracking) a prohledávání s dopřednou kontrolou (forward checking). Principem obou algoritmů je postupné dosazování hodnot za proměnné a průběžná kontrola konzistence podmínek.

Prohledávání s navracením (*backtracking*) kontroluje pouze podmínky mezi dosud ohodnocenými proměnnými. V případě, že narazí na nekonzistenci, vrací se k poslední ohodnocené proměnné a přiřadí jí další hodnotu z její domény. Pokud už jsou prvky domény vyčerpány, vrací se algoritmus k předchozí proměnné. Tento postup může být někdy dosti neefektivní, a proto byly vyvinuty metody *inteligentního navracení* (*intelligent backtracking*), kdy se algoritmus při navracení vrací až k proměnné, která způsobila konflikt. Nalezení příčiny konfliktu ale také není jednoduché, navíc inteligentní navracení vlastně řeší až „zotavení“ z konfliktu, místo aby samotnému konfliktu předcházelo.

Předcházením konfliktu se naopak vyznačují algoritmy založené na *dopředné kontrole* (*forward checking*). Ty po přiřazení hodnoty proměnné kontrolují všechny podmínky a hodnoty proměnných, které dosud nebyly ohodnoceny (tzv. budoucí proměnné) a které jsou ve sporu s dosud nalezeným částečným ohodnocením, jsou dočasně vyřazeny z domény příslušné proměnné. Pokud po kontrole konzistence dojde k tomu, že doména některé budoucí proměnné je prázdná, víme ihned, že současné částečné řešení je nekonzistentní, a algoritmus proto zkouší pro právě ohodnocovanou proměnnou jinou hodnotu resp. v případě vyčerpání domény se vrací k předchozí proměnné. Algoritmy dopředné kontroly takto dokáží odhalit větve v prohledávacím stromu vedoucí k neúspěchu dříve než při jednoduchém prohledávání s navracením.

Příklad 1.4: (navracení vs. dopředná kontrola)

Nechť máme za úkol rozmístit dvě dámy na šachovnici 2×2, tak, aby se navzájem neohrožovaly. Bez újmy na obecnosti můžeme úkol zjednodušit takto: najdi umístění první dámy v první řádku šachovnice a druhé dámy ve druhém řádku šachovnice tak, aby se navzájem neohrožovaly. Každá dáma je tedy reprezentována proměnnou s doménou {1,2} obsahující pořadová čísla sloupců. V systému je jediná binární podmínka zaručující, že se dámy neohrožují. Předpokládejme dále, že není použit test hranové konzistence, který by neřešitelnost problému ihned odhalil. Prohledávací stromy vzniklé při hledání hodnot proměnných potom vypadají takto (čísla určují pořadí procházení, tlusté křížky označují neúspěšné větve a křížky uvnitř šachovnic ukazují zakázané pozice určené při dopředné kontrole):



navracení *dopředná kontrola*

Jak je vidět, potřebuje prohledávání s navracením (backtracking) pro odhalení neřešitelnosti úkolu projít všechny možnosti, zatímco prohledávání s dopřednou kontrolou (forward checking) zjistí neřešitelnost již při umisťování první dámy. U příkladů, kde je více kombinací hodnot proměnných, se výhody dopředné kontroly projeví ještě výrazněji.

Efektivitu prohledávacích algoritmů lze dále ovlivnit vhodnou volbou pořadí, v jakém jsou ohodnocovány proměnné. Toto uspořádání může určit uživatel znalý řešeného problému, častěji ho ale určuje systém sám na základě různých heuristik. Jednou z běžně používaných heuristik je princip první chyby (first-fail), který lze popsat slovy *„aby jsi uspěl, zkus nejprve ty možnosti, kde nejspíše neuspěješ“*. V praxi to znamená výběr takové proměnné pro ohodnocení, jejíž doména je nejmenší. Tato volba vychází z předpokladu, že pravděpodobnost přítomnosti hodnoty v řešení je stejná pro všechny hodnoty. U proměnných, které mají v doméně více hodnot, je tak větší pravděpodobnost, že některá z jejích hodnot bude v řešení. Hledáme-li tedy proměnnou, jejíž ohodnocení nejspíše neuspěje, bude to ta proměnná, jejíž současná doména je nejmenší.

Pokud hledáme pouze jediné řešení soustavy podmínek, může efektivitu prohledávacího algoritmu ovlivnit také pořadí, v jakém jsou ohodnocované proměnné přiřazovány jednotlivé hodnoty z domény. Pro určení tohoto pořadí se opět používají různé heuristiky, jejichž cílem je upřednostnit ty hodnoty, které nejspíše povedou k řešení.

Úvodu do splňování podmínek jsou věnovány práce [Tsa93] a [Smi95], splňování podmínek v logických programech je zase tématem knihy [VaH89]. Mezi první publikace z oblasti konzistenčních technik patří [Wal72], tyto techniky byly dále rozvíjeny mimo jiné v [Fre78], [Gas74], [HaEl80], [Mac77] a [Mon74]. Další konzistenční techniky jsou popsány v [GrSm96], [Pro96] a [AtBa94]. Příklad implementace jednoduchého systému CSP lze nalézt v Příloze A.

1.1.2 CLP (Constraint Logic Programming)

Logické programování s omezujícími podmínkami (CLP-Constraint Logic Programming) [JaLa87] je třída programovacích jazyků kombinujících deklarativní vlastnosti logického programování s efektivitou řešení podmínek. Asi nejdůležitější výhodou jazyků CLP jsou krátké časy nutné pro vývoj aplikací při zachování efektivity vytvořených aplikací srovnatelné s imperativními jazyky.

Logické programování je známé svou jednoduchou a elegantní operační i deklarativní sémantikou. Tato sémantika má ale svá omezení. Objekty, se kterými se v logických programech manipuluje, jsou neinterpretované struktury, tj. termy, které lze sestavit z funkcí a konstant v daném programu. Každý sémantický objekt zde musí být explicitně zakódován do termu a rovnost pak platí pouze mezi objekty, které jsou syntakticky identické.

Příklad 1.5: (neinterpretované struktury PROLOGu)

Řešení dotazu `?-3=1+2` v PROLOGu neuspěje, protože termy `3` a `1+2` nejsou unifikovatelné.

Druhý problém svázaný s jazyky logického programování je důsledkem použitého uniformního ale jednoduchého výpočtového pravidla, prohledávání do hloubky. Jeho použití totiž vede k chování typu *generuj a testuj (generate and test)* se známými problémy efektivity u větších aplikací.

CLP je pokusem o překonání obou nastíněných problémů rozšířením jazyků typu PROLOG o mechanismus řešení omezujících podmínek. Základní myšlenkou je nahrazení výpočtového srdce systémů logického programování - unifikace, systémem řešení podmínek ve zvolené doméně. Toto obecné schéma, nazývané CLP(X), bylo poprvé navrženo v práci [JaLa87]. Dosazením konkrétní domény za X potom dostáváme instance obecného schématu, jako jsou CLP(R) pro reálná čísla, CLP(Z) pro celá čísla nebo CLP(FD) pro konečné domény ([VaH89], [Mei96]). Ve zvolené doméně jsou potom termy interpretovány.

CLP bylo silně ovlivněno také pracemi v oblasti konzistenčních technik (consistency techniques) [Gal85]. Aplikace těchto technik vede k aktivnímu použití podmínek pro omezení prohledávacího prostoru místo jejich pouhého pasivního testování, které vede k chování generuj a testuj. Nové paradigma může být charakterizováno jako technika *omez a generuj (constrain and generate)*. Příkladem systému, ve kterém se úspěšně používají konzistenční techniky, je CHIP (Constraint Handling in Prolog).

Příklad 1.6: [FHK+93] (programování v CLP(R))

Následující klauzule realizuje v CLP(R) násobení komplexních čísel:

```
zmul(R1,I1,R2,I2,R3,I3):-
     R3 = R1*R2 - I1*I2,
     I3 = R1*I2 + R2*I1.
```

Na dotaz `?-zmul(1,2,3,4,R3,I3)` odpoví systém:

```
R3 = -5
I3 = 10
```

Stejnou odpověď bychom ale dostali i v PROLOGu, pokud by místo = byl v programu `zmul` použit operátor `is`. V CLP(R) ovšem stejný program umí vyřešit i dotaz `?-zmul(R1,2,R2,4,-5,10),R2<3`, na který dá odpověď

```
R1 = 1.5
R2 = 2
```

v případě, že systém řešení podmínek umí řešit i nelineární rovnice, nebo

```
R1 = -0.5*R2 + 2.5
3 = R1*R2
R2<3
** maybe
```

pokud systém řešení podmínek umí řešit pouze lineární rovnice. V tomto druhém případě systém vrátí částečně vyřešené podmínky, o jejichž splnitelnosti není schopen rozhodnout, a doplní je hlášením `maybe` (možná). Tím se neurčitá odpověď liší od podobné odpovědi, kdy jsou také vráceny některé podmínky, které ovšem znamenají nekonečně mnoho řešení v závislosti na parametrech. Systém v tomto případě vydá jako odpověď minimální množinu podmínek svazujících proměnné z dotazu. Například na dotaz `?-zmul(1,2,R2,I2,R3,I3)` vrátí systém odpověď:

```
I2 = 0.2*I3 - 0.4*R3
R2 = 0.4*I3 + 0.2*R3
```

znamenající nekonečně mnoho řešení v závislosti na parametrech `I3` a `R3`.

CLP je věnováno velké množství prací. Obecné schéma CLP(X) bylo poprvé navrženo v práci [JaLa87], první jazyk logického programování explicitně používající podmínky, PROLOG II, je ale ještě starší [Col83]. Přehled teorie, implementačních technik, ukázky konkrétních aplikací a rozsáhlý seznam literaratury věnované omezujícím podmínkám lze nalézt v [VaH89] a [JaMa94]. Úvod k základním myšlenkám a technikám, které se skrývají za CLP, je také v [FHK+93]. Konkrétním systémům řešení podmínek jsou věnovány práce [Car94], [MRS95], [VHAK95] a [VHM95]. Návrh otevřené architektury pro implementaci CLP jazyků použité v systému ECL^{i}PS^e je popsán v [Neu90], [Mei95b], [MeBr95] a [MeSc95]. Tvorbě vlastních systémů řešení podmínek a uživatelsky definovaným podmínkám jsou věnovány práce [LePW93], [Frü93] a [FrBr95].

1.2 Hierarchie omezujících podmínek

V klasickém programování s omezujícími podmínkami musí být všechny podmínky, které se v systému vyskytnou, splněny. V mnoha aplikacích v oblastech jako je interaktivní grafika nebo plánování je ale někdy potřeba dát přednost splnění jedněch podmínek před druhými. Toho se v klasických systémech dosahovalo různými „triky“, které ovšem kazily příjemný deklarativní charakter omezujících podmínek.

Pro zavedení jistých preferencí omezujících podmínek hovořily i další zkušenosti. Pokud se totiž do systému zadá velké množství omezujících podmínek (někdy stačí dvě jednoduché „protichůdné“ podmínky typu X=1, X=2), je zde velká pravděpodobnost, že se nepodaří najít žádné řešení splňující všechny podmínky. Takovéto systémy se nazývají *příliš omezené* (*over-constrained*). Jejich opakem jsou *příliš volné* systémy (*under-constrained*), kde je omezujících podmínek málo, což vede k velkému množství nalezených řešení. Vlastní řešení tak de facto musí vybrat uživatel.

Vytvořit vyvážený systém omezujících podmínek, který lze splnit a který zároveň nedává příliš mnoho řešení, je velice komplikované. Přirozeným postupem proto došlo k rozdělení podmínek na *nutné „pevné“* podmínky (*hard constraints*), jejichž splnění je bezpodmínečně vyžadováno, a *volné* neboli *měkké* podmínky (*soft constraints*), které splněny být nemusí. Vyřešením nutných podmínek se najde množina přijatelných řešení, ze které jsou potom pomocí měkkých podmínek vybrána preferovaná řešení. Přednost se samozřejmě dává takovým řešením, která splňují více měkkých podmínek. Měkké podmínky tak vlastně fungují jako jakýsi usměrňovač výběru řešení.

Od jedné úrovně měkkých podmínek je to jen krůček k celé hierarchii podmínek skládající se z vrstev stejně preferovaných podmínek. V této hierarchii jsou potom silnější omezující podmínky preferovány na úkor slabších podmínek. Mají v tomto vztahu jakési právo veta, které říká, že splnění jedné více preferované podmínky má přednost před splněním libovolného množství méně preferovaných podmínek.

A co se stane, pokud se do sporu dostanou dvě podmínky na stejné preferenční úrovni? Potom dostane přednost ta, jejíž splnění získá větší podporu mezi ostatními podmínkami téže úrovně. Teprve v případě, že se na dané úrovni nelze dohodnout, tj. podpora obou „znesvářených“ podmínek je stejná, obrací se pozornost na nejvyšší nižší preferovanou úroveň atd. Pokud na žádné z dalších úrovní nedojde k rozhodující podpoře jednoho nebo druhého řešení, potom je výsledkem prostě více stejně dobrých řešení. Drobnou výjimku v tomto rozhodovacím procesu hraje úroveň nutně splnitelných (required) podmínek. Ty musí být bez výjimky splněny všechny a přirozeně tak v hierarchii reprezentují „pevné“ podmínky.

Spory mezi podmínkami na stejné preferenční úrovni lze také řešit upřednostněním jedné podmínky před druhou. Toho se dosáhne třeba tak, že každá podmínka má kromě své preference také tzv. *váhu* (*weight*). Podmínkám s větší vahou je potom na dané úrovni dávána přednost před ostatními podmínkami. V tomto případě ale může více

podmínek s menší vahou převážit jednu podmínku s velkou váhou a neplatí zde tak právo veta charakteristické pro preference. Z téhož důvodu není možné spojit preference a váhy do jediného pojmu.

Tolik k „lidovému“ popisu hierarchií omezujících podmínek a teď jsou na řadě formální definice. Čerpat při nich budeme především z prací [BMMW89, WiBo89, WiBo93] vytvořených na University of Washington, rodišti hierarchií omezujících podmínek.

1.2.1 Základní definice k hierarchiím

Omezující podmínka (*constraint*) je relace nad doménou D. Příkladem domény mohou být reálná čísla a omezující podmínkou nad touto doménou je třeba relace "≤". Spolu s doménou D je dána množinu 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. *Označená omezující podmínka* (*labeled constraint*) je omezující podmínka doplněná *preferencí* (*strength*). Často se pro ni používá zápis *lc* nebo *c@l*, kde *c* je omezující podmínka a *l* je preference. O množině preferencí se předpokládá, že je lineárně uspořádaná. Ve většině implementací je uživateli dovoleno definovat vlastní symbolická jména pro preference (např. *required*, *strong*, *weak*…). Tato jména jsou potom mapována na přirozená čísla $0,1,\ldots,n$, kde n je počet „měkkých“ preferenčních úrovní. Číslo 0 je přitom vyhrazeno pro nutné podmínky.

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 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 tj. preferovanější podmínky než v H_j.

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 omezující podmínky, tj. všechny 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.

Příklad 1.7:

1) doména R (reálná čísla)

$\Pi_D = \{=, \leq, \geq\}$

příklady omezujících podmínek: X=5, 2*X+Y≥Z

symbolické názvy preferenčních úrovní: required, strong, weak

hierarchie H omezujících podmínek:

required	X=Y
required	X≤Z
strong	2*X+Y≥Z
weak	Z=5

úrovně H_i potom vypadají takto:

$H_0 = \{X=Y, X\leq Z\}$

$H_1 = \{2*X+Y\geq Z\}$

$H_2 = \{Z=5\}$

$H_i = \emptyset$ pro i>2

ohodnocení {X/2, Y/2, Z/5} padne do řešení, protože splňuje všechny omezující podmínky, tj. splňuje všechny nutné podmínky a měkké podmínky splňuje lépe nebo stejně jako libovolné jiné ohodnocení

2) doména H (Herbrandovo univerzum)

$\Pi_D = \{=, \neq\}$

preferenční úrovně: required, strong, weak

hierarchie H omezujících podmínek:

required	X=f(Y)
weak	Y≠b
weak	X=a

úrovně H_i vypadají takto:

$H_0 = \{X=f(Y)\}$

$H_1 = \emptyset$

$H_2 = \{Y\neq b, X=a\}$

$H_i = \emptyset$ pro i>2

příklad ohodnocení {X/f(c), Y/c}.

Pro srovnání různých ohodnocení se používají metody, které se nazývají komparátory. Právě formálním definicím různých komparátorů bude věnována následující část.

1.2.2 Komparátory

Než přistoupíme k definicím komparátorů je třeba nejprve formalizovat množinu řešení hierarchie omezujících podmínek (v některých případech, kdy to bude jasné z kontextu, budeme slovem řešení označovat i jednotlivá ohodnocení z řešení).

Při hledání řešení se jako první vytvoří množina S_0 všech ohodnocení, která splňují nutné podmínky dané hierarchie. Nazvěme tato ohodnocení *potenciální ohodnocení*. Potom se použitím množiny S_0 definuje hledaná množina řešení S tak, že se ze všech potenciálních ohodnocení vyřadí ta, která jsou horší než nějaké jiné potenciální ohodnocení, a to prostřednictvím komparátoru *better*. Jinými slovy to znamená, že v množině S všech řešení zůstanou pouze ta ohodnocení z S_0, která nejsou horší než jiná ohodnocení z S_0. Formálně vypadá definice zmiňovaných množin takto:

Definice 1.1: [WiBo89] (řešení hierarchie podmínek)

množina potenciálních ohodnocení hierarchie H:

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

množina řešení hierarchie H:

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

kde $c\theta$ znamená výsledek aplikace ohodnocení θ na podmínku c.

Predikát *better* se nazývá *komparátor* a při pevně dané hierarchii H vlastně určuje binární relaci nad množinou ohodnocení. Na tuto relaci klademe další podmínky. Předně chceme, aby relace definovaná predikátem better, byla ireflexivní a tranzitivní, tj.

$\forall H \ \forall \theta \ \neg \ \text{better}(\theta,\theta,H)$ (ireflexivita)

$\forall H \ \forall \theta \ \forall \sigma \ \forall \tau \ \ (\text{better}(\theta,\sigma,H) \ \& \ \text{better}(\sigma,\tau,H) \Rightarrow \text{better}(\theta,\tau,H))$ (tranzitivita).

Poznamenejme také, že díky ireflexivitě a tranzitivitě dostáváme další vlastnost komparátoru better a tou je antisymetrie:

$\forall H \ \forall \theta \ \forall \sigma \ \neg \ (\text{better}(\theta,\sigma,H) \ \& \ \text{better}(\sigma,\theta,H))$.

Relace na množině ohodnocení definovaná komparátorem better tak určuje částečné uspořádání na množině ohodnocení, které je parametrizováno zvolenou hierarchií. Obecně není vyžadováno, aby toto uspořádání bylo lineární, což znamená, že mohou

existovat taková ohodnocení θ a σ , která nejsou srovnatelná, tj. θ není lepší než σ a ani σ není lepší než θ, formálně:

$\neg$ better(θ,σ,H) & $\neg$ better(σ,θ,H).

Poznamenejme také, že pro různé hierarchie může stejný komparátor better určovat různé relace uspořádání ohodnocení.

Některé systémy řešení hierarchie omezujících podmínek (viz. IHCS, kapitola 1.5.9) potřebují, aby komparátor better vždy definoval lineární relaci na množině ohodnocení. Toho se dosahuje jeho jednoduchým zúplněním, třeba tak, že ze dvou stejně dobrých ohodnocení je jako lepší vybráno to, které bylo nalezeno dříve.

Obecně je na komparátor better kladen ještě další požadavek a tím je *respektování hierarchie omezujících podmínek* (viz. právo veta). Znamená to, že pokud nějaké ohodnocení z S_0 splňuje všechny podmínky až do nějaké úrovně *k*, potom také každé ohodnocení z řešení S musí splňovat všechny podmínky až do úrovně *k*, tj.

pokud $\exists\theta\in S_0\ \exists k>0\ \ \forall i\in\{1,..,k\}\ \forall c\in H_i\ \ c\theta$ platí

potom také $\forall\sigma\in S\ \forall i\in\{1,..,k\}\ \forall c\in H_i\ \ c\sigma$ platí .

Definice 1.2: (respektování hierarchie podmínek)

Říkáme, že komparátor *better* respektuje hierarchii podmínek H, pokud splňuje následující podmínku:

$\forall k>0$

$((\exists\theta\in S_0\ \forall i\in\{1,..,k\}\ \forall c\in H_i\ c\theta$ platí & $\exists\sigma\in S_0\ \exists j\in\{1,..,k\}\ \exists c'\in H_j\ c'\sigma$ neplatí)

$\Rightarrow \exists\nu\in S_0$ better(ν,σ,H).

Jinými slovy, pokud nějaké ohodnocení z S_0 splňuje všechny podmínky až do nějaké úrovně *k*, potom pro každé ohodnocení $\sigma\in S_0$, které nesplňuje některou z podmínek na úrovních 1,…,k, existuje ohodnocení $\nu\in S_0$, které je lepší, tj. platí better(ν,σ,H). Tím je zajištěno, že ohodnocení σ nepadne do množiny řešení S.

Této vlastnosti komparátoru better lze dosáhnout opatrnější definicí řešení hierarchie omezujících podmínek (viz. [WiBo93]). Podle mého názoru se tím ale trochu znepřehlední celý přístup k hierarchiím, na druhou stranu je pak možné definovat i jiné druhy komparátorů (viz. regionally-better).

Příklad 1.8:

Nechť je dána hierarchie H nad doménou reálných čísel:

required	X=Y+Z
strong	X=5
weak	Y=2
weak	Z=4

a nechť máme ohodnocení:

ν={X/5, Y/2, Z/4}, δ={X/6, Y/2, Z/4}, σ={X/5, Y/2, Z/3}, θ={X/5, Y/1, Z/4}.

Potom zřejmě:

$\delta,\theta,\sigma\in S_0$ a $\nu\notin S_0$,

protože ohodnocení δ,θ,σ splňují nutnou podmínku, zatímco ohodnocení ν ji nesplňuje. Pro libovolný komparátor better bude určitě také platit better(σ,δ,H) i better(θ,δ,H), aby byl splněn požadavek respektování hierarchie podmínek (ohodnocení θ,σ splňují všechny strong podmínky, zatímco δ je nesplňuje).

Nyní je na čase, abychom představili některé konkrétní komparátory. Než ale přistoupíme k jejich definicím, je třeba ještě zvolit *chybovou funkci* (*error function*) *e(cθ)*, která vrací

nezáporné reálné číslo indikující, jak dobře je splněna podmínka *c* při ohodnocení θ. Chybová funkce *e* musí mít navíc následující vlastnost:

$e(c\theta)=0 \Leftrightarrow c\theta$ platí.

V libovolné doméně D můžeme zvolit triviální chybovou funkci *e* takovou, že:

$e(c\theta)=0 \Leftrightarrow c\theta$ platí.

$e(c\theta)=1$ v ostatních případech.

Komparátory využívající takovéto funkce potom budou mít v názvu slovo predicate (viz. locally-predicate-better) a můžeme je nazývat *predikátové komparátory* nebo *komparátory predikátového typu.*

V doménách, které jsou metrickými prostory, je možné chybovou funkci definovat třeba následujícím způsobem:

$e(X=Y) = dist(X,Y)$,

kde dist(X,Y) je vzdálenost bodů X a Y v daném metrickém prostoru. Dostaneme potom tzv. *metrické komparátory.*

Pokud se budeme pohybovat v oblasti fuzzy množin [Zad65, Nov86, Mat93], lze chybovou funkci definovat přirozeně následujícím způsobem:

$e(x\in A) = 1\text{-}mem(x,A)$,

kde mem(x,A) určuje míru příslušnosti prvku x do množiny A. Ta se pohybuje v intervalu <0,1>, přičemž hodnoty 1 nabývá u prvků, které do množiny určitě patří, a hodnoty 0 u prvků mimo množinu.

Komparátory používající obecně nějakou netriviální chybovou funkci bez jejího přesného určení budou mít často v názvu slovo *error* (např. locally-error-better)

Komparátory lze v principu rozdělit do dvou základních skupin: komparátory lokální a globální. Podle novější klasifikace[WiBo93] lze definovat ještě komparátory regionální, ty ale nesplňují podmínku tranzitivity, a tak je formálně nemůžeme za komparátory tak, jak zde byly definovány, považovat.

Zatímco *lokální komparátory*, kterými náš přehled začneme, srovnávají dvě ohodnocení na základě porovnání chyb jednotlivých omezujících podmínek, *globální komparátory* využívají kombinační funkci *g*, která sdružuje chyby všech omezujících podmínek na dané úrovni, a teprve její výsledky jsou porovnány. *Regionální komparátory* mohou být považovány za odrůdu komparátorů lokálních, protože podobně jako tato třída komparátorů jsou i regionální komparátory založeny na individuálním porovnání jednotlivých omezujících podmínek. Na rozdíl od lokálních komparátorů ale mohou být dvě ohodnocení, která nejsou na vyšší preferenční úrovni porovnatelná, srovnána regionálním komparátorem na méně preferované úrovni a jedno z nich pak vybráno jako lepší. Díky této vlastnosti mohou regionální komparátory srovnat více ohodnocení, a lze tak říci, že částečné uspořádání na ohodnoceních generované regionálními komparátory je „úplnější“ než uspořádání určené lokálními komparátory.

Definice 1.3: (lokální komparátor)

Říkáme, že ohodnocení θ je *lokálně lepší* (locally-better) než jiné ohodnocení σ, pokud je hodnota chybové funkce pro všechny omezující podmínky až do nějaké úrovně *k-1* po aplikaci θ stejná jako po aplikaci σ a na úrovni *k* je chyba po aplikaci θ ostře menší u alespoň jedné omezující podmínky a menší nebo rovna u všech ostatních podmínek než chyba po aplikaci σ.

Formální zápis této definice je podstatně kratší a přehlednější:

locally-better(θ,σ,H) $\equiv_{def}$

$\exists k>0\ \forall i \in \{1,\dots,k-1\}\ \forall c \in H_i\ \ e(c\theta)=e(c\sigma)\ \&$

$\exists c' \in H_k\ e(c'\theta)<e(c'\sigma)\ \&\ \forall c \in H_k\ e(c\theta)\leq e(c\sigma)$,

kde H_i je i-tá úroveň hierarchie H a cθ výsledek aplikace ohodnocení θ na podmínku c.

Tvrzení 1.1:

Relace locally-better splňuje podmínky kladené na komparátor.

Důkaz:

a) ireflexivita

zřejmé, protože $\neg\exists c\ \ e(c\theta)<e(c\theta)$, tj. ¬ locally-better(θ,θ,H)

b) tranzitivita

locally-better(θ,σ,H) & locally-better(σ,δ,H) ≡

$\exists k>0\ \forall i \in \{1,\dots,k-1\}\ \forall c \in H_i\ \ e(c\theta)=e(c\sigma)\ \&$

$\exists c' \in H_k\ e(c'\theta)<e(c'\sigma)\ \&\ \forall c \in H_k\ e(c\theta)\leq e(c\sigma)\ \&$

$\exists n>0\ \forall i \in \{1,\dots,n-1\}\ \forall c \in H_i\ \ e(c\sigma)=e(c\delta)\ \&$

$\exists c'' \in H_n\ e(c''\sigma)<e(c''\delta)\ \&\ \forall c \in H_n\ e(c\sigma)\leq e(c\delta)$

zvolme l=min{k,n}, potom

$l>0\ \&$

$\forall i \in \{1,\dots,l-1\}\ \forall c \in H_i\ e(c\theta)=e(c\sigma)=e(c\delta)\ \&\ \forall c \in H_l\ e(c\theta)\leq e(c\sigma)\leq e(c\delta)\ \&$

$\exists c^+ \in H_l\ e(c^+\theta)<e(c^+\delta)$

protože stačí $c^+=c'$, je-li l=k, tj. $e(c'\theta)<e(c'\sigma)\leq e(c'\delta)$, nebo

$c^+=c''$, je-li l=n, tj. $e(c''\theta)\leq e(c''\sigma)<e(c''\delta)$

zřejmě tedy

$\exists l>0\ \forall i \in \{1,\dots,l-1\}\ \forall c \in H_i\ \ e(c\theta)=e(c\delta)\ \&$

$\exists c^+ \in H_l\ e(c^+\theta)<e(c^+\delta)\ \&\ \forall c \in H_l\ e(c\theta)\leq e(c\delta) \equiv$

locally-better(θ,δ,H)

c) respektování hierarchie podmínek

nechť pro libovolné k>0 platí:

$\exists \theta \in S_0\ \forall i \in \{1,..,k\}\ \forall c \in H_i\ \ c\theta$ platí (tj. $\forall i \in \{1,..,k\} \forall c \in H_i\ \ e(c\theta)=0$)

& $\exists \sigma \in S_0\ \exists j \in \{1,..,k\}\ \exists c' \in H_j\ \ c'\sigma$ neplatí (tj. $e(c'\sigma)>0$),

potom zřejmě locally-better(θ,σ,H), protože

$\forall i \in \{1,\dots,j\}\ \forall c \in H_i\ \ e(c\theta)=0\leq e(c\sigma)\ \&\ \exists c' \in H_j\ 0=e(c\theta)<e(c'\sigma)$.

❑

Někdy se místo obecného názvu *locally-better* používá název *locally-error-better* zdůrazňující přítomnost nějaké netriviální chybové funkce *e*.

Pokud za *e* zvolíme triviální chybovou funkci, potom dostaneme tzv. *locally-predicate-better komparátor* (LPB), který formálně můžeme přepsat třeba takto:

locally-predicate-better(θ,σ,H) $\equiv_{def}$

$\exists k>0\ \forall i \in \{1,\dots,k-1\}\ \forall c \in H_i$ (cθ platí ⇔ cσ platí) &

$\exists c' \in H_k$ (c'θ platí & c'σ neplatí) & $\forall c \in H_k$ (cθ neplatí ⇒ cσ neplatí).

Locally-predicate-better lze dobře popsat také množinově. Nechť $H_{k,\theta}$ je množina všech podmínek z úrovně H_k, které jsou splněny při ohodnocení θ:

$H_{k,\theta} = \{ c \mid c \in H_k \ \& \ c\theta \text{ platí}\}$.

Potom lze locally-predicate-better komparátor definovat také takto:

$$\text{locally-predicate-better}(\theta,\sigma,H) \equiv_{def}$$
$$\exists k>0 \ \forall i \in \{1,\dots,k-1\} \ H_{i,\theta}=H_{i,\sigma} \ \&$$
$$H_{k,\theta} \supset H_{k,\sigma},$$

kde symbol ⊃ značí ostrou inkluzi.

Příklad 1.9: (použití LPB komparátoru)

Nechť je dána hierarchie omezujících podmínek nad Herbrandovým univerzem:

```
required   X=Y
  strong   X=a
  strong   Y=b
    weak   Z=c
    weak   X=Z.
```

Potom locally-predicate-better dá následující čtveřici stejně dobrých řešení: {X/a,Y/a,Z/c}, {X/a,Y/a,Z/a}, {X/b,Y/b,Z/c} a {X/b,Y/b,Z/b}. Tato ohodnocení totiž splňují nutnou podmínku a po jedné podmínce z úrovní strong a weak. Navzájem jsou nesrovnatelná LPB komparátorem a žádné jiné ohodnocení splňující nutnou podmínku není lepší. Poznamenejme, že například ohodnocení {X/c,Y/c,Z/c} splňující nutnou podmínku i obě weak podmínky je horší než všechna řešení, protože nesplňuje žádnou ze strong podmínek.

Použitím metrické chybové funkce *e* dostaneme další z lokálních komparátorů tzv. *locally-metric-better* komparátor.

Principem srovnání chyb u jednotlivých omezujících podmínek se lokálním komparátorům podobají komparátory regionální. Zatímco ale lokální komparátory vyžadují, aby se chyby u jednotlivých podmínek až do určité úrovně rovnaly, regionálním komparátorům stačí, pokud nejsou dvě ohodnocení na stejných úrovních srovnatelná, tj. nelze rozhodnou, které je lepší.

Definice 1.4: (regionálně nesrovnatelná ohodnocení)

Dvě ohodnocení *nelze na dané úrovni srovnat*, pokud jsou chyby u odpovídajících podmínek stejné (tj. jako u lokálních komparátorů) nebo pokud existuje podmínka s menší chybou při jednom ohodnocení a zároveň existuje jiná podmínka, která má menší chybu při druhém ohodnocení.

Definice 1.5: (regionální komparátor)

Říkáme, že ohodnocení θ je *regionálně lepší* (regionally-better) než jiné ohodnocení σ, pokud nejsou ohodnocení θ a σ až do nějaké úrovně $k-1$ srovnatelná a na úrovni k je chyba po aplikaci θ ostře menší u alespoň jedné omezující podmínky a menší nebo rovna u všech ostatních podmínek než chyba po aplikaci σ.

Formální zápis definice regionálního komparátoru vypadá takto:

$$\text{regionally-better}(\theta,\sigma,H) \equiv_{def}$$
$$\exists k>0 \ \forall i \in \{1,\dots,k-1\}$$
$$((\forall c \in H_i \ e(c\theta)=e(c\sigma)) \lor ((\exists c_1 \in H_i \ e(c_1\theta)<e(c_1\sigma)) \ \& \ (\exists c_2 \in H_i \ e(c_2\theta)>e(c_2\sigma))))$$
$$\& \ \exists c' \in H_k \ e(c'\theta)<e(c'\sigma) \ \& \ \forall c \in H_k \ e(c\theta) \le e(c\sigma).$$

Podobně jako u lokálních komparátorů lze i u komparátorů regionálních získat volbou chybové funkce *e* jejich speciální tvary.

Regionally-predicate-better komparátor definovaný množinově potom může vypadat třeba takto:

regionally-predicate-better(θ,σ,H) $\equiv_{def}$

$\exists k>0\ \forall i\in\{1,\ldots,k-1\}\ (\neg\ H_{i,\theta}\supseteq H_{i,\sigma}\ \&\ \neg\ H_{i,\theta}\subseteq H_{i,\sigma})\ \&$

$H_{k,\theta}\supset H_{k,\sigma}$,

kde $H_{k,\theta}$ má stejný význam jako v množinové definici locally-predicate-better komparátoru.

Tvrzení 1.2:

Relace regionally-better je ireflexivní a respektuje hierarchii podmínek.

Důkaz:

a) ireflexivita

zřejmé, protože $\neg\exists c\ \ e(c\theta)<e(c\theta)$, tj. $\neg$ regionally-better(θ,θ,H)

b) respektování hierarchie podmínek

nechť pro libovolné k>0 platí:

$\exists\theta\in S_0\ \forall i\in\{1,..,k\}\ \forall c\in H_i\ \ c\theta$ platí (tj. $\forall i\in\{1,..,k\}\forall c\in H_i\ \ e(c\theta)=0$)

& $\exists\sigma\in S_0\ \exists j\in\{1,..,k\}\ \exists c'\in H_j\ \ c'\sigma$ neplatí (tj. $e(c'\sigma)>0$),

potom zřejmě regionally-better(θ,σ,H), protože

$\forall i\in\{1,\ldots,j\}\ \forall c\in H_i\ \ e(c\theta)=0\leq e(c\sigma)\ \&\ \exists c'\in H_j\ 0=e(c\theta)<e(c'\sigma)$.

❑

Jak již bylo zmíněno v úvodu ke komparátorům, nesplňuje relace regionally-better obecně podmínku tranzitivity, a nemůžeme ji proto považovat za komparátor tak, jak jsme jej definovali v této práci. Na druhou stranu je tato relace ireflexivní a respektuje hierarchii podmínek, a tak se v praxi někdy jako komparátor používá. Důvodem je to, že umí srovnat více ohodnocení než lokální komparátory.

Příklad 1.10: (regionally-better není tranzitivní)

Nechť máme dánu následující hierarchii H nad Herbrandovým univerzem:

strong	X=b
strong	Y=a
strong	X=a
medium	X≠a
weak	X=c

a nechť máme trojici ohodnocení:

θ={X/b, Y/c}, σ={X/a, Y/a}, δ={X/c, Y/a}.

Potom při použití regionally-predicate-better komparátoru platí:

regionally-predicate-better(θ,σ,H) - rozlišeny na úrovni medium

regionally-predicate-better(σ,δ,H) - rozlišeny na úrovni strong

regionally-predicate-better(δ,θ,H) - rozlišeny na úrovni weak.

Pokud by byl komparátor tranzitivní, potom by platilo regionally-predicate-better(θ,θ,H), což neplatí, protože regionální komparátory jsou ireflexivní. Komparátor regionally-predicate-better tedy není v tomto případě tranzitivní.

Odlišnost lokálních a regionálních komparátorů demonstruje následující jednoduchý příklad.

Příklad 1.11: (odlišnost lokálních a regionálních komparátorů)

Nechť je dána hierarchie podmínek nad Herbrandovým univerzem:

strong X=a
strong X=b
weak $X \neq b$.

Potom použitím locally-predicate-better komparátoru dostaneme dvojici „stejně dobrých" řešení {X/a} a {X/b}, které splňují po jedné strong podmínce, zatímco regionally-predicate-better upřednostní jediné řešení {X/a}, které kromě jedné strong podmínky splňuje také weak podmínku.

K lokálním a regionálním komparátorům ještě jedna poznámka. Vzhledem k tomu, že je zde každá omezující podmínka posuzována samostatně, nemá u lokálních ani regionálních komparátorů smysl použití vah, o kterých jsme mluvili v úvodu k hierarchiím.

Globální komparátory se od lokálních i regionálních liší použitím kombinační funkce *g*, která sdružuje chyby jednotlivých podmínek na dané úrovni. Tato funkce přiřazuje každému ohodnocení na zvolené úrovni nějaké nezáporné reálné číslo a musí splňovat podmínku:

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

Kombinační funkce *g* je potom dalším parametrem do schématu *globally-better*, jehož definice následuje.

Definice 1.6: (globální komparátor)

Říkáme, že ohodnocení θ je *globálně lepší* (globally-better) než jiné ohodnocení σ, pokud je kombinovaná chyba na každé úrovni až do nějaké úrovně *k-1* stejná po aplikaci θ jako po aplikaci σ a na úrovni *k* je kombinovaná chyba po aplikaci θ ostře menší než kombinovaná chyba po aplikaci σ.

Formálně:

$$\text{globally-better}(\theta,\sigma,H,g) \equiv_{def}$$
$$\exists k>0 \ \forall i \in \{1,\ldots,k-1\} \ g(\theta,H_i)=g(\sigma,H_i) \ \& $$
$$g(\theta,H_k)<g(\sigma,H_k).$$

Tvrzení 1.3:

Pro libovolnou kombinační funkci *g* je relace globally-better komparátorem.

Důkaz:

Stačí ukázat, že globally-better je ireflexivní, tranzitivní a že respektuje hierarchii podmínek:

a) ireflexivita

zřejmé, protože $\neg\exists i>0 \ g(\theta,H_i)<g(\theta,H_i)$ tj. $\neg$ globally-better(θ,θ,H,g)

b) tranzitivita

$$\text{globally-better}(\theta,\sigma,H,g) \ \& \ \text{globally-better}(\sigma,\delta,H,g) \equiv$$
$$\exists k>0 \ \forall i \in \{1,\ldots,k-1\} \ g(\theta,H_i)=g(\sigma,H_i) \ \& \ g(\theta,H_k)<g(\sigma,H_k) \ \&$$
$$\exists n>0 \ \forall i \in \{1,\ldots,n-1\} \ g(\sigma,H_i)=g(\delta,H_i) \ \& \ g(\sigma,H_n)<g(\delta,H_n)$$

zvolme $l=\min\{k,n\}$, potom

$$l>0 \ \&$$
$$\forall i \in \{1,\ldots,l-1\} \ g(\theta,H_i)=g(\sigma,H_i)=g(\delta,H_i) \ \& \ g(\theta,H_l)<g(\delta,H_l)$$

protože $g(\theta,H_l)<g(\sigma,H_l) \leq g(\delta,H_l)$, je-li $l=k$ nebo

$g(\theta,H_l) \leq g(\sigma,H_l) < g(\delta,H_l)$, je-li $l=n$

zřejmě tedy

$\exists l>0\ \forall i\in\{1,\dots,l-1\}\ g(\theta,H_i)=g(\delta,H_i)\ \ \&\ g(\theta,H_l)<g(\delta,H_l)\equiv$

globally-better(θ,δ,H,g)

c) respektování hierarchie podmínek

nechť pro libovolné k>0 platí:

$\exists\theta\in S_0\ \forall i\in\{1,..,k\}\ \forall c\in H_i\ \ c\theta$ platí (tj. $\forall i\in\{1,..,k\}\ g(\theta,H_i)=0$)

& $\exists\sigma\in S_0\ \exists j\in\{1,..,k\}\ \exists c\in H_j\ \ c\sigma$ neplatí (tj. $g(\sigma,H_j)>0$)

potom zřejmě globally-better(θ,σ,H,g), protože

$\forall i\in\{1,..,j-1\}\ g(\theta,H_i)=0\le g(\sigma,H_i)\ \&\ 0=g(\theta,H_j)<g(\sigma,H_j)$.

❑

Volbou kombinační funkce dostaneme různé globální komparátory. V následující části ukážeme několik druhů globálních komparátorů, které využívají také možnosti přiřadit každé omezující podmínce tzv. *váhu* (*weight*). Váhu podmínky *c* označujeme w_c a jedná se o kladné reálné číslo, které určuje vliv neboli sílu podmínky na dané hierarchické úrovni. Větší váha pak odpovídá většímu vlivu. Jinými slovy větší váha znamená větší tlak na to, aby příslušná podmínka byla splněna, protože její nesplnění by přišlo příliš „draho“. Zde jsou tedy tři příklady globálních komparátorů, které definujeme tak, že v obecném schématu pro *globally-better* určíme konkrétní kombinační funkci *g*:

weighted-sum-better(θ,σ,H) $\equiv_{def}$ globally-better(θ,σ,H,g),

kde $g(\tau,H_i)=\sum_{c\in H_i} w_c * e(c\tau)$

worst-case-better(θ,σ,H) $\equiv_{def}$ globally-better(θ,σ,H,g),

kde $g(\tau,H_i)=\max_{c\in H_i}\{w_c * e(c\tau)\}$

least-squares-better(θ,σ,H) $\equiv_{def}$ globally-better(θ,σ,H,g),

kde $g(\tau,H_i)=\sum_{c\in H_i} w_c * e^2(c\tau)$.

Je zřejmé, že uvedené funkce *g* splňují podmínku kladenou na kombinační funkci.

Podobně jako u lokálních komparátorů můžeme i zde volbou vhodné chybové funkce *e* získat speciální případy právě definovaných globálních komparátorů. Pokud například použijeme triviální chybovou funkci u komparátorů *weighted-sum-better* a *least-squares-better*, dostaneme stejný globální komparátor. Ten se, v případě že navíc zvolíme váhu 1 u každé podmínky, nazývá *unsatisfied-count-better*. Při jeho aplikaci se systém snaží minimalizovat počet resp. vliv (v případě použití nejedničkových vah) nesplněných podmínek.

Příklad 1.12: [WiBo93] (porovnání komparátorů)

Nechť je dána hierarchie omezujících podmínek nad reálnými čísly:

required	C=A+B
strong	C=7
weak	A=2
weak	B=3

Potom jednotlivé komparátory dají následující řešení:

locally-predicate-better: {A/2,B/5,C/7} a {A/4,B/3,C/7}

locally-metric-better: nekonečně mnoho řešení tvaru {A/x,B/7-x,C/7} pro $x\in[2\dots4]$

regionální komparátory dávají v tomto příkladě stejná řešení jako příslušné komparátory lokální

weighted-sum-predicate-better: {A/2,B/5,C/7} a {A/4,B/3,C/7}, pokud jsou váhy u obou weak podmínek stejné, jinak vybere jedno z řešení podle toho, která váha je větší (např. {A/2,B/5,C/7}, je-li větší váha u podmínky A=2)

weighted-sum-metric-better: nekonečně mnoho řešení tvaru {A/x,B/7-x,C/7} pro x∈[2…4], pokud jsou váhy u obou weak podmínek stejné, jinak vybere jedno z řešení podle toho, která váha je větší, tj. {A/2,B/5,C/7}, je-li větší váha u podmínky A=2, resp. {A/4,B/3,C/7}, je-li větší váha u podmníky B=3

worst-case-predicate-better: {A/x,B/7-x,C/7}, kde x∈R, pokud jsou váhy u obou weak podmínek stejné, jinak vybere jedno z řešení podle toho, která váha je větší, tj. {A/2,B/5,C/7}, je-li větší váha u podmínky A=2, resp. {A/4,B/3,C/7}, je-li větší váha u podmníky B=3

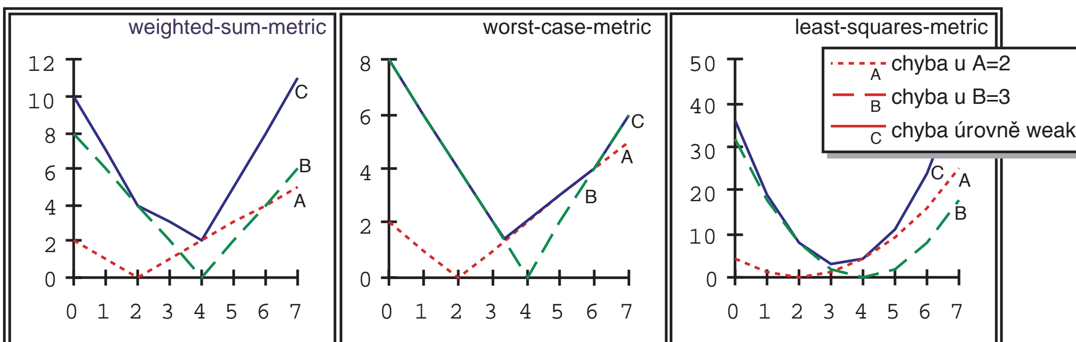
worst-case-metric-better: {A/x,B/7-x,C/7}, kde $x = \dfrac{4 * w_{B=3} + 2 * w_{A=2}}{w_{B=3} + w_{A=2}}$, tj. pokud jsou váhy u obou weak podmínek stejné, je řešením {A/3,B/4,C/7}

least-squares-predicate-better: vždy dává stejné výsledky jako weighted-sum-predicate-better komparátor (protože $0^2=0$ a $1^2=1$)

least-squares-metric-better: {A/x,B/7-x,C/7}, kde $x = \dfrac{2 * w_{B=3} + 4 * w_{A=2}}{w_{B=3} + w_{A=2}}$, tj. pokud jsou váhy u obou weak podmínek stejné, je řešením {A/3,B/4,C/7}

Za povšimnutí stojí, že pro libovolnou dvojici vah (nezáporná reálná čísla) bude u posledních dvou metrických komparátorů vždy x∈[2…4]. Podmínky na úrovni weak tak zajišťují, že se řešení vždy (s vyjímkou worst-case-predicate-better komparátoru při stejných vahách) nachází v „rozumných" mezích, tj. že ohodnocení typu {A/1000000,B/-999993,C/7}, které také splňuje všechny podmínky z úrovní required a strong, nepatří do řešení.

Následující odstavec objasňuje způsob nalezení řešení u globálních metrických komparátorů. Protože všechna (a žádná jiná) ohodnocení tvaru {A/x,B/7-x,C/7} splňují všechny podmínky na úrovních required a strong, je jasné (komparátory respektují hierarchii), že se řešení bude hledat mezi těmito ohodnoceními a to podle míry splnění podmínek úrovně weak. V praxi to znamená nalezení takových hodnot proměnné *x*, že chyba na úrovni weak je nejmenší (na ostatních úrovních je chyba nulová). Následující obrázky ukazují grafy chybové funkce v závislosti na *x* jak jednotlivých podmínek tak i celé úrovně weak. Nyní stačí nalézt minimum chybové funkce a určit hodnotu(y) *x*, kde je minimum nabýváno. Pro ilustraci byly u jednotlivých podmínek úrovně weak zvoleny následující váhy: $w_{A=2} = 1$, $w_{B=3} = 2$.



1.2.3 Existence řešení hierachie

Pokud je množina S_0 potenciálních ohodnocení, tj. ohodnocení splňujících nutné podmínky, neprázdná, intuitivně bychom očekávali, že potom i množina S všech řešení je také neprázdná. Nicméně v některých patologických případech tomu tak není.

Příklad 1.13: [WiBo93] (neexistence řešení)

Nechť je dána následující hierarchie nad doménou reálných čísel:

required X>0
strong X=0

a nechť používáme nějaký metrický komparátor. Potom platí:

$S_0 = \{ \{X/r\} \mid r \in R^+ \}$ (tj. množina ohodnocení, která mapují X na kladné reálné číslo)

$S = \emptyset$, protože pro libovolné ohodnocení $\{X/r\}$ můžeme najít jiné ohodnocení, např. $\{X/(\frac{r}{2})\}$, které lépe vyhovuje strong podmínce X=0.

Přes problémy nastíněné v příkladě 1.13 můžeme formulovat některá tvrzení o existenci řešení hierarchie omezujících podmínek.

Tvrzení 1.4: [WiBo93]

Pokud je množina S_0 potenciálních ohodnocení hierarchie H neprázdná a konečná, potom je i množina S všech řešení hierarchie H neprázdná.

Důkaz:

Nechť je množina S_0 potenciálních ohodnocení hierarchie H neprázdná a konečná. Předpokládejme pro spor, že množina S je prázdná. Protože množina S_0 je neprázdná, můžeme v ní vybrat nějaké ohodnocení $\theta_1 \in S_0$. Vzhledem k tomu, že S je prázdná, platí $\theta_1 \notin S$, a tedy existuje nějaké ohodnocení $\theta_2 \in S_0$ takové, že platí better(θ_2,θ_1,H), kde H je hierarchie, jejíž řešení hledáme (existence θ_2 plyne z definice 1.1 množiny S). Protože opět $\theta_2 \notin S$ (S je prázdná), můžeme najít další ohodnocení $\theta_3 \in S_0$ takové, že platí better(θ_3,θ_2,H). Stejným způsobem nalezneme celý nekonečný řetězec ohodnocení $\theta_1,\theta_2,\theta_3,\theta_4,\ldots$, ve kterém díky tranzitivitě komparátoru *better* platí: $\forall i,j>0$ ($i>j \Rightarrow$ better(θ_i,θ_j,H)). Protože *better* je zároveň ireflexivní, víme že $\forall i>0 \neg$ better(θ_i,θ_i,H), a tudíž jsou všechna ohodnocení θ_i navzájem různá a je jich tedy nekonečně mnoho. Protože $\forall i>0\ \theta_i \in S_0$, je také množina S_0 nekonečná, což je ovšem spor s předpokladem tvrzení.

Množina S tedy nemůže být za daných předpokladů prázdná.

❑

Tvrzení 1.5: [WiBo93]

Pokud je množina S_0 potenciálních ohodnocení hierarchie H neprázdná a používáme predikátový komparátor (tj. komparátor s triviální chybovou funkcí), potom je i množina S všech řešení hierarchie H neprázdná.

Důkaz:

Nechť je množina S_0 potenciálních ohodnocení hierarchie H neprázdná. Připomeňme, že hierarchie H je konečná (multi)množina označených podmínek. Předpokládejme pro spor, že množina S je prázdná. Stejným způsobem jako v důkazu tvrzení 1.4 získáme nekonečný řetězec $\theta_1,\theta_2,\theta_3,\theta_4,\ldots$ navzájem různých ohodnocení. Protože používáme komparátor predikátového typu, nemůže být jedno ohodnocení lepší než druhé, pokud obě ohodnocení splňují přesně stejné podmnožiny podmínek hierarchie H. Z toho plyne, že každé ohodnocení θ_i musí

splňovat různou podmožinu podmínek z H, a těchto podmnožin je tedy nekonečně mnoho. Proto také hierarchie H musí být nekonečná, což je ovšem spor s konečností hierarchie H.
Množina S tedy nemůže být za daných předpokladů prázdná.

❑

Tvrzení 1.6:
Je-li doména, nad kterou jsou definovány omezující podmínky, konečná a pokud je množina S_0 potenciálních ohodnocení hierarchie H neprázdná, potom je i množina S všech řešení hierarchie H neprázdná.

Důkaz:
Protože hierarchie H je konečná (multi)množina omezujících podmínek, obsahuje pouze konečně mnoho proměnných. Vzhledem k tomu, že také doména, nad kterou jsou definovány omezující podmínky, je konečná, existuje pouze konečně mnoho navzájem různých ohodnocení. Proto je i množina S_0 obsahující pouze ohodnocení splňující nutné podmínky konečná a z předpokladu věty také neprázdná. Nyní můžeme využít tvrzení 1.4 a dostáváme, že množina všech řešení hierarchie H je neprázdná.

❑

1.2.4 HCLP-logické programování s hierarchiemi omezujících podmínek

Příkladem použití hierarchií omezujících podmínek je třeba systém Multi-Garnet [SaBo92] pro tvorbu grafických uživatelských rozhraní nebo imperativní programovací jazyk Kaleidoscope [FBB92b, LFBB94b] (viz. kapitola 1.4). V této části ale bude řeč o logickém programování s hierarchiemi omezujících podmínek (Hierarchical Constraint Logic Programming - HCLP) [BMMW89,WiBo93]. Jedná se o přirozené rozšíření logického programování s omezujícími podmínkami (CLP), které je dnes velice populární [BeCo93,Car94,FHK+93,JaMa94,Mei96,SaVH95,Smi95,VaH89], o hierarchie.

HCLP pravidlo (klauzule) má tvar:

$p(t) :- q_1(t), \dots, q_m(t), l_1 c_1(t), \dots, l_n c_n(t).$

kde t je seznam termů, $p(t), q_1(t), \dots, q_m(t)$, jsou atomy a $l_1 c_1(t), \dots, l_n c_n(t)$ jsou označené omezující podmínky.

HCLP program je potom seznam (multi-množina) HCLP pravidel a cíl (dotaz) je multi-množina atomů. V praxi může cíl také obsahovat označené podmínky, pro zjednodušení teoretického popisu se ale cíl uvažuje bez podmínek. Každý cíl obsahující označené podmínky totiž můžeme bez újmy na obecnosti přejmenovat na nový predikát, který se potom stane novým cílem, viz. příklad:

cíl s podmínkami

$?- q_1(t), \dots, q_m(t), l_1 c_1(t), \dots, l_n c_n(t).$

předěláme tak, že k programu přidáme nové pravidlo, kde p je nový predikátový symbol

$p(t) :- q_1(t), \dots, q_m(t), l_1 c_1(t), \dots, l_n c_n(t).$

a cíl změníme na:

$?- p(t).$

Z operačního hlediska se cíl redukuje stejně jako v CLP, s tím rozdílem, že měkké podmínky nezasahují do výpočtu a jsou pouze shromažďovány. Po té, co je cíl úspěšně zredukován, ještě pořád můžeme dostat více odpovědí nebo také žádnou odpověď. Nashromážděná hierarchie měkkých podmínek se předá k vyřešení příslušnému systému, který používá vhodnou metodu pro zvolenou doménu a použitý komparátor. Pokud obsahuje množina řešení více ohodnocení, jsou další ohodnocení získávána navracením (backtracking).

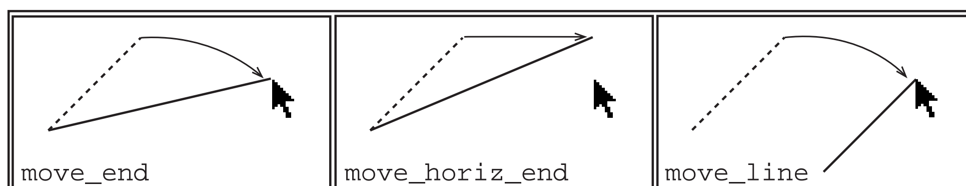
Právě popsaný postup běhu HCLP programu je stručným výkladem operační sémantiky HCLP programů. Popis její konkrétní implementace lze nalézt v části věnované systémům řešení hierarchií omezujících podmínek (kapitola 1.5). Úplný popis operační sémantiky HCLP programů spolu s popisem modelově-teoretické (model-theoretic) sémantiky a sémantiky pevných bodů (fixed-point) včetně jejich srovnání je v práci [WiBo93].

Následující příklad ukazuje možnosti využití HCLP při tvorbě interaktivních grafických aplikací.

Příklad 1.14: (HCLP pro interaktivní grafiku)

Situace: S pomocí HCLP lze snadno vytvářet grafické editory typu MacDraw. Jednou z typických akcí v takovém editoru bývá prodloužení resp. zkrácení nakreslené čáry v horizontálním směru metodou táhni a pusť (drag & drop). To se provádí tak, že uživatel myší „uchopí“ jeden konec čáry a přesune ho na požadované místo. Samozřejmě myší lze pohybovat nejen horizontálně ale také vertikálně, a proto konec čáry nenásleduje myš úplně, ale sleduje pouze její horizontální souřadnici.

Řešení: K vyřešení nastíněného úkolu použijeme naprogramování jednoduššího problému a tím je přesné sledování pohybu myši druhým bodem čáry (`move_end`). Použitím tohoto „programu“ a přidáním nutné podmínky zajišťující setrvání druhého bodu čáry na stejné horizontální souřadnici získáme řešení úkolu (`move_horiz_end`). Podobným způsobem můžeme nyní také naprogramovat pohyb celé čáry (`move_line`). Jednotlivé akce nejlépe objasní obrázek:



K řešení nám budou stačit tři úrovně podmínek. Nutné podmínky budou zajišťovat, že čára zůstane v horizontální poloze (`move_horiz_end`) resp. že čára nezmění svůj tvar (`move_line`). Podmínky úrovně *medium* svazují druhý koncový bod čáry s myší. V případě `move_horiz_end` bude jedna z nich „přebita“ nutnou podmínkou. Navíc zde budou slabé podmínky úrovně *weak*, které zajišťují setrvání bodu na svém místě. Weak podmínky tak dávají systému stabilitu a starají se o to, aby při drobném pohybu myší nedošlo třeba k několikanásobnému prodloužení čáry.

Implementace: Každá akce bude reprezentována jedním pravidlem se třemi argumenty - souřadnicemi čáry před akcí, po akci a změnou polohy myši.

```
move_end(line(OldX1,OldY1,OldX2,OldY2),
         line(NewX1,NewY1,NewX2,NewY2),
         delta(DX,DY)):-
     medium OldX2+DX=NewX2, medium OldY2+DY=NewY2,
     weak OldX1=NewX1, weak OldY1=NewY1,
     weak OldX2=NewX2, weak OldY2=NewY2.
```

```
move_horiz_end(line(OldX1,OldY1,OldX2,OldY2),
               line(NewX1,NewY1,NewX2,NewY2),
               Delta):-
     move_end(line(OldX1,OldY1,OldX2,OldY2),
               line(NewX1,NewY1,NewX2,NewY2),
               Delta),
     required OldY2=NewY2.

move_line(line(OldX1,OldY1,OldX2,OldY2),
          line(NewX1,NewY1,NewX2,NewY2),
          Delta):-
     move_end(line(OldX1,OldY1,OldX2,OldY2),
               line(NewX1,NewY1,NewX2,NewY2),
               Delta),
     required OldX2-OldX1=NewX2-NewX1,
     required OldY2-OldY1=NewY2-NewY1.
```

1.2.5 Mezi-hierarchické porovnání

V originální definici logického programování s hierarchiemi omezujících podmínek (HCLP) jsou porovnávána řešení dané hierachie podmínek a pouze ta „nejlepší“ jsou vracena jako odpověď. Nejsou ale srovnávána řešení vzešlá z výběru různých pravidel, což může někdy vést k neintuitivním výsledkům, jak ukazuje následující příklad.

Příklad 1.15: [WiBo93] (neintuitivní chování intra-hierarchického porovnání)

Nechť je dán následující HCLP program pro hledání vhodného času schůzky skupiny lidí:

```
find_times([Person|More],Start,End):-
     free(Person,StartFree,EndFree),
     medium StartFree≤Start,medium EndFree≥End,
     find_times(More,Start,End).
find_times([],Start,End).
```

a databáze faktů s volnými časy jednotlivých osob:

```
free(peter,8,12).
free(peter,18,21).
free(eve,17,21).
```

Na dotaz pro vyhledání vhodného času hodinového jednání

```
?-find_times([peter,eve],S,E),required E-S=1.
```

vrátí HCLP(R,WSMB) (použit weighted-sum-metric-better komparátor) dvě různé odpovědi. První odpověď, libovolný hodinový interval mezi 11. a 18. hodinou, je výsledkem použití prvního faktu `free(peter,8,12)`, zatímco druhá odpověď, libovolný hodinový interval mezi 18. a 21. hodinou, vzešla z druhého faktu. Ve skutečnosti byly při řešení zkonstruovány dvě hierarchie:

required	E-S=1	required	E-S=1
medium	8≤S	medium	18≤S
medium	17≤S	medium	17≤S
medium	E≤12	medium	E≤21
medium	E≤21	medium	E≤21

použitím různých pravidel pro Petrův volný čas, přičemž každá odpověď je příslušným řešením dané hiearchie. Je zřejmé, že druhá odpověď je očekávaným řešením dotazu, protože splňuje preference obou lidí, nicméně klasické HCLP vrátí postupně obě odpovědi.

Jedním ze způsobů, jak řešit popsané neintuitivní chování klasických HCLP systémů, je umožnit srovnání odpovědí vzešlých z jedné hierarchie s odpověďmi z hierarchie druhé. Porovnávat se samozřejmě musí to, jak odpověď vyhovuje „své“ hierarchii, protože ohodnocení z jedné hierarchie nemusí vůbec splňovat nutné podmínky jiné hierarchie. Tohoto tzv. *mezi-hierarchického* (*inter-hierarchy*) porovnání lze celkem jednoduše dosáhnout rozšířením definice 1.1 řešení hierarchie omezujících podmínek na definici 1.7 řešení množiny hierarchií.

Řešením množiny Δ hierarchií omezujících podmínek je množina ohodnocení všech volných proměnných v Δ. V HCLP se normálně množina Δ skládá z jednotlivých hierarchií, které vzniknou volbou různých pravidel v programu. Poznamenejme, že pokud množina Δ obsahuje pouze jedinou hierarchii, je definice 1.7 řešení s mezi-hierarchickým srovnáním stejná jako klasická definice 1.1 řešení hierarchie omezujících podmínek.

Nejprve je definována množina $S_{0\Delta}$ ohodnocení, která splňují všechny nutné podmínky některé z hierarchií v Δ. Každé ohodnocení v $S_{0\Delta}$ je indexováno příslušnou hierarchií, kterou splňuje. Použitím $S_{0\Delta}$ je potom definována množina S_Δ podobně jako v klasickém případě pouze s tím rozdílem, že komparátor better je parametrizován množinou hierarchií Δ.

Definice 1.7: (řešení množiny hierarchií)

množina potenciálních ohodnocení množiny hierarchií Δ:

$$S_{0\Delta} = \{ \theta_H \mid H \in \Delta \ \& \ \forall c \in H_0 \ \ c\theta_H \text{ platí} \}$$

množina řešení množiny hierarchií Δ:

$$S_\Delta = \{ \theta_H \mid \theta_H \in S_{0\Delta} \ \& \ \forall \sigma_J \in S_{0\Delta} \ \neg \ \text{better}(\sigma_J, \theta_H, \Delta) \},$$

kde $c\theta_H$ znamená výsledek aplikace ohodnocení θ_H na podmínku c a H_0 jsou nutné podmínky hierarchie H.

Podobně jako v definici řešení jedné hierarchie podmínek je i v tomto případě požadováno, aby komparátor better byl irreflexivní, tranzitivní a respektoval množinu hierarchií omezujících podmínek.

Definice 1.8: (respektování množiny hierarchií)

Říkáme, že komparátor *better* respektuje množinu hierarchií, právě když splňuje následující implikaci:

pokud existuje nějaké ohodnocení v $S_{0\Delta}$, které splňuje všechny podmínky příslušné hierarchie až do nějaké úrovně *k*, potom také všechna ohodnocení z řešení S_Δ musí splňovat všechny podmínky ze „svých“ hierarchií až do úrovně *k*, tj.:

$$\text{pokud } \exists \theta_H \in S_{0\Delta} \ \exists k>0 \ \forall i \in \{1,..,k\} \ \forall c \in H_i \ \ c\theta_H \text{ platí}$$

$$\text{potom také } \ \forall \sigma_J \in S_\Delta \ \forall i \in \{1,..,k\} \ \forall c \in J_i \ \ c\sigma_J \text{ platí},$$

kde H_i resp. J_i jsou jednotlivé úrovně příslušných hierarchií H resp. J.

Jinými slovy, pokud nějaké ohodnocení z $S_{0\Delta}$ splňuje všechny podmínky své hierarchie až do nějaké úrovně *k*, potom pro každé ohodnocení $\sigma_J \in S_{0\Delta}$, které nesplňuje některou z podmínek na úrovních 1,…,k „své“ hierarchie J, existuje ohodnocení $\theta_H \in S_{0\Delta}$, které je lepší, tj. platí better(θ_H,σ_J,Δ). Tím je zajištěno, že ohodnocení σ_J nepadne do množiny řešení S_Δ.

Protože lokální i regionální komparátory uvažují každou podmínku v hierarchii samostatně, je zřejmé, že obdoby těchto komparátorů nelze pro mezi-hierarchické porovnání definovat (snad s jednou vyjímkou, viz. kapitola 1.3.2). Na druhou stranu globální komparátory se s mezi-hierarchickým porovnáním vyrovnají celkem snadno.

Definice 1.9: (globální komparátor pro mezi-hierarchické porovnání)

Říkáme, že ohodnocení θ_H je globálně lepší (globally-better) než jiné ohodnocení σ_J, pokud je kombinovaná chyba na každé úrovni až do nějaké úrovně *k-1* stejná po aplikaci θ_H na podmínky hierarchie *H* jako po aplikaci σ_J na podmínky hierarchie *J* a na úrovni *k* je ostře menší.

Formálně:

$$\text{globally-better}(\theta_H,\sigma_J,\Delta,g) \equiv \exists k>0\ \forall i\in\{1,\dots,k-1\}\ g(\theta_H,H_i)=g(\sigma_J,J_i)\ \&\ g(\theta_H,H_k)<g(\sigma_J,J_k),$$

kde H_i resp. J_i jsou jednotlivé úrovně příslušných hierarchií H resp. J.

Důkaz korektnosti definice 1.9 globálních komparátorů pro mezi-hierarchické porovnání je stejný jako v případě intra-hierarchického porovnání (Tvrzení 1.3). Kombinační funkce definované pro jednoduché hierachie se také nemění a jejich použitím můžeme získat různé varianty globally-better komparátoru pro mezi-hierarchické porovnání.

Existuje mnoho příkladů, kdy mezi-hierarchické porovnání vede k intuitivnějším výsledkům než porovnání v rámci jedné hierarchie (intra-hierarchy). Jsou zde ale také důvody, proč zůstat u jednoduchých hierarchií. Kromě toho, že pro mezi-hierarchické porovnání lze používat pouze globální komparátory, je zde především důvod efektivity. Pro nalezení řešení HCLP programu s mezi-hierarchickým porovnáním je potřeba projít všechny „větve“ běhu programu. Protože těchto větví může být potenciálně nekonečně mnoho, může i hierarchií v množině Δ být nekonečně mnoho. V praxi to snadno nastane, pokud jsou v HCLP programu rekurzivní pravidla jako v následujícím příkladu.

Příklad 1.16: [WiBo89] (nekonečný běh)

```
f(X):-g(X), prefer X<0.
g(1).
g(X):-g(X-1).
```

Řešení tohoto nekonečného prohledávání v sobě potenciálně zahrnuje řešení problému zastavení (halting-problem), a nekonečnému prohledávání proto nelze obecně zabránit. Problém s generováním nekonečně velké hierarchie má v sobě ovšem také klasické HCLP s porovnáváním v rámci jedné hierarchie.

Jiným důvodem pro použití jednoduchých hierarchií může být požadavek na vrácení všech řešení jednotlivých hierarchií.

1.2.6 Vlastnosti komparátorů

Jednoduchý systém řešení hierarchie omezujících podmínek, jehož operační sémantika byl nastíněna v kapitole 1.2.4 o HCLP a jehož příklady jsou v části věnované systémům řešení hierarchií (jednoduchý interpret a DeltaStar) a v Příloze B, dostane vždy jako vstup celou hierarchii, kterou řeší. V mnoha praktických aplikacích jako jsou třeba interaktivní grafická prostředí, je ale vhodnější řešit hierarchii postupně tak, jak jsou přidávány resp. ubírány omezující podmínky. Tento způsob řešení se nazývá inkrementální a spočívá v úpravě dosud nalezeného řešení na řešení nové hierarchie vzniklé přidáním resp. ubráním nějaké podmínky.

Bohužel, jak uvidíme dále, komparátory, které respektují hierarchii, tento inkrementální způsob řešení trochu komplikují. Konkrétně přidání další podmínky k hierarchii může vést k řešení, které není zjemněním předchozího řešení, ale řešením zcela novým. To ztěžuje vyvinutí efektivního obecného a zároveň inkrementálního systému řešení hierarchie omezujících podmínek, protože ten po přidání nové podmínky bude v některých případech nucen zcela přepracovat řešení.

Definice 1.10: [WiBo89] (spořádanost komparátoru)

Nechť H a J jsou hierarchie podmínek a nechť C je komparátor. Potom říkáme, že komparátor C je *spořádaný* (*orderly*) právě tehdy, když:

$\forall H \ \forall J \ \ S_{H \cup J}(C) \subseteq S_H(C)$,

kde $S_H(C)$ a $S_{H \cup J}(C)$ jsou řešení příslušných hierarchií H resp. $H \cup J$ při zvoleném komparátoru C. Komparátor, který není spořádaný, je *nespořádaný* (*disorderly*).

Poznamejme, že spořádanost komparátoru můžeme přirozeně rozšířit i na mezi-hierarchické porovnání.

Tvrzení 1.7: [WiBo89] (nespořádanost komparátoru)

Nechť D je netriviální doména, tj. doména obsahující alespoň dva prvky. Potom každý komparátor, který respektuje hierarchii, je nespořádaný.

Důkaz:

Položme H={weak X=a} a J={strong X=b}, kde a, b jsou dva různé prvky domény D. Nechť C je dále libovolný komparátor respektující hierarchii.

Potom zřejmě $S_H=\{\{X/a\}\}$ a $S_{H \cup J}=\{\{X/b\}\}$, a tedy $S_{H \cup J} \not\subset S_H$. Komparátor C proto není spořádaný.

❏

Vzhledem k tomu, že k úplnému přepracování řešení nedochází vždy, existuje řada inkrementálních systémů řešení hierarchií podmínek, z nichž některé jsou představeny dále. Tyto systémy využívají toho, že při přidání podmínky, která není ve sporu se současným řešením, resp. ubráním podmínky, kterou současné řešení nesplňuje, nedochází k drastickému přepracování současného řešení. Proto jsou inkrementální systémy v mnoha praktických aplikacích velice efektivní.

Jedním ze způsobů, jak překonat vlastnost nespořádanosti komparátoru, je zavedení zvláštního tzv. commit operátoru, tedy jakési obdoby řezu z PROLOGu. „Přechod“ tohoto operátoru pro systém znamená pokyn, aby vyřešil dosud nashromážděnou hierarchii a její řešení považoval v dalším průběhu výpočtu za nutné podmínky na použité proměnné. Přidání dalších podmínek potom může dojít k zjemnění nalezeného řešení, nelze ho ale učinit neplatným.

Poznamejme také, že systémy CLP nemají problémy s nespořádaností, protože všechny podmínky v nich musí být splněny. Přidání podmínky nekompatibilní se současným řešením zde znamená, že řešení neexistuje.

Dosud jsme se zabývali tím, jak se řešení hierarchie podmínek chová po přidání dalších podmínek. V HCLP s mezi-hierarchickým porovnáním, je navíc možné přidávat další pravidla (klauzule), tj. další hierarchie, což opět vede ke změně řešení. To nastoluje otázku monotonie komparátoru.

Klasická logika je monotónní v tom smyslu, že přidáním nových axiomů k teorii může dojít pouze k zvětšení množiny tvrzení, která lze odvodit. Nikdy není potřeba vyřadit staré závěry s příchodem nových znalostí. Naproti tomu u nemonotónních logik může po přidání axiomů dojít k tomu, že dosud odvozená tvrzení přestávají platit. Podobně se také chovají HCLP programy s mezi-hierarchickým porovnáním, jak to ukazuje následující příklad.

Příklad 1.17: [WiBo89] (nemonotonie komparátorů)

Nechť máme následující triviální HCLP program nad doménou reálných čísel:

```
f(X):-g(X), prefer X>0.
g(-1).
```

Na dotaz `?-f(A)` vrátí interpret HCLP odpověď `A=-1`.

Předpokládejme nyní, že k programu přidáme fakt `g(1)`. Na stejný dotaz nyní vrátí interpret HCLP při použití komparátoru LPB, tj. při intra-hierarchickém porovnání, obě odpovědi `A=-1` a `A=1`. Pokud ale použijeme mezi-hierarchické porovnání dostaneme odpověď jedinou a tou je `A=1`. Je vidět, že při mezi-hierarchickém porovnání není stará množina odpovědí podmožinou odpovědí odvozených z programu po přidání nového pravidla.

Formálně pojem monotonie komparátoru zavádí následující definice.

Definice 1.11: [WiBo89] (monotonie komparátoru)

Nechť Δ a Γ jsou množiny hierarchií podmínek a C je komparátor. Potom říkáme, že komparátor C je *monotónní* právě tehdy, když:

$\forall\Delta\ \forall\Gamma\ \ S_\Delta(C)\subseteq S_{\Delta\cup\Gamma}(C)$, kde

$S_\Delta(C)$ a $S_{\Delta\cup\Gamma}(C)$ znamenají řešení příslušných množin hierarchií Δ resp. $\Delta\cup\Gamma$ při zvoleném komparátoru C. Komparátor, který není monotónní, je *nemonotónní.*

K definici monotonie komparátoru poznamejme, že ji nelze aplikovat na lokální komparátory, protože pro ně není $S_{\Delta\cup\Gamma}(C)$ definováno.

Tvrzení 1.8: [WiBo89] (nemonotonie komparátoru)

Nechť D je netriviální doména, tj. doména obsahující alespoň dva prvky. Potom každý komparátor, který respektuje množinu hierarchií, je nemonotónní.

Důkaz:

Položme Δ={{required X=a, weak X=b}} a Γ={{required X=b}} (tj. každá množina hierarchií obsahuje právě jednu hierarchii), kde a, b jsou dva různé prvky domény D. Nechť C je dále libovolný komparátor respektující množinu hierarchií. Potom zřejmě $S_\Delta(C)$={{X/a}} a $S_\Gamma(C)$={{X/b}}, a protože ohodnocení z $S_\Gamma(C)$ splňuje všechny podmínky své hierarchie zatímco ohodnocení z $S_\Delta(C)$ ne, platí také $S_{\Delta\cup\Gamma}(C)$={{X/b}}. Je tedy zřejmé, že $S_\Delta(C)\not\subset S_{\Delta\cup\Gamma}(C)$, a tedy komparátor C není monotónní.

❑

Vlastnost nemonotonie komparátorů nám v budoucnu umožní u expertních systémů založených na hierarchiích zamítnutí staré odpovědi přidáním nového pravidla, tj. nové znalosti. Hierarchie podmínek podobně jako nemonotónní logiky také poskytují řešení známého problému rámce (frame problem).

1.2.7 Rozšíření klasické teorie hierarchií omezujících podmínek

Klasická teorie hierarchií omezujících podmínek a všechny současné systémy řešení hierarchií předpokládají lineární uspořádání preferencí. V závěru práce [WiBo93] je také zmíněna možnost zobecnění na částečně uspořádané množiny preferencí. Potom je možné mít hierarchie s úrovněmi A, B a C, kde úrovně A a B jsou preferovány nad úrovní C, ale mezi úrovněmi A a B není žádný preferenční vztah. Praktické využití tohoto zobecnění ani jeho teoretické základy nebyly dosud dostatečně zpracovány.

Z uživatelského hlediska se praktičtější jeví možnost dynamického rozšiřování počtu preferenčních úrovní. Jedná se o možnost vložit do hierarchie další prázdnou úroveň, do které jsou později přidávány podmínky. To je umožněno tím, že systému, který používá symbolické názvy preferencí, stačí pouze znalost jejich uspořádání a

nepotřebuje znát, zda *strong* odpovídá úrovni 1 nebo zda odpovídá úrovni 3 a úrovně 1 a 2 jsou prázdné.

Při vkládání úrovní, které bylo zmíněno v předchozím odstavci, je ale třeba dát pozor na to, aby komparátor nepoužívat pro srovnání podmínek na různých úrovních různé principy. Podmínky na úrovni *strong* je pak možné srovnávat třeba weighted-sum-better komparátorem, zatímco pro úroveň *weak* se použije locally-better komparátor. Tuto možnost nastiňuje rozšíření hierarchií podmínek v pracích [HMT+94] a [HMY96].

Zajímavé a v oblasti CLP ([Frü93], [FrBr95]) již zkoumané jsou uživatelem definované podmínky. Ty dávají možnost na uživatelské úrovni definovat vlastní systém řešení podmínek.

1.3 Alternativní přístupy

Dosud prezentovaná teorie hierarchií omezujících podmínek není přirozeně jediným přístupem k této problematice. Patří ale k nejvíce rozšířeným, a tak jí byl věnován náležitý prostor. V následující části ji z téhož důvodu budeme nazývat klasická teorie. Stručně představeny zde budou dva další přístupy k hierarchiím omezujících podmínek a jeden přístup, který hierarchie zcela obchází. První z „hierachických" přístupů, teorie před-měr a před-řešení, se od klasické teorie nějak dramaticky neliší, jedná se vlastně o trochu jiné zpracování stejných myšlenek. Druhý přístup, kompozicionální teorie, se zaměřuje na jednu z vlastností hierarchií omezujících podmínek a tou je nemonotonie. Přehled alternativních přístupů ale začneme u podmínek vyšších řádů, které se pojmu hierarchie úplně vyhnuly.

1.3.1 Podmínky vyšších řádů

Kromě zavedení hierarchie omezujících podmínek existuje také jiný způsob překonání problémů s příliš omezenými systémy a tím je práce s podmínkami vyššího řádu. Místo jednoduché podmínky je zde možné pracovat s konjunkcí nebo s disjunkcí podmínek. Zvláště disjunkce je v tomto ohledu zajímavá, protože pokud potom chceme vyjádřit, že podmínka c nemusí být nutně splněna, stačí ji nahradit disjunkcí této podmínky s podmínkou, která je splněna vždy, (tj. $c \vee true$). Tím se dosáhne toho, že pokud systém není schopen splnit podmínku c, vezme její alternativu, a protože ta je splněna vždy, je i celá disjunkce vždy splnitelná. Tímto způsobem lze do jisté míry nahradit hierarchie omezujících podmínek, částečně se tím ale ztrácí deklarativní charakter podmínek a záleží pak na konkrétním systému, jak danou soustavu podmínek vyřeší (např. při řešení $a \vee b$ zkus nejprve splnit a a v případě neúspěchu zkus b). Výhodou může být snazší zakomponování do stávajících jazyků typu PROLOG s omezujícími podmínkami. Některé systémy (viz. IHCS [MeBa95], kapitola 1.5.9) používají podobný přístup pro simulaci mezi-hierarchického porovnání.

1.3.2 Před-řešení a před-míry

Následující přístup k hierarchiím omezujících podmínek vypracovali Michael Maher a Peter Stuckey [MaSt89]. Místo množiny S_0 všech ohodnocení splňujících nutné omezující podmínky využívá jejich teorie tzv. *před-řešení* (*pre-solutions*) pro jednotlivé hierarchie. Před-řešení není nic jiného než přiřazení hodnot proměnným tak, že jsou v dané hierarchii splněny všechny nutné (required) omezující podmínky. Ve druhé fázi jsou před-řešení pomocí funkce g nazývané *před-míra* (*pre-measure*) zobrazena do zvolené škály, ve které jsou navzájem porovnatelná.

Definice 1.12: (srovnání předřešení)

Říkáme, že před-řešení α je *lepší* než před-řešení β, pokud platí:

$$(g(\alpha,H_1),\dots,g(\alpha,H_n)) \geq (g(\beta,H_1),\dots,g(\beta,H_n))$$

při lexikografickém uspořádání na škále $S \times \dots \times S$. Množiny omezujících podmínek $H_1,\dots,H_n$ jsou takové podmnožiny hierarchie H, že v H_i jsou všechny omezující podmínky z úrovně *i* hierarchie H s odstraněným symbolem preference. Před-míru tvoří funkce g: $\Phi \times P(C) \rightarrow S$, kde Φ je množina všech před-řešení, P(C) je množina všech podmnožin množiny C všech omezujících podmínek a *S* je množina zvolená za škálu.

Použitím různých škál a před-měr lze, podobně jako v klasickém přístupu, získat různé komparátory. Například známý komparátor *locally-predicate-better* (LPB) dostaneme tak, že za škálu *S* zvolíme množinu všech podmnožin omezujících podmínek s přirozeným uspořádáním inkluzí (S=P(C), kde P(C) je množina všech podmnožin množiny C všech omezujících podmínek). Před-míra je tedy v tomto případě definována jako zobrazení g: $\Phi \times P(C) \rightarrow P(C)$, které z dané množiny omezujících podmínek vybere pouze podmínky splněné při daném ohodnocení:

$$g(\theta,H_i)=\{c \mid c\in H_i \ \&\ c\theta \text{ platí}\}.$$

Pokud škály a před-míru trochu upravíme a použijeme v nich chybovou funkci *e*, dostaneme jiný druh lokálního komparátoru nazývaný *locally-error-better* (LEB). Pro připomenutí, locally-error-better komparátor je lokální komparátor používající netriviální chybovou funkci *e*. Škála je v tomto případě definována jako množina všech množin dvojic $[c,v]$, kde *c* je omezující podmínka a *v* je reálné číslo (S=P(C × R), kde P(C × R) je množina všech podmnožin množiny $C \times R=\{[c,v] \mid c\in C \ \&\ v\in R\}$, C je množina všech omezujících podmínek a R je množina všech reálných čísel). Uspořádání na škále *S* je potom definováno následujícím způsobem:

množina *s* ze škály *S* je *větší nebo rovna* než množina *s*' ze škály *S*, právě tehdy když *s* je nadmnožinou množiny *s*' a hodnoty *v* jednotlivých prvků z množiny *s* nejsou větší (tj. jsou menší nebo rovny) než hodnoty *v* odpovídajících prvků v *s*' (viz. příklad 1.18).

Příklad 1.18: (porovnání škál pomocí LEB)

{[x=2; 0,3], [x+y=5; 0]}≥{[x=2; 0,4], [x+y=5; 0]}
{[x=2; 0,3], [x+y=5; 0]}≥{[x=2; 0,3]}

Před-míra je potom pro LEB definována jako zobrazení g: $\Phi \times P(C) \rightarrow P(C \times R)$ takto:

$$g(\theta,H_i)=\{[c,v] \mid c\in H_i \ \&\ v=e(c\theta)\}.$$

Toto zobrazení vlastně nedělá nic jiného, než že dané omezující podmínce *c* přiřadí hodnotu *v* určující míru splnění resp. nesplnění podmínky při ohodnocení θ (pro připomenutí v=0 právě když $c\theta$ je splněno).

Globální komparátory z klasické teorie se do teorie před-řešení a před-měr přenesou přirozeně tak, že za škálu *S* zvolíme množinu reálných čísel a před-míra *g* je definována jako zobrazení g:$\Phi \times P(C) \rightarrow R$ následujícím způsobem:

$$g(\theta,H_i) = 0 - k(\theta,H_i),$$

kde *k* je příslušná kombinační funkce globálního komparátoru z klasické teorie (tam se ale značila *g*, viz. kapitola 1.2.2). Například před-míra pro *worst-case-better* komparátor potom vypadá takto:

$$g(\theta,H_i) = 0 - \max\{w_c * e(c\theta) \mid c\in H_i\},$$

kde w_c je váha podmínky *c*.

Teorii před-řešení a před-měr je možné snadno rozšířit tak, aby umožňovala *mezi-hierarchické* porovnání, tedy srovnání před-řešení různých hierarchií.

Definice 1.13: (mezi-hierarchické srovnání předřešení)

Nechť α je před-řešení hierachie H a β je před-řešení hierarchie J a nechť n je pořadové číslo maximální úrovně v H a J. Potom říkáme, že před-řešení α je *lepší* než před-řešení β, pokud platí

$$(g(\alpha,H_1),\ldots,g(\alpha,H_n)) \geq (g(\beta,J_1),\ldots,g(\beta,J_n))$$

při lexikografickém uspořádání na škále $S \times \ldots \times S$. Množiny omezujících podmínek $H_1,\ldots,H_n$ resp. $J_1,\ldots,J_n$ jsou příslušnými „vrstvami“ hierarchií H resp. J takové, že pokud v dané hierarchii úroveň i neexistuje, je množina H_i resp. J_i prázdná.

Při této definici lze pro mezi-hierarchické porovnání používat nejen globální ale také lokální komparátory. To je zajímavé z toho důvodu, že klasická teorie definici lokálních komparátorů pro mezi-hierarchické porovnání vylučuje. Na druhou stranu použití lokálních komparátotů při mezi-hierarchickém porovnání může vést k neintuitivním výsledkům a navíc, vzhledem k jejich charakteru, bude často docházet k tomu, že dvě před-řešení z různých hierarchií budou nesrovnatelná, tj. stejně dobrá. Toto anomální chování podporuje domněnku, že lokální komparátory by se měly používat pouze pro porovnávání řešení v rámci jedné hierarchie.

Následující příklad je ukázkou neintuitivního chování locally-predicate-better komparátoru.

Příklad 1.19: (neintuitivní chování LPB)

```
f(X):-g(X), prefer X≥5.
g(X):-required X≥0.
g(X):-prefer X=6.
```

Při dotazu `?-f(A)` a použití komparátoru LPB dostaneme pouze odpověď A=6 splňující všechny podmínky hierarchie {prefer A≥5, prefer A=6}, přestože také odpověď A≥5 splňuje všechny podmínky „své“ hierarchie {required A≥0, prefer A≥5}.

1.3.3 Kompozicionální teorie

Žádanou vlastností formálních systémů pro řešení problémů je jejich kompozicionalita. Kompozicionalita je, neformálně řečeno, vlastnost, která umožňuje z řešení podproblémů snadno složit řešení problému vzniklého spojením těchto podproblémů. Kompozicionalita tak zajišťuje, že systém lze efektivně implementovat. Klasické logické programování i logické programování s omezujícími podmínkami (CLP) jsou v tomto smyslu kompozicionální, zatímco systémy s hierachiemi podmínek kompozicionální nejsou (viz. kapitola 1.2.6 o vlastnostech komparátorů).

Michael Jampel [Jam95a] proto navrhl jiný přístup k hierarchiím podmínek, který umožňuje řešení hierarchie rozdělit na kompozicionální a nekompozicionální část. Kompozicionální část lze potom efektivně inkrementálně implementovat, zatímco část nekompozicionální slouží k „usměrnění“ řešení tak, aby výsledek odpovídal klasickému HCLP.

V kompozicionální části nazývané *BCH* (*Bags for Composition of Hierarchies*) se využívá teorie multimnožin (multiset, bag). Multimnožiny jsou struktury podobné klasickým množinám s tím rozdílem, že v multimnožině může mít prvek vícenásobný výskyt. Podobně jako jsou množiny jednoznačně charakterizovány svými členskými (membership) funkcemi, má také každá multimnožina jednoznačný popis členskou funkcí. Každé multimnožině X tak jednoznačně odpovídá jediná členské funkce, označme

ji #X: $D \rightarrow N_0^+$ (D - doména, nad kterou děláme multimnožiny; N_0^+-množina přirozených čísel včetně nuly), určující počet výskytů daného prvku v multimnožině X. Pomocí členských funkcí potom můžeme definovat klasické operace sjednocení $\cup$, průniku $\cap$ a dále třeba operaci aditivního sjednocení $\cup^+$. Nově je zaveden binární operátor stráže (guard) *[]*:

Definice 1.14: [Jam95a] (základní operace na multimnožinách)

$e\#(A \cup B) =$	$\max(e\#A, e\#B)$	(sjednocení)
$e\#(A \cap B) =$	$\min(e\#A, e\#B)$	(průnik)
$e\#(A \cup^+ B) =$	$e\#A + e\#B$	(aditivní sjednocení)
$e\#(A\ []\ B) =$	$e\#A$, pokud $e\#B>0$	(stráž; říkáme, že A je strážené B)
$=$	0, pokud $e\#B=0$,	

kde *e#X* znamená počet výskytů prvku *e* v multimnožině *X*.

Příklad 1.20: [Jam95a] (operace na multimnožinách)

$\{a,a\} \cup^+ \{a,b\}=\{a,a,a,b\}$

$\{a,a\} \cup \{a,b\}=\{a,a,b\}$

$\{a,a\} \cap \{a,b\}=\{a\}$

$\{a,a,c\}[]\{a,b\}=\{a,a\}$

$\{a,a,c\}[]\{b\}=\{\}$

V BCH se pracuje s multimnožinami definovanými nad doménou všech možných ohodnocení. Řešení hierachie H je potom definováno jako n-tice $[S_0(H),S_1(H),\ldots,S_n(H)]$, kde $S_0(H)$ je multimnožina řešení nutných podmínek, tj. ohodnocení splňujících všechny nutné podmínky, a $S_i(H)$ pro i>0 jsou příslušná řešení podmínek na úrovni *i* strážená multimnožinou $S_0(H)$. *n* je počet preferenčních úrovní hierarchie H. Multimnožina $S_0(H)$ se získá použitím klasického CLP a můžeme se na ni také dívat jako na průnik řešení jednotlivých nutných podmínek. Multimnožiny $S_i(H)$ pro i>0 zase vzniknou aditivním sjednocením řešení jednotlivých podmínek dané úrovně a následnou aplikací stráže. Stráž tak vyřadí ta řešení, která nejsou kompatibilní s řešením nutných podmínek. Můžeme proto říci, že BCH respektuje nutné podmínky, ale protože například v $S_2(H)$ mohou být řešení, která nejsou v $S_1(H)$, nerespektuje BCH hierarchii podmínek tak, jak jsme to definovali dříve (kapitola 1.2.2).

Nyní je možné definovat asociativní a komutativní operátor kompozice $\bullet$ pro skládání řešení hierarchií následujícím způsobem:

$$S_0(\bullet H^i) =_{def} \cap_i S_0(H^i)$$

$$S_k(\bullet H^i) =_{def} (\cup^+_i S_k(H^i))[]\ S_0(\bullet H^i),\ k>0,$$

kde H^i jsou jednotlivé skládané hierarchie.

Příklad 1.21: (skládání hierarchií)

Nechť máme dány hierarchie H a H' nad reálnými čísly:

H: required	X=Y	H': required	X=2
strong	X=1		
weak	X=2	weak	Y=2
weak	Y=3,		

potom jednotlivé složky řešení v rámci BCH vypadají takto:

$S_0(H) = \{\{X/x,Y/x\} \mid x\in R\}$	$S_0(H') = \{\{X/2,Y/x\} \mid x\in R\}$
$S_1(H) = \{\{X/1,Y/1\}\}$	$S_1(H') = \{\}$
$S_2(H) = \{\{X/2,Y/2\},\{X/3,Y/3\}\}$	$S_2(H') = \{\{X/2,Y/2\}\}$.

Pro ilustraci, multimnožina $S_2(H)$ vznikne aditivním sjednocením řešení podmínky *weak X=2*, tj. $\{\{X/2,Y/y\} \mid y\in R\}$, s řešením podmínky *weak Y=3*, tj. $\{\{X/x,Y/3\} \mid x\in R\}$ a následnou aplikací stráže vzhledem k $S_0(H) = \{\{X/x,Y/x\} \mid x\in R\}$:

$$S_2(H) = (\{\{X/2,Y/y\} \mid y\in R\} \cup^+ \{\{X/x,Y/3\} \mid x\in R\})\ [\,]\ \{\{X/x,Y/x\} \mid x\in R\}$$

Řešení složené hierarchie H•H':

$S_0(H\bullet H') = \{\{X/2,Y/2\}\}$,

$S_1(H\bullet H') = \{\}$,

$S_2(H\bullet H') = \{\{X/2,Y/2\}\{X/2,Y/2\}\}$.

Vztah řešení získaných pomocí BCH k řešením z HCLP lze neformálně zapsat následovně [Jam95a]:

$$HCLP \subseteq BCH,$$

BCH tedy vrací řešení obsahující všechna řešení získaná HCLP.

Druhou částí kompozicionální teorie hierarchií je její nekompozicionální složka nazývaná *FGH* (*Filters, Guards and Hierarchies*). Ta má za úkol vzít výsledky BCH a použít je k nalezení řešení ekvivalentního tomu, které by dalo HCLP. FGH k tomu používá tzv. filtrů *f*, které jsou vlastně funkcemi z multimnožin ohodnocení do multimnožin ohodnocení. Filtry zde představují obdobu komparátorů z HCLP. Cílem je tedy dosáhnou následující rovnosti:

$$HCLP_{H\cup H'} = F(H\bullet H'),$$

kde *F* je filtr indukovaný funkcí *f* následujícím způsobem (poznamejme, že se používá jak na n-tice, tak na jejich prvky):

$$F([S_0(H),S_1(H),\ldots,S_n(H)]) =_{def} [F(S_0(H)),F(S_1(H)),\ldots,F(S_n(H))]$$
$$F(S_0(H)) =_{def} S_0(H)$$
$$F(S_i(H)) =_{def} f(S_i(H)\ [\,]\ F(S_{i-1}(H)))\ \text{pro}\ i>0.$$

Příkladem filtru může být funkce f_{max}, která je v FGH obdobou známého locally-predicate-better komparátoru. Filtr f_{max} vybírá z multimnožiny jen ty prvky, jejichž počet výskytů je v rámci multimnožiny maximální. Opět ho můžeme definovat pomocí členské funkce:

$$e\#f_{max}(A) = e\#A \quad \text{pokud } e\#A=\max\{e'\#A \mid e'\in A\}$$
$$= 0 \quad \text{v ostatních případech.}$$

Příklad 1.22: [Jam95a] (filtr f_{max})

$f_{max}(\{a,a,b\}) = \{a,a\}$

$f_{max}(\{a,b\}) = \{a,b\}$

Vztah mezi HCLP, BCH a FGH nejlépe ozřejmí následující příklad ukazující řešení tří hierarchií a jejich sjednocení/složení za použití locally-predicate-better komparátoru resp. filtru f_{max}.

Příklad 1.23: [Jam95a] (srovnání HCLP, BCH, FGH)

Nechť máme dány tři hierarchie podmínek P, Q, R nad reálnými čísly:

	P	Q	R
required	X>0	0<X<20	0<X<23
strong	X<10	X>5	X>15
weak	X=4	X mod 7 =0	X mod 8=0

HCLP dá u jednotlivých hierarchií následující řešení (bráno po úrovních). Řešení hierarchie tak, jak bylo definováno v kapitole 1.2.2, je vlastně v řádce S_2:

	P	Q	R
S_0	X>0	0<X<20	0<X<23
S_1	0<X<10	5<X<20	15<X<23
S_2	X=4	X=7, X=14	X=16

BCH dá při řešení jednotlivých hierarchií stejná řešení jako HCLP s výjimkou úrovně S_2 u hierarchie R, kam navíc přidá X=8.

Řešení spojení hierarchií P, Q a R použitím HCLP, BCH a FGH ukazuje následující tabulka:

	HCLP	BCH	FGH
S_0	0<X<20	0<X<20	0<X<20
S_1	5<X<10, 15<X<20	$0<X\leq 5$, $(5<X<10)^2$, $10\leq X\leq 15$, $(15<X<20)^2$	$(5<X<10)^2$, $(15<X<20)^2$
S_2	X=7, X=8, X=16	X=4, X=7, X=8, X=14, X=16	X=7, X=8, X=16

Podrobně je kompozicionální teorie hierarchií podmínek včetně rozboru složitosti objasněna v pracích [Jam95a] a [Jam95b]. Její další zobecnění integrující dva současné přístupy k příliš omezeným systémům, HCLP a PCSP (Partial Constraint Satisfaction Problems), je popsáno v pracích [JJG96] a [JJGH96].

1.4 Omezující podmínky a imperativní programování

Cílem při zavádění omezujících podmínek a vlastně i hierarchií podmínek byla snaha o deklarativní zápis většího množství problémů. V typické aplikaci se ale jen některé části zadávají dobře pomocí omezujících podmínek, zatímco jiné části se lépe popisují použitím imperativních technik. To vedlo ke vzniku nového rámce, ve kterém se tato dvě paradigmata, tedy imperativní programování a programování s omezujícími podmínkami, kombinují. Vzniká tak imperativní programování s omezujícími podmínkami (CIP-Constraint Imperative Programming), jehož typickým představitelem je objektově orientovaný jazyk Kaleidoscope ([FBB92b], [LFBB94b]).

Kombinace omezujících podmínek a imperativních technik s sebou přináší řadu problémů. Zatímco v imperativním programu má nad změnou hodnoty proměnné plnou kontrolu uživatel, v systému s omezujícími podmínkami určuje hodnoty proměnných systém řešení podmínek a uživatel může tyto hodnoty ovlivňovat pouze přidáváním resp. ubíráním podmínek. Problémy tak činí i jednoduché přiřazovací příkazy typu `x:=x+1`, které po přímém přepsaní na omezující podmínku $x=x+1$ vedou k neřešitelnému systému podmínek.

Řešením se ukázalo zavedení pojmu času do běhu programu. Místo současného modelu ukládání proměnných charakterizovaného funkcí *pozice (v paměti)→hodnota* se zavádí nový model, který lze popsat funkcí *pozice×čas→hodnota*. Problematický přiřazovací příkaz můžeme nyní přepsat na podmínku $x_{t+1}=x_t+1$. Díky této konvenci je možné používat i přiřazení, která mají složitější výrazy na obou stranách jako je `b+2*c:=x+sin(y)`, které se transformuje na $b_{t+1}+2*c_{t+1}=x_t+sin(y_t)$.

V imperativním programování se předpokládá, že se hodnota proměnné nemění, pokud to není explicitně požadováno přiřazovacím příkazem. Toho se v CIP dosahuje použitím velmi slabých podmínek (i zde se pracuje s hierarchiemi podmínek), které zajišťují, že se hodnota proměnné s časem nemění: $\forall x\ \forall t$ *very_weak* $x_{t+1}=x_t$. Tyto velmi

slabé podmínky mohou být samozřejmě „přebity" silnějšími podmínkami reprezentujícími třeba přiřazovací příkaz.

Aby nedocházelo k různým paradoxům, kdy budoucnost modifikuje minulost, je potřeba u podmínek, kde se vyskytují proměnné z různých časů, zajistit „dopředný" tok v čase. Přiřazení `x:=x+1` tak vlastně přechází na podmínku $x_{t+1}=x_t?+1$, kde $x_t?$ označuje proměnnou určenou pouze pro čtení (read-only).

V klasickém programování s omezujícími podmínkami trvá platnost podmínky od jejího vstupu do systému až do konce běhu programu resp. do jejího explicitního vyřazení. V CIP se ale, alespoň na konceptuální úrovni, pracuje s oddělenými systémy podmínek pro každý čas. To by znamenalo opětovné explicitní zadávání podmínky, která má platit po celou dobu běhu programu. Aby se tomu zabránilo, pracuje se místo s konkrétní podmínkou se schématem podmínky. Zápis `always: 2*a=b-5` potom znamená $\forall t\ 2*a_t=b_t-5$, neboli podmínku, která platí po celou dobu programu. Podmínka platná pouze v jeden okamžik se zapisuje `once: c=40` a znamená třeba $c_7=40$. Je možné také zadávat podmínky, které platí jen určitý omezený časový úsek, tj. třeba po dobu provádění bloku programu nebo cyklu. Ty se zapisují například takto: *`condition`* `during` *`program_block`*.

Uživatelé moderních imperativních a zvláště pak objektově orientovaných jazyků jsou zvyklí na možnost definovat vlastní typy dat a operací nad těmito typy. V CIP to přináší problémy, protože uživatelem definované datové typy zpravidla neposkytují dostatek sémantické informace pro vytvoření systému pro řešení podmínek nad těmito typy. Jedním z řešení může být zavedení jakéhosi konstruktoru, který podmínky nad uživatelskými typy dat transformuje na podmínky nad základními typy dat.

Předchozí odstavce nastínili některé problémy, které se vyskytují v CIP včetně jejich řešení. Podrobnější informace lze nalézt v článcích [FBB92b] a [LFBB94b], kde je také popsán abstraktní stroj, tzv. K-machine, pro CIP jazyky. Obecné problematice integrace omezujících podmínek do objektově orientovaných jazyků jsou věnovány práce [FBB92a] a [LFBB94a].

1.5 Systémy řešení hierarchií omezujících podmínek

Pro řešení omezujících podmínek bylo vytvořeno a upraveno mnoho systémů, počínaje třeba simplexovým algoritmem až třeba k systému Newton [VHM95] pro intervalové řešení polynomiálních podmínek. Náplní této práce jsou ale hierarchie omezujících podmínek a tak je následující pasáž věnována pouze vybraným systémům řešení hierarchií podmínek. Začneme od „prehistorických" systémů, které ještě svým způsobem spoléhají na systémy CLP, pokračovat budete přes rychlé inkrementální algoritmy tzv. lokální propagace, které ovšem řeší omezené množiny podmínek, zastavíme se u algoritmu, který kombinuje různé další algoritmy pro řešení specifických podmínek abychom nakonec skončili u dvojice relativně obecných algoritmů.

1.5.1 Jednoduchý interpret pro HCLP

První interprert pro HCLP ([BMMW89], [WiBo93]) vznikl na University of Washington jako nadstavba stávajících systémů CLP. Cílem při jeho tvorbě byla snaha o ověření myšlenek hierarchií omezujících podmínek, otázky efektivity byly proto zatím odsunuty do pozadí. Tento jednoduchý interpret je založen na technikách metaprogramování a na využití podkladového systému CLP. V praxi se jednalo o interpret pro HCLP(R,LPB) využívající systému řešení omezujících podmínek CLP(R). Jak ale uvidíme dále, použití domény reálných čísel R není výlučné a aplikací stejných myšlenek lze libovolný systém CLP(D) rozšířit na HCLP(D,LPB). Na druhou stranu práce s locally-predicate-better

komparátorem byla bezpodmínečně vyžadována. Podívejme se ale konečně jak takový jednoduchý interpret pro HCLP vlastně vypadá. Jeho běh se skládá ze dvou fází.

V první fázi se používá meta-interpret, který se příliš neliší od jiných známých meta-interpretů [AbRo89, Bar93, Ste86, Ste88, StSh86]. Jeho úkolem je zredukovat zadaný cíl použitím pravidel z programu. Nutné (required/hard) omezující podmínky jsou při tom ihned předávány systému CLP(R) k vyřešení, zatímco měkké (soft) podmínky jsou shromažďovány pro další použití. Jakmile se podaří původní cíl zcela zredukovat, poznamenejme také, že pak jsou splněny všechny nutné podmínky, je nashromážděný zásobník měkkých podmínek předán jako hierarchie omezujících podmínek do druhé fáze.

Úkolem druhé fáze je potom nalézt všechna řešení hierachie omezujících podmínek vzniklé v první fázi a to použitím locally-predicate-better komparátoru. Algoritmus druhé fáze k tomu využívá rekurzivního volání procedury *Solve*, která jako vstup dostává zásobník *Untried* dosud nevyzkoušených podmínek, tj. hierarchii měkkých podmínek z první fáze, a dosud nalezenou odpověď, tj. v prvním kroku řešení nutných podmínek. Jako výstup vrací nalezené řešení/odpověď. Procedura *Solve* pracuje následujícím způsobem. Nechť s je nejsilnější preference v zásobníku *Untried*. Potom pro každou podmínku c ze zásobníku *Untried*, která má preferenci s, přidá procedura *Solve* řešení podmínky c k dosud nalezené odpovědi a ze zásobníku *Untried* dosud netestovaných podmínek vyřadí všechny podmínky, které se buď staly nesplnitelnými tím, že podmínka c platí, nebo jsou implikovány současnou odpovědí. Nakonec procedura *Solve* rekurzivně volá sebe sama na nově vzniklý zásobník *Untried* a vytvořenou částečnou odpověď. K řešení se dopracujeme tehdy, je-li hierarchie netestovaných podmínek prázdná. Opakovaným voláním procedury *Solve* lze použitím navracení (backtracking) získat také další řešení. Teprve po vyčerpání všech řešení dané hierarchie je řízení předáno zpět meta-interpretu z první fáze, který může hledat další řešení výběrem alternativních pravidel.

Výhodou takto pojatého interpretu je jeho snadné, rychlé a přehledné naprogramování použitím technik metainterpretace a samozřejmě také přímočaré využití podkladového systému CLP (viz. Příloha B). Bohužel i nevýhody pro praktické použití jsou zřejmé. Předně druhá fáze interpretu není inkrementální, což znamená, že vždy zcela od začátku počítá všechny LPB odpovědi místo toho, aby inkrementálně upravovala odpovědi podle toho, jak jsou nové podmínky přidávány nebo ubírány v průběhu výběru alternativních pravidel. Výsledkem je podstatná neefektivnost systému při hledání alternativních řešení.

Vzhledem k použití podkladového systému CLP také není možné používat metrické komparátory. Vlastně tak, jak byl systém resp. procedura *Solve* popsán, lze používat pouze locally-predicate-better komparátor. Mimo hru jsou tedy i všechny regionální a globální komparátory.

1.5.2 DeltaStar

Algoritmus popsaný v předchozí části podporuje pouze locally-predicate-better komparátor. Nicméně v aplikacích jako je interaktivní grafika nebo rozvrhování je v případě, že podmínka není splněna, vhodnější, aby chyba při nesplnění byla co nejmenší. Stejně tak je v těchto aplikacích často užitečné použití globálních komparátorů. Proto byl vytvořen další interpret HCLP ([WiBo93]), opět nad reálnými čísly, který podporuje weighted-sum-metric-better, worst-case-metric-better a locally-metric-better komparátory.

Důsledkem podpory metrických komparátorů je to, že systém nemůže být přímočaře postaven nad CLP(R), protože tentokrát záleží nejenom na tom, zda je podmínka splněna nebo ne, ale v případě nesplnění podmínky je nutné znát také chybu. Tuto informaci ovšem běžné systémy CLP(R) neposkytují.

Popisovaný interpret je založený na algoritmu *DeltaStar*, který vlastně představuje třídu algoritmů parametrizovaných podkladovým „plochým“ (tj. bez podpory hierarchií) systémem řešení podmínek. Tento řešič podmínek musí především poskytovat funkci filter

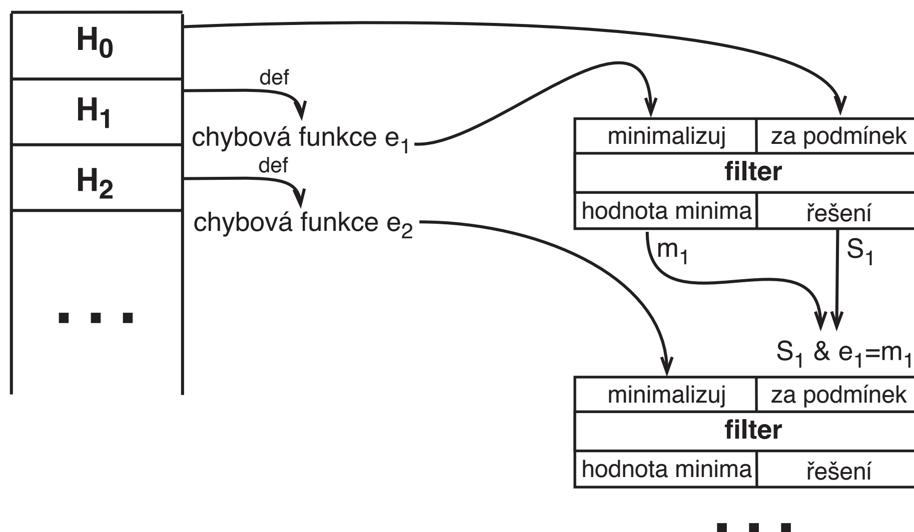
filter(S:Solution, C:Set of Constraints) → *Solution*,

která z daného řešení *S* vybere podmnožinu minimalizující chybu při plnění podmínek *C*. Implementace funkce *filter* tak zároveň definuje příslušný komparátor. Navíc musí podkladový systém poskytovat funkce pro efektivní určení, zda je nová podmínka kompatibilní se současným řešením, a pro rychlé přidání podmínky k danému řešení za předpokladu, že je s ním kompatibilní.

Prvním systémem využívajícím algoritmu DeltaStar byla jeho původní implemetace v jazyce Common Lisp na University of Washington, kde jako podkladový systém řešení podmínek sloužil klasický simplexový algoritmus. Řešení hierarchie podmínek spočívalo v následujícím postupu. Nejprve se minimalizovala chyba na úrovni H_1 vzhledem k podmínkám z úrovně H_0. K tomu stačilo vyjádřit chybu na úrovni H_1 formou funkce (tzv. objektivní funkce) podle zvoleného komparátoru a předat tuto funkci k minimalizaci simplexovému algoritmu spolu se soustavou podmínek z H_0. Jedná se tak vlastně o první volání funkce *filter*, kde jako parametr *S* je předána množina H_0 představující implicitní popis řešení nutných podmínek a jako parametr *C* množina H_1. Než funkce *filter* předá soustavu podmínek *S* a objektivní funkci vytvořenou z množiny podmínek *C* k vyřešení simplexovému algoritmu, musí samozřejmě provést řadu standardních úprav vycházejích z vlastností simplexového algoritmu (ten například požaduje nezápornost proměnných a podmínky ve tvaru neostrých nerovností).

Pokud nyní simplexový algorimus vrátí jediné řešení, jsme hotovi. V opačném případě, tj. když problém lineárního programování má více řešení, je k množině podmínek z H_0 přidána další podmínka ve tvaru *objektivní funkce*=*vypočtená minimální hodnota*. Tím vlastně dostaneme implicitní popis řešení úrovně 1. Nyní vytvoříme nový problém lineárního programování, ve kterém budeme minimalizovat chybu úrovně H_2. Tímto způsobem se pokračuje případně i u dalších úrovní. Přidávané podmínky nám vždy zaručují, že řešení nalezená na dalších úrovních jsou také nejlepšími řešeními předchozích úrovní.

Popsaný postup ilustruje následující obrázek (pro přehlednost je vytvoření chybové funkce vyjmuto z filtru, ve skutečnosti ale vytváří chybovou funkci filtr sám).



Lokální propagace
Dosud popsaná dvojice interpretů hierarchií omezujících podmínek vždy spoléhala na přítomnost podkladového „plochého“ systému řešení podmínek, který rozšířila o podporu hierarchií. V následující části se budeme zabývat algoritmy, které hierarchie podmínek řeší přímo bez přítomnosti dalšího systému řešení podmínek a jsou založeny na tzv. *lokální propagaci.*

Technika lokální propagace vždy používá pouze jednu podmínku pro určení hodnoty proměnné. Jakmile je známa hodnota jedné proměnné, může systém použít jinou podmínku pro získání hodnoty další proměnné atd. Aby bylo možné využívat techniky lokální propagace, je každá podmínka reprezentována jednou nebo více metodami.

Metoda není nic jiného než kus kódu realizujícího funkci, jejímiž argumety jsou tzv. vstupní proměnné podmínky a která počítá hodnoty výstupních proměnných tak, aby podmínka byla splněna.

Příklad 1.24: [SFMB92] (metody podmínky)
podmínka A+B=C může mít obecně tři metody C ← A+B, A ← C–B, B ← C–A

Obecně může být podmínka použita pro určení hodnoty její libovolné proměnné. Označením některých proměnných jako read-only, pouze pro čtení, lze ale snadno dosáhnout toho, že podmínka nebude metody počítající tyto proměnné používat.

Příklad 1.25: [SFMB92] (read-only proměnné)
podmínka A?+B?=C s read-only proměnnými A a B má pouze metodu C ← A+B

Příklad 1.26: [SFMB92] (lokální propagace)
nechť máme podmínky A+B=C a C+D=E
po změně hodnoty proměnné A můžeme použít třeba metodu C ← A+B pro výpočet nové hodnoty proměnné C a následně metodu E ← C+D pro výpočet nové hodnoty proměnné E

Metody nejsou omezeny jen na numerické funkce. Protože se jedná o obecný kus kódu, může metoda třeba svázat název fontu, tj. text, se souborem na disku obsahujícím popis fontu.

Výhodou systémů založených na lokální propagaci je efektivnost a díky použití metod také dostatečná obecnost. Nevýhodou je to, že tyto systémy neumí pracovat se všemi možnými druhy podmínek a například soustavu rovnic nad stejnými proměnnými neumí vyřešit. Tuto nevýhodu překonává zobecněná lokální propagace [HMY96], jejíž konkrétní implementací je systém DETAIL [HMT+94].

Systémy používající lokální propagaci lze klasifikovat podle počtu a druhu metod. Pokud má každá podmínka pouze jednu metodu, hovoříme o jednocestných (one-way) podmínkách resp. systémech, v případě více metod na podmínku, jsou to podmínky resp. systémy vícecestné (multi-way). Je-li výstupem každé metody právě jedna proměnná, jedná se o podmínky resp. systémy s jedním výstupem (one-output), při přítomnosti metod s výstupem několika proměnných hovoříme o podmínkách resp. systémech s více výstupy (multiple output).

1.5.3 DeltaBlue

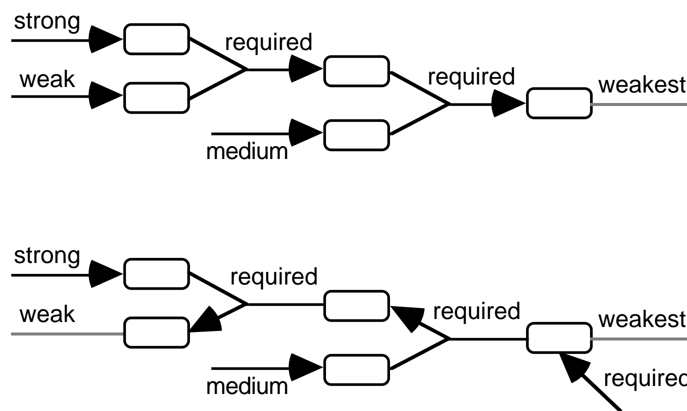
V oblastech jako jsou interaktivní grafická rozhraní se množiny omezujících podmínek často mění a na všechny změny je zde potřeba dostatečně rychle reagovat. Výhodou je pak inkrementálnost algoritmu pro splňování podmínek, tj. možnost přidávat a ubírat podmínky bez nutnosti znova řešit celý systém podmínek od začátku. Inkrementální systémy místo řešení od začátku využívají dosud nalezeného řešení, které pouze upraví.

Mezi inkrementální algoritmy patří algoritmus *DeltaBlue* [SFMB92] vytvořený na University of Washington, který je představitelem vícecestných systémů s jedním výstupem založených na lokální propagaci. Tento algorimus je určen pro řešení hierarchií podmínek použitím locally-predicate-better komparátoru. Protože ale neumí vyřešit

všechny množiny podmínek, které lze použitím locally-predicate-better komparátoru řešit, hovoří se často spíše o použití locally-graph-better komparátoru (jeho přesnější definice je uvedena v následující kapitole o SkyBlue).

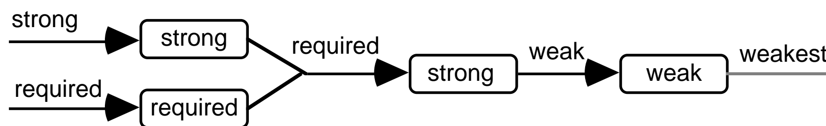
Algoritmus DeltaBlue uschovává nalezené řešení v podobě tzv. *řešícího grafu*, který popisuje, jak se mají hodnoty proměnných přepočítat, aby byly splněny všechny splnitelné podmínky. Uzly řešícího grafu jsou jednotlivé proměnné, zatímco hrany reprezentují omezující podmínky. Šipka na hraně indikuje, jaká metoda podmínky je použita pro její splnění, a tedy jaká proměnná se bude počítat. Cílem systému, je nalézt takový řešící graf, který neobsahuje orientovaný cyklus a do každé proměnné vede nejvýše jedna hrana, což v řeči metod znamená, že každou proměnnou počítá nejvýše jedna metoda.

Následující obrázek ukazuje grafickou reprezentaci řešícího grafu (nahoře). Každá podmínka je zde ohodnocena svojí sílou (preferencí), podmínky, které nejsou splněny, tj. není u nich vybrána metoda, jsou zobrazeny tečkovanou hranou. Obrázek také ukazuje, jak se změní řešící graf po přidání nutné podmínky (dole).



DeltaBlue je inkrementální algorimus, který umožňuje snadné a rychlé přidání resp. ubrání podmínky. K docílení efektivnosti při tom používá tzv. *průchozích preferencí* (*walkabout strength*). Průchozí preference jsou přiřazeny každé proměnné v řešícím grafu a indikují preferenci nejslabší splněné podmínky v řešícím grafu, kterou je možné vyřadit (udělat nesplnitelnou), aby danou proměnnou mohla vypočítat metoda nové podmínky. Při výpočtu průchozích preferencí se tedy berou v úvahu pouze splněné podmínky, tj. podmínky, u kterých je vybrána nějaká metoda. Formální definici průchozí preference lze nalézt v práci [SFMB92].

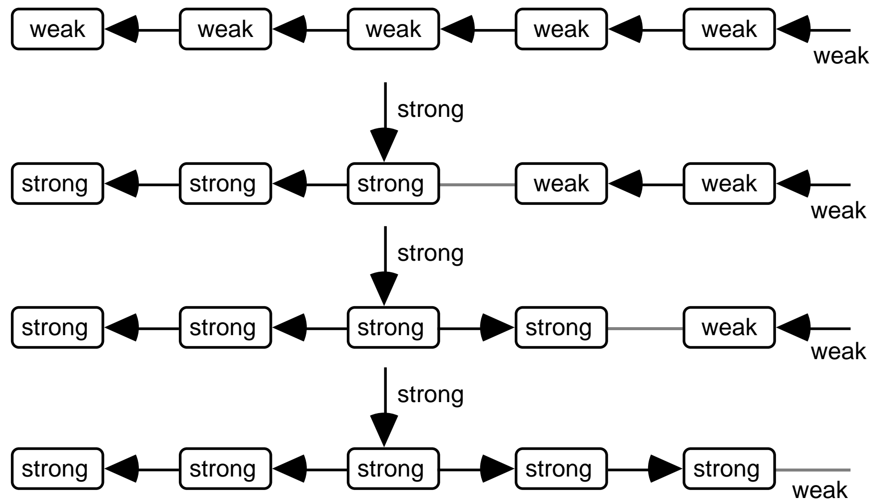
Následující obrázek ukazuje vypočtené průchozí preference. Preference u hran jsou preferencemi podmínek, zatímco preference v uzlech jsou průchozími preferencemi.



Přidání podmínky do řešícího grafu znamená vybrání některé, případně žádné (potom podmínka není splněna) její metody. Díky průchozím preferencím stačí vybrat tu metodu, jejíž výstupní proměnná má nejslabší průchozí preferenci a ta je zároveň menší než je preference přidávané podmínky. Tuto proměnnou od této chvíle váže přidávaná podmínka a podmínka, která ji vázala dosud, se stává nesplněnou, tj. nemá vybránu žádnou metodu. V dalším kroku se algoritmus snaží do řešícího grafu zpětně zakomponovat podmínku, kterou v předchozím kroku udělal nesplněnou. Takto se rekurzivně pokračuje až do chvíle, kdy se podaří podmínku přidat bez vyřazení jiné podmínky, případně když podmínku přidat nelze (tj. když výstupní proměnné všech metod podmínky mají alespoň tak silné průchozí preference jako má podmínka sama)

nebo se narazí na cyklus. Výsledkem je nový řešící graf, který podmínky splňuje nejlépe, jak je to možné podle locally-predicate-better komparátoru.

Následující obrázek ilustruje proces přidání podmínky s preferencí *strong* do řešícího grafu. Podmínky (hrany), které v grafu nemají určeny preferenci, mají preferenci *required*, průchozí preference proměnných jsou v uzlech grafu. Posloupnost vzniklých grafů (shora dolů) ukazuje, jak jsou postupně dvě podmínky s preferencí *required* udělány nesplněnými, tj. nemají vybránu metodu (v grafu označeno tečkovanou hranou), aby je algoritmus vždy hned v následujícím kroku opět splnil vybráním jiné metody. Algoritmus nakonec skončí u podmínky s preferencí *weak*, kterou už nemůže dále splnit, tj. vybrat pro ní metodu.



Ubrání podmínky z řešící grafu je jednoduché v případě, že tato podmínka není splněna (nemá vybránu metodu). Takováto podmínka se jednoduše z grafu vypustí a vzniklý graf je opět řešícím grafem. Pokud je ale ubíraná podmínka v řešícím grafu splněna, tj. má vybránu metodu, je potřeba po jejím vyřazení z grafu otestovat, zda potom nelze splnit nějakou další podmínku, která byla dosud nesplněna. Při obou postupech, přidání resp. ubrání podmínky, je samozřejmě nutné příslušně přepočítat průchozí preference.

DeltaBlue předpokládá implicitní přítomnost slabých podmínek pro každou proměnnou, které zajišťují, že se hodnota proměnné rovná předchozí hodnotě, pokud silnější podmínka nepožaduje něco jiného. Tím je zajištěno to, že systém podmínek nebude nikdy příliš volný (viz. kapitola 1.2).

Algoritmus DeltaBlue má dvě důležitá omezení. Prvním z nich je to, že neumí zpracovat cykly podmínek. Cyklem podmínek je zde myšlen orientovaný cyklus v řešícím grafu. Pokud DeltaBlue narazí na cyklus, ohlásí chybu. Praktické zkušenosti s použitím DeltaBlue při konstrukci grafických uživatelských rozhraní naštěstí ukazují, že lze tyto aplikace vytvářet i bez použití cyklů podmínek. Přítomnost cyklu je zároveň ten případ, kdy může existovat řešení použitím locally-predicate-better komparátoru, zatímco locally-graph-better komparátor řešení nenajde.

Druhé omezení bylo zmíněno již v úvodu. DeltaBlue pracuje pouze s metodami, jejichž výstupem je jedna proměnná, a jedná se tak o systém s jedním výstupem. Toto omezení také není příliš svazující.

Na tomto místě bychom měli poznamenat, že DeltaBlue nalezne vždy jen jedno řešení hierarchie podmínek (pokud vůbec nějaké), tj. jedno přiřazení proměnných. Tím se liší třeba od jednoduchého interpretu (viz. kapitola 1.5.1), který postupně vrací všechna řešení.

V některých aplikacích jako jsou tabulkové procesory zůstává množina podmínek neměnná a mění se jen hodnoty některých proměnných. Od systému se potom očekává, že dopočte hodnoty zbývajících proměnných tak, aby byly podmínky splněny. DeltaBlue

podporuje tvorbu takovýchto aplikací tím, že má možnost generovat tzv. *plány*. Plán je posloupnost metod, jejichž provedení zajistí po změně daných proměnných úpravu ostatních proměnných tak, aby podmínky byly splněny. Plány se díky řešícímu grafu vytvářejí jednoduše tak, že se graf topologicky setřídí a metody ve vzniklém pořadí pak tvoří plán.

Nejhorší časová složitost algoritmu DeltaBlue při přidání resp. ubrání podmínky je O(NM), kde N je celkový počet omezujících podmínek v grafu a M je maximální počet metod na jednu podmínku. Protože počet metod M je většinou omezen nějakým malým číslem, je maximální složitost v podstatě O(N).

Úplný popis algoritmu DeltaBlue včetně pseudokódu, rozboru složitosti a ukázek praktického použití lze nalézt v práci [SFMB92].

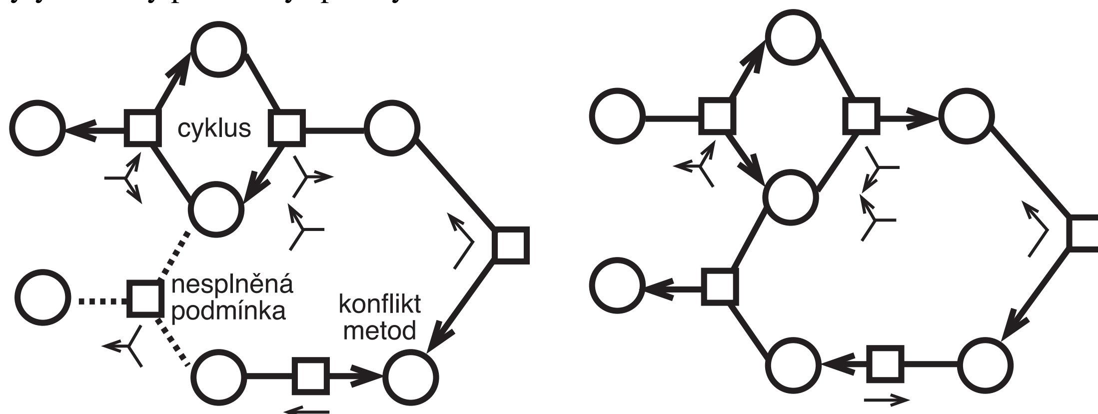
1.5.4 SkyBlue

Dalším „modrým“ algorimem, o kterém zde bude řeč, je *SkyBlue* [San92]. Tento algoritmus je přímým následníkem algoritmu DeltaBlue, který podporuje cykly podmínek a umožňuje používat metody s více výstupy. Podle uvedené klasifikace se tedy jedná o vícecestný algoritmus s více výstupy založený na lokální propagaci. Algoritmus je opět inkrementální, tj. umožňuje efektivní přidání resp. ubrání podmínky z hierarchie.

Základní myšlenky, které stojí za SkyBlue, jsou stejné jako u algoritmu DeltaBlue. Každá podmínka je opět reprezentována jako jedna nebo více metod, metody tentokrát mohou mít výstup s více proměnnými. Řešení se reprezentuje formou orientovaného grafu, který byl ve SkyBlue přejmenován na graf metod. Cílem algoritmu je zkonstruovat tzv. locally-graph-better graf metod, neboli graf reprezentující řešení hierarchie podmínek použitím locally-predicate-better komparátoru.

Graf metod je locally-graph-better (LGB), pokud v něm není žádný konflikt metod a neexistuje v něm nesplněná podmínka, která by šla splnit nesplněním jedné nebo více slabších podmínek. Konflikt metod nastává tehdy, je-li jedna z výstupních proměnných některé vybrané metody zároveň výstupní proměnnou vybrané metody jiné podmínky.

Konflikt metod, nesplněnou podmínku a orientovaný cyklus ukazuje graf metod v následujícím obrázku nalevo. Proměnné jsou v něm označeny kolečkem, zatímco podmínky čtverečkem. Zvolená metoda podmínky je v grafu vyznačena orientovanými hranami, ostatní metody podmínky jsou zobrazeny formou piktogramů. Napravo je zobrazen graf metod se stejnými podmínkami, tentokrát jsou ale metody vybrány tak, aby byly všechny podmínky splněny.



Algoritmus SkyBlue je postaven na podobných principech jako DeltaBlue. Podpora metod s více výstupy si samozřejmě vynutila některé změny například v definici průchozí preference.

První verze algoritmu umožňovala pracovat s cykly podmínek přímo, nebyla ale schopna splnit všechny podmínky nacházející se na cyklu. Novější verze volají v případě

nalezení cyklu specializované systémy řešení jako je algoritmus pro řešení soustav rovnic.

Při práci s podmínkami s jedním výstupem (one-output) vykazuje SkyBlue stejnou nejhorší časovou složitost při přidání resp. ubrání podmínky jako DeltaBlue, tedy O(NM), kde N je celkový počet omezujících podmínek v grafu a M je maximální počet metod na jednu podmínku. Nicméně při použití podmínek s více výstupy (multiple-output) může být časová složitost i podstatně horší. V práci [Mal91] je dokázáno, že problém nalezení LGB grafu pro podmínky s více výstupy je NP-úplný. Nejhorší časová složitost algoritmu SkyBlue pro graf s N podmínkami, z nichž každá má M metod s více výstupy, je potom $O(M^N)$. Grafy, které vykazují exponenciální složitost algoritmu, se ale v reálných aplikacích vyskytují zřídka.

Algoritmus SkyBlue je použit například v balíku Multi-Garnet [SaBo92] určeném pro konstrukci uživatelských rozhraní nebo v imperativním jazyce Kaleidoscope [FBB92b], [LFBB94b]. Přesný popis algoritmu včetně pseudokódu a rozboru složitosti lze nalézt v pracích [San92], [San94b] a [San94c]. Pohled na problematiku debuggingu hierarchií vícecestných podmínek je spolu s popisem schopností systému CNV při analýze sítí podmínek v práci [San94a].

1.5.5 QuickPlan

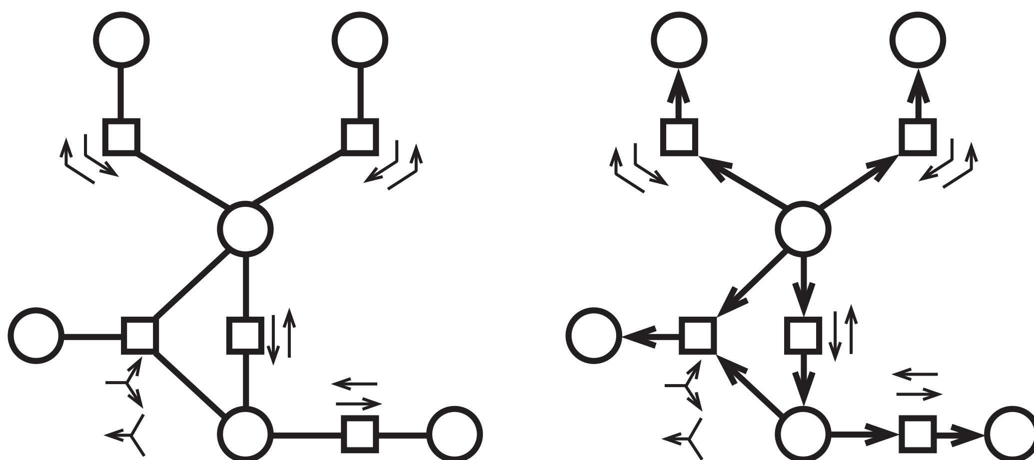
Prvním z algoritmů vyvinutých mimo University of Washington, který zde představíme, je *QuickPlan* [Zan95] vytvořený Bradem Vander Zanden na University of Tennessee. QuickPlan patří do stejné kategorie jako SkyBlue, a je to tedy algoritmus podporující vícecestné podmínky s více výstupy.

Obecně je problém splňování vícecestných podmínek s více výstupy NP-úplný [Mal91]. Algoritmus QuickPlan ale vyžaduje, aby každá metoda podmínky používala všechny proměnné podmínky buď jako vstup nebo jako výstup. Problém splňování vícecestných podmínek s více výstupy, které vyhovují předchozímu omezení, je potom možné řešit v polynomiálním čase [Zan95].

Podobně jako předchozí dvojice algoritmů využívá také QuickPlan grafové reprezentace systému podmínek, přičemž každá podmínka je opět reprezentována jednou nebo více metodami, které mohou mít na výstupu více proměnných. Běh algoritmu se potom skládá ze dvou fází: plánování, kdy je jsou vybrány metody tak, aby byla hierarchie podmínek co nejlépe splněna podle LGB komparátoru, a vlastní exekuce, tj. realizace metod v navrženém pořadí, kdy dochází k výpočtu hodnot proměnných.

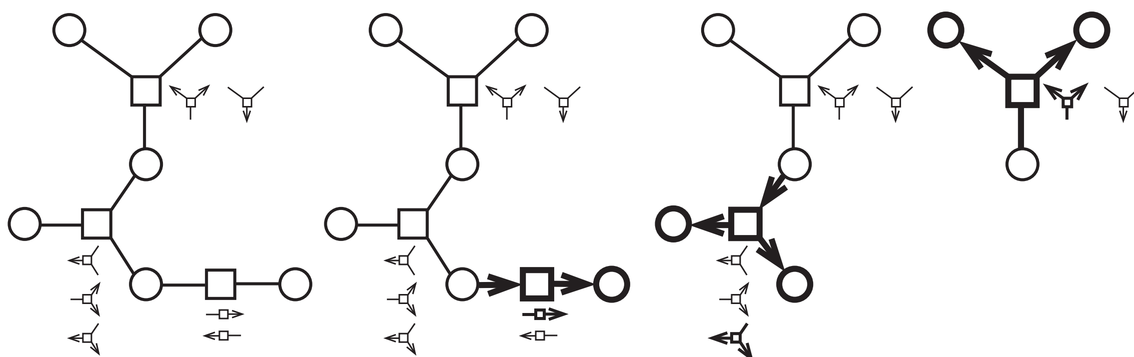
Ve fázi plánování je nejprve systém podmínek reprezentován neorientovaným bipartitním grafem G_C=(V,C,E), kde V a C jsou množiny vrcholů reprezentujících proměnné resp. omezující podmínky a E je množina neorientovaných hran spojujících proměnné s podmínkami, ve kterých se vyskytují. Cílem algoritmu je orientovat hrany v G_C výběrem příslušných metod u podmínek. Vstup metody je potom reprezentován jako orientovaná hrana od příslušné proměnné k podmínce, výstup metody je reprezentován jako hrana od podmínky k výstupní proměnné. Požadováno je aby výsledný orientovaný graf byl acyklický a do každé proměnné vedla maximálně jedna hrana. Takovýto graf se nazývá řešící.

Následující obrázek ukazuje reprezentaci systému podmínek neorientovaným grafem (nalevo) a reprezentaci téhož systému orientovaným grafem (napravo) s vybranými metodami pro splnění podmínek. Proměnné jsou zobrazeny kolečkem, podmínky čtverečkem a všechny možné metody u podmínek formou piktogramu.



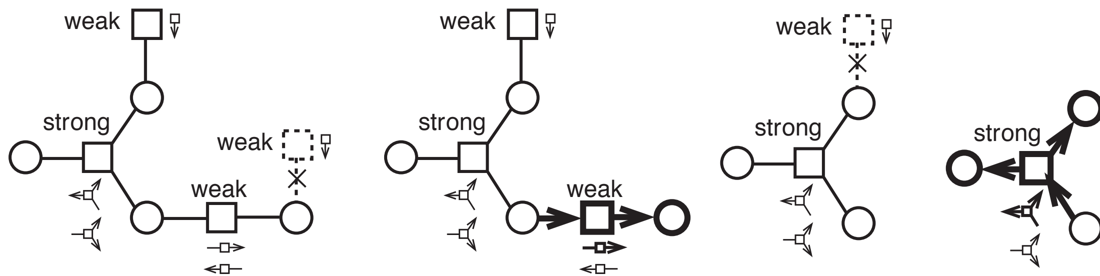
Metody se vybírají známým postupem propagace stupňů volnosti. V neorientovaném grafu G_C jsou nejprve nalezeny proměnné, označme jejich množinu V', které jsou spojeny pouze s jednou podmínkou, označme ji C', a zároveň jsou výstupem některé z metod, označme ji M', této podmínky. Takovéto proměnné se nazývají volné proměnné v grafu G_C. Metoda M' nyní definuje orientaci hran vycházejících z podmínky C', a proto je možné z grafu G_C podmínku C' spolu s hranami, které z ní vedou, a s proměnnými V' vyřadit (eliminovat). Na nově vzniklý graf aplikujeme rekurzivně stejný postup.

Následující obrázek ukazuje postupnou eliminaci podmínek (zleva doprava). Eliminované podmínky, proměnné a hrany jsou označena tučně.

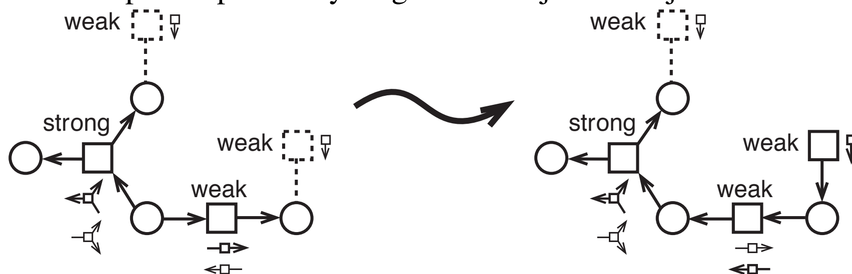


Algoritmus končí tehdy, pokud v grafu G_C zůstanou pouze uzly reprezentující proměnné (resp. žádné uzly) nebo pokud nelze nalézt žádné volné proměnné pro další eliminaci. V prvním případě se podařilo nalézt metodu pro každou podmínku, a každou podmínku tak lze splnit. Ve druhém případě přichází ke slovu ohodnocení podmínek preferencemi. Protože v grafu jsou ještě podmínky, které nemají vybranou metodu, a neexistují v něm volné proměnné, je potřeba některou podmínku vypustit (tj. bude nesplněna), a tím změnit stupeň volnosti proměnných svázaných s touto podmínkou. Vypouští se přirozeně ta podmínka, jejíž preference je z podmínek v grafu nejslabší. Po vypuštění podmínky opět pokračujeme v hledání volných proměnných a eliminaci podmínek.

Následující obrázek ukazuje proces eliminace podmínek (zleva doprava) zahrnující i vypuštění podmínky, abychom v grafu získali volné proměnné. Vypouštěné (tj. nesplněné) podmínky jsou označeny tečkovaně a jsou přeškrtnuty. Eliminované podmínky, proměnné a hrany jsou opět vyznačeny tučně.



Popsaný postup zaručuje, že eliminaci ukončíme grafem obsahujícím pouze proměnné resp. s prázdným grafem. Bohužel v tomto bodě jsme stále nenalezli orientovaný graf poskytující nejlepší řešení, protože někdy je možné podmínky, které byly vypuštěny bez přiřazení metody, ještě do grafu zakomponovat. To lze provést třeba výběrem jiné metody u některé splněné podmínky nebo vypuštěním podmínky se slabší preferencí. Podmínky vypuštěné bez přiřazení metody se proto setřídí od nejpreferovanější k nejslabší a zkouší se přidat ke splněným podmínkám. Při tom může dojít k tomu, že některé slabší podmínky, které dříve byly splněny budou nyní nesplněny. Tyto podmínky jsou zařazeny do seznamu nesplněných podmínek a až na ně dojde řada, pokusíme se je do grafu opět zakomponovat. Pokud se podmínku C do grafu nepodaří zakomponovat, tj. nelze změnit použité metody nebo vypustit slabší podmínku tak, aby podmínka C byla splněna, potom tuto podmínku nelze v nejlepším řešení splnit. Zakomponování nesplněné podmínky do grafu ukazuje následující obrázek.



Právě popsaný algoritmus je jednodušší neinkrementální verzí algoritmu QuickPlan. Vander Zanden navrhl také rychlejší inkrementální verzi využívající toho, že při přidávání podmínky do orientovaného grafu není potřeba kontrolovat všechny podmínky tj. začínat od začátku, ale stačí prověřit jen některé přesně specifikované podmínky. V inkrementální verzi se také využívá obdoba průchozích preferencí ze SkyBlue.

Algoritmus QuickPlan zaručuje nalezení acyklického řešení soustavy podmínek (viz. kapitola 1.5.4), pokud takové řešení existuje. Pokud jediné existující řešení je cyklické (třeba soustava N lineárně nezávislých rovnic s N neznámými), QuickPlan cyklus podmínek identifikuje a pro jeho řešení musí stejně jako SkyBlue použít specializovaný řešič.

Popis neinkrementální i inkrementální verze algoritmu QuickPlan s příklady aplikací a srovnáním se SkyBlue je v práci [Zan95].

1.5.6 Indigo

Předchozí trojice systémů (DeltaBlue, SkyBlue, QuickPlan) založených na lokální propagaci je určena pro řešení hierachií tzv. funkcionálních omezujících podmínek.

Říkáme, že omezující podmínka je *funkcionální* (*dataflow*), pokud pro libovolnou proměnnou v podmínce existuje jednoznačně určená hodnota za předpokladu, že jsou dány hodnoty ostatních proměnných. Funkce pro výpočet této hodnoty se často nazývá *metoda* (viz. zmiňované algoritmy).

V mnoha praktických aplikacích, např. grafických uživatelských rozhraních, se ale vyskytují podmínky, které vlastnost funkcionality nesplňují. Jednoduchým příkladem

mohou být lineární nerovnice (ObjectA.X ≤ ObjectB.X), které se mohou hojně vyskytovat třeba právě v grafických aplikacích. Protože řešení hierarchií takovýchto podmínek použitím simplexového algoritmu (viz. DeltaStar), iteračních numerických metod nebo technikami backtrackingu je výpočtově příliš náročné, vytvořili na University of Washington algoritmus Indigo ([BAFM96a], [BAFM96b]) založený na lokální propagaci. Tento algoritmus umožňuje řešení acyklických sítí omezujících podmínek zahrnujících nerovnice.

Při použití omezujících podmínek v grafických aplikacích, kam je Indigo především mířeno, není vhodné pracovat s triviální chybovou funkcí, protože potom můžeme dostávat neintuitivní výsledky. Indigo proto od začátku pracuje s locally-error-better komparátorem, kde je jako metrika použita vzdálenost dvou reálných čísel. Algoritmus lze ale snadno upravit pro práci s locally-predicate-better komparátorem.

Základní myšlenkou Indiga je propagace dolní a horní meze hodnoty proměnné místo propagace konkrétní hodnoty jako to dělaly předchozí algoritmy. Omezující podmínky jsou zpracovávány od nejsilnějším k nejslabším, přičemž v každém kroku jsou omezeny hodnoty proměnných dané podmínky a v případě nutnosti i hodnoty proměnných již zpracovaných podmínek. Na konci zpracování mají všechny proměnné specifickou hodnotu, tj. hodnoty dolní a horní meze jsou stejné, a to díky tomu, že systém obsahuje podmínky nejslabší preference, které každou proměnnou svazují s její předchozí hodnotou.

Příklad 1.27: [BAFM96b] (algoritmus Indigo)

nechť je dána následující hierarchie podmínek nad reálnými čísly:

c_1:	required	$a \geq 10$
c_2:	required	$b \geq 20$
c_3:	required	$a+b=c$
c_4:	required	$c+25=d$
c_5:	strong	$d \leq 100$
c_6:	medium	$a=50$
c_7:	weak	$a=5$
c_8:	weak	$b=5$
c_9:	weak	$c=100$
c_{10}:	weak	$d=200$

algoritmus Indigo vyřeší hierarchii takto:

akce	a	b	c	d	poznámka
	(-∞,∞)	(-∞,∞)	(-∞,∞)	(-∞,∞)	počáteční rozložení intervalů
add c_1	[10,∞)	(-∞,∞)	(-∞,∞)	(-∞,∞)	
add c_2	[10,∞)	[20,∞)	(-∞,∞)	(-∞,∞)	
add c_3	[10,∞)	[20,∞)	[30,∞)	(-∞,∞)	
add c_4	[10,∞)	[20,∞)	[30,∞)	[55,∞)	
add c_5	[10,∞)	[20,∞)	[30,∞)	[55,100]	
	[10,∞)	[20,∞)	[30,75]	[55,100]	propagace mezí podmínkou c_4
	[10,55]	[20,65]	[30,75]	[55,100]	propagace mezí podmínkou c_3
add c_6	[50,50]	[20,65]	[30,75]	[55,100]	
	[50,50]	[20,25]	[70,75]	[55,100]	propagace mezí podmínkou c_3
	[50,50]	[20,25]	[70,75]	[95,100]	propagace mezí podmínkou c_4
add c_7	[50,50]	[20,25]	[70,75]	[95,100]	podmínka c_7 není splněna
add c_8	[50,50]	[20,20]	[70,75]	[95,100]	podmínka c_8 není splněna, ale minimalizuje se její chyba
	[50,50]	[20,20]	[70,70]	[95,100]	propagace mezí podmínkou c_3
	[50,50]	[20,20]	[70,70]	[95,95]	propagace mezí podmínkou c_4
add c_9	[50,50]	[20,20]	[70,70]	[95,95]	podmínka c_9 není splněna
add c_{10}	[50,50]	[20,20]	[70,70]	[95,95]	podmínka c_{10} není splněna

Algoritmus je korektní a úplný (viz. [BAFM96b]), existuje ale i jeho slabší verze, která je pouze korektní, tj. nalezené řešení je správné, ne vždy je ale řešení nalezeno. Časová složitost algoritmu je v nejhorším případě O(NM), kde N je počet proměnných a M počet omezujících podmínek. Ve většině případů je časová složitost podstatně menší, i když existují příklady, kdy je nejhorší časové složitosti dosahováno.

Současná implementace algoritmu je ve skutečnosti slabou verzí, a to proto, že se pro každou proměnnou pracuje pouze s jedním intervalem možných hodnot místo se sjednocením intervalů. Výhodou je snadnější implementace a rychlejší běh.

Slabá verze algoritmu je popsána v práci [BAFM96a], popis téhož algoritmu včetně důkazu korektnosti a úplnosti lze nalézt také v [BAFM96b].

V interaktivních aplikacích je někdy potřeba řešit stejnou hierarchii omezujících podmínek (stejný graf podmínek) s různými vstupnímy hodnotami některých proměnných (např. různé souřadnice ukazatele). V těchto případech je dobré, pokud lze předem vytvořit plán řešení podmínek, tj. zkompilovat graf podmínek do plánu, který je potom opakovaně prováděn pro každou vstupní hodnotu. Indigo podporuje přímočarou kompilaci plánů, i když algoritmus kompilace nebyl dosud implementován.

Další důležitou vlastností, která dělá algoritmy založené na lokální propagaci efektivnější, je inkrementalita. Inkrementální algoritmy mají možnost po přidání resp. ubrání podmínky inkrementálně upravit současné řešení místo jeho opětovného hledání „od nuly". Bohužel se zdá, že vytvoření inkrementální verze Indiga nebude jednoduché a přímočaré.

1.5.7 Ultraviolet

Jak napovídá název, patří systém Ultraviolet [BFB95] od rodiny algoritmů vyvinutých na University of Washington. Nejedná se vlastně o další algoritmus pro řešení hierachií omezujících podmínek jako spíše o nadstavbu nad několika systémy pro řešení různých typů podmínek. Ultraviolet rozděluje graf omezujících podmínek na samostatné části, pro jejichž vyřešení potom volá specializované řešiče. Pro řešení funkcionálních podmínek nad libovolnými objekty se používá algoritmus Blue, pro nerovnosti a další numerické podmínky zase algoritmus Indigo. Protože oba tyto algoritmy neumí řešit grafy obsahující cykly podmínek, jsou příslušné podgrafy s cykly předávány systému Purple v případě cyklu lineárních rovnic resp. systému Deep Purple při cyklu lineárních nerovnic. Jednotlivé systémy řešení podmínek mezi sebou potom komunikují přes sdílené proměnné. Verze systému Ultraviolet zatím neobsahující nový algoritmus Indigo je popsána v [BFB95].

1.5.8 Houria

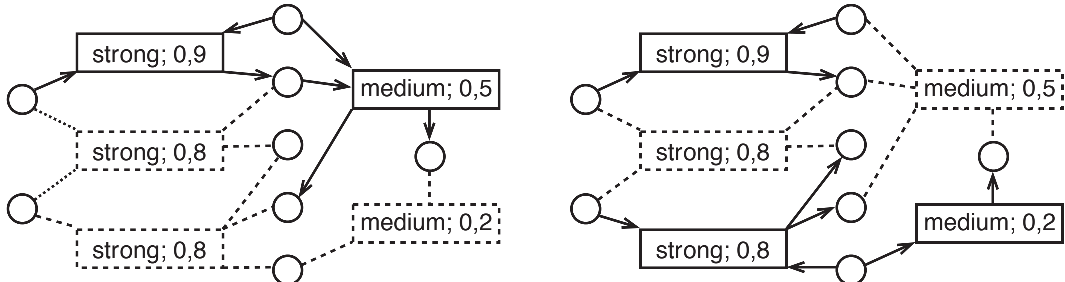
Další z algoritmů, který zde představíme, je Houria III [BNH96] vyvinutá v institutu INRIA. Opět se jedná o algoritmus řešící hierarchie funkcionálních vícecestných podmínek s více výstupy. Na rozdíl od algoritmů SkyBlue a QuickPlan, ale používá globální komparátor, konkrétně weighted-sum-predicate-better komparátor. Kromě preference tak umožňuje mít u podmínky přiřazenu také váhu (viz. globální komparátory-kapitola 1.2.2). Houria III je nástupcem algoritmů Houria a Houria II, která pracovaly s unsatisfied-count-better komparátorem (Houria II umožňoval použití vah u podmínek).

Podmínky jsou zde opět reprezentovány jednou nebo více metodami a řešení, tj. výběr metod u podmínek, je stejně jako v QuickPlanu reprezentováno bipartitním orientovaným grafem. Tentokrát je ale řešící graf definován jako tzv. lexicographic-weighted-sum-better graf.

Lexicographic-weighted-sum-better (LWSB) graf je takový orientovaný bipartitní graf, že neobsahuje žádný konflikt metod, žádný orientovaný cyklus (viz. kapitola 1.5.4) a žádnou nesplněnou podmínku, tj. podmínku bez vybrané metody, takovou, že jejím

splněním, tj. vybráním metody, dostaneme lepší řešení podle weighted-sum-predicate-better komparátoru.

Následující obrázek ukazuje dvojici grafů reprezentujících stejnou hierarchii. Zatímco graf nalevo není LWSB grafem (existuje lepší řešení), graf napravo odpovídá požadavku LWSB. Proměnné jsou zde reprezentovány kolečky, podmínky obdélníky s preferencí a váhou, výběr metod ukazují šipky. Nesplněné podmínky jsou označeny tečkovaně.



Houria III postupuje při hledání řešení následujícím způsobem. Nejprve vytvoří množinu všech řešících grafů obsahujících pouze nutné podmínky. Protože nutné podmínky musí být všechny splněny, je v těchto grafech každé podmínce vybrána metoda a grafy se tak navzájem liší pouze orientací hran, tj. přiřazením metod. V další fázi jsou postupně ke všem grafům přidávány zbylé podmínky a grafy jsou vždy setříděny vzhledem ke komparátoru weighted-sum-predicate-better. Přidání podmínky ke grafu znamená vybrání její metody tak, aby byla kompatibilní se současným grafem, tj. nevznikl konflikt metod ani orientovaný cyklus. Pokud je s grafem kompatibilních více metod podmínky, vznikne příslušný počet dalších grafů. Pokud žádná metoda podmínky není s grafem kompatibilní zůstane graf beze změny. U každého grafu si systém pomatuje součet vah splněných podmínek, který je potom mezi jednotlivými grafy porovnáván, aby se zjistilo, který graf obsahuje nejlepší řešení. Na konci tak uživatel dostává všechna řešení zadané hierarchie, což je další odlišnost od algoritmů SkyBlue a QuickPlan, které vracely jediné řešení.

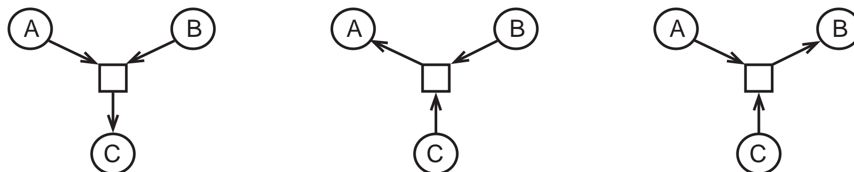
Příklad 1.28: (algoritmus Houria)

nechť máme dánu následující hierarchii (čísla v závorkách jsou váhy podmínek):

required	A+B=C	(u nutných podmínek nemá váha smysl)
strong	C=5	(0,5)
strong	A=3	(0,8)
strong	B=3	(0,8)

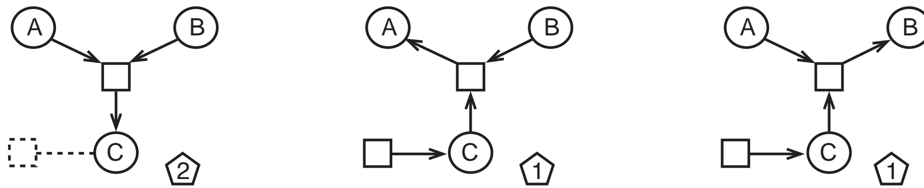
1) nejprve vytvoříme všechny řešící grafy nutné podmínky

předpokládejme, že podmínku A+B=C tvoří tři metody: A+B→C, C–B→A, C–A→B

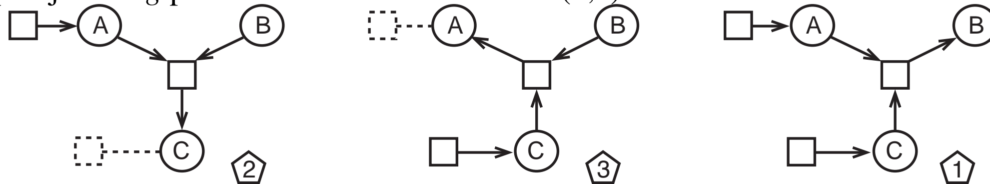


2) potom ke každému grafu přidáme podmínku C=5 resp. její jedinou metodu 5→C

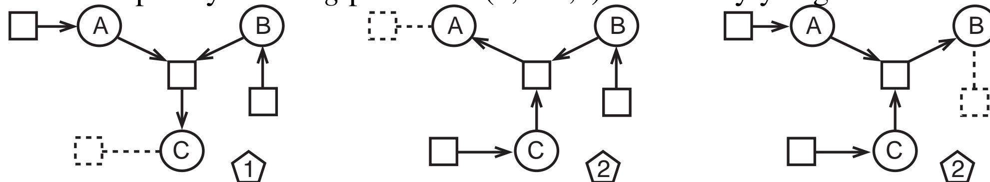
čísla u grafů určují jejich pořadí v uspořádání podle komparátoru *weighted-sum-predicate-better*; řešícími grafy je tedy dvojice napravo, protože v obou jsou splněny všechny podmínky (nesplněná podmínka v grafu nalevo je označena tečkovaně)



3) po přidání podmínky A=3 (3→A) dostaneme jediný řešící graf (úplně napravo), který splňuje všechny podmínky; o pořadí zbylé dvojice grafů rozhodlo to, že graf nalevo splňuje *strong* podmínku A=3 s větší vahou (0,8) než graf uprostřed, který splňuje *strong* podmínku C=5 s menší vahou (0,5)



4) přidáním podmínky B=3 (3→B) se výsledným řešícím grafem stane graf nalevo, protože součet vah splněných *strong* podmínek (0,8+0,8) je zde větší než součet vah splněných *strong* podmínek (0,8+0,5) v obou zbylých grafech



Protože při přidání podmínky využívá Houria dosud nalezeného řešení, můžeme tento algoritmus považovat za inkrementální. Popis algoritmu včetně jeho dalších optimalizací lze nalézt v práci [BNH96].

Jako nadstavba nad systémem Houria byl vytvořen algoritmus pro mezi-hierarchické porovnání v HCLP [BH96]. Princip algoritmu je jednoduchý. Nejprve jsou podobně jako v jednoduchém interpretu (viz. kapitola 1.5.1) „posbírány" všechny hierarchie, které vzniknou při běhu HCLP programu. Potom se postupnou aplikací algoritmu Houria na vzniklé hierarchie vybere nejlepší řešení v rámci mezi-hierarchického porovnání. Nejedná se tak o inkrementální algoritmus, ale o algoritmus dávkový, který jako vstup dostane množinu hierarchií a jako výstup vrátí její řešení.

1.5.9 IHCS

Po Hourii druhý zástupce „evropských" algoritmů pro řešení hierarchií omezujících podmínek, který bude v této práci představen, je Incremental Hierarchical Constraint Solver (IHCS) ([MBC93], [MeBa95]) z Universidade Nova de Lisboa. Jedná se o inkrementální algoritmus pro řešení hierarchií podmínek nad konečnými doménami použitím obecného predikátového komparátoru.

IHCS řeší všechny, tedy i měkké podmínky ihned po jejich objevení. Tímto „optimistickým" přístupem se liší třeba od jednoduchého interpretu, který měkké podmínky pouze shromažďuje a řeší je až na konci běhu programu. Výhodou aktivního použití měkkých podmínek je omezení prohledávacího prostoru při běhu HCLP programu, nevýhodou může být nutnost „zpětného" chodu při neúspěchu.

Pro řešení hierarchie podmínek definuje IHCS tzv. konfiguraci hierarchie.

Konfigurace Φ hierarchie H je trojice AS•RS•US množin podmínek takových, že AS obsahuje aktivní neboli splněné podmínky, RS relaxované tj. nesplněné podmínky a US představuje množinu dosud nezařazených podmínek. Tyto množiny jsou navzájem

disjunktní a jejich sjednocením dostaneme hierarchii H. Množina RS navíc nesmí obsahovat žádné nutné podmínky.

Všechny možné konfigurace dané hierarchie je možné setřídit použitím příslušného komparátoru. Tím dostaneme částečné uspořádání na konfiguracích, které je pro potřeby algoritmu potřeba ještě nějakým způsobem zúplnit (třeba očíslováním podmínek a lexikografickým porovnáním konfigurací, které nejsou srovnatelné použitým komparátorem).

Příklad 1.29: [MeBa95] (uspořádání konfigurací)

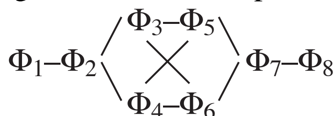
Nechť je dána následující hierarchie podmínek

required	c_1
strong	c_2
strong	c_3
weak	c_4.

Potom existují tyto konfigurace s prázdnou množinou US

$\Phi_1=\{c_1,c_2,c_3,c_4\}\bullet\{\}\bullet\{\}$	$\Phi_5=\{c_1,c_2\}\bullet\{c_3,c_4\}\bullet\{\}$
$\Phi_2=\{c_1,c_2,c_3\}\bullet\{c_4\}\bullet\{\}$	$\Phi_6=\{c_1,c_3\}\bullet\{c_2,c_4\}\bullet\{\}$
$\Phi_3=\{c_1,c_2,c_4\}\bullet\{c_3\}\bullet\{\}$	$\Phi_7=\{c_1,c_4\}\bullet\{c_2,c_3\}\bullet\{\}$
$\Phi_4=\{c_1,c_3,c_4\}\bullet\{c_2\}\bullet\{\}$	$\Phi_8=\{c_1\}\bullet\{c_2,c_3,c_4\}\bullet\{\}$

Částečné uspořádání definované unsatisfied-count-better komparátorem vypadá takto (bráno zleva doprava, tj. nalevo jsou lepší konfigurace než napravo; konfigurace Φ_3 a Φ_4 resp. Φ_5 a Φ_6 jsou nesrovnatelné):



Lineární uspořádání získáme zúplněním částečného uspořádání, tj. určením uspořádání mezi nesrovnatelnými konfiguracemi (nesrovnatelné konfigurace se setřídí lexikograficky):

$$\Phi_1–\Phi_2–\Phi_3–\Phi_4–\Phi_5–\Phi_6–\Phi_7–\Phi_8$$

Postup řešení hierarchie H podmínek potom není nic jiného než přechod od konfigurace $\emptyset\bullet\emptyset\bullet H$ k nejlepší konfiguraci $AS\bullet RS\bullet\emptyset$., tj. takové konfiguraci, kde množina nezařazených podmínek je prázdná, podmínky z AS jsou navzájem bezesporné a v uspořádání konfigurací neexistuje lepší konfigurace, která by měla prázdnou množinu nezařazených podmínek. Pro přechod od jedné konfigurace k druhé je definována řada pravidel.

Tzv. *dopředné (forward) pravidlo* jednoduše přemístí podmínku z množiny US nevyzkoušených podmínek do množiny AS splněných podmínek, tj. o podmínce se optimisticky předpokládá, že bude splněna. Pokud se tím množina AS stane sporná, tj. podmínky z AS nemohou platit současně, aplikuje se *zpětné (backward) pravidlo*. To nalezne jakousi konfliktní podkonfiguraci, pomocí které přímístí některé podmínky z AS do RS a jiné podmínky případně přesunete opět do množiny US dosud nezařazených podmínek. Postupnou aplikací této dvojice pravidel dojde buď k nalezení řešení hierarchie nebo se zjistí, že daná hierarchie nemá řešení. Stejná pravidla je pak možné použít i při přidání podmínky k hierarchii.

Příklad 1.30: [MeBa95] (algoritmus IHCS)

nechť je dána následující hierarchie podmínek

c_1: strong $X+Y=15$

c_2: strong $3X-Y<5$

c_3: weak $X>Y+1$

c_4: weak $X<7$

předpokládejme dále, že počáteční domény proměnných X a Y jsou D(X)=1…10 a D(Y)=1…10.

akce	konfigurace	D(X)	D(Y)	pravidlo
add c_1	$\{\ \}\bullet\{\ \}\bullet\{c_1\}$	1…10	1…10	fw
	$\{c_1\}\bullet\{\ \}\bullet\{\ \}$	5…10	5…10	
add c_2	$\{c_1\}\bullet\{\ \}\bullet\{c_2\}$	5…10	5…10	fw
	$\{c_1,c_2\}\bullet\{\ \}\bullet\{\ \}$	∅	∅	bw
	$\{c_1\}\bullet\{c_2\}\bullet\{\ \}$	5…10	5…10	
add c_3	$\{c_1\}\bullet\{c_2\}\bullet\{c_3\}$	5…10	5…10	fw
	$\{c_1,c_3\}\bullet\{c_2\}\bullet\{\ \}$	7…10	5…8	
add c_4	$\{c_1,c_3\}\bullet\{c_2\}\bullet\{c_4\}$	7…10	5…8	fw
	$\{c_1,c_3,c_4\}\bullet\{c_2\}\bullet\{\ \}$	∅	∅	bw
	$\{c_1,c_3\}\bullet\{c_2,c_4\}\bullet\{\ \}$	7…10	5…8	

Aby bylo možné získat všechna řešení dané hierarchie, definuje IHCS tzv. *alternativní pravidlo*. Toto pravidlo umožňuje přechod k další konfiguraci, která je v zúplněném uspořádání konfigurací horší než současná konfigurace, ale v částečném uspořádání leží obě konfigurace na stejné úrovni, tj. ani jedna z nich není lepší než ta druhá.

Algoritmus IHCS umožňuje také inkrementální odstranění podmínky. K tomu se používají dvě pravidla podle toho, zda je odstraňovaná podmínka v množině relaxovaných podmínek nebo v množině aktivních podmínek. Odstranění podmínky v prvním případě je jednoduché. Protože je v množině relaxovaných podmínek, je nesplněna, a tedy neovlivňuje další podmínky. Stačí ji proto pouze vypustit z příslušné množiny. Odstranění aktivní, tj. splněné podmínky je náročnější. Jejím odstraněním se totiž mohou některé nesplněné (relaxované) podmínky stát splněnými, zatímco jiné splněné (aktivní) podmínky se stanou nesplněnými. V tomto případě jsou tedy kandidáti na změnu stavu (splněné↔nesplněné) vybráni z množiny aktivních podmínek, dostaneme tak množinu Reset, i z množiny relaxovaných podmínek, dostaneme množinu Activate. Množiny Reset a Activate se potom sloučí do množiny US dosud nezařazených podmínek a na nově vzniklou konfiguraci se aplikují příslušná pravidla pro přidání podmínky.

IHCS nabízí také zvláštní pravidlo pro řešení disjunktivních podmínek (viz. podmínky vyšších řádů-kapitola 1.3.1). Díky podpoře disjunktivních podmínek je tak v IHCS možné částečně simulovat mezi-hierarchické porovnání.

Pro rychlé nalezení konfliktních konfigurací a množin typu Reset a Activate používá IHCS tzv. *závislostního (dependency) grafu*. Závislostní graf je orientovaný graf, jehož uzly reprezentují podmínky a hrana vede od podmínky c_1 k podmínce c_2, pokud podmínka c_1 nějak omezuje nějakou proměnnou v, která je přítomna v podmínce c_2.

Popis algoritmu IHCS lze nalézt v práci [MBC93], podrobnější popis včetně důkazu korektnosti a úplnosti algoritmu je v [MeBa95].

Část druhá

Expertní systémy a hierarchie omezujících podmínek

2.1 Expertní systémy

Expertní systémy [Neb88, PaCh88] jsou asi nejrozšířenějším a určitě nejznámějším výsledkem výzkumů v oblasti umělé inteligence. Dnes se jim v popularitě a rozšířenosti vyrovnávají různí „inteligentní“ agenti pomáhající uživatelům orientovat se ve složitých systémech a softwarových aplikacích. Opět se ale nejedná o nic jiného než o „převlečené“ expertní systémy simulující chování specialisty na danou problematiku.

Expertní nebo také jinak znalostní systém je krátce řečeno softwarová aplikace simulující chování lidského experta. Jejím úkolem je nabídnout uživateli odbornou expertízu v přesně určené oblasti a to nejčastěji formou otázek a odpovědí. Prvotní dotaz zadává systému zpravidla uživatel, další doplňující otázky pro upřesnění prvotního dotazu potom klade systém sám. Uživatel se v průběhu dialogu často může systému dotazovat proč mu klade danou otázku nebo jak dospěl k určitému závěru.

Typickým příkladem, na kterém se často prezentuje použití expertních systémů, je lékařství. Zde tyto systémy pomáhají praktickým lékařům v diagnózách, pro které by jinak bylo nutné přizvat specialistu. Z tohoto příkladu je také vidět, že expertní systém zpravidla není určen pro úplného laika v oblasti expertízy, ale předpokládají se určité znalosti, například terminologie, bez kterých by uživatel nebyl schopen správně reagovat na dotazy systému a interpretovat nalezené řešení problému. Existují ale také jednodušší expertní systémy určené pro širokou veřejnost, které předpokládají jen základní znalosti dané problematiky.

Jedním z cílů při vývoji expertních systémů bylo umožnit uživatelům, kteří nejsou špičkovými experty v dané oblasti, provádět expertízy bez nutnosti konzultací s těmito experty. Důvodem je nedostatek a z něj plynoucí vytíženost špičkových odborníků. K vytvoření expertního systému sloužícího pro rozmisťování nákladu v nákladovém prostoru raketoplánu například přistoupila i NASA, když odešel do důchodu jeden z dvojice specialistů na tuto problematiku.

Strukturu typického expertního systému lze jednoduše popsat následující rovnicí:

expertní (znalostní) systém = báze znalostí + inferenční mechanismus.

Báze znalostí obsahuje fakta týkající se dané oblasti expertízy a tzv. heuristické znalosti, tj. znalosti, jak s fakty zacházet. Zatímco fakta lze celkem snadno získat například z učebnic, heuristické znalosti tvoří osobnost experta, který je získal vlastními i předanými zkušenostmi. Tyto znalosti jsou potom prostřednictvím znalostního inženýra kódovány do báze znalostí a právě jejich kvalita často určuje výslednou kvalitu expertního systému. Pro expertní systémy totiž platí, že síla je ve znalostech.

Dalším paradigmatem, které charakterizuje znalosti, je to, že znalost je nejistá. Znamená to, že zkušenosti experta, které jsou zakódovány do databáze znalostí, nejsou 100% spolehlivé. Nejistota je také ukryta v nepřítomnosti některých vstupních dat, které se potom musí nějakým způsobem odvodit z přítomných údajů nebo se místo nich použije nějaká typická (default) hodnota. Vycházíme-li tedy z nepřesných údajů, dostaneme nutně také nepřesný výsledek. Pro práci s neurčitou informací dnes existuje řada postupů od klasické kompozicionální metody [Haj85] reprezentované systémy MYCIN [Sho76, BuSh84] a PROSPECTOR [DHN+78, HDE78] přes pravděpodobnostní přístup a Dempster-Shaferovu teorii evidence [Dem68, Sha76] až k fuzzy logice [Zad65, Zad83, Nov86] a nemonotónním logikám [MD80, DPP95].

Způsob reprezentace znalostí se v různých expertních systémech liší. V této práci nás bude zajímat asi nejtypičtější metoda reprezentace báze znalostí formou pravidel. Hovoříme potom o pravidlově orientovaných expertních systémech, kde pravidla mají tvar implikace:

`pokud` *platí předpoklady* `potom` *odvoď závěr*.

Druhou část expertního systému tvoří *inferenční mechanismus* sloužící pro odvozování závěrů z dané báze znalostí. Jedná se o relativně samostatnou součást expertního systému nezávislou na konkrétní bázi znalostí. Používané inferenční mechanismy jsou téměř výlučně založeny na prohledávání. Používáno je buď vstřícné řetězení (forward chaining) vycházející z daných faktů, ze kterých postupně odvozuje závěry, nebo zpětné řetězení (backward chaining/goal-directed), které naopak začíná od zadaného cíle/dotazu, který se snaží zjednodušováním převést až na známá fakta z báze znalostí. V některých systémech (PROSPECTOR) jsou dokonce obě metody kombinovány tak, že vstřícné řetězení nejprve z prvotních faktů dodaných uživatelem odvodí několik hypotéz, které jsou potom ověřovány zpětným řetězením za použití dodatečných dotazů na uživatele.

Protože inferenční mechanismus je relativně samostatnou součástí expertního systému, vytvářejí se také tzv. prázdné expertní systémy, které obsahují právě jen inferenční mechanismus spolu s metodou kódování báze znalostí. Tyto prázdné expertní systémy se potom naplní konkrétní bází znalostí, čímž lze získat expertní systémy pro rozličné oblasti expertízy.

Klíčovou součástí každého expertního systému musí být schopnost vysvětlit a zdůvodnit navržené řešení. Díky tomuto vysvětlovacímu mechanismu potom může uživatel, který není specialistou v oblasti expertízy, ověřit řešení navržené programem a nemusí mu jen slepě důvěřovat. Vysvětlovací mechanismus se používá také ve fázi naplňování expertního systému znalostmi, kdy umožňuje tvůrci systému „odladit" bázi znalostí. Nepřítomnost vysvětlovacího mechanismu je slabinou expertních systémů založených na umělých neuronových sítí.

Klasické expertní systémy lze rozdělit do dvou kategorií: diagnostické a plánovací expertní systémy. Diagnostické expertní systémy mají za úkol přiřadit zadané sadě příznaků odpovídající diagnózu, tj. například druh nemoci nebo závady. Příkladem diagnostických expertních systémů jsou systémy MYCIN [Sho76, BuSh84] pro diagnostiku infekčních onemocnění a PROSPECTOR [DHN+78, HDE78] pro určování ložisek nerostů. Plánovací expertní systémy naproti tomu mají jako vstup sadu parametrů/vlastností konstruovaného předmětu a jejich cílem je navrhnout přesné rozložení částí předmětu tak, aby výsledný objekt měl požadované vlastnosti. Mezi plánovací expertní systémy patří DENDRAL [BuFe78, LBFL80] pro určování strukturních vzorců organických sloučenin ze zadaného hmotového spektrogramu a R1 (XCON) [McD79] pro navrhování uživatelských konfigurací počítačů DEC podle zadaných parametrů konfigurace.

V současnosti jsou předmětem intenzivních výzkumů expertní systémy druhé generace. Jedná se o distribuované expertní systémy [EEP91], ve kterých je báze znalostí roztroušena na různých počítačích, a o expertní systémy se schopností učení, které se během používání sami zdokonalují, tj. zvyšují přesnost svých řešení.

V předchozím textu byly také zmíněny expertní systémy založené na umělých neuronových sítích [MP43]. Tyto systémy vynikají svojí adaptibilitou a schopností učit se z příkladů a vytvářet tak vlastní zkušenosti. Jejich slabinou je ale nepřítomnost přímého vysvětlovacího mechanismu, která souvisí s tím, že znalost je v neuronové síti rozložena po síti neuronů a spojení mezi nimi.

2.2 Expertní systémy založené na omezujících podmínkách

Jedním z cílů této práce je navrhnout nový přístup k tvorbě expertních systémů (ES), který by byl založen na využití hierarchií omezujících podmínek. V této práci navrhované znalostní systémy jsou pravidlově orientované a jsou postaveny na logickém programování s hierarchiemi omezujících podmínek (HCLP) a s mezi-hierarchickým porovnáváním (kapitola 1.2.5). V této kapitole bude ukázán rozdíl mezi navrhovanými znalostními systémy založenými na omezujících podmínkách a klasickými

kompozicionálními expertními systémy. Budou zde rozebrány očekávané vlastnosti těchto systémů z hlediska koncových uživatelů i tvůrců systému a chybět nebude ani soubor požadavků, které navrhované systémy kladou na podkladový systém řešení podmínek.

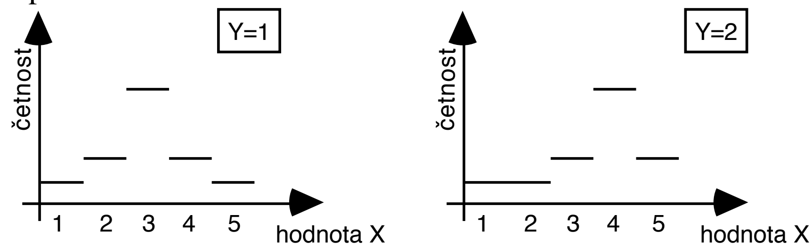
Při návrhu expertních systémů používajících omezující podmínky budeme vycházet z klasických pravidlově orientovaných expertních systémů. Jako základní paradigma použijeme logické programování s hierarchiemi omezujících podmínek (HCLP) a s podporou mezi-hierarchického srovnávání. To nám umožní přirozeně vyjadřovat pravidla expertního systému v jazyce HCLP, zatímco hierarchie podmínek zde budou sloužit pro vyjádření neurčité informace (viz. dále).

Aby bylo možné jistým způsobem slučovat příspěvky jednotlivých pravidel k řešení podobně jako v kompozicionálních systémech, je potřeba použít mezi-hierarchické srovnávání. Zatímco ale v klasických kompozionálních expertních systémech se výsledky použití jednotlivých pravidel skládají dohromady, aby se získalo výsledné řešení, v expertních systémech založených na hierarchiích omezujících podmínek tak, jak jsou navrhovány v této práci, se pro získání řešení použije vždy jen jedna „větev“ výpočtu a to ta, která dává nejlepší řešení (viz. příklady 1.15 a 2.1). Jakási forma skládání řešení vzniklých v jednotlivých větvích výpočtu je zde implicitně obsažena ve výběru řešení vzniklé množiny hierarchií. Tento způsob práce s pravidly podobající se klasickému logickému programování by měl tvůrci expertního systému usnadnit kontrolu nad chováním vytvářeného systému. Jednodušeji se také bude vyjadřovat odůvodnění získaného řešení, které může mít například podobu posloupnosti použitých pravidel ve zvolené větvi dávající nejlepší řešení.

V předchozích odstavcích bylo nastíněno, že pro práci s neurčitou informací mohou v navrhovaných expertních systémech přirozeně sloužit hierarchie podmínek. Následující příklad 2.1 ukazuje, jakým způsobem lze například vyjádřit neurčitou informaci kolem tzv. default hodnot pomocí hierarchie podmínek.

Příklad 2.1: (default hodnoty pomocí podmínek)

Nechť následující dvojice grafů zobrazuje rozložení relativních četností výskytů hodnoty X při zvolené hodnotě Y.



Potom tuto situaci můžeme kódovat například následující dvojicí HCLP pravidel:

```
p(X,Y):-
   X in {1,2,3,4,5}@required,
   X in {2,3,4}@prefer,
   X=3@weak,
   Y=1@strong.
p(X,Y):-
   X in {1,2,3,4,5}@required,
   X in {3,4,5}@prefer,
   X=4@weak,
   Y=2@strong.
```

Předpokládejme dále, že se používá některý predikátový typ komparátoru (tj. podmínka je buď zcela splněna nebo splněna není, přesnost splnění podmínky nás nezajímá). Jestliže uživatel nic neví o hodnotách X a Y, dostane jako odpověď na dotaz `?-p(X,Y)` dvojici stejně dobrých řešení `{X/3,Y/1}` a `{X/4,Y/2}`, tj. default hodnoty pocházející z použití jednotlivých pravidel (žádné z ohodnocení

není preferováno, protože obě splňují všechny podmínky příslušných hierarchií). Pokud ale například známe hodnotu proměnné `X=2`, tj. volá se dotaz `?-p(2,Y)`, vybere systém řešení `{X/2,Y/1}` pocházející z prvního pravidla, protože je zde splněna podmínka se sílou `prefer`, zatímco v druhém pravidle stejně silná podmínka splněna není. Všimněte si také, že pokud známe hodnotu proměnné Y, záleží vypočtené řešení na síle podmínky, která tuto hodnotu určuje. Například, volá-li se cíl `?-p(2,Y),Y=2@weak`, vydá systém řešení `{X/2,Y/1}` (`strong` podmínka prvního pravidla „přebije“ zadanou `weak` podmínku určující hodnotu proměnné Y), ale při volání cíle `?-p(2,Y),Y=2@strong` bude vypočtené řešení `{X/2,Y/2}`.

Pro vyjádření neurčité informace lze využít i další mechanismy, jako jsou například váhy jednotlivých podmínek (viz. kapitola 1.2) nebo metrické chybové funkce (kapitola 1.2.2) určující jak přesně je podmínka splněna. Kombinace jednotlivých neurčitých příspěvků k řešení se potom přirozeně dosáhne použitím různých druhů komparátorů. Kromě tohoto „vestavěného“ zpracování neurčité informace lze používat i tradiční metody například práci s fuzzy podmínkami [Mat93]. Další možností je používat podmínky pracující s intervaly hodnot, které tak vyjadřují neurčitost hodnoty. Na práci s intervaly je například založen systém pro výpočet nájemného v Mnichově (The Munich Rent Advisor [FrAb96]) podle zadané velikosti, vybavenosti a umístění bytu.

Dosud jsme hovořili o vnitřní struktuře navrhovaných expertních systémů, důležitá je ale také interaktivita mezi systémem a uživatelem. Uživatel systému postaveného na omezujících podmínkách bude moci například postupně měnit vstupní hodnoty, typicky tažením posuvníku myší v grafickém uživatelském rozhraní, a systém podle nastavených hodnot interaktivně přepočítá hodnoty výstupních parametrů podobně jako v tabulkovém procesoru. Vzhledem k charakteru systému s hierarchiemi podmínek a mezi-hierarchickým porovnáváním zde lze očekávat nemonotónní chování (viz. kapitola 1.2.6), které se bude projevovat například tím, že drobná změna jednoho vstupního parametru vyvolá nalezení řešení zcela odlišného od řešení nalezeného před změnou.

Z dosud uvedeného popisu navrhovaných expertních systémů plynou jasné požadavky na podkladový systém řešení hierarchií podmínek. Bude se jednat o systém HCLP s mezi-hierarchickým porovnáváním, přičemž mezi-hierarchické porovnávání dále vyžaduje použití globálních komparátorů. Implementace systému pro řešení hierarchií podmínek tedy musí podporovat globální komparátory a navíc by měla být dostatečně efektivní, aby umožňovala interaktivní práci se systémem. Tento požadavek žádný současný algoritmus řešení podmínek nesplňuje. Většina efektivních algoritmů založených na lokální propagaci podporuje jen určitý druh podmínek a pouze lokální komparátory. Algoritmy podporující globální komparátory jsou zase buď časově nebo prostorově náročné a opět se zaměřují jen na určité druhy podmínek. Z nastíněných požadavků tedy vyplývá, že je nejprve potřeba vyvinout efektivní a dostatečně obecný algoritmus pro řešení hierarchií omezujících podmínek. Popis takového algoritmu je náplní zbytku této práce.

2.3 Obecný systém pro řešení hierarchií omezujících podmínek

V předchozí kapitole bylo nastíněno, že pro tvorbu expertních systémů založených na hierarchiích omezujících podmínek je nejprve potřeba vytvořit dostatečně efektivní algoritmus pro řešení hierarchií podmínek, který bude podporovat globální komparátory.

Současné algoritmy, které jsou dostatečné obecné, aby zvládly různé druhy podmínek a podporovaly globální komparátory, jsou založeny na tzv. zjemňovací (refining) metodě. Principem této metody je postupné řešení hierarchie podmínek od vrstvy s nejsilnějšími podmínkami až po vrstvu s nejslabšími podmínkami. Typickými reprezentanty této skupiny jsou jednoduchý systém řešení podmínek (kapitola 1.5.1) a

algoritmus DeltaStar (kapitola 1.5.2). Obecný systém pro řešení hierarchií podmínek založený na zjemňovací metodě je popsán v příloze B a práci [Bar96], kde je také ukázána obecné architektura systému pro řešení hierarchií podmínek. Nevýhodou algoritmů založených na zjemňovací metodě je jejich neefektivnost, vyžadující vždy po přidání nebo ubrání podmínky úplný přepočet řešení od začátku, tj. nelze použít výsledků dosud provedeného výpočtu.

Současné efektivní algoritmy řešení hierarchií podmínek, které jsou založeny na lokální propagaci (viz. kapitola 1.5), jsou omezené hned v několika směrech. Předně ve většině případů podporují pouze lokální komparátory. Pokud jsou podporovány i komparátory globální (Houria-kapitola 1.5.8), je toho dosaženo podstatným zvětšením prostoru nutného pro výpočet. Algoritmy lokální propagace vždy najdou pouze jediné řešení a nejsou schopny nalézt alternativní řešení. Nemohou také řešit hierarchie podmínek, ve kterých se vyskytují cykly (kapitola 1.5.4). Řešení cyklů podmínek lze dosáhnou pouze voláním „externí“ procedury, která tak narušuje eleganci vlastního algoritmu (kapitola 1.5.7). Asi největším omezením algoritmů lokální propagace je schopnost práce pouze s podmínkami ve tvaru rovnosti nebo obecněji s funkčními podmínkami, tj. takovými podmínkami, které jednoznačně „vypočtou“ hodnotu výstupní proměnné z hodnot vstupních proměnných. Při práci s obecnějším typem podmínek jako jsou nerovnosti, se částečně ztrácí efektivita výpočtu (Indigo-kapitola 1.5.6).

V následujících kapitolách bude postupně popsána teorie umožňující implementaci systémů pro efektivní řešení hierarchií podmínek a bude zde ukázán konkrétní algoritmus pro řešení hierarchií podmínek, který překonává zmíněné nedostatky lokální propagace a dovoluje tak efektivně pracovat s globálními i lokálními komparátory a s širší třídou podmínek zahrnující například nerovnosti.

2.3.1 Teorie řešení hierarchií podmínek

Aby bylo možné vytvořit formální systém pro řešení hierarchií podmínek, je nejprve potřeba přesně definovat pojem řešení hierarchie podmínek. Oproti kapitole 1.2, kde je uvedena původní formální definice řešení hierarchie podmínek prostřednictvím komparátorů, budeme nyní používat přesnější a více omezující definice. Některé základní pojmy (omezující podmínka, označená omezující podmínka, hierarchie omezujících podmínek, úrovně hierarchie, ohodnocení proměnných a chybová funkce) převezmeme beze změn z kapitoly 1.2.

Nejprve nahradíme obecný komparátor *better* jemnější definicí pomocí úrovňových komparátorů. Tato a následující definice potom také implicitně vyjadřují ideu pojmu respektování hierarchie podmínek, který nebude ve svém původním tvaru (Definice 1.2) dále používán.

Definice 2.1: (úrovňový komparátor)

Říkáme, že relace $\overset{C}{\leq}$ na množině ohodnocení je *úrovňový komparátor*, pokud jsou splněny následující podmínky (C a C' jsou množiny omezujících podmínek, σ, θ a π jsou ohodnocení proměnných a *e* je chybová funkce):

a) $\sigma \overset{C}{\leq} \theta \wedge \theta \overset{C}{\leq} \pi \Rightarrow \sigma \overset{C}{\leq} \pi$ (tranzitivita relace $\overset{C}{\leq}$)

b) $\forall c \in C \;\; e(c\sigma) \leq e(c\theta) \Rightarrow \sigma \overset{C}{\leq} \theta$

c) $\theta \overset{C \cup C'}{\leq} \sigma \wedge \sigma \overset{C}{\leq} \theta \Rightarrow \theta \overset{C}{\leq} \sigma$

d) $\sigma \overset{C}{\leq} \theta \wedge \sigma \overset{C'}{\leq} \theta \Rightarrow \sigma \overset{C \cup C'}{\leq} \theta.$

Pomocí relace $\overset{C}{\leq}$ nyní můžeme definovat také relace $\overset{C}{<}$ a $\overset{C}{\sim}$ takto:

$$\sigma \overset{C}{<} \theta \equiv_{def} \sigma \overset{C}{\leq} \theta \wedge \neg \theta \overset{C}{\leq} \sigma$$

$$\sigma \overset{C}{\sim} \theta \equiv_{def} \sigma \overset{C}{\leq} \theta \wedge \theta \overset{C}{\leq} \sigma .$$

Pro zjednodušení zápisu můžeme dále definovat relace $\overset{C}{\geq}$ a $\overset{C}{>}$ takto:

$$\sigma \overset{C}{\geq} \theta \equiv_{def} \theta \overset{C}{\leq} \sigma$$

$$\sigma \overset{C}{>} \theta \equiv_{def} \theta \overset{C}{<} \sigma .$$

Podmínky a) a b) definice 2.1 jsou intuitivně očekávanými vlastnostmi úrovňového komparátoru. Podmínka a) říká, že pokud ohodnocení σ není horší (tj. je lepší nebo stejně dobré) než ohodnocení θ, které zároveň není horší než π, potom také σ není horší než π na dané množině podmínek *C*. Podmínka b) zase zajišťuje, že pokud nějaké ohodnocení σ má menší nebo stejnou chybu u všech podmínek dané množiny *C* než jiné ohodnocení θ, potom je ohodnocení σ lepší nebo stejně dobré na celé množině *C* než ohodnocení θ. Smysl podmínek c) a d) možná není na první pohled tak viditelný jako u předchozí dvojice podmínek. Tyto podmínky vlastně říkají, že srovnání ohodnocení na dané úrovni (tj. množině stejně preferovaných podmínek) nemusíme dělat pro celou úroveň najednou, ale že úroveň je možné rozdělit na několik menších množin podmínek a srovnání provést odděleně na těchto množinách.

Prezentovaná čtveřice podmínek kladených na úrovňový komparátor nám později umožní dokázat zajímavé tvrzení o efektivní metodě řešení hierarchie podmínek. Nejprve se ale podívejme na další vlastnosti úrovňového komparátoru a z něj definovaných relací, které zachycuje následující tvrzení.

Tvrzení 2.1: (vlastnosti úrovňového komparátoru)

a) $\sigma \overset{C}{\leq} \theta \Rightarrow \sigma \overset{C}{<} \theta \vee \sigma \overset{C}{\sim} \theta$

b) $\sigma \overset{C}{\leq} \sigma$ (reflexivita relace $\overset{C}{\leq}$)

c) $\sigma \overset{C}{\sim} \sigma$ (reflexivita relace $\overset{C}{\sim}$)

d) $\sigma \overset{C}{\sim} \theta \Rightarrow \theta \overset{C}{\sim} \sigma$ (symetrie relace $\overset{C}{\sim}$)

e) $\sigma \overset{C}{\sim} \theta \wedge \theta \overset{C}{\sim} \pi \Rightarrow \sigma \overset{C}{\sim} \pi$ (tranzitivita relace $\overset{C}{\sim}$)

f) $\neg \sigma \overset{C}{<} \sigma$ (ireflexivita relace $\overset{C}{<}$)

g) $\sigma \overset{C}{<} \theta \Rightarrow \neg \theta \overset{C}{<} \sigma$ (antisymetrie relace $\overset{C}{<}$)

h) $\sigma \overset{C}{<} \theta \wedge \theta \overset{C}{<} \pi \Rightarrow \sigma \overset{C}{<} \pi$ (tranzitivita relace $\overset{C}{<}$)

i) $\sigma \overset{C}{<} \theta \wedge \theta \overset{C}{\sim} \pi \Rightarrow \sigma \overset{C}{<} \pi$

j) $\sigma \overset{C}{\sim} \theta \wedge \theta \overset{C}{<} \pi \Rightarrow \sigma \overset{C}{<} \pi$

Důkaz:

důkaz jednotlivých částí tvrzení je jednoduchou aplikací podmínek a) a b) definice 2.1 pro ilustraci uvedeme důkaz bodu i)

podle definice 2.1 platí

$$\sigma \overset{C}{<} \theta \wedge \theta \overset{C}{\sim} \pi$$

$$\Leftrightarrow \sigma \overset{C}{\leq} \theta \wedge \neg \sigma \overset{C}{\geq} \theta \wedge \theta \overset{C}{\leq} \pi \wedge \theta \overset{C}{\geq} \pi$$

$$\Rightarrow \sigma \overset{C}{\leq} \pi$$

kdyby $\pi \overset{C}{\leq} \sigma$, potom $\theta \overset{C}{\leq} \pi \overset{C}{\leq} \sigma$, tj. $\theta \overset{C}{\leq} \sigma$, a tedy $\neg \sigma \overset{C}{<} \theta$, což je spor s předpokladem tedy $\neg \pi \overset{C}{\leq} \sigma$, což dohromady s $\pi \overset{C}{\geq} \sigma$ dává $\sigma \overset{C}{<} \pi$

❑

Po definici úrovňového komparátoru, jehož úkolem bude srovnávat jednotlivá ohodnocení na dané úrovni podmínek, můžeme přistoupit k definici hierarchického komparátoru, který bude srovnávat ohodnocení vzhledem k celé hierarchii podmínek.

Definice 2.2: (hierarchický komparátor)

Nechť relace $\overset{H}{<}$ na množině ohodnocení je definována následujícím způsobem (H je hierarchie podmínek, H_l jsou její příslušné úrovně a σ a θ jsou ohodnocení proměnných):

$$\sigma \overset{H}{<} \theta \equiv_{def} \exists k > 0 \ \ \forall l \in \{1,\ldots,k-1\} \ \ \sigma \overset{H_l}{\sim} \theta \ \wedge \sigma \overset{H_k}{<} \theta$$

Potom říkáme, že relace $\overset{H}{<}$ je *hierarchický komparátor*. Podobně jako v předchozí definici můžeme definovat další relace $\overset{H}{\leq}$ a $\overset{H}{\sim}$ takto:

$$\sigma \overset{H}{\leq} \theta \equiv_{def} \forall l \ \ \sigma \overset{H_l}{\leq} \theta$$

$$\sigma \overset{H}{\sim} \theta \equiv_{def} \forall l \ \ \sigma \overset{H_l}{\sim} \theta$$

případně relace $\overset{H}{\geq}$ a $\overset{H}{>}$ takto:

$$\sigma \overset{H}{\geq} \theta \equiv_{def} \theta \overset{H}{\leq} \sigma$$

$$\sigma \overset{H}{>} \theta \equiv_{def} \theta \overset{H}{<} \sigma.$$

Poznamenejme, že definice hierarchického komparátoru v sobě implicitně obsahuje ideu pojmu respektování hierarchie podmínek, tj. silnější podmínky mají přednost před slabšími podmínkami a mají v tomto vztahu právo veta. Dokonce lze říci, že definice 2.2 vystihuje ideu respektování hierarchie podmínek lépe než formální definice 1.2 tohoto pojmu, která byla vytvořena tak trochu uměle. Formálně ovšem netvrdíme, že každý hierarchický komparátor podle definice 2.2 splňuje vlastnost respektování hierarchie podmínek z definice 1.2.

Za povšimnutí také stojí to, že všechny vlastnosti lokálního komparátoru, resp. relací $\overset{C}{\leq}$, $\overset{C}{<}$ a $\overset{C}{\sim}$, které jsou popsány v tvrzení 2.1 a které zachycují podmínky a)-d) definice 2.1, lze přímo přenést na hierarchický komparátor, resp. relace $\overset{H}{\leq}$, $\overset{H}{<}$ a $\overset{H}{\sim}$. Pro důkaz tohoto tvrzení stačí postupně procházet jednotlivé úrovně od nejsilnější po nejslabší a používat odpovídající vlastnosti úrovňového komparátoru.

To, že definice 2.2 hierarchického komparátoru je rozumná, ukazují následující odstavce. Zde ukážeme konkrétní příklady úrovňových komparátorů takových, že příslušný hierarchický komparátor bude odpovídat lokálnímu (definice 1.3) resp. vybraným globálním (definice 1.6) komparátorům. Poznamenejme, že regionální komparátor (definice 1.5) nemá ve zde navrhované teorii obdobu, protože jemu odpovídající úrovňový komparátor nesplňuje podmínku tranzitivity.

Definice 2.3: (lokální úrovňový komparátor)

Říkáme, že komparátor $\overset{C}{\leq}$ je *lokální*, pokud je relace $\overset{C}{\leq}$ definována takto (*e* je chybová funkce):

$$\sigma \overset{C}{\leq} \theta \equiv_{def} \forall c \in C \ \ e(c\sigma) \leq e(c\theta).$$

Všechny podmínky definice 2.1 lokální komparátor triviálně splňuje, a proto můžeme hovořit o lokálním úrovňovém komparátoru. Pokud nyní pomocí tohoto úrovňového komparátoru definujeme příslušný (lokální) hierarchický komparátor, zjistíme, že jeho definice se přesně shoduje s definicí 1.3 lokálního komparátoru. Navrhovaná teorie řešení hierarchií podmínek tedy podporuje běžné lokální komparátory.

Definice 2.4: (globální úrovňový komparátor)

Říkáme, že komparátor $\overset{C}{\leq}$ je *globální*, pokud je relace $\overset{C}{\leq}$ definována takto:

$$\sigma \overset{C}{\leq} \theta \equiv_{def} g(\sigma, C) \leq g(\theta, C),$$

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

a) $g(\sigma, C) = 0 \Leftrightarrow \forall c \in C \;\; c\sigma$ platí
b) $\forall c \in C \;\; e(c\sigma) \leq e(c\theta) \Rightarrow g(\sigma, C) \leq g(\theta, C)$
c) $g(\theta, C \cup C') \leq g(\sigma, C \cup C') \wedge g(\sigma, C') \leq g(\theta, C') \Rightarrow g(\theta, C) \leq g(\sigma, C)$
d) $g(\sigma, C) \leq g(\theta, C) \wedge g(\sigma, C') \leq g(\theta, C') \Rightarrow g(\sigma, C \cup C') \leq g(\theta, C \cup C')$

Poznamenejme, že podmínky b), c), d) definice 2.4 zajišťují splnění odpovídajících podmínek definice 2.1 a že podmínka a) definice 2.1 je triviálně splněna díky tranzitivitě relace ≤ na reálných číslech. Právě definovaný globální komparátor proto splňuje podmínky kladené na úrovňový komparátor a můžeme tak hovořit o globálním úrovňovém komparátoru. Stejně jako v předchozím případě můžeme nyní definovat příslušný (globální) hierarchický komparátor, jehož definice je opět shodná s definicí 1.6 globálního komparátoru.

Podmínka a) definice 2.4 není ve zde prezentované teorii řešení hierarchií nezbytně vyžadována. Její splnění ale zajišťuje, že všechny globální hierarchické komparátory jsou zároveň globálními komparátory podle definice 1.6. Zde definované globální hierarchické komparátory jsou tedy podmnožinou globálních komparátorů. Mezi globální hierarchické komparátory patří například známé komparátory weighted-sum-better a least-squares-better (kapitola 1.2.2), jejichž kombinační funkce *g* vyhovují podmínkách definice 2.4. Naopak globální komparátor worst-case-better není globálním hierarchickým komparátorem, protože jeho kombinační funkce obecně nesplňuje podmínku c) definice 2.4.

Nyní tedy můžeme srovnávat jednotlivá ohodnocení vzhledem k dané hierarchii. Aby bylo možné formálně zachytit způsob řešení hierarchie podmínek, zavádíme pojem operátoru splňování hierarchie podmínek.

Definice 2.5: (operátor splňování hierarchie podmínek)

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

$$S(\Theta, H) = \{\sigma \in \Theta \mid \neg\exists \theta \in \Theta \;\; \theta \overset{H}{<} \sigma\}.$$

Operátor splňování hierarchie podmínek vybírá z dané množiny ohodnocení proměnných pouze taková ohodnocení, která nejlépe splňují danou hierarchii podmínek, resp. taková ohodnocení, která nejsou horší než jiná ohodnocení. Tento operátor nám nyní umožní snadno definovat řešení hierarchie podmínek.

Definice 2.6: (řešení hierarchie podmínek)

Řešením dané hierarchie H podmínek je taková množina ohodnocení, označme ji $S(H)$, 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 množiny H_0.

Poznamenejme, že definice 2.6 společně s definicí 2.5 určuje řešení hierarchie podmínek stejným způsobem jako to činí definice 1.1. Oproti přístupu popsaném v první části této práce tedy zůstává jedinou odlišností definice komparátorů, která je nyní více restriktivní. To nám na druhou stranu umožní dokázat zajímavá tvrzení o efektivním řešení hierarchií podmínek.

Poznamenejme dále, že teorii prezentovanou v této kapitole lze snadno rozšířit tak, aby umožňovala použití různých úrovňových komparátorů na různých úrovních hierarchie. To například klasická teorie hierarchií z [WiBo93] vylučuje.

Následující dvojice tvrzení ukazuje zajímavé vlastnosti operátoru splňování hierarchie podmínek. Při důkazu těchto tvrzení se ukáže potřebnost podmínek c) a d) z definice 2.1.

Tvrzení 2.2 říká, že pokud máme nějaké ohodnocení σ z řešení hierarchie H, které zároveň splňuje všechny podmínky jiné hierarchie H', potom toto ohodnocení patří také do řešení hierarchie vzniklé sjednocením hierarchií H a H'.

Tvrzení 2.2:

$\forall\sigma\in S(\Theta,H)\ \ (\forall c\in H'\ e(c\sigma)=0 \Rightarrow \sigma\in S(\Theta,H\cup H')\ )$, kde σ je ohodnocení, Θ je množina ohodnocení, H a H' jsou hierarchie podmínek a e je chybová funkce.

Důkaz:

sporem

nechť platí předpoklady tvrzení $\sigma\in S(\Theta,H)\ \&\ \forall c\in H'\ e(c\sigma)=0$

pro spor předpokládejme, že $\sigma\notin S(\Theta,H\cup H')$, z definice 2.5 potom dostaneme

$$\exists\theta\in\Theta\ \ \theta \overset{H\cup H'}{<} \sigma \text{ tj. } \exists k>0\ \forall l\in\{1,\ldots,k-1\}\ \ \theta \overset{(H\cup H')_l}{\sim} \sigma \ \wedge\ \theta \overset{(H\cup H')_k}{<} \sigma \qquad \textit{(1)}$$

protože $\forall c\in H'\ e(c\sigma)=0$ dostaneme podle definice 2.1 bodu b)

$$\forall l\ \forall\pi\ \ \sigma \overset{H'_l}{\leq} \pi, \text{ konkrétně tedy } \forall l\ \ \sigma \overset{H'_l}{\leq} \theta \qquad \textit{(2)}$$

použitím bodu c) definice 2.1 na formule (1) a (2) nyní dostáváme

$$\forall l\in\{1,\ldots,k\}\ \ \theta \overset{H_l}{\leq} \sigma \qquad \textit{(3)}$$

1) pokud by platilo $\exists l\in\{1,\ldots,k-1\}\ \neg\sigma \overset{H_l}{\leq} \theta$, potom stačí vzít nejmenší takové l a dostaneme $\theta \overset{H}{<} \sigma$, a tedy $\sigma\notin S(\Theta,H)$, což je ovšem spor s předpokladem tvrzení

2) musí tedy platit druhá možnost, tj.

$$\forall l\in\{1,\ldots,k-1\}\ \ \sigma \overset{H_l}{\leq} \theta \ + \text{spojení s (3) dává } \theta \overset{H_l}{\sim} \sigma \qquad \textit{(4)}$$

podívejme se nyní na úroveň k hierarchií H, H' a H∪H'

2.1) kdyby platilo $\sigma \overset{H_k}{\leq} \theta$, potom spojením s (2) a použitím bodu d) definice 2.1 dostaneme

$\sigma \overset{(H\cup H')_k}{\leq} \theta$, což je ve sporu s (1)

2.2) musí tedy platit druhá možnost, tj. $\neg\sigma \overset{H_k}{\leq} \theta$, potom ale podle (3) dostaneme

$$\theta \overset{H_k}{<} \sigma \qquad \textit{(5)}$$

spojením (4) a (5) dostáváme $\theta \overset{H}{<} \sigma$, a tedy $\sigma \notin S(\Theta, H)$, což je ve sporu s předpokladem tvrzení

Ukázali jsme, že předpoklad $\sigma \notin S(\Theta, H \cup H')$ vždy vede ke sporu, a tedy musí platit $\sigma \in S(\Theta, H \cup H')$.

❑

Následující tvrzení 2.3 ukazuje, že pokud máme nějaké ohodnocení σ z řešení hierarchie H a jiné ohodnocení θ je s ním ekvivalentní, tj. je na každé úrovni hierarchie H je stejně dobré jako ohodnocení σ, potom toto ohodnocení θ také patří do řešení hierarchie H.

Tvrzení 2.3:

$\forall \sigma, \theta \in \Theta$ ($(\sigma \in S(\Theta, H)$ & $\sigma \overset{H}{\sim} \theta) \Rightarrow \theta \in S(\Theta, H)$), kde σ,θ jsou ohodnocení, Θ je množina ohodnocení a H je hierarchie podmínek.

Důkaz:

sporem

nechť platí předpoklady tvrzení $\sigma \in S(\Theta, H)$ & $\sigma \overset{H}{\sim} \theta$ & $\theta \in \Theta$

pro spor předpokládejme, že $\theta \notin S(\Theta, H)$, z definice 2.5 potom dostaneme

$$\exists \pi \in \Theta \;\; \pi \overset{H}{<} \theta \;\text{ tj. }\; \exists k > 0 \;\forall l \in \{1, \ldots, k-1\} \;\; \pi \overset{H_l}{\sim} \theta \;\wedge\; \pi \overset{H_k}{<} \theta \qquad (1)$$

z předpokladu tvrzení $\sigma \overset{H}{\sim} \theta$ dostáváme

$$\forall l \;\; \theta \overset{H_l}{\sim} \sigma \qquad (2)$$

spojením (1) a (2) použitím bodů e) a i) tvrzení 2.1 dostaneme

$$\forall l \in \{1, \ldots, k-1\} \;\; \pi \overset{H_l}{\sim} \sigma \;\wedge\; \pi \overset{H_k}{<} \sigma, \text{ a tedy } \pi \overset{H}{<} \sigma, \text{ tj. } \sigma \notin S(\Theta, H)$$

tj. je ovšem spor s předpokladem tvrzení.

Ukázali jsme, že předpoklad $\theta \notin S(\Theta, H)$ vede ke sporu, a tedy platí $\theta \in S(\Theta, H)$.

❑

V další části se budeme věnovat teoretickým základům efektivní metody řešení hierarchií podmínek. Tato metoda bude založena na rozkladu dané hierarchie podmínek na co nejmenší množiny podmínek (tzv. buňky), které potom budou řešeny postupnou aplikací operátoru splňování hierarchie podmínek. Cílem teoretické části tedy bude ukázat, že ohodnocení získaná postupnou aplikací operátoru splňování hierarchie na posloupnost buněk padnou do množiny řešení (korektnost metody) a že každé ohodnocení z řešení lze tímto způsobem získat (úplnost metody). O tom, že rozklad hierarchie na buňky tak, aby postupná aplikace operátoru splňování hierarchie dala ohodnocení z řešení, nemůže být zcela libovolný, svědčí následující tvrzení.

Tvrzení 2.4:

Existují takové hierarchie podmínek H a H', že neplatí $S(H \cup H') \supseteq S(S(H), H')$ ani $S(H \cup H') \subseteq S(S(H), H')$ (neplatí tak ani $S(H \cup H') = S(S(H), H')$).

Důkaz:

položme H={x=1@weak} a H'={x=2@strong}

potom zřejmě:

S(H)={{x/1}}

$S(H \cup H')$={{x/2}}

S(S(H),H')=S({{x/1}},H')={{x/1}} ❑

Pokud by se podařilo vytvořit rozklad hierarchie podmínek H do posloupnosti hierarchií $B^1,\ldots,B^n$ (tj. $H=B^1\cup\ldots\cup B^n$ a $\forall i\neq j$ $B^i\cap B^j=\emptyset$) tak, aby alespoň platilo $S(S\ldots S(S(\Theta,B^1),B^2),\ldots,B^n)\subseteq S(\Theta,B^1\cup\ldots\cup B^n)$, potom víme, že ohodnocení získaná postupnou aplikací operátoru splňování hierarchie na posloupnost buněk $B^1,\ldots,B^n$ patří do řešení hierarchie H. Jednoduchost důkazu tvrzení 2.4 ovšem ukazuje, že ne každý rozklad je vhodný pro navrhovanou metodu řešení hierarchií.

Následující definice popisuje podmínku, kterou musí posloupnost $B^1,\ldots,B^n$ splňovat, abychom získali požadovanou vlastnost rozkladu. Hierarchie B^i z rozkladu budeme dále nazývat *buňky*.

Definice 2.7: (vlastnost postupného oslabování)

Říkáme, že posloupnost buněk $B^1,\ldots,B^n$ splňuje *vlastnost postupného oslabování*, pokud $\forall i\in\{1,\ldots n-1\}$ platí následující implikace:

$$\exists\sigma\in S(S\ldots S(S(\Theta,B^1),B^2),\ldots,B^{i+1})\cup S(\Theta,B^1\cup\ldots\cup B^{i+1})\ \ \exists c@l\in B^{i+1}\ \ e(c\sigma)>0$$

$$\Rightarrow$$

$$\forall j\leq i\ \ \forall c@l\in B^j\ \ \forall c'@l'\in B^{i+1}\ \ l<l'$$

Vlastnost postupného oslabování zaručuje, že pokud v průběhu výpočtu získáme nějaké ohodnocení takové, že některá podmínka z buňky B^{i+1} není při tomto ohodnocení „beze zbytku" splněna, potom jsou ve všech předcházejích buňkách B^j ($j\leq i$) pouze silnější podmínky (podmínky se silnější preferencí) než jsou podmínky v buňce B^{i+1}. Tato vlastnost nám zaručí, že pokud v průběhu výpočtu narazíme na nesplněnou podmínku, potom její nesplnění není zapříčiněno nějakou slabší nebo stejně silnou podmínkou, která byla splněna v předchozí části výpočtu. Jedná se tak vlastně o „operační" popis myšlenky respektování hierarchie podmínek.

Následující dvojice tvrzení jsou pomocná tvrzení pro větu ukazující, že vlastnost postupného oslabování je postačující podmínkou pro získání podmožiny řešení hierarchie podmínek postupnou aplikací operátoru splňování hierarchie na posloupnost buněk.

Tvrzení 2.5:

Nechť $\theta\in\Theta$, posloupnost buněk $B^1,\ldots,B^n$ splňuje vlastnost postupného oslabování, $\forall i\in\{1,\ldots n\}$ $S(\ldots S(\Theta,B^1),\ldots,B^i)\subseteq S(\Theta,B^1\cup\ldots\cup B^i)$ a $\sigma\in S(\ldots S(\Theta,B^1),\ldots,B^n)$.

Potom platí následující implikace:

$$\forall i\in\{1,\ldots n-1\}\ \ \sigma\overset{B^1\cup\ldots\cup B^{i+1}}{\sim}\theta\Rightarrow\sigma\overset{B^1\cup\ldots\cup B^i}{\sim}\theta.$$

Důkaz:

zvolme libovolné $i\in\{1,\ldots,n-1\}$ a předpokládejme $\sigma\overset{B^1\cup\ldots\cup B^{i+1}}{\sim}\theta$

protože $\sigma\in S(\ldots S(\Theta,B^1),\ldots,B^n)$, potom také $\sigma\in S(\ldots S(\Theta,B^1),\ldots,B^i)$ podle definice 2.5 a dále podle předpokladu věty

$$\sigma\in S(\Theta,B^1\cup\ldots\cup B^i)\qquad\textit{(1)}$$

Rozebereme následující dva případy:

1) nechť $\exists c@l\in B^{i+1}\ e(c\sigma)>0$

podle vlastnosti postupného oslabování jsou potom v B^{i+1} slabší podmínky než v $B^1\cup\ldots\cup B^i$

nechť $m=\min\{l\mid c@l\in B^{i+1}\}$, potom zřejmě $\forall l<m\ (B^1\cup\ldots\cup B^{i+1})_l=(B^1\cup\ldots\cup B^i)_l$

tj. úrovně silnější než m obsahují pouze podmínky z $B^1,\ldots,B^i$

navíc v $B^1,\ldots,B^i$ nejsou podmínky slabší nebo stejně silné než m

proto $\forall l<m\ \ \sigma\overset{(B^1\cup\ldots\cup B^{i+1})_l}{\sim}\theta\Leftrightarrow\sigma\overset{(B^1\cup\ldots\cup B^i)_l}{\sim}\theta$, a tedy $\sigma\overset{B^1\cup\ldots\cup B^i}{\sim}\theta$

2) nechť naopak $\forall c@l \in B^{i+1}\ e(c\sigma)=0$

potom podle bodu b) definice 2.1 a podle definice 2.2 platí

$$\forall \pi\ \sigma \overset{B^{i+1}}{\leq} \pi, \text{ a tedy konkrétně } \sigma \overset{B^{i+1}}{\leq} \theta$$

protože $\sigma \overset{B^1 \cup \ldots \cup B^{i+1}}{\sim} \theta$ můžeme použitím bodu c) definice 2.1 a definice 2.2 ukázat

$$\theta \overset{B^1 \cup \ldots \cup B^i}{\leq} \sigma \qquad (2)$$

kdyby $\neg\sigma \overset{B^1 \cup \ldots \cup B^i}{\leq} \theta$, tj. $\theta \overset{B^1 \cup \ldots \cup B^i}{<} \sigma$, potom $\sigma \notin S(\Theta, B^1 \cup \ldots \cup B^i)$, což je ve sporu s (1)

musí tedy platit $\sigma \overset{B^1 \cup \ldots \cup B^i}{\leq} \theta$, tj. ve spojení s (2) $\theta \overset{B^1 \cup \ldots \cup B^i}{\sim} \sigma$.

❑

Tvrzení 2.5 vlastně říká ještě více, konkrétně za daných předpokladů platí implikace:

$$\sigma \overset{B^1 \cup \ldots \cup B^n}{\sim} \theta \Rightarrow \forall i \in \{1, \ldots n-1\}\ \sigma \overset{B^1 \cup \ldots \cup B^i}{\sim} \theta.$$

Následující tvrzení 2.6 je zobecněním tvrzení 2.3. Říká se v něm, že pokud postupnou aplikací operátoru splňování hierarchie na posloupnost buňek $B^1, \ldots, B^n$ získáme nějaké ohodnocení σ a zároveň máme další ohodnocení θ, které je s ním ekvivalentní na sjednocení buněk $B^1 \cup \ldots \cup B^n$, potom také ohodnocení θ můžeme získat aplikací operátoru splňování hierarchie na danou posloupnost buněk. Předpokladem tohoto tvrzení je, že příslušná posloupnost buněk splňuje vlastnost postupného oslabování a aplikace operátoru splňování hierarchie je korektní, tj. ohodnocení získaná aplikací operátoru splňování hierarchie patří do řešení hierarchie vzniklé sjednocením příslušných buněk.

Tvrzení 2.6

Nechť $\theta \in \Theta$, posloupnost buněk $B^1, \ldots, B^n$ splňuje vlastnost postupného oslabování, $\forall i \in \{1, \ldots n\}\ S(\ldots S(\Theta, B^1), \ldots, B^i) \subseteq S(\Theta, B^1 \cup \ldots \cup B^i)$, $\sigma \in S(\ldots S(\Theta, B^1), \ldots, B^n)$ a $\sigma \overset{B^1 \cup \ldots \cup B^n}{\sim} \theta$. Potom platí $\theta \in S(\ldots S(\Theta, B^1), \ldots, B^n)$.

Důkaz:

indukcí ukážeme, že $\forall i \in \{1, \ldots, n\}\ \theta \in S(\ldots S(\Theta, B^1), \ldots, B^i)$

z důsledku tvrzení 2.5 víme: $\forall i \in \{1, \ldots n\}\ \sigma \overset{B^1 \cup \ldots \cup B^i}{\sim} \theta$

1) i=1 (začátek indukce)

protože $\sigma \overset{B^1}{\sim} \theta$ a $\sigma \in S(\Theta, B^1)$, víme podle tvrzení 2.3, že $\theta \in S(\Theta, B^1)$

2) i+1 (indukční krok)

víme, že $\theta \in S(\ldots S(\Theta, B^1), \ldots, B^i)$ (indukční předpoklad),

$\sigma \in S(\ldots S(\Theta, B^1), \ldots, B^{i+1})$ (přímý důsledek předpokladu tvrzení) a

$$\sigma \overset{B^1 \cup \ldots \cup B^{i+1}}{\sim} \theta \qquad (1)$$

předpokládejme pro spor, že $\theta \notin S(\ldots S(\Theta, B^1), \ldots, B^{i+1})$, z definice 2.5 víme:

$$\exists \pi \in S(\ldots S(\Theta, B^1), \ldots, \mathrm{B}^i)\ \pi \overset{B^{i+1}}{<} \theta \text{ tj. } \exists k > 0\ \forall l \in \{1, \ldots, k-1\}\ \pi \overset{B_l^{i+1}}{\sim} \theta\ \wedge\ \pi \overset{B_k^{i+1}}{<} \theta$$

protože $\pi \overset{B_k^{i+1}}{<} \theta$ víme, že $\exists c \in B_k^{i+1}\ e(c\theta) > 0$ (kdyby $\forall c \in B_k^{i+1}\ e(c\theta) = 0$, pak $\theta \overset{B_k^{i+1}}{\leq} \pi$)

z vlastnosti postupného oslabování dostáváme, že v B^{i+1} jsou slabší podmínky než v $B^1 \cup \ldots \cup B^i$, úrovně z B^{i+1} a $B^1 \cup \ldots \cup B^i$ jsou tak disjunktní, a proto z (1) plyne

$\theta \overset{B^{i+1}}{\sim} \sigma$

dohromady platí $\pi \overset{B_k^{i+1}}{<} \theta \wedge \theta \overset{B_k^{i+1}}{\sim} \sigma$ z čehož podle části i) tvrzení 2.1 plyne $\pi \overset{B_k^{i+1}}{<} \sigma$

zároveň $\forall l < k \;\; \pi \overset{B_l^{i+1}}{\sim} \theta \wedge \theta \overset{B_l^{i+1}}{\sim} \sigma$ z čehož podle části e) tvrzení 2.1 plyne $\pi \overset{B_l^{i+1}}{\sim} \sigma$

složením dostaneme $\pi \overset{B^{i+1}}{<} \sigma$, a tedy $\sigma \notin S(\ldots S(\Theta, B^1), \ldots, B^{i+1})$, což je spor s předpokladem tvrzení

Ukázali jsem, že předpoklad $\theta \notin S(\ldots S(\Theta,B^1),\ldots,B^{i+1})$ vede ke sporu, musí tedy platit $\theta \in S(\ldots S(\Theta,B^1),\ldots,B^{i+1})$.

❑

Nyní je již vše připraveno pro jednu z hlavních vět této práce. Tato věta ospravedlňuje metodu řešení hierarchie podmínek postupnou aplikací operátoru splňování hierarchie na posloupnost buněk rozkladu hierarchie. Jediným předpokladem tohoto tvrzení je to, že daná posloupnost buněk splňuje vlastnost postupného oslabování.

Věta 2.7 (o korektnosti)

Nechť posloupnost buněk $B^1,\ldots,B^n$ splňuje vlastnost postupného oslabování, potom platí $S(\ldots S(\Theta, B^1), \ldots, B^n) \subseteq S(\Theta, B^1 \cup \ldots \cup B^n)$.

Důkaz:

indukcí podle n

1) n=1 (začátek indukce)

$S(\Theta,B^1) \subseteq S(\Theta,B^1)$

2) n+1 (indukční krok)

indukční předpoklad $\forall i \in \{1,\ldots n\} \;\; S(\ldots S(\Theta, B^1), \ldots, B^i) \subseteq S(\Theta, B^1 \cup \ldots \cup B^i)$

sporem

nechť $\exists \sigma \in S(\ldots S(\Theta, B^1), \ldots, B^{n+1})$ tak, že $\sigma \notin S(\Theta, B^1 \cup \ldots \cup B^{n+1})$ ***(1)***

z indukčního předpokladu a (1) víme, že $\sigma \in S(\Theta, B^1 \cup \ldots \cup B^n)$ ***(2)***

Rozeberme následující dva případy:

2.1) nechť $\forall c@l \in B^{n+1}$ $e(c\sigma)=0$

spolu s (2) a tvrzením 2.2 dostáváme $\sigma \in S(\Theta, B^1 \cup \ldots \cup B^{n+1})$, což je ovšem ve sporu s předpokladem (1)

2.2) nechť $\exists c@l \in B^{n+1}$ $e(c\sigma)>0$

podle vlastnosti postupného oslabování jsou tedy v B^{n+1} slabší podmínky než v $B^1 \cup \ldots \cup B^n$; označme $m = \min\{l \mid c@l \in B^{n+1}\}$

z (1) a definic 2.2 a 2.5 dostáváme formuli:

$\exists \pi \in \Theta \;\; \pi \overset{B^1 \cup \ldots \cup B^{n+1}}{<} \sigma$ tj. $\exists k > 0 \; \forall l \in \{1, \ldots, k-1\} \;\; \pi \overset{(B^1 \cup \ldots \cup B^{n+1})_l}{\sim} \sigma \wedge \pi \overset{(B^1 \cup \ldots \cup B^{n+1})_k}{<} \sigma$ ***(3)***

Uvažujme dále následující dva případy:

2.2.1) nechť k<m

potom $\forall j \le k \;\; (B^1 \cup \ldots \cup B^{n+1})_j = (B^1 \cup \ldots \cup B^n)_j$

ze (3) potom dostáváme $\pi \overset{B^1 \cup \ldots \cup B^n}{<} \sigma$ tj. $\sigma \notin S(\Theta, B^1 \cup \ldots \cup B^n)$, což je ovšem ve sporu s (2)

2.2.2) nechť k≥m

víme (z vlastnosti postupného oslabování):

$\forall j < m \;\; (B^1 \cup \ldots \cup B^{n+1})_j = (B^1 \cup \ldots \cup B^n)_j$ a $\forall j \ge m \; (B^1 \cup \ldots \cup B^n)_j = \emptyset$

spojením s (3) dostaneme $\pi \overset{B^1 \cup \ldots \cup B^n}{\sim} \sigma$

podle tvrzení 2.6 potom dostáváme: $\pi \in S(\ldots S(\Theta, B^1), \ldots, B^n)$

protože zároveň $\pi \overset{B^{n+1}}{<} \sigma$ (vlastnost postupného oslabování + (3)), platí $\sigma \notin S(\ldots S(\Theta, B^1), \ldots, B^{n+1})$, což je sporu s (1)

Ukázali jsme, že předpoklad (1) vždy vede ke sporu, a tedy platí:

$\forall \sigma \ (\sigma \in S(\ldots S(\Theta, B^1), \ldots, B^{n+1}) \Rightarrow \sigma \in S(\Theta, B^1 \cup \ldots \cup B^{n+1}))$, tj.

$S(\ldots S(\Theta, B^1), \ldots, B^{n+1}) \subseteq S(\Theta, B^1 \cup \ldots \cup B^{n+1})$.

❑

Věta 2.7 poskytuje návod na efektivní řešení hierarchie podmínek rozkladem do posloupnosti buněk a postupnou aplikací operátoru splňování hierarchie podmínek na tuto posloupnost. Bohužel nám toto tvrzení pouze zajišťuje, že ohodnocení získaná popsaným postupem patří do řešení hierarchie (metoda je korektní), nic ale není řečeno o tom, zda takto najdeme všechna ohodnocení z řešení, tj. nevíme, zda metoda je úplná. Zpřísněním podmínek kladených na úrovňový komparátor a dalšími předpoklady získáme v následující části tvrzení, které již bude zajišťovat nalezení všech ohodnocení z řešení, a tedy úplnost metody.

Definice 2.8: (lineární komparátory)

Říkáme, že *úrovňový komparátor* $\overset{C}{\leq}$ je *lineární*, pokud splňuje následující podmínku:

$$\forall \sigma, \theta \ \ \sigma \overset{C}{\leq} \theta \vee \theta \overset{C}{\leq} \sigma.$$

Říkáme, že *hierarchický komparátor* $\overset{H}{\leq}$ je *lineární,* pokud je definován pomocí lineárního úrovňového komparátoru.

Lineární úrovňový komparátor zajišťuje, že každá dvě ohodnocení jsou porovnatelná. Mezi lineární úrovňové komparátory tedy obecně nemůžeme počítat lokální úrovňové komparátory (definice 2.3), které tuto podmínku nesplňují. Naopak globální úrovňové komparátory (definice 2.4) podmínku linearity triviálně splňují (uspořádání ≤ na reálných číslech je lineární), a jsou tedy lineárními úrovňovými komparátory.

Schopnost porovnat každá dvě ohodnocení se z lineárního úrovňového komparátoru přenáší také na lineární hierachický komparátor, který tak pro každá dvě ohodnocení vždy rozhodne, zda je jedno lepší než druhé ($\overset{H}{<}$) nebo zda jsou obě stejně dobrá (tj. ekvivalentní podle relace $\overset{H}{\sim}$).

Tvrzení 2.8:

Pro libovolná dvě ohodnocení σ a θ a lineární hierarchický komparátor $\overset{H}{<}$ platí:

$$\sigma \overset{H}{<} \theta \vee \sigma \overset{H}{>} \theta \vee \sigma \overset{H}{\sim} \theta.$$

Důkaz:

Vzhledem k linearitě úrovňového komparátoru platí $\forall l \ \ \sigma \overset{H_l}{\leq} \theta \vee \theta \overset{H_l}{\leq} \sigma$ *(1)*.

Pokud pro každou úroveň platí obě nerovnosti (1), potom $\sigma \overset{H}{\sim} \theta$.

V opačném případě vezmeme nejmenší *l* takové, že jedna z nerovností (1) neplatí.

Podle toho, která z nerovností neplatí, dostaneme $\sigma \overset{H}{<} \theta$ resp. $\sigma \overset{H}{>} \theta$.

❑

Díky linearitě navíc můžeme získat následující tvrzení, která lze chápat jako doplněk k tvrzení 2.3

Tvrzení 2.9:

Pro libovolná dvě ohodnocení σ a θ a hierarchii podmínek H platí:

$$\sigma, \theta \in S(\Theta, H) \Rightarrow \sigma \overset{H}{\sim} \theta.$$

Důkaz:

Nechť platí: $\sigma, \theta \in S(\Theta, H)$

Z tvrzení 2.8 víme, že platí jedna z následujících možností:

$$\sigma \overset{H}{<} \theta \vee \sigma \overset{H}{>} \theta \vee \sigma \overset{H}{\sim} \theta.$$

Kdyby platila libovolná z prvních dvou možností, potom buď $\theta \notin S(\Theta, H)$ nebo $\sigma \notin S(\Theta, H)$, což je spor s předpokladem tvrzení.

Musí tedy platit $\sigma \overset{H}{\sim} \theta$.

❑

Linearita komparátoru je natolik silný předpoklad, že můžeme rovnou přistoupit k tvrzení, které zajistí, že metoda postupného řešení hierarchie je úplná. Nejprve ukážeme úplnost na jednom kroku metody.

Tvrzení 2.10:

Nechť B^1,B^2 jsou hierarchie podmínek (buňky) splňující jako posloupnost vlastnost postupného oslabování. Nechť použitý úrovňový komparátor je lineární a $S(S(\Theta,B^1),B^2)\neq\emptyset$. Potom platí: $S(S(\Theta,B^1),B^2)=S(\Theta,B^1\cup B^2)$.

Důkaz:

1) $S(S(\Theta,B^1),B^2)\subseteq S(\Theta,B^1\cup B^2)$ víme z věty 2.7

2) $S(S(\Theta,B^1),B^2)\supseteq S(\Theta,B^1\cup B^2)$

<u>sporem</u>

nechť $\exists\sigma\in S(\Theta,B^1\cup B^2)$ tž. $\sigma\notin S(S(\Theta,B^1),B^2)$ ***(1)***

Rozeberme následující dva případy:

2.1) nechť $\sigma\in S(\Theta,B^1)$

z (1) a definice 2.5 víme $\exists\theta \in S(\Theta, B^1)$ $\theta \overset{B^2}{<} \sigma$

potom ovšem $\exists c@l\in B^2$ tž. $e(c\sigma)>0$ (kdyby $\forall c@l\in B^2$ $e(c\sigma)=0$, pak $\sigma \overset{B^2}{\leq} \theta$)

z vlastnosti postupného oslabování tak víme, že v B^1 jsou silnější podmínky než v B^2 (B^1 a B^2 tedy nemají společné úrovně)

protože $\theta, \sigma \in S(\Theta, B^1)$ víme, že $\theta \overset{B^1}{\sim} \sigma$ podle tvrzení 2.9

dohromady tedy $\theta \overset{B^1 \cup B^2}{<} \sigma$, tj. $\sigma \notin S(\Theta, B^1 \cup B^2)$, což je ovšem ve sporu s (1)

2.2) nechť $\sigma\notin S(\Theta,B^1)$ ***(2)***

protože $S(S(\Theta,B^1),B^2)\neq\emptyset$ víme, že

$\exists\theta\in S(S(\Theta,B^1),B^2)$ (také $\theta\in S(\Theta,B^1)$) ***(3)***

podle tvrzení 2.8 platí jedna z následujících možností:

$$\sigma \overset{B^1}{<} \theta \vee \sigma \overset{B^1}{>} \theta \vee \sigma \overset{B^1}{\sim} \theta$$

Uvažujme tedy následující tři případy:

2.2.1) nechť $\sigma \overset{B^1}{<} \theta$, potom $\theta \notin S(\Theta, B^1)$, což je ve sporu s (3)

2.2.2) nechť $\sigma \overset{B^1}{\sim} \theta$, potom $\sigma \in S(\Theta, B^1)$, což je ve sporu s (2)

2.2.3) nechť $\sigma \overset{B^1}{>} \theta$ ***(4)***

víme $\theta\in S(S(\Theta,B^1),B^2)\subseteq S(\Theta,B^1\cup B^2)$ ((3) a část 1), tedy $\theta\in S(\Theta,B^1\cup B^2)$

protože $\sigma,\theta \in S(\Theta, B^1 \cup B^2)$, platí podle tvrzení 2.9 $\sigma \overset{B^1 \cup B^2}{\sim} \theta$

ve spojení s (4) a podle definice 2.1 bodů c) a d) platí $\sigma \overset{B^2}{<} \theta$

potom ovšem $\exists c@l \in B^2$ tž. $e(c\theta)>0$ (kdyby $\forall c@l \in B^2$ $e(c\theta)=0$, pak $\theta \overset{B^2}{\leq} \sigma$)
z vlastnosti postupného oslabování tak víme, že v B^1 jsou silnější podmínky než v B^2 (B^1 a B^2 tedy nemají společné úrovně)
protože $\sigma \overset{B^1}{>} \theta$ je také $\sigma \overset{B^1 \cup B^2}{>} \theta$, a tedy $\sigma \notin S(\Theta, B^1 \cup B^2)$, což je spor s (1)

Ukázali jsme, že předpoklad (1) vždy vede ke sporu. Pro každé σ tedy platí:
$\sigma \in S(\Theta, B^1 \cup B^2) \Rightarrow \sigma \in S(S(\Theta, B^1), B^2)$, tj. $S(\Theta, B^1 \cup B^2) \subseteq S(S(\Theta, B^1), B^2)$

Dohromady jsme tedy ukázali, že:
$S(S(\Theta, B^1), B^2) \subseteq S(\Theta, B^1 \cup B^2)$ a $S(S(\Theta, B^1), B^2) \supseteq S(\Theta, B^1 \cup B^2)$,
tj. $S(S(\Theta, B^1), B^2) = S(\Theta, B^1 \cup B^2)$.

❑

Tvrzení 2.10 ukazuje, že jeden krok metody postupného řešení hierarchie podmínek aplikací operátoru splňování hierarchie je korektní a úplný. Kromě linearity komparátoru stojí za povšimnutí další z předpokladů věty, totiž $S(S(\Theta, B^1), B^2) \neq \emptyset$. Neprázdnost množiny řešení resp. existence řešení hierarchie je podrobněji rozebírána v kapitole 1.2.3.

Nyní je možné přistoupit k tvrzení o korektnosti a úplnosti navrhované metody řešení hierarchií.

Věta 2.11 (o úplnosti)
Nechť posloupnost buněk $B^1, \ldots, B^n$ splňuje vlastnost postupného oslabování a $S(\ldots S(\Theta, B^1), \ldots, B^n) \neq \emptyset$. Nechť použitý úrovňový komparátor je lineární. Potom platí: $S(\ldots S(\Theta, B^1), \ldots, B^n) = S(\Theta, B^1 \cup \ldots \cup B^n)$.

Důkaz:
indukcí podle n
1) n=1 (začátek indukce)
$S(\Theta, B^1) = S(\Theta, B^1)$
2) n+1 (indukční krok)
víme $S(\ldots S(\Theta, B^1), \ldots, B^n) = S(\Theta, B^1 \cup \ldots \cup B^n)$ (indukční předpoklad)
tedy $S(S(\ldots S(\Theta, B^1), \ldots, B^n), B^{n+1}) = S(S(\Theta, B^1 \cup \ldots \cup B^n), B^{n+1}) \neq \emptyset$
podle tvrzení 2.10 ovšem víme:
$$S(S(\Theta, B^1 \cup \ldots \cup B^n), B^{n+1}) = S(\Theta, B^1 \cup \ldots \cup B^n \cup B^{n+1})$$
dohromady tedy: $S(S(\ldots S(\Theta, B^1), \ldots, B^n), B^{n+1}) = S(\Theta, B^1 \cup \ldots \cup B^n \cup B^{n+1})$

❑

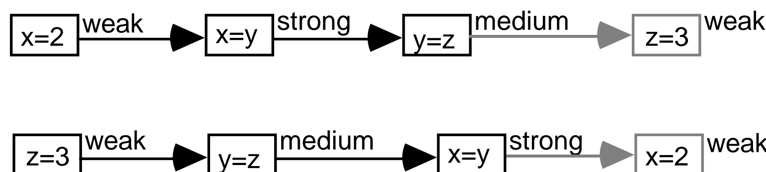
Věty 2.7 a 2.11 poskytují teoretické základy pro řadu metod řešení hierarchie podmínek rozkladem do posloupnosti buněk a následnou postupnou aplikaci operátoru splňování hierarchie.

Nejjednodušší rozklad hierarchie splňující podmínku postupného oslabování je přirozeně „rozklad“ do jediné buňky. Tento rozklad ovšem neposkytuje žádný návod na řešení hierarchie, které se provede v jednom „super“ kroku aplikací operátoru splňování hierarchie na celou hierarchii.

Další triviální rozklad splňující vlastnost postupného oslabování je rozklad do jednotlivých úrovní. Postupná aplikace operátoru splňování hierarchie potom odpovídá

zjemňovací (refining) metodě, která hierarchii řeší od nejsilnějších podmínek po nejslabší. Navrhovaný teoretický podklad metody řešení hierarchií podmínek tedy zahrnuje i zjemňovací metodu. Vzhledem k již několikrát zmíněné neefektivnosti této metody budou ale zajímavější další instance navrhované teorie, které budou používat menší buňky než jsou celé úrovně. Čím jemněji se totiž podaří hierarchii rozložit, tím více bude možné využívat principů lokální propagace a tím efektivnější bude algoritmus řešení hierarchie.

V této kapitole navržená teorie řešení hierarchií podmínek vychází z principů lokální propagace (kapitola 1.5), kdy jsou ohodnocení postupně propagována a filtrována grafem podmínek. Měla by tedy také poskytovat teoretické základy pro různé algoritmy lokální propagace. Vezmeme-li ale klasické grafy podmínek (kapitoly 1.5.3-1.5.5), zjistíme, že není možné přímo ztotožnit uzly grafu, tj. podmínky, s buňkami z teorie. Grafy podmínek totiž v sobě implicitně obsahují výběr některého z ohodnocení z řešení, jak to ukazuje následující obrázek, kde jsou zobrazeny dva různé grafy podmínek reprezentující dvě různá řešící ohodnocení téže hierarchie (nesplněné podmínky a do nich vedoucí hrany jsou šrafované.



Vezmeme-li nyní uzly grafů topologicky uspořádané, dostaneme dvě posloupnosti buněk (první posloupnost odpovídá hornímu grafu):

{x=2@weak},{x=y@strong},{y=z@medium},{z=3@weak}

{z=3@weak},{y=z@medium},{x=y@strong},{x=2@weak}

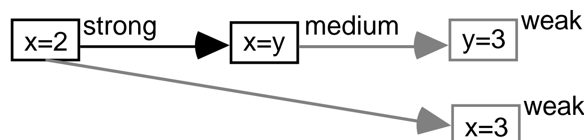
Ani jedna z těchto posloupností ovšem nesplňuje vlastnost postupného oslabování (problém je u weak podmínek). Existují dva stejně jemné rozklady podmínek do buněk, které tuto vlastnost splňují:

{x=y@strong},{y=z@medium},{x=2@weak,z=3@weak}

{y=z@medium},{x=y@strong},{x=2@weak,z=3@weak}.

Za pozornost stojí to, že zatímco první z těchto rozkladů odpovídá rozkladu do úrovní, a tedy vlastnost postupného oslabování je triviálně splněna, druhý, stejně dobrý rozklad nezachovává pořadí úrovní a přesto také splňuje vlastnost postupného oslabování. To ukazuje na to, že zde navrhovaná teorie řešení hierarchií je obecnější než zjemňovací metoda. Oba rozklady přirozeně umožňují nalézt všechna řešící ohodnocení.

Navrhovaná metodologie pro řešení hierarchií podmínek, jak ji popisují věty 2.7 a 2.11, je ve svém principu lineární. Tím je míněno, že představuje způsob řešení typu krok za krokem, bez jakýchkoliv větvení a navracení. Grafy podmínek ale naopak různých větvení s úspěchem používají, protože umožňují jemněji rozložit hierarchii podmínek do nezávislých částí. Příklad takto „rozvětveného“ grafu podmínek ukazuje následující obrázek.



Opět není možné přímo ztotožnit jednotlivé uzly zobrazeného grafu podmínek s buňkami. Ani jedno z topologických uspořádání uzlů, tj.:

{x=2@strong},{x=y@medium},{y=3@weak},{x=3@weak}

{x=2@strong},{x=y@medium},{x=3@weak},{y=3@weak}

{x=2@strong},{x=3@weak},{x=y@medium},{y=3@weak}

totiž nesplňuje vlastnost postupného oslabování (problém je vždy u posledních weak podmínek). Nejjemnější rozklad zobrazené hierarchie splňující vlastnost postupného oslabování je například triviální rozklad do úrovní:

{x=2@strong},{x=y@medium},{y=3@weak,x=3@weak}.

Na rozdíl od předchozího příkladu, kdy nám spojení podmínek do jedné buňky umožnilo reprezentovat všechna řešení, zde toto chování není žádoucí, protože podmínky y=3@weak a x=3@weak lze řešit nezávisle (vzhledem k řešení silnějších podmínek), a bylo by proto vhodné, aby obě podmínky tvořily samostatné buňky.

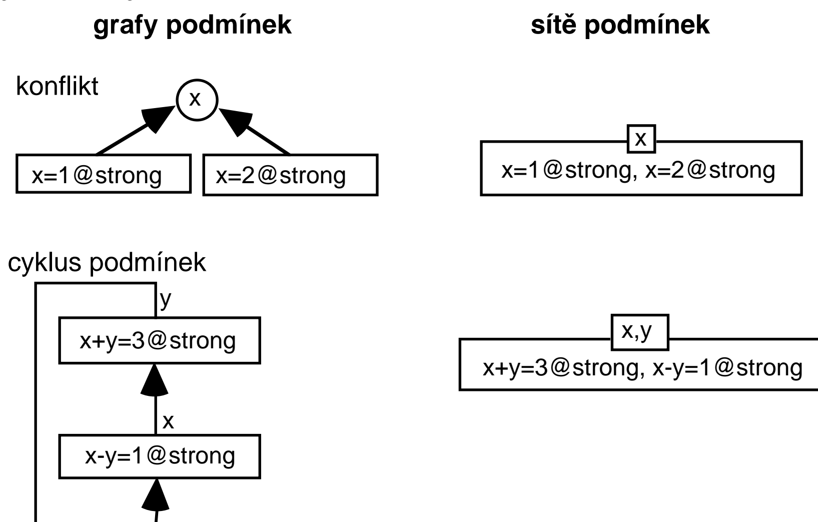
V následující kapitole je z právě nastíněného důvodu zaveden pojem sítě podmínek, který podporuje jemnější rozklady do buněk při zachování korektnosti řešení metodou postupného splňování podmínek z buněk. Je zde také popsána idea, která tyto jemnější rozklady ospravedlňuje, tj. ukazuje, že i když není zcela splněna vlastnost postupného oslabování, jsou ohodnocení získaná postupným průchodem sítě podmínek z množiny řešení hierarchie.

2.3.2 Buňky a sítě podmínek

Sítě podmínek navrhované v této práci vznikly zobecněním grafů podmínek, které jsou úzce spjaty s algoritmy lokální propagace (kapitoly 1.5.3-1.5.5). Toto zobecnění má za cíl překonat problémy grafů podmínek resp. lokální propagace při zachování efektivity řešení hierarchie podmínek. Pro připomenutí zde stručně zopakujeme typické problémy lokální propagace a grafů podmínek, pro jejichž řešení neposkytují klasické grafy podmínek dostatečně silné prostředky:

- konflikty mezi podmínkami nejsou obecně vyřešeny
- lokální propagace neumí řešit cykly podmínek
- lokální propagace pracuje pouze s rovnosti resp. funkčními podmínkami
- lokální propagace podporuje pouze locally-predicate-better komparátory
- lokální propagace nehledá alternativní řešení.

Většina problémů klasických grafů podmínek a tedy i klasických algoritmů lokální propagace je vyřešena zavedením pojmu *buňky podmínek*. Protože buňka může obsahovat více podmínek, lze tak přirozeně řešit konflikty podmínek i cykly podmínek, jak to ukazuje následující obrázek.



Zapouzdření více navzájem souvisejících podmínek do jediné buňky také umožňuje použití většího spektra komparátorů včetně komparátorů globálních.

Sítě podmínek mohou obsahovat tzv. *funkční* buňky, což je jediný typ buněk povolený v klasických grafech podmínek. Funkční buňka je tvořena jedinou funkční

podmínkou, tj. takovou podmínkou, která „jednoznačně“ vypočte hodnotu výstupní proměnné (proměnných) z libovolných hodnot vstupních proměnných tak, aby podmínka byla splněna. Poznamenejme také, že funkční buňka nijak nemění hodnoty vstupních proměnných. Tyto vlastnosti propagace ohodnocení přes funkční buňku lze formálně popsat takto:

Nechť Θ_S je množina ohodnocení před vstupem do funkční buňky, Θ_E je množina ohodnocení po propagaci funkční buňkou (platí $\Theta_S \supseteq \Theta_E$) a In resp. Out jsou množiny vstupních resp. výstupních proměnných dané funkční buňky. To, že je funkční buňka vždy splněna, vyjadřuje následující formule:

$$\Theta_E \neq \emptyset \ \& \ \forall \sigma \in \Theta_E \ \ c\sigma \text{ platí,} \qquad (1)$$

kde c je podmínka z funkční buňky

Neměnnost hodnot vstupních proměnných zase vyjádříme formálně takto:

$$\forall \sigma \in \Theta_S \ \exists \theta \in \Theta_E \ \sigma{\downarrow}In=\theta{\downarrow}In, \qquad (2)$$

kde $\sigma{\downarrow}V$ znamená restrikci ohodnocení σ na proměnné z V, např. $\{x/1,y/2,z/3\}{\downarrow}\{x,z\}=\{x/1,z/3\}$.

Jednoznačné ohodnocení výstupních proměnných formálně vyjadřuje následující formule (poznamejme, že oproti (2) jsou tentokrát obě ohodnocení z Θ_E):

$$\forall \sigma,\theta \in \Theta_E \ \ \sigma{\downarrow}In=\theta{\downarrow}In \ \Rightarrow \ \sigma{\downarrow}Out=\theta{\downarrow}Out. \qquad (3)$$

Poznamenejme, že podmínka ve funkční buňce je splněna nezávisle na hodnotách vstupních proměnných a jedná se tak vlastně o totální funkci.

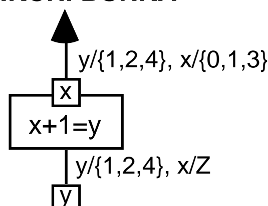
Nově jsou v sítích podmínek zavedeny buňky *generativní* a *testovací*, které rozšiřují prostředky řešení hierarchií Jedná se o zobecnění funkčních buněk, které nevyžaduje, aby podmínka v buňce byla funkční. Buňka dokonce může obsahovat více podmínek.

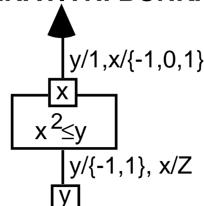
Generativní buňky se podobají buňkám funkčním tím, že mohou přiřazovat hodnoty výstupních proměnných. Na dané ohodnocení vstupních proměnných ale mohou generovat více ohodnocení výstupních proměných tak, že podmínky v buňce jsou splněny. Odtud pochází název generativní buňka.

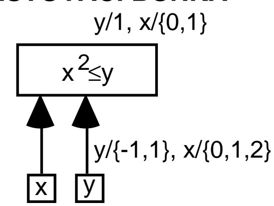
Testovací buňky se od buněk generativních liší vlastně jen v tom, že nemají žádné výstupní proměnné. Místo výpočtu hodnot výstupních proměnných tak pouze testují splnitelnost podmínek vzhledem k hodnotám vstupních proměnných (odtud testovací buňky). Mohou také omezit hodnoty vstupních proměnných tak, aby podmínky v buňce byly splněny.

V generativních a testovacích buňkách se mohou nacházet libovolné podmínky, tedy třeba i rovnosti. Poznamenejme dále, že tyto buňky mohou ovlivňovat také vstupní proměnné, tj. neplatí v nich podmínka (2). Podmínky z těchto buněk se dokonce nemusí podařit při daném ohodnocení vstupních proměnných splnit a neplatí v nich tedy ani podmínka (1). Navíc generativní buňky nemusí přiřazovat hodnoty výstupních proměnných jednoznačně při daných vstupních proměnných, tj. nemusí splňovat podmínku (3).

Příklady jednotlivých buněk ukazuje následující obrázek, ve kterém je také naznačen způsob propagace hodnot proměnných přes jednotlivé buňky.







Jednotlivé buňky jsou spojeny do sítě, tj. tvoří uzly orientovaného grafu. Tento graf je vytvářen a udržován v plánovací fázi algoritmu řešení hierarchie podmínek. Poznamenejme, že síť podmínek, a tedy i plánovací fáze, je zcela nezávislá na používaném komparátoru a že je díky různým typům buněk schopna pojmout libovolnou podmínku. Plánovacím algoritmům je věnována kapitola 2.3.3.

Síť podmínek je ve druhé, tzv. prováděcí fázi algoritmu pro řešení hierarchií procházena a použitím konkrétního komparátoru a systému pro řešení podmínek jsou ohodnocení propagována sítí podmínek. V praktické implementaci se sítí podmínek nepropagují ohodnocení, protože jich je příliš mnoho resp. nekonečně, ale přípustné množiny hodnot proměnných (viz. obrázek na předchozí stránce). Toto zjednodušení potom vyžaduje zpětnou kontrolu splnitelnosti podmínek v již prošlých buňkách při změně množiny hodnot vstupních proměnných v generativní nebo testovací buňce. Podrobněji je prováděcí fáze včetně zpětné kontroly diskutována v kapitole 2.3.4.

Následující definice formálně zavádí pojmy buňky, typu buňky a sítě podmínek.

Definice 2.9: (buňka podmínek)

Nechť C je konečná neprázdná množina označených podmínek se stejnou preferencí a V je množina všech proměnných z těchto podmínek. Pro libovolné množiny proměnných $\mathrm{In},\mathrm{Out}\subseteq V$ takových, že $\mathrm{In}\cup\mathrm{Out}=V$ a $\mathrm{In}\cap\mathrm{Out}=\emptyset$ definujeme *buňku podmínek* jako trojici (C,In,Out).

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

Množiny *In* a *Out* z buňky (C,In,Out) nazýváme *vstupní* resp. *výstupní* proměnné.

Říkáme také, že buňka (C,In,Out) *určuje* (determinuje) každou proměnnou z množiny *Out*.

Definice 2.10: (klasifikace buněk podmínek)

Buňky podmínek rozdělujeme na následující typy:

- *volná proměnná* ({},{},{v})
- *funkční buňka* je taková buňka podmínek ({c@l},In,Out), že $\mathrm{Out}\neq\emptyset$ 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 $\mathrm{In}=\emptyset$, potom chceme, aby existovalo právě jedno ohodnocení σ výstupních proměnných z Out takové, že $c\sigma$ platí.
- *generativní buňka* je taková buňka podmínek (C,In,Out), že $C\neq\emptyset$, $\mathrm{Out}\neq\emptyset$ a (C,In,Out) není funkční podmínka
- *testovací buňka* (C,In,$\emptyset$)
- *nerozhodnutelná buňka* je buď generativní nebo testovací buňka.

Volné proměnné a funkční buňky jsou známé z klasických grafů podmínek, generativní a testovací buňky jsou nově zaváděny v této práci. Zatímco o podmínkách z funkčních buněk vždy víme, že jsou splněny nezávisle na hodnotách vstupních proměnných, u generativních a testovacích buněk toto rozhodnutí nemůžeme v plánovací fázi učinit. Proto je zde zaveden shrnující pojem nerozhodnutelné buňky, který vyjadřuje právě to, že během plánování nejsme schopni rozhodnout o splnitelnosti podmínek z těchto buněk. Poznamenejme dále, že nerozhodnutelné buňky mohou na rozdíl od funkčních buněk omezovat při propagaci také vstupní proměnné.

Než přistoupíme k definici sítě podmínek, zavedeme pojem vnitřní síly (preference) buňky. Protože všechny podmínky v buňce mají stejnou preferenci, je možné tuto preferenci svázat s celou buňkou.

Definice 2.11: (vnitřní síla buňky)

Vnitřní sílu (preferenci) buňky (C,In,Out) definujeme jako preferenci libovolné označené podmínky z C, pokud je množina C neprázdná. V opačném případě, tj. u volných proměnných ({},{},{v}), definujeme *vnitřní sílu* jako „free“, což je preference, která je slabší než libovolná jiná preference u podmínek.

Definice 2.12: (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 obsahujících pouze podmínky z *H*, tj.
   $\forall$Cell$\in$CC Cell=(C,In,Out) & 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) & 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) & 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) & Cell'=(C',In',Out') & In$\cap$Out'$\neq\emptyset$ $\Rightarrow$ (Cell',Cell)$\in$E
6) pro každou nerozhodnutelnou buňku *Cell* neexistuje buňka se stejnou nebo slabší vnitřní sílou, která by se nacházela proti proudu od *Cell*, tj.
   $\forall$Cell$\in$CC
   Cell je nerozhodnutelná $\Rightarrow$ $\forall$Cell'$\in$CC tž. v grafu existuje orientovaná cesta z Cell' do Cell (Cell' je proti proudu od 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.
   $\forall$Cell,Cell'$\in$CC
   Cell a Cell' jsou nerozhodnutelné & v grafu neexistuje orientovaná cesta z Cell do Cell' ani z Cell' do Cell
   $\Rightarrow$
   $\forall$Cell''$\in$CC tž. Cell'' je proti proudu jak od Cell tak od Cell',
   Cell'' není nerozhodnutelná (tj. je to funkční buňka).

Prvních pět bodů z definice 2.12 je obdobou předpokladů z klasických grafů omezujících podmínek rozšířených tak, aby zahrnovaly pojem buňky podmínek. Sítě podmínek jsou tak zobecněním grafů podmínek. Nově přidané předpoklady 6) a 7) se týkají především nerozhodnutelných buněk a jsou novým přínosem sítí podmínek. Právě tato dvojice podmínek zajišťuje „korektnost“ algoritmů používajících pro řešení hierarchií sítě podmínek.

Algoritmy pro řešení hierarchií založené na sítích podmínek se skládají ze dvou fází. První fází je plánování (kapitola 2.3.3), kdy je hierarchie podmínek, tj. konečná množina označených omezujících podmínek převedena na odpovídající síť podmínek. Druhá tzv. prováděcí fáze (kapitola 2.3.4) potom prochází vytvořenou síť podmínek a propaguje přes ní množiny hodnot proměnných. Jinými slovy prováděcí fáze postupně omezuje množinu ohodnocení tak, že na konci v ní zbydou pouze řešící ohodnocení. Protože buňky jsou v prováděcí fázi procházeny postupně tak, že buňka je zpracována

teprve tehdy, když jsou zpracovány všechny buňky na hranách vedoucích do této buňky, můžeme procházení sítě linearizovat topologickým setříděním buněk podle orientovaných hran. Tím dostaneme posloupnost buněk a můžeme tak využít teoretických výsledků z kapitoly 2.3.1. Zpracování buňky resp. propagace hodnot přes buňku totiž není nic jiného než aplikace operátoru splňování hierarchie. Pokud vzniklá posloupnost buněk splňuje vlastnost postupného oslabování, můžeme bez úprav použít výsledků vět 2.7 a 2.11 o korektnosti a úplnosti algoritmu řešení hierarchie podmínek. Jak ale bylo naznačeno v závěru kapitoly 2.3.1 sítě podmínek jsou obecnější a ne každá posloupnost vzniklá uspořádáním buněk sítě splňuje poměrně přísnou vlastnost postupného oslabování. V následujících odstavcích proto ukážeme základní myšlenky, proč splnění podmínek 6) a 7) definice 2.12 zajišťuje korektnost navrhované metody řešení hierarchií.

Nejprve je třeba si uvědomit, že každá omezující podmínka omezuje pouze proměnné, které se v této podmínce vyskytují. To jinými slovy znamená, že pokud máme dvě ohodnocení, která jsou na všech proměnných dané hierarchie stejná, potom jsou tato ohodnocení ekvivalentní. Formálně můžeme toto tvrzení vyjádřit takto:

$$\sigma \downarrow \mathrm{var}(H) = \theta \downarrow \mathrm{var}(H) \Rightarrow \sigma \overset{H}{\sim} \theta, \quad (4)$$

[1]

kde var(*H*) jsou všechny proměnné z podmínek v H a $\sigma\downarrow V$ znamená restrikci ohodnocení σ na proměnné z V.

Důsledkem (4) podle tvrzení 2.3 je potom to, že pokud jedno z dvojice ohodnocení identických na proměnných hierarchie padne do řešení této hierarchie, potom tam padne i druhé ohodnocení. To znamená, že máme-li nějaké ohodnocení σ, které je z řešení hierarchie *H*, potom každé ohodnocení θ lišící se od σ pouze v ohodnocení proměnných nenacházejících se v podmínkách z *H* (jinými slovy $\sigma\downarrow\mathrm{var}(H)=\theta\downarrow\mathrm{var}(H)$) padne také do řešení hierarchie *H*. Tím jsme stručně ukázali, že podmínky omezují pouze vlastní proměnné, zatímco ostatní proměnné zůstávají volné.

Důsledkem tvrzení předchozích odstavců je to, že máme-li dvě hierarchie H a H’ takové, že nemají žádné společné proměnné, potom lze řešení spojené hierarchie $H\cup H'$ získat přímo z řešení jednotlivých hierarchií takto:

$$\mathrm{var}(H)\cap\mathrm{var}(H')=\emptyset \;\Rightarrow\; S(H\cup H') = S(H)\cap S(H') = S(S(H),H') = S(S(H'),H). \quad (5)$$

Toto tvrzení (bez formálního důkazu) také říká, že nezáleží na pořadí v jakém budeme hierarchie H a H’ řešit.

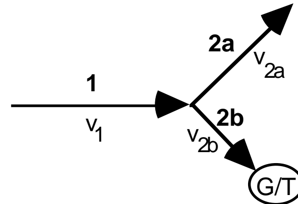
Nyní je možné přistoupit k popisu základní myšlenky korektnosti metody řešení hierarchií podmínek propagací ohodnocení přes buňky sítě podmínek. V definici vlastnosti postupného oslabování hrála důležitou roli buňka obsahující nějakou nesplněnou podmínku. Přítomnost takové buňky potom vyžadovala, aby se v poslupnosti buněk před ní vyskytovaly pouze buňky se silnějšími podmínkami. Tím bylo zaručeno, že nesplnění podmínky v inkriminované buňce nebylo způsobeno splněním nějaké slabší podmínky v některé předchozí buňce.

Vzhledem k definici 2.10 mohou být jedinými buňkami obsahujícími nesplněnou podmínku nerozhodnutelné buňky. Funkční podmínky lze totiž při libovolném ohodnocení vstupních proměnných vždy splnit (viz. (1)) a protože výstupní proměnné funkčních buněk se v předchozím výpočtu (tj. předchozích buňkách v topologickém uspořádání) nikde nevyskytovaly, zůstaly podle úvah z předchozích odstavců tyto proměnné volné. Podmínka z funkční buňky tak může „navázat“ výstupní proměnné tak, aby byla splněna.

---

[1] Důkaz je přímým důsledkem definic 2.1 a 2.2.

Splnění bodu 6) definice 2.12 zaručuje, že se na cestě před nerozhodnutelnou buňkou vyskytují pouze silnější buňky. Jedinou možností jak by se před tuto buňku mohla v topologickém uspořádání buněk dostat slabší nebo stejně silná buňka je, že tato buňka „přišla“ z paralelní větve. Obecně tuto situaci zachycuje následující obrázek, ve kterém je nerozhodnutelná buňka s nesplněnou podmínkou ve větvi 2b. Proměnné z podmínek z buněk v jednotlivých větvích jsou označeny v_1,v_{2a} a v_{2b}.



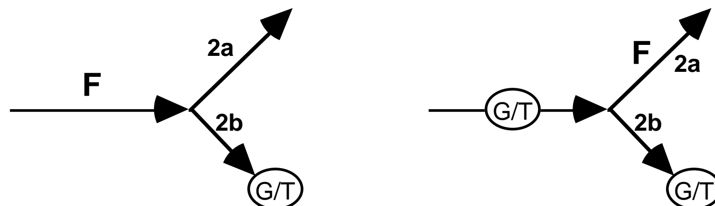
Protože mezi větvemi 2a a 2b nevedou žádné hrany, víme podle podmínky 5) definice 2.12, že platí:

$$v_{2a} \cap v_{2b} \subseteq v_1 \qquad (6)$$

tj. jediné společné proměnné podmínek z větví 2a a 2b pocházejí z podmínek větve 1. Pokud se nám podaří ukázat, že splňování podmínek z větve 2a neovlivňuje proměnné v_{2b}, víme z (5), že podmínky z větve 2a můžeme řešit před podmínkami z větve 2b, aniž bychom nějak ovlivnili řešení větve 2b. Vzhledem ke vztahu (6) stačí ukázat, že řešení podmínek z větve 2a neovlivní proměnné z v_1.

Poznamenejme, že úsek 1 může být prázdný, čímž dostáváme dvě paralelní větve 2a a 2b bez společných proměnných. To, že větev 2a potom neovlivňuje proměnné v_{2b} je zřejmé.

Nechť je tedy větev 1 neprázdná. Vzhledem k tomu, že síť splňuje podmínku 7) definice 2.12 mohou nyní nastat jen dva případy. Buď jsou ve větvi 1 samé funkční buňky nebo větev 1 obsahuje nějakou nerozhodnutelnou buňku. Ve druhém případě potom víme, že větev 2a neobsahuje nerozhodnutelnou buňku a jsou v ním tedy samé funkční buňky.



Pokud větev 1 obsahuje pouze funkční buňky, víme, že jsou všechny podmínky z těchto buněk jednoznačně splněny (viz. (3)). Všechny proměnné z v_1 tedy mají po průchodu větví 1 přiřazenu právě jednu hodnotu. Následné řešení podmínek z větve 2a potom již nemůže ovlivnit ohodnocení proměnných z v_1.

Ve druhém případě obsahuje větev 1 nějakou nerozhodnutelnou buňku. Vzhledem ke splnění podmínky 7) definice 2.12 víme, že ve větvi 2a jsou pouze funkční buňky. O splňování podmínek z funkčních buněk ovšem víme, že neovlivňuje vstupní proměnné těchto buněk (viz. (2)). Ve větvi 2a jsou tedy ovlivňovány pouze proměnné z v_{2a}–v_1, což jinými slovy znamená, že proměnné z v_1 nejsou ovlivněny.

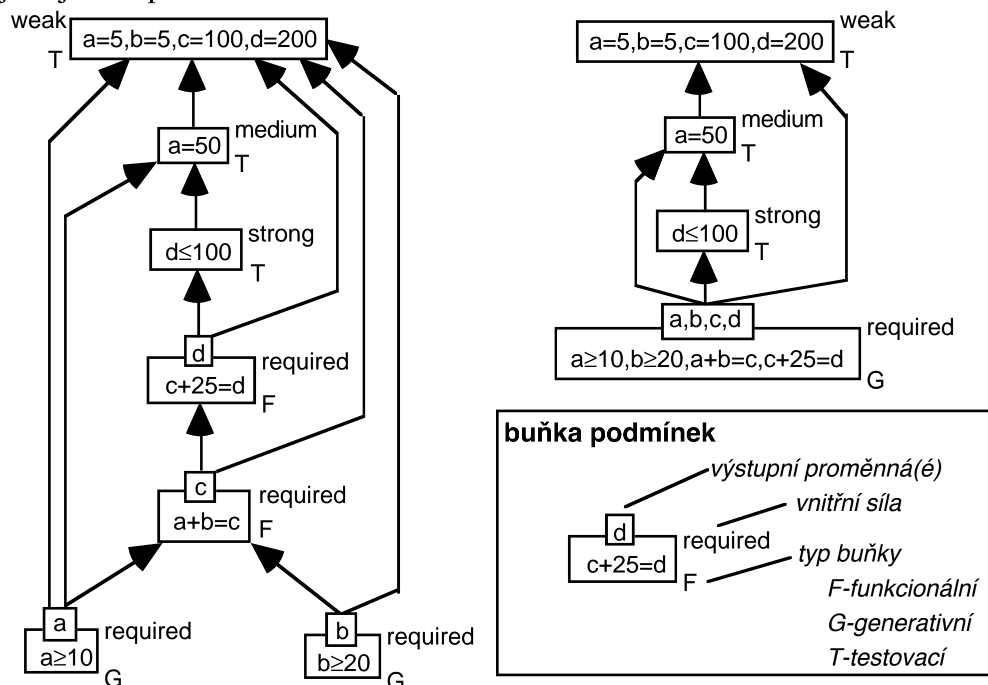
Ukázali jsme, že průchod buňkami větve 2a neovlivňuje řešení větve 2b. Případné nesplnění vlastnosti postupného oslabování předřazením slabší nebo stejně silné buňky z větve 2a před nerozhodnutelnou buňku větve 2b tedy nemá podle (5) na korektnost výpočtu vliv.

Myšlenky prezentované ve této kapitole představují kostru důkazu korektnosti metody řešení hierarchií založené na propagaci ohodnocení sítěmi podmínek. Formální důkaz je jen podrobnějším rozpracováním tohoto postupu.

2.3.3 Tvorba sítí podmínek (plánovací fáze)

Definice 2.12 sítí podmínek je natolik obecná, že umožňuje jednu hierarchii reprezentovat různými sítěmi podmínek. Je proto možné vytvořit řadu různých korektních plánovacích algoritmů, které budou vytvářet a udržovat sítě podmínek. V této kapitole představíme dva takové plánovací algoritmy, z nichž jeden je triviální a odpovídá vlastně plánování pro zjemňovací metodu řešení hierarchií (všechny podmínky se stejnou silou jsou zde shromažďovány do jediné buňky). Druhý plánovací algoritmus je více sofistikovaný. Tento algoritmus se snaží síť podmínek maximálně strukturalizovat, tj. udržuje minimálně možný počet podmínek v buňkách. Více strukturalizované sítě podmínek umožňují lepší využití metod lokální propagace, což přináší větší efektivitu prováděcí fáze.

Následující obrázek ukazuje dvě sítě podmínek reprezentující stejnou hierarchii tak, jak je vytvořily právě zmíněné algoritmy (síť napravo odpovídá plánovacímu algoritmu zjemňovací metody). Buňky v sítích jsou indexovány svojí vnitřní silou, typem a jsou v nich vyznačeny výstupní proměnné (viz. legenda). Předpokládejme dále, že pracujeme s preferencemi required, strong, medium a weak, kde required je nejsilnější preference a weak je nejslabší preference.



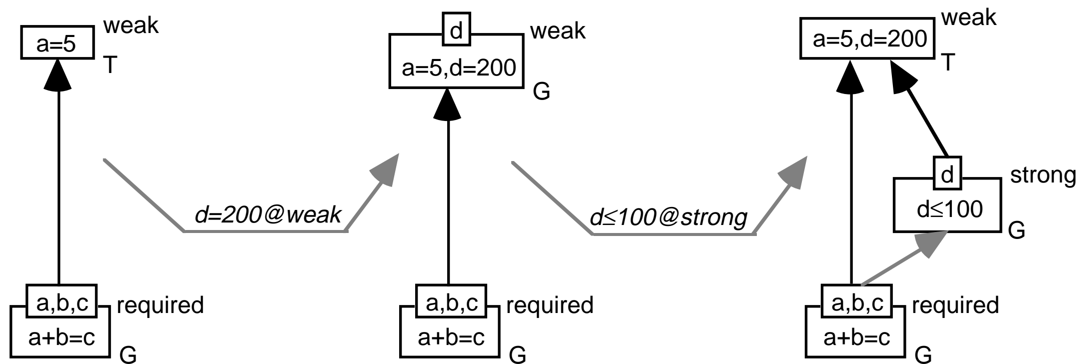
Asi nejjednodušší plánovací algoritmus pro tvorbu sítí podmínek je ten, který shromažďuje všechny podmínky stejné síly v jediné buňce. Tento algoritmus se při přidání podmínky chová následovně. Pokud již v síti existuje buňka s vnitřní silou rovnou preferenci přidávané podmínky, potom je podmínka přidána do této buňky. V opačném případě, tj. taková buňka v síti neexistuje, vytvoří algoritmus buňku novou obsahující pouze přidávanou podmínku.

Ve druhé fázi algoritmus rozdělí proměnné přidávané podmínky do množin vstupních resp. výstupních proměnných. Pokud se daná proměnná vyskytuje v buňce se silnější vnitřní silou, je zařazena mezi vstupní proměnné, v opačném případě je přidána mezi výstupní proměnné. V případě, že jsme proměnnou zařadili mezi výstupní proměnné a tato proměnná je zároveň výstupní proměnnou nějaké buňky se slabší vnitřní silou, potom je tato proměnná ve slabší buňce přeřazena mezi vstupní proměnné. Tímto krokem je zajištěno splnění podmínky 4) definice 2.12.

Na závěr algoritmus přidá resp. ubere hrany tak, aby byly splněny podmínky 5), 6) a 7) definice 2.12. Poznamenejme, že při právě popsaném rozložení proměnných do množin vstupních a výstupních proměnných jednotlivých buněk, je splnění podmínky 6)

automaticky zajištěno. Splnění podmínky 7) zajistíme jednoduše tak, že do sítě přidáme hrany vedoucí vždy od dané buňky k buňce, která je nejsilnější ze slabších buněk. Taková buňka je zřejmě jediná, protože preference jsou lineárně uspořádané a s danou vnitřní silou je v grafu vždy maximálně jedna buňka. Formálně je plánovací algoritmus zjemňovací metody popsán v příloze C.

Právě popsaný proces přidávání podmínek do sítě podmínek ilustruje následující obrázek (čteno zleva do prava). Za povšimnutí stojí mimo jiné přeřazení výstupní proměnné *d* mezi vstupní proměnné v buňce s vnitřní silou *weak* při přidání podmínky `d≤100@strong`.



Poznamenejme, že šedá hrana mezi *required* a *strong* buňkou v síti napravo není definicí 2.12 striktně vyžadována, na druhou stranu přítomnost této hrany není ani explicitně zakázána. Jedná se právě o hranu vedoucí od dané buňky k nejsilnější slabší buňce tak, jak to bylo zmíněno v závěru popisu plánovacího algoritmu v předchozím odstavci. Tyto hrany potom zajistí, že prováděcí algoritmus bude síť procházet postupně od nejsilnějších buněk po nejslabší, jak je to vlastní zjemňovací metodě.

O co jednodušší (efektivnější) je právě popsaný plánovací algoritmus zjemňovací metody, o to náročnější (výpočtově) bude prováděcí fáze zjemňovací metody. Protože plánování vlastně neprovedlo téměř žádnou práci, kromě rozdělení podmínek do úrovní, je veškerá náročnost řešení hierarchie skryta v prováděcí fázi. Ta vždy řeší celou úroveň stejně silných podmínek najednou a nemůže tak využít žádné další vztahy mezi různě silnými podmínkami. Příklady konkrétních algoritmů prováděcí fáze jsou uvedeny v kapitolách 1.5.1 a 1.5.2, v příloze B a v práci [Bar96].

Zajímavějšími plánovacími algoritmy budou jistě ty, které se snaží síť podmínek více strukturalizovat, aby tak přesněji zachycovala vztahy mezi jednotlivými podmínkami (viz. síť podmínek nalevo v obrázku z úvodu této kapitoly). Prováděcí algoritmy potom budou moci lépe využít technik lokální propagace a budou tak efektivnější než zjemňovací metoda (viz. následující kapitola). Navíc při změně sítě, tj. při přidání podmínky, bude stačit „přepočítat" menší část sítě pro získání nového řešení zatímco výpočty provedené v ostatních částech sítě zůstanou v platnosti. Navrhovaný algoritmus je tak „více inkrementální" než zjemňovací metoda při zachování dostatečné obecnosti. Bohužel v některých případech, viz. kapitola 1.2.6, se kompletnímu přepočtu řešení nevyhneme.

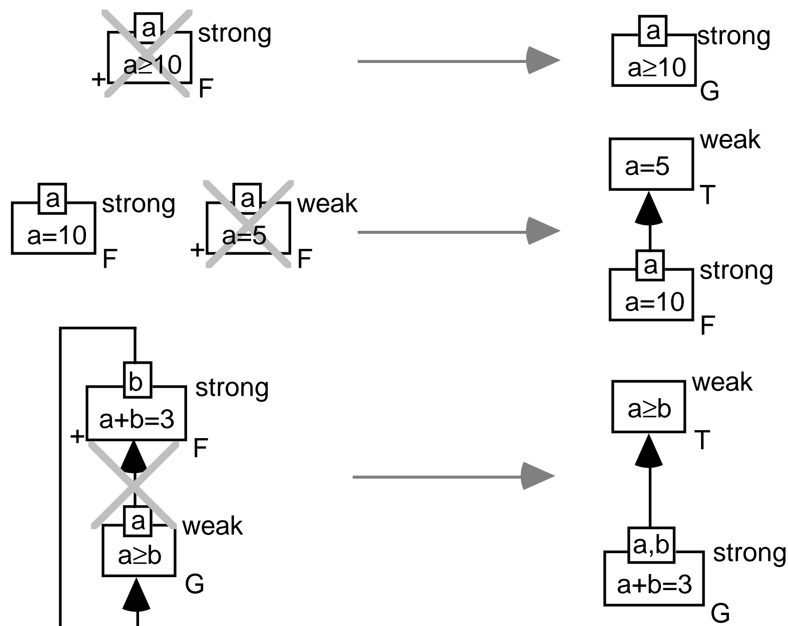
Jak víme z kapitoly 2.3.2 a na příkladu uvidíme v kapitole 2.3.4 způsobují neefektivnost prováděcího algoritmu především nerozhodnutelné, tj. generativní a testovací buňky. Pro rychlejší zpracování buňky je také výhodnější, pokud je v buňce méně podmínek. Vztahy mezi podmínkami z různých buněk se totiž řeší pomocí levné lokálné propagace, zatímco vztahy mezi podmínkami v jedné buňce je potřeba řešit složitějšími algoritmy (simplexová metoda, řešení lineárních rovnic …). Z naznačeným důvodů vyplývá, že sofistikovaný plánovací algoritmus pro tvorbu a udržování sítí podmínek by se měl snažit udržet maximální počet podmínek ve funkčních buňkách a

zároveň by měl udržovat nerozhodnutelné buňky tak malé, jak jen to jen možné vzhledem k podmínkám 6) a 7) definice 2.12.

Základní princip sofistikovaného plánovacího algoritmu není složitý. Nejprve se tento algorimus snaží přidat podmínku do sítě jako novou funkční buňku, což lze například provést podobně jako v algoritmu DeltaBlue (kapitola 1.5.3). V případě, že tento krok neuspěje, je podmínka do sítě přidána jako nová generativní resp. testovací buňka.

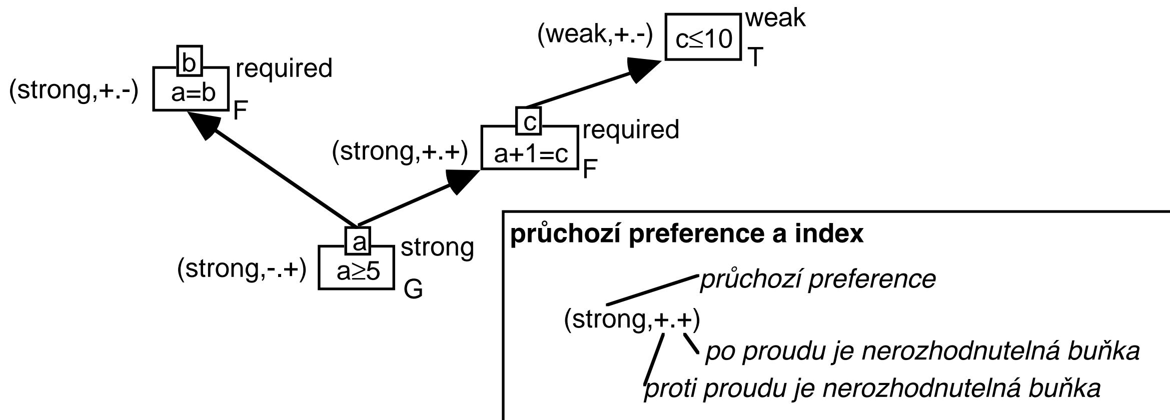
Důvodů možných neúspěchů při přidávání podmínky do sítě jako funkční buňky je více. Předně některé typy podmínek, např. nerovnosti, nemohou nikdy tvořit funkční buňku podle definice 2.10. V tomto případě se tedy podmínka do sítě rovnou přidává jako generativní nebo testovací buňka. Dalším možným důvodem neúspěchu při přidávání funkční buňky může být to, že všechny proměnné z přidávané buňky jsou určovány silnějšími nebo stejně silnými buňkami. Algoritmy klasické lokální propagace v tomto případě přidají podmínku do grafu jako nesplněnou, zde prezentovaný plánovací algoritmus ale přidá podmínku do sítě jako testovací buňku a o její splnitelnosti či nesplnitelnosti rozhodne až prováděcí algoritmus. Posledním důvodem, proč se nemusí podařit přidat podmínku do sítě jako funkční buňku, je vznik cyklu po přidání buňky. Protože cykly jsou v sítích podmínek stejně jako v grafech podmínek zakázány, nelze takovou buňku do sítě zařadit. Klasické algoritmy lokální propagace v tomto případě nejčastěji ohlásí chybu, sofistikovaný plánovací algoritmus pro sítě podmínek ale přidá buňku jako generativní nebo testovací, případně ji spojí s dalšími buňkami a tím se cyklům vyhne.

Jednotlivé důvody neúspěchů při přidávání funkčních buněk a způsob jejich řešení ukazuje následující obrázek (buňka, kterou se pokoušíme přidat jako funkční buňku, je označena +). Sítě v levé části, kdy jsme podmínky přidali jako funkční buňku, nejsou korektní, správně se má podmínka přidat tak, jak to ukazují sítě napravo.

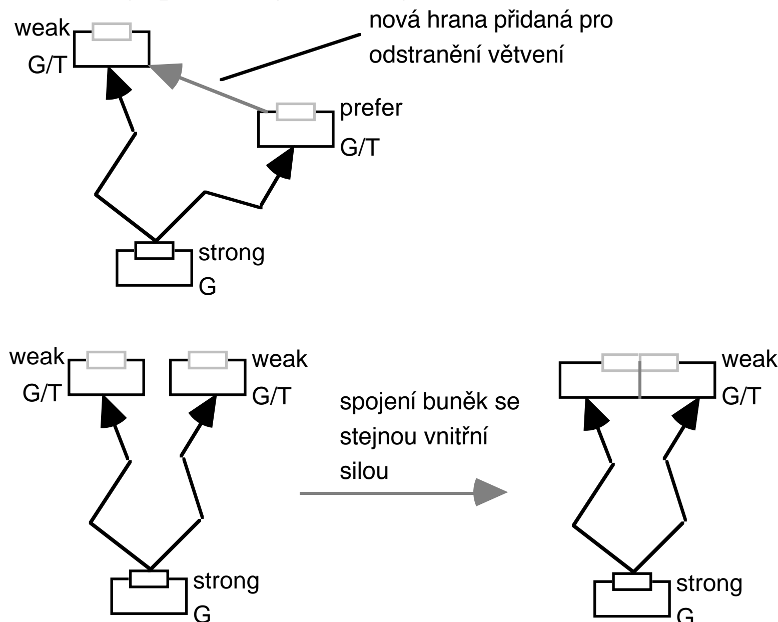


Podobně jako algoritmy lokální propagace používají pojmu *průchozí preference* (kapitola 1.5.3), aby nemusely při přidávání podmínky vždy procházet celý graf, je možné podobný pojem zavést i v sítích podmínek. Průchozí preference (síla) dané buňky je definována jako nejslabší síla z vnitřní síly dané buňky a vnitřních sil buněk, ze kterých vede do dané buňky orientovaná cesta (buňky proti proudu). Pojem průchozí preference se bude dobře hodit pro zajištění platnosti podmínek 6) a 7) definice 2.12. Ze stejného důvodu se ukazuje praktické zavést také pojem *průchozího indexu*, který bude říkat zda se po proudu resp. proti proudu od dané buňky nachází nějaká nerozhodnutelná buňka.

Oba tyto pojmy ilustruje následující obrázek zobrazující síť podmínek s vypočtenými průchozími preferencemi a indexy u buněk.



Prostým přidáním funkční buňky do grafu činnost plánovacího algoritmu nekončí. Je potřeba ještě zajistit platnost podmínky 7) definice 2.12[2], tedy odstranit zakázaná větvení po proudu, která mohla přidáním buňky vzniknout. K efektivnímu zjištění přítomnosti takovéhoto větvení se budou používat právě zavedené průchozí indexy. Způsob odstraňování větvení nejlépe ilustruje následující obrázek.



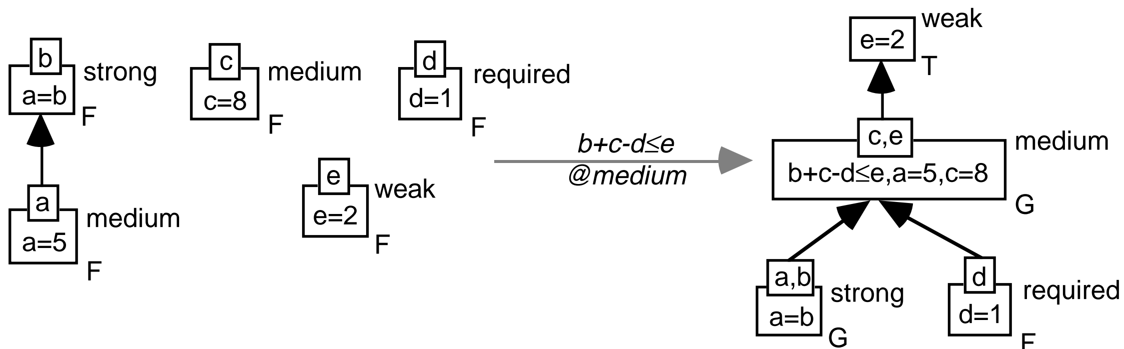
Zatím jsme rozebrali přidání podmínky do sítě v podobě funkční podmínky. To je obdobné jako v klasických algoritmech lokální propagace s tím, že je potřeba vzít v úvahu další vlastnosti sítí podmínek (větvení po proudu).

Pokud podmínka není do sítě přidána formou funkční buňky, pokouší se algoritmus přidat tuto podmínku jako generativní buňku. Je tedy vytvořena nová generativní buňka obsahující přidávanou podmínku a proměnné této podmínky jsou rozloženy do množina vstupních a výstupních proměnných následujícím způsobem. Všechny proměnné, které jsou určovány buňkami se silnější průchozí preferencí než je preference přidávané podmínky, jsou zařazeny mezi vstupní proměnné. Podobně, všechny proměnné, které jsou určovány buňkami se slabší průchozí preferencí než je preference přidávané podmínky, jsou zařazeny mezi výstupní proměnné. Ještě zbývá zařadit proměnné, které jsou určovány buňkami se stejnou průchozí preferencí jako je

[2] Ostatní podmínky definice 2.12 jsou splněny vzhledem ke způsobu přidání funkční buňky.

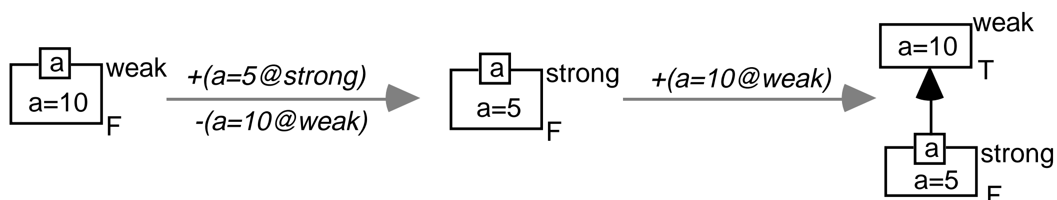
preference přidávané podmínky. To znamená, že buď přímo buňka určující danou proměnnou nebo buňka na cestě proti proudu od ní má stejnou vnitřní preferenci jako přidávaná podmínka. Tyto buňky jsou posbírány a spojeny s přidávanou buňkou do jedné velké buňky. Rozložení zbylých proměnných pak vyplyne ze způsobu spojení.

Přidání generativní buňky demonstruje následující ilustrace, kde jsou také zachyceny právě popsané varianty distribuce proměnných do množin vstupních resp. výstupních proměnných buňky. V levé části obrázku je síť podmínek před přidáním podmínky b+c-d≤e@medium, v pravé části je potom síť po přidání podmínky. Poznamenejme ještě, že pokud množina výstupních proměnných nové buňky zůstana prázdná, změní se typ buňky z generativní na testovací.



Do sítě jsou přirozeně přidány všechny hrany tak, aby podmínka 5) definice 2.12 byla splněna, a opět jsou odstraněna případná větvení po proudu zakázaná podmínkou 7) definice 2.12.

Aby se zjednodušil a zpřehlednil algoritmus přidání podmínky, jsou po přidání nové buňky do sítě dočasně ze sítě odstraněny buňky (a tedy i podmínky) dosud určující proměnné, které nyní určuje nová buňka. Odstraněné podmínky jsou následně použitím stejného algoritmu přidávání do sítě opět zařazeny. Toto dočasné vyřazení a opětovné zařazení podmínek používají i klasické algoritmy lokální propagace jako je DeltaBlue (kapitola 1.5.3). Ilustruje ho následující obrázek (přidávané resp. ubírané podmínky jsou označeny + resp. -).



Právě popsaný algoritmus přidání podmínky do sítě podmínek tvoří jádro plánovacího algoritmu. Jeho úkolem je převést hierarchii podmínek zadanou jako konečná množina označených omezujících podmínek na odpovídající síť podmínek. Protože algoritmus tvoří síť podmínek postupným přidáváním označených podmínek, je možné síť inkrementálně upravovat přidáváním dalších podmínek. Sofistikovaný plánovací algoritmus byl implemtován v jazyce PROLOG a příklad přidávání podmínek do sítě tímto algoritmem je uveden v Příloze D.

Vytvořená síť podmínek je ve druhé tzv. prováděcí fázi postupně po proudu procházena a je počítáno konkrétní ohodnocení proměnných, které řeší zadanou hierarchii podmínek. Algoritmům prováděcí fáze je věnována následující kapitola.

2.3.4 Procházení sítí podmínek (prováděcí fáze)

V následujících odstavcích představíme jeden z možných prováděcích algoritmů, který umí řešit rovnosti a nerovnosti nad reálnými čísly použitím locally-metric-better komparátoru (tj. lokální komparátor používající netriviální chybovou funkci). Tento

algoritmus je založen na propagaci intervalů reálných čísel přes síť podmínek a ideově tak vychází z algoritmu Indigo (kapitola 1.5.6). Oproti Indigu je ale schopen nalézt také alternativní ohodnocení z řešení.

Popisovaný prováděcí algoritmus nejprve topologicky setřídí buňky ze sítě. Vzhledem k teorii z kapitoly 2.3.1 a dodatku v závěru kapitoly 2.3.2 nezáleží na tom, kterou z možných posloupností vzniklých topologickým uspořádáním sítě zvolíme.

Po uspořádání buněk prochází prováděcí algoritmus přes jednotlivé buňky ve zvolené posloupnosti a propaguje přes ně intervaly reálných čísel. Na začátku je interval u každé proměnné nastaven na $(-\infty,+\infty)$. Pokud buňka obsahuje jedinou omezující podmínku, je propagace přímočará stejně jako v Indigu. V případě, že je v buňce více podmínek, tyto podmínky libovolně setřídíme a opět přes vzniklou posloupnost postupně propagujeme intervaly reálných čísel. Tím, že zvolíme jiné uspořádání podmínek v buňce, můžeme získat jiné ohodnocení proměnných, které také padne do řešení. Možnost propagovat intervaly přes buňku tak, že je postupně propagujeme přes jednotlivé podmínky v buňce, je dána použitím lokálního komparátoru. V případě, že bychom používali jiný typ komparátoru, například nějaký globální, musely by se hodnoty přes buňky s více podmínkami propagovat komplikovanějším způsobem. Algoritmus prováděcí fáze končí přechodem přes poslední buňku. V případě, že jsou žádána další alternativní ohodnocení z řešení, stačí u některé z buněk obsahujících více podmínek zvolit jiné uspořádání podmínek a propagaci od této buňky provést znova.

Za pozornost jistě stojí to, že právě načrtnutý algoritmus prováděcí fáze může procházet sítě podmínek vytvořené různými plánovacími algoritmy. Jeho efektivita ovšem bude záležet na strukturovanosti sítě podmínek. Pokud se totiž síť skládá z buněk obsahujících méně podmínek, není potřeba při hledání alternativních ohodnocení generovat tolik různých uspořádání jako v případě sítí s „velkými“ buňkami. Výhoda větší strukturovanosti sítě se projeví také při hledání prvního ohodnocení z řešení. Metoda propagace intervalů totiž vyžaduje v případě zmenšení intervalu u vstupní proměnné buňky (může se stát pouze u nerozhodnutelných buněk) provést zpětnou kontrolu podmínek z již prošlých buněk (viz. příklad 1.27). Strukturovanější sítě umožňují chytřejší zpětnou kontrolu tím, že se po hranách proti proudu dojde k buňkám obsahujícím podmínky omezující inkriminované proměnné. To také zvýhodňuje navrhovaný prováděcí algoritmus oproti Indigu, které zpětně kontroluje všechny (aktivní) podmínky. Když už jsme u srovnání s Indigem, tak navrhovaný prováděcí algoritmus má oproti Indigu další výhodu. Zatímco Indigo vždy setřídí podmínky podle preference od silnějších po slabší, topologické setřídění buněk (a tedy i podmínek) je mnohem volnější a slabší buňky se tak v uspořádání mohou dostat před silnější buňky. Pokud potom umístíme co nejvíce funkčních buněk na začátek posloupnosti a ze zbylých buněk umístíme co největší počet funčních buněk na konec posloupnosti tak, aby bylo respektováno topologické uspořádání, můžeme více využít metod lokální propagace, která nevyžaduje žádnou zpětnou kontrolu jako Indigo. Propagace hodnot sítí je pak efektivnější. Tuto myšlenku zachycuje následující obrázek, kdy symboly F, G a T označují funkční, generativní resp. testovací buňky.

schéma sítě podmínek

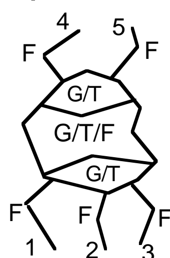
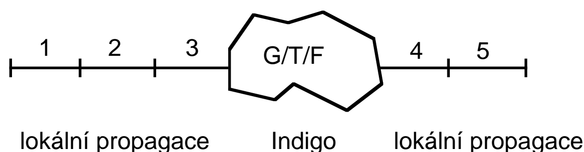
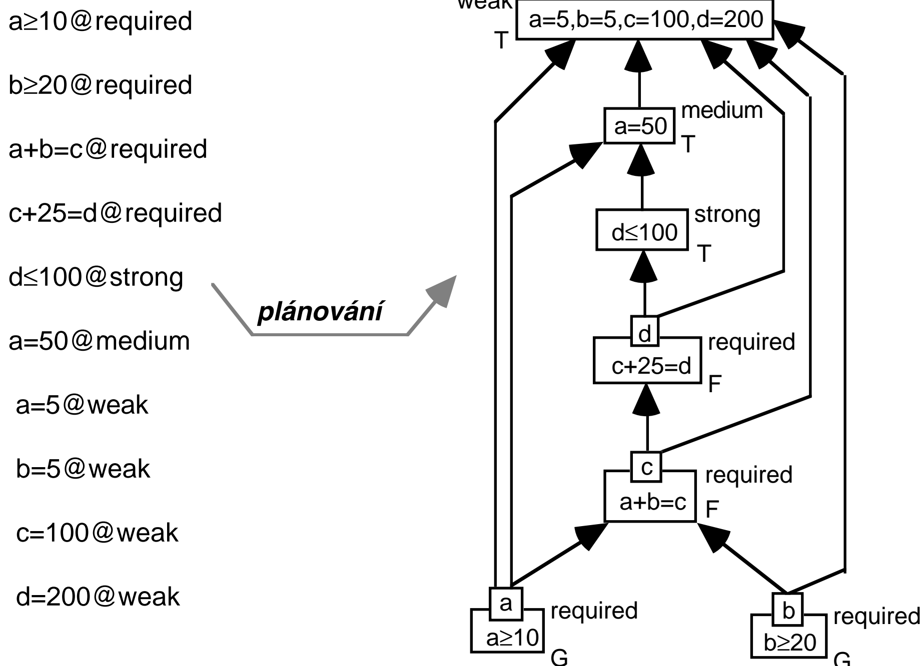


schéma uspořádání buněk



Použití právě popsaného prováděcího algoritmu si ukážeme na hierarchii z příkladu 1.27, na které byl prezentován algoritmus Indigo (kapitola 1.5.6) a jejíž síť podmínek je zobrazena v úvodu kapitoly 2.3.3 (nalevo). Jedná se o následující hierarchii podmínek a jí odpovídající síť podmínek:



Algoritmus nejprve topologicky setřídí buňky sítě, v tomto konkrétním případě existují dvě různá uspořádání (zleva doprava):

a≥10, b≥10, a+b=c, c+25=d, d≤100, a=50, {a=5,b=5,c=100,d=200}

b≥10, a≥10, a+b=c, c+25=d, d≤100, a=50, {a=5,b=5,c=100,d=200}.

Pro další postup zvolíme první posloupnost buněk. Na první šestici buněk z této posloupnosti nyní můžeme přímo aplikovat algorimus Indigo, protože každá buňka obsahuje jedinou podmínku. Dostaneme následující částečné ohodnocení (viz. příklad 1.27):

a/50, b/{20…25}, c/{70…75}, d/{95…100}.

Abychom získali ohodnocení z řešení, musíme ještě provést propagaci hodnot přes poslední buňku posloupnosti. Následující tabulka ukazuje získaná ohodnocení při různě zvoleném uspořádání podmínek v poslední buňce.

uspořádání podmínek v buňce	ohodnocení z řešení
začíná s b=5, např. b=5, a=5, c=100, d=200	a/50, b/20, c/70, d/95
začíná s c=100 nebo d=200, např. c=100, a=5, b=5, d=200 nebo d=200, a=5, b=5, c=100	a/50, b/25, c/75, d/100

2.4 Algoritmus pro mezi-hierarchické porovnání

Obecný algoritmus pro řešení hierarchií podmínek diskutovaný v kapitole 2.3 neobsahuje přímou podporu pro mezi-hierarchické porovnávání (kapitola 1.2.5). Protože se ale jedná o jeden z mála algoritmů podporujících globální komparátory, lze ho využít i pro mezi-hierarchické porovnávání. Připomeňme, že mezi-hierarchické porovnávání umožňuje

srovnávat ohodnocení z různých hierarchií. V následujících odstavcích popíšeme nadstavbu algoritmu z kapitoly 2.3, která umožní jeho využití při mezi-hierarchickém porovnávání v HCLP.

Klasické systémy HCLP s intra-hierarchickým porovnáváním (kapitola 1.2.4) hledají řešení stejným způsobem jako systémy založené na jazyku PROLOG, tj. prohledáváním do hloubky (depth-first). Tato metoda je u nich zvolena především z důvodů efektivity, nemalý význam má jistě také možnost postavit HCLP jako nadstavbu nad jazykem PROLOG.

Systém HCLP zpravidla v průběhu redukce cíle řeší pouze nutné podmínky zatímco měkké podmínky jsou shromažďovány. Po zredukování cíle je vyřešena nashromážděná hierarchie podmínek a nalezené ohodnocení proměnných je vráceno uživateli. Při dotazu na další řešení se nejprve hledají další ohodnocení z řešení hierarchie a pokud takové ohodnocení již neexistuje, zkouší se stejně jako v PROLOGu alternativní pravidla pro zredukování cíle (viz kapitola 1.5.1 a příloha B).

HCLP s mezi-hierarchickým porovnáváním může využívat stejný přístup s tím rozdílem, že je potřeba nashromáždit všechny hierarchie podmínek získané alternativními redukcemi cíle a teprve z řešení těchto hierarchií vybrat to nejlepší. Tento přístup je například zvolen v práci [BH96]. Problém ovšem mohou způsobit nekonečné větve výpočtu (viz. příklad 1.16) a to i v případě, že lze řešící ohodnocení získat z jiné konečné větve. Z tohoto důvodu se jako vhodnější jeví prohledávání do šířky (breadth-first), které umožní (většinou) najít řešení, pokud existuje. Protože při mezi-hierarchickém porovnávání je potřeba projít všechny větve výpočtu, je prohledávání do šířky stejně efektivní jako prohledávání do hloubky. Nevýhodou prohledávání do šířky je ale větší paměťová náročnost.

Jednoduché prohledávání do šířky lze v případě HCLP s mezi-hierarchickým porovnáním dále zefektivnit. Využijeme při tom poznatku, že pokud ohodnocení z řešení nějaké hierarchie podmínek H použitím globálního komparátoru je lepší než ohodnocení z řešení jiné hierarchie H', potom přidáním dalších podmínek do hierarchie H' (získanou hierarchii označme H^{ext}) nemůžeme dostat lepší řešení než je řešení hierarchie H (viz. příklad 2.2). V případě mezi-hierarchického porovnávání to znamená, že žádné ohodnocení z řešení hierarchie H^{ext} nepadne do řešení množiny hierarchií obsahující hierarchie H a H^{ext}.

Předchozí pozorování se při prohledávání do šířky stromu výpočtu využije následujícím způsobem. Jakmile na nějaké úrovni zcela zredukujeme cíl v některé větvi a získáme tak finální hierarchii podmínek H této větve, získanou hierarchii H vyřešíme například použitím algoritmu z kapitoly 2.3. V ostatních větvích zatím máme jen částečné hierarchie, které se budou při dalších redukcích příslušných cílů rozšiřovat o další podmínky. Také tyto částečné hierarchie ovšem můžeme vyřešit a v případě, že řešení některé částečné hierarchie H' je horší než řešení hierarchie H, víme podle pozorování z předchozího odstavce, že nemusíme v redukcích na větvi s hierarchií H' dále pokračovat, protože nepovedou k hierarchii, jejíž řešení by bylo lepší než již získané řešení hierarchie H. Řešení hierarchie H tak představuje „horní“ odhad celkového řešení. Právě popsaným způsobem můžeme některé výpočtové větve eliminovat, aniž bychom na nich získali finální hierarchii po úplném zredukování cíle. Někdy se tak podaří vyhnout i nekonečným větvím (viz. příklad 2.2).

Prohledávání do šířky stromu výpočtu s úpravou navrženou v předchozím odstavci ukončíme ve chvíli, kdy je každá větev buď zakončena (tj. cíl je zredukován) nebo je eliminována, protože její částečná hierarchie dává horší řešení než je řešení hierarchie z nějaké ukončené větve. Při tomto postupu nemusíme procházet celý výpočtový strom a je proto efektivnější než jednoduché prohledávání do hloubky nebo do šířky, které pro nalezení řešení musí projít celý výpočtový strom. Schéma algoritmu pro řešení cílů v HCLP s mezi-hierarchickým porovnáváním je uvedeno v příloze E.

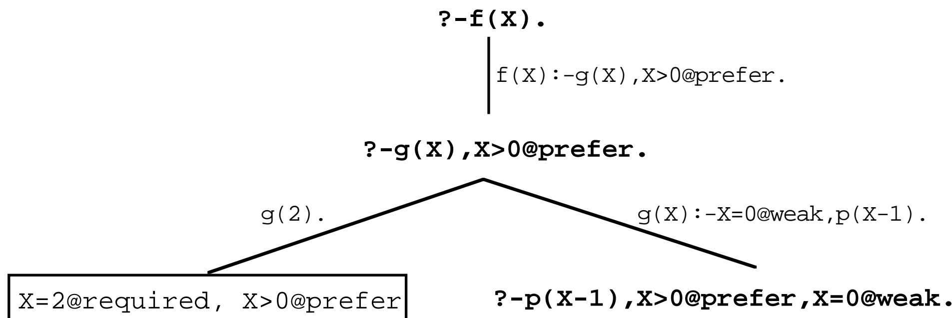
Právě navržený princip HCLP s mezi-hierarchickým porovnáváním je demonstrován následujícím příkladem.

Příklad 2.2

Nechť je dán následující HCLP program:

```
f(X):-g(X), X>0@prefer.
g(2).
g(X):-X=0@weak,p(X-1).
p(1).
p(X):-p(X-1).
```

Řešíme-li cíl `?-f(X)` použitím mezi-hierarchického porovnávání nad doménou celých čísel s komparátorem weighted-sum-better, dostaneme následující částečný strom výpočtu:



V levé větvi je již cíl zredukován a dostali jsme hierarchii:

`X=2@required, X>0@prefer`,

jejímž řešení je ohodnocení {X/2}.

Pravá větev ještě není zcela zredukována a dává tedy částečnou hierarchii:

`X>0@prefer, X=0@weak`.

Řešením této hierarchie je ohodnocení {X/1} splňující prefer podmínku a minimalizující chybu u weak podmínky.

Zatímco ohodnocení {X/2} splňuje všechny podmínky „své“ hierarchie, ohodnocení {X/1} nesplňuje weak podmínku své hierarchie. Nemá tedy cenu pokračovat v další redukci pravé větve, protože ať se přidá jakákoliv další podmínka, nikdy už nebudou splněny všechny podmínky vzniklé hierarchie a žádné její řešení tedy nemůže být lepší než již získané řešení {X/2}, které splňuje všechny podmínky své hierarchie.
Poznamejme ještě, že tímto „useknutím“ výpočtu jsme se zbavili nekonečné větve, která by nutně vznikla, pokud bychom prováděli další redukce v pravé větvi výpočtu. V případě použití klasické metody, kdy jsou nejprve získávány všechny hierarchie, by tak výpočet byl nekonečný.

Navrženým rozšířením prohledávání do šířky, které eliminuje některé větve výpočtu, se podařilo zefektivnit procházení stromu výpočtu. Toto vylepšení klade na druhou stranu větší nároky na systém řešení hierarchií, protože se kromě finálních hierarchií po zredukování cíle řeší také částečné hierarchie vzniklé v průběhu redukcí. To ovšem ve svém důsledku umožní eliminovat některé větve výpočtu a tím se případně zbavit řešení řady finálních hierarchií, které by redukcemi v této větvi vznikly. Nelze proto obecně říci, zda navržený algoritmus využívající řešení částečných hierarchií je nucen vyřešit větší nebo menší množství hierarchií podmínek než jednoduché prohledávání do šířky.

Řešení hierarchie H^{ext}, která vznikne přidáním podmínek do hierarchie H, navíc nemusí být díky algoritmu z kapitoly 2.3 výpočtově tak náročné, jako řešení hierarchie

H^{ext} zcela od začátku. Algoritmus navržený v kapitole 2.3 totiž často dovoluje využít výsledků řešení částečné hierarchie H při řešení hierarchie H^{ext}.

Z předchozích odstavců je zřejmé, že pokud se nám brzy podaří najít nějakou finální hierarchii s „dobrým“ řešením, můžeme tím eliminovat výpočtové větve ještě před tím, než se rozvětví do podstromu výpočtu. Tím se dále můžeme zbavit nutnosti řešit množství částečných hierarchií. Z tohoto důvodu se zdá lepší nahradit „slepé“ prohledávání do šířky nějakým řízeným prohledáváním, které najde finální hierarchii s dobrým řešením dříve. Návrhu tohoto řízeného prohledávání jsou věnovány následující odstavce.

Nejjednoduším způsobem, jak získat potřebné informace pro řízené prohledávání, je po každé redukci vyřešit vzniklou částečnou hierarchii a vždy se potom vydat po té větvi, která poskytuje nejlepší řešení své částečné hierarchie. Tímto přístupem by se ale příliš zvedl počet řešených hierarchií, což by svém důsledku znamenalo zpomalení celého prohledávání. Efektivnější proto bude využít pouze odhad řešení, který se získá méně výpočtově náročným postupem.

Při hledání odhadu řešení může dobře posloužit plánovací fáze algoritmu z kapitoly 2.3. Protože podmínky jsou do sítě podmínek přidávány inkrementálně po každé redukci, provádí se plánování resp. přeplánování v každém kroku redukce a horní odhad řešení částečné hierarchie tak vlastně získáme zadarmo. Připomeňme, že podmínky z funkčních buněk sítě podmínek lze vždy splnit, zatímco o splnitelnosti podmínek v nerozhodnutelných buňkách nemůžeme ve fázi plánování rozhodnout. Pro horní odhad řešení tedy můžeme předpokládat, že tyto podmínky nejsou splněny.

Je také vidět, že pro výpočet odhadu jsou lepší více strukturované sítě, protože je v nich méně nerozhodnutelných buněk a odhad je pak přesnější. Plánovací algoritmus zjemňovací metody proto dává horší odhady než sofistikovaný plánovací algoritmus. Po „ohodnocení“ jednotlivých částečných řešení vybere řízené prohledávání tu větev výpočtu, kde je odhad nejhoršího řešení nejlepší.

Poznamejme, že ve schématu algoritmu pro řešení cílů v HCLP s mezi-hierarchickým porovnáním z přílohy E se právě navržené vylepšení projeví tak, že procedura `insert_to_list`, která v případě prohledávání do šířky zařazovala prvky na konec seznamu, je bude nyní zařazovat na místo podle horního odhadu řešení tak, aby cíle s lepším částečným řešením (resp. jeho odhadem) byly v seznamu před cíly s horším částečným řešením.

V předchozích odstavcích jsme navrhli metodu pro řešení HCLP programů s mezi-hierarchickým porovnáváním, která využívá vlastností algoritmu pro řešení hierarchií navrženého v kapitole 2.3. Ukázali jsme, že i mezi-hierarchické porovnávání lze řešit dostatečně efektivně, což otevírá cestu k expertním systémům založeným na hierarchiích omezujících podmínek.

Závěr

Předkládaná práce zpracovává problematiku efektivních algoritmů pro řešení hierarchií omezujících podmínek. Nosnou linií práce je myšlenka expertních systémů založených na hierarchiích omezujících podmínek, která odůvodňuje nutnost výzkumu efektivních algoritmů pro řešení hierarchií podmínek.

Práce je rozdělena do dvou částí. V první části jsou nejprve stručně představeny omezující podmínky (kapitola 1.1), aby mohl být zbytek první části věnován poměrně podrobnému přehledu hierarchií omezujících podmínek (kapitoly 1.2-1.4). Velký prostor je také dán představení řady algoritmů pro řešení hierarchií (kapitola 1.5).

Druhá část přináší do oblasti algoritmů pro řešení hierarchií omezujících podmínek nové poznatky. V úvodu druhá části jsou nejprve stručně představeny klasické pravidlově orientované expertní systémy (kapitola 2.1) a nově jsou zde navrženy expertní systémy založené na hierarchiích omezujících podmínek (kapitola 2.2). Jsou tady také rozebrány konkrétní požadavky těchto expertních systémů na podkladový systém řešení hierarchií (kapitola 2.2), kterému je věnován zbytek práce. Zde je především navržena alternativní teorie hierarchií podmínek (kapitola 2.3.1), která se v mnoha bodech podobá přístupu z první části. Důležité je, že tato teorie odpovídá intuitivnímu přístupu k hierarchiím a že zároveň podporuje efektivní metody řešení hierarchií podmínek rozkladem do buněk a postupným (iterativním) řešením buněk. Hlavním výsledkem v práci navržené teorie jsou věty o korektnosti a úplnosti nově navržené metody řešení hierarchií. V dalších kapitolách je potom prezentována konkrétní instance navržené metody založená na nově zavedeném pojmu sítí podmínek (kapitola 2.3.2). Pro práci se sítěmi podmínek byly nově vyvinuty algoritmy umožňující vytváření sítí podmínek postupným přidáváním podmínek (kapitola 2.3.3) a procházení sítí podmínek resp. propagaci ohodnocení sítí (kapitola 2.3.4). Na závěr je představen nově vyvinutý obecný algoritmus pro mezi-hierarchické porovnávání postavený jako nadstavba nad algoritmy pro řešení hierarchií (kapitola 2.4).

Přestože v práci je podán ucelený soubor poznatků o algoritmech pro řešení hierarchií podmínek (kapitoly 1.5, 2.3 a 2.4), neznamená to, že by byl výzkum v této oblati již uzavřen. Právě naopak, předkládaná práce by měla být impulsem pro další bádání v oblasti algoritmů pro řešení hierarchií podmínek, protože ukazuje možnost efektivní implementace takových algoritmů.

V teoretické části se lze zabývat dalším rozvojem teorie vypracované v kapitole 2.3.1. Zvláště zajímavé může být hledání slabších omezení (alternativních definic) pro komparátory případně oslabení vlastnosti postupného oslabování tak, aby prezentovaná tvrzení zůstala v platnosti.

U sítí podmínek může být zajímavé zkoumat, zda lze oslabit některé podmínky kladené na tyto sítě, například zda přítomnost různě silných omezujících podmínek v jedné buňce umožní sítě ještě více zobecnit. Velkou oblastí pro další práce jsou algoritmy pro tvorbu a údržbu sítí podmínek. V této práci byla popsána dvojice algoritmů pro přidání podmínky do sítě, otevřena zůstává otázka efektivních algoritmů umožňujících odstranění podmínky ze sítě. A potom jsou tady algoritmy prováděcí fáze umožňující propagaci ohodnocení sítí podmínek. Těch lze vyvinout celou řadu v závislosti na použití konkrétních omezujících podmínek a komparátorů.

V oblasti znalostních systémů založených na hierarchiích omezujících podmínek je zajímavou a stále ještě otevřenou kapitolou zpracování neurčité informace použitím hierarchií podmínek. V práci navržené expertní systémy také zdůvodňují užitečnost mezi-hierarchického porovnávání. Dosud vytvořené algoritmy pro mezi-hierarchické porovnávání pracují jako nadstavba nad algoritmy pro řešení hierarchií. Z důvodu větší efektivity by jistě bylo zajímavé vytvoření algoritmu s přímou podporou mezi-hierarchického porovnávání.

Literatura

[AbRo89] Abramson, H. and Rogers, M.H. (eds.), *Meta-Programming in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1989

[AtBa94] Athanasiu, I., Bal, H.E., The Arc Consistency Problem: a Case Study in Parallel Programming with Shared Objects, 7th International Conference on Parallel and Distributed Computing Systems, Las Vegas, October 1994

[Bar93] Barták, R., Metainterpretace logických programů (in Czech), Diploma Thesis, Charles University, Prague, 1993

[Bar96] Barták, R., Plug-In Architecture of Constraint Hierarchy Solvers, Tech. Report No 96/8, Department of Theoretical Computer Science, Charles University, December 1996 (submitted to PACT'97)

[Bar97] Barták, R., A Generalized Algorithm for Solving Constraint Hierarchies, Tech. Report No 97/1, Department of Theoretical Computer Science, Charles University, January 1997 (submitted to JFPLC'97)

[BaŠt95] Barták, R. and Štěpánek, P., Meta-Interpreters and Expert Systems, Tech. Report No 115, Department of Computer Science, Charles University, October 1995

[BAFM96a] Borning, A., Anderson, R., Freeman-Benson, B., Indigo: A Local Propagation Algorithm for Inequality Constraints, Preliminary Report, Department of Computer Science and Engineering, University of Washington, 1996

[BAFM96b] Borning, A., Anderson, R., Freeman-Benson, B., The Indigo Algorithm, Tech. Report 96-05-01, Department of Computer Science and Engineering, University of Washington, July 1996

[BDFB+87] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., Woolf, M., Constraint Hierarchies, in: *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp.48-60, ACM, October 1987

[BFB95] Borning, A., Freeman-Benson, B., The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces, in: *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pp. 624-628, Cassis, France, September 1995

[BeCo93] Benhamou, F. and Colmerauer, A. (eds.), *Constraint Logic Programming-Selected Research*, The MIT Press, Cambridge, Massachusetts, 1993

[BNH96] Bouzoubaa, M., Neveu, B., Hasle, G., Houria III: Solver for Hierarchical System, Planning of Lexicographic Weight Sum Better Graph For Functional Constraints, in: *the Fifth INFORMS Computer Science Technical Section Conference on Computer Science and Operations Research*, Dallas, Texas, Jan. 8-10, 1996

[BH96] Bouzoubaa, M., Hasle, G., Equational Constraint Hierarchies in Constraint Logic Programming Languages: Algorithm for Inter-Hierarchy Comparisons, in: *Proceedings of PACT`96*, pp. 417-426, London, April 24-26, 1996

[BMMW89] Borning, A., Maher, M., Martindale, A., Wilson, M., Constraint Hierarchies and Logic Programming, in: *Proceedings of the Sixth International Conference on Logic Programming*, pp. 149-164, Lisbon, June, 1989

[Bor81] Borning, A., The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory, in: *ACM Transactions on Programming Languages and Systems 3(4)*, pp. 252-387, October 1981

[Buch85] Buchberger, B., Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory, in: *Multidimensional Systems Theory*, pp. 184-232, D. Reidel Publ. Comp., 1985

[BuFe78] Buchanan, B.G., Feigenbaum, E.A., DENDRAL and Meta-DENDRAL: Their Application Dimensions, in: *Artificial Intelligence*, Vol. 11, 1978

[BuSh84] Buchanan, B.G., Shortliffe, E.H. (eds.), Rule-Based Expert Systems-The MYCIN experiments, Addison-Wesley, Reading, Massachusetts, 1984

[Car94] Carlsson, M., Boolean Constraints in SISCtus Prolog, Tech. Report T91:09, SISC, November 1994

[ClMe81] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, Berlin, 1981

[Col83] Colmerauer, A., Prolog in 10 Figures, in: *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pp. 487-499, 1983

[Cov89] Covington, M.A., Efficient Prolog: A Practical Guide, Tech. Report AI-1989-08, The University of Georgia, August 1989

[Dem68] Dempster, A.P., A generalization of Bayesian inference, in: *Journal of Royal Statistic Society 30(B)*, pp. 208-247, 1968

[DHN+78] Duda, R.O., Hart, P.E., Nilsson, N.J., Barre, P., Gaching, J.G., Reboh, R., Development of the Prospector Consultation System for Mineral Exploration, Stanford Research Institute, Menlo Park, 1978

[DPP95] Dix, J., Pereira, L.M., Przymusinsky, T.C. (eds.), Non-Monotonic Extensions of Logic Programming, Lecture Notes in Artificial Intelligence 927, Springer Verlag, Berlin, 1995

[EEP91] Eisinger, N., Elshiewy, N., Pareschi, R., Distributed Artificial Intelligence-An Overview, Tech. Report ECRC-91-7, ECRC, 1991

[FBB92a] Freeman-Benson, B.N., Borning A., Integrating Constraints with an Object-Oriented Language, in: *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pp. 268-286, 1992

[FBB92b] Freeman-Benson, B.N., Borning A., The Design and Implementation of Kaleidoscope '90, A Constraint Imperative Programming Language, in: *Proceedings of ICCL'92*, pp. 174-180, 1992

[FrAb96] Frühwirth, T., Abdennadher, S., The Munich Rent Advisor, 1st Workshop on Logic Programming Tools for Internet Applications in conjunction with JICSLP'96, Bonn, Germany, September 1996

[Fre78] Freuder, E.C., Synthesizing Constraint Expressions, in: *Communications of the ACM*, 21, pp.958-966, November 1978

[Frü93] Frühwirth, T., Constraint Simplification Rules, Tech. Report ECRC-92-18, ECRC, February 1993

[FHK+93] Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E., Wallace, M., Constraint Logic Programming - An Informal Introduction, Tech. Report ECRC-93-5, ECRC, February 1993

[FrBr95] Frühwirth, T., Brisset, P., High-Level Implementations of Constraint Handling Rules, Tech. Report ECRC-TR-95-20, ECRC, June 1995

[Gal85] Gallaire, H., Logic programming: Further developments, in: *IEEE Symposium on Logic Programming*, pp. 88-99, IEEE, Boston, July 1985

[Gas74] Gasching, J., A Constraint Satisfaction Method for Inference Making, in: *Proceedings of the 12th Annual Allerton Conference on Circuit System Theory*, pp. 866-874, U. Illinois, 1974

[GrSm96] Grant, S.A., Smith, B.M., The Arc and Path Consistency Phase Transitions, Tech. Report 96.09, School of Computer Studies, University of Leeds, March 1996

[Haj85] Hájek, P., Combining functions for certainty degrees in consulting systems, in: *Int.J. Man-Machine Studies 22*, pp. 59-76, 1985

[HaEl80] Haralick, R.M., Elliot, G.L., Increasing Tree Search Efficiency for Constraint Satisfaction Problems, in: *Artificial Intelligence*, 14, pp. 263-313, 1980

[HDE78] Hart, P.E., Duda, R.O., Einaudi, M.T., PROSPECTOR-A Computer Based Consultation System for Mineral Exploration, in: *Mathematical Geology 10*, pp. 589-610, 1978

[HMT+94] Hosobe, H., Miyashita, K., Takahashi, S., Matsuoka, S., Yonezawa, A., Locally Simultaneous Constraint Satisfaction, in: *Principles and Practice of Constraint Programming---PPCP'94 (A. Borning ed.), no. 874 in Lecture Notes in Computer Science*, pp. 51-62, Springer-Verlag, October 1994

[HMY96] Hosobe, H., Matsuoka, S., Yonezawa, A., Generalized Local Propagation: A Framework for Solving Constraint Hierarchies, in: *Principles and Practice of Constraint Programming---CP'96 (E. Freuder ed.), Lecture Notes in Computer Science*, Springer-Verlag, August 1996

[ChaM85] Charniak, E., McDermot, D., *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, Massachussetts, 1985

[JaMa94] Jaffar, J., Maher, M.J., Constraint Logic Programming: A Survey, in: Journal of Logic Programming 19, pp. 503-581, 1994

[JaLa87] Jaffar, J., Lassez, J.-L., Constraint Logic Programming, in: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111-119, Munich, Germany, January 1987

[JSK95] Jain, A., Sterling, L. and Kirschenbaum, M., Towards Reusability Based Upon Similar Computational Behaviour, in: *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, Rockville, Maryland, USA, June 1995

[JJG96] Jampel, M., Jacquet, J.-M., Gilbert, D., A General Framework for Integrating HCLP and PCSP, Tech. report RP-96-009, Institut d'Informatique, F.U.N.D.P., Belgium, April 1996

[JJGH96] Jampel, M., Jacquet, J.-M., Gilbert, D., Hunt, S., Transformations between HCLP a PCSP, Tech. report, Institut d'Informatique, F.U.N.D.P., Belgium, July 1996

[JaGi94] Jampel, M., Gilbert, D., Fair Hierarchical Constraint Logic Programming, Tech. Report TCU/CS/1994/9, Department of Computer Science, City University, July 1994

[Jam95a] Jampel, M., A Compositional Theory of Constraint Hierarchies, Tech. Report TCU/CS/1995/5, Department of Computer Science, City University, March 1995

[Jam95b] Jampel, M., A Compositional Theory of Constraint Hierarchies (Operational Semantics), in: *Proceedings of CP'95*, 1995

[KRKW89] Kandri-Rodi, A., Kapur, D., Winkler, F., Knuth-Bendix Procedure and Buchberger Algorithm–A Synthesis, in: *Proceedings of ISSAC'89*, Portland, August 1989

[Kov88] Kovacec, A., Buchberger's Theory of Gröbner Bases, Tech. Report 88-91.0, RISC Linz, December 1988

[LBFL80] Lindsay, R.K., Buchanan, B.G., Feigenbaum, E.A., Lederberg, J., Applications of Artificial Intelligence to Chemistry, The DENDRAL Project, McGraw-Hill, New York, 1980

[LePW93] Le Provost, T., Wallace, M., Generalised Constraint Propagation Over the CLP Scheme, Tech. Report ECRC-92-1, ECRC, February 1993

[LSB95] Lakhotia, A., Sterling, L. and Bojantchev, D., Development of a Prolog Tracer by Stepwise Enhancement, in: *Proceedings of the Third International Conference on Practical Applications of Prolog*, Paris, April 1995

[Llo84] Lloyd, J.W., *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984

[LFBB94a] Lopez, G., Freeman-Benson, B., Borning, A., Constraints and Object Identity, Tech. Report 94-03-07, University of Washington, March 94

[LFBB94b] Lopez, G., Freeman-Benson, B., Borning, A., Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine, Tech. Report 94-07-07, University of Washington, July 1994

[Mac77] Mackworth, A.K., Consistency in Network of Relations, in: *AI Journal*, 8, pp. 99-118, 1977

[MaSt89] Maher, M.J., Stuckey, P.J., Expanding Query Power in Constraint Logic Programming, in: *Proceedings of the North Americal Conference on Logic Programming*, Cleveland, October 1989

[Mal91] Maloney, J., Using Constraints for User Interface Construction, PhD Thesis, Tech. Report 91-08-12, Department of Computer Science and Engineering, University of Washington, August 1991

[Mat93] Matyska, L., Logic Programming with Fuzzy Sets, Tech. Report TCU/CS/1993/4, Department of Computer Science, City University, London, December 1993

[McD79] McDermot, J., R1: A Rule-Based Configurer of Computer Systems, in: *Artificial Intelligence*, Vol. 1, No 1, 1979

[MD80] McDermot, D., Doyle, J., Nonmonotonic Logic, in: *Artificial Intelligence* Vol. 13, No 1, November 1980

[MP43] McCulloch, W., Pitts, W., A Logical Calculus of the Ideas Immanent in Nervous System Activity, in: *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115-133, 1943

[Mei95a] Meier, M., Better Late Than Never, Tech. Report ECRC-95-08, ECRC, 1995

[Mei95b] Meier, M., Event Handling in Prolog, Tech. Report ECRC-95-09, ECRC, 1995

[Mei96] Meier, M., Finite Domain Constraint Programming, Tutorial in PACT`96, London, April 24-26 1996

[MeBr95] Meier, M., Brisset, P., Open Architecture for CLP, TR ECRC-95-10, ECRC, 1995

[MeSc95] Meier, M., Schimpf, J., An Architecture for Prolog Extensions, TR ECRC-95-6, ECRC, 1995

[MBC93] Menezes, F., Barahoma, P., Codognet, P., An Incremental Hierarchical Constraint Solver, in: *Proceedings of PPCP'93*, pp. 190-199, Newport, Rode Island, 1993

[MeBa95] Menezes, F., Barahona, P., An Incremental Hierarchical Constraint Solver, in: [SaVH95]

[MiOr95] Michaylov, S., Ordóñez, I., Time and Money: A Case Study in Systematic Development of Constraint Logic Programs, Tech. Report OSU-CISRC-5/95-TR24, Department of Computer and Information Science, The Ohio State University, 1995

[Mon74] Montanary, U., Networks of Constraints: Fundamental Properties and Applications to Picture Processing, in: *Information Science*, 7, pp. 95-132, 1974

[MRS95] Monfroy, E., Rusinowitch, M., Schott, R., Implementing non-linear constraints with cooperative solvers, Tech. Report No 2747, INRIA, December 1995

[Neb88] Nebendahl, D. (ed.), Expert Systems-Introduction to the Technology and Applications, John Wiley & Sons, Berlin, 1988

[Neu90] Neumerkel, U., Extensible Unification by Metastructures, in: *Proceedings of META`90*, 1990

[Nil71] Nilsson, N.J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971

[Nov86] Novák, V., *Fuzzy množiny a jejich aplikace*, volume 23 of Matematický seminář SNTL, SNTL, Praha, 1986

[PaCh88] Parsaye, K. and Chignell, M., *Expert Systems for Experts*, John Wiley & Sons, New York, 1988

[Pro96] Prosser, P., MAC-CBJ: maintaining arc consistency with conflict-directed backjumping, Submitted to ECAI-96

[SaBo92] Sannella, M., Borning, A., Multi-Garnet: Integrating Multi-Way Constraints with Garnet, Tech. Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992

[San92] Sannella, M., The SkyBlue Constraint Solver, Tech. Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993

[San94a] Sannella, M., Analyzing and Debugging Hierarchies of Multi-Way Local Propagation Constraints, in: *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, Springer-Verlag, 1994

[San94b] Sannella, M., SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction, in: *Proceedings UIST '94*, 1994

[San94c] Sannela, M., The SkyBlue Constraint Solver and Its Applications, in: *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*, MIT Press, 1994

[SaVH95] Saraswat, V. and Van Hentenryck, P. (eds.), *Principles and Practice of Constraint Programming*, The MIT Press, Cambridge, Massachusetts, 1995

[SFMB92] Sannella, M., Freeman-Benson, B., Maloney, J., Borning,A., Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm, Tech. Report 92-07-05, Department of Computer Science and Engineering, University of Washington, July 1992

[Sha76] Shafer, G., A mathematical theory of evidence, Princeton University Press, 1976

[Sho76] Shortliffe, E.H., Computer-Based Medical Consultations: MYCIN, Elsevier, New York, 1976

[Smi95] Smith, B.M., A Tutorial on Constraint Programming, Tech. Report 95.14, School of Computer Studies, University of Leeds, April 1995

[Ste86] Sterling, L., Meta-Interpreters: The Flavors of Logic Programming?, in: *Proceedings of Workshop on foundation of Logic Programming and Deductive Databases*, Washington, 1986

[Ste88] Sterling, L., Constructing Meta-Interpreters for Logic Programs, in: *Advanced School on Foundations of Logic Programming*, Alghero, Sardinia, Italy, September 1988

[SJK93] Sterling, L., Jain, A. and Kirschenbaum, M., Composition Based on Skeletons and Techniques, Work presented at ILPS `93 Post Conference Workshop on Methodologies for Composing Logic Programs

[StKi93] Sterling, L. and Kirschenbaum, M., Applying Techniques to Skeletons, in: *Constructing Logic Programs*, J.M.J. Jacquet (editor), John Wiley & Sons, 1993

[StLa88] Sterling, L. and Lakhotia, A., Composing Prolog Meta-Interpreters, in: *Proceedings of 5th International Logic Programming Conference*, Seattle, 1988

[StSh86] Sterling, L. and Shapiro, E., *The Art of Prolog*, The MIT Press, Cambridge, Massachusetts, 1986

[Stu89] Sturmfels, B., Dynamic Versions of the Buchberger Algorithm, Tech. Report 89-16.0, RISC Linz, April 1989

[Sut63] Sutherland, I., A Man-Machine Graphical Communication System, Ph.D. thesis, MIT, January 1963

[Tsa93] Tsang, E., *Foundations of Constraint Satisfaction*, Academic Press, 1993

[VaH89] Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, 1989

[VHAK95] Van Hentenryck, P., McAllister, D., Kapur, D., Solving Polynomial Systems Using a Branch and Prune Approach, in: *SIAM Journal on Numerical Analysis*, 1995

[VHM95] Van Hentenryck, P., Michel, L., Newton: Constraint Programming over Nonlinear Real Constraints, Tech. Report CS-95-25, Department of Computer Science, Brown University, 1995

[Wal72] Waltz, D., Generating Semantic Descriptions from Drawings of Scenes with Shadows, Tech. Report AI271, MIT, MA, November 1972

[Wal75] Waltz, D., Understanding Line Drawings of Scenes with Shadows, in: *Psychology of Computer Vision*, McGraw-Hill, New York, 1975

[WiBo89] Wilson, M., Borning, A., Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison, Tech. Report 89-05-04, Department of Computer Science and Engineering, University of Washington, July 1989

[WiBo93] Wilson, M., Borning, A., Hierarchical Constraint Logic Programming, TR 93-01-02a, Department of Computer Science and Engineering, University of Washington, May 1993

[Win84] Winston, P.H., *Artificial Intelligence*, Addison-Wesley, Reading, Massachussetts, 1984

[YaSt88] Yalçinalp, L.Ü. and Sterling, L., An Integrated Interpreter for Explaining PROLOG's Successes and Failures, Case Western Reserve University, CES TR-88-04, April 1988

[Zad65] Zadeh, L.A., Fuzzy Sets, in: *Information and Control*, Vol. 8, 1965

[Zad83] Zadeh, L.A., The Role of Fuzzy Logic in Management of Uncertainty in Expert Systems, in: *Fuzzy Sets and Systems*, Vol. 11, pp. 199-227, 1983

[Zan95] Vander Zanden, Brad, An Incremental Algorithm for Satisfying Hierarchies of Multi-way Dataflow Constraints, Tech. Report, Department of Computer Science, University of Tennessee, March 1995

Přílohy

Příloha A

CSP (Constraint Satisfaction Problems)

V první části přílohy je popsána ohodnocovací část (labelling) systému pro splňování podmínek (viz. kapitola 1.1.1) naprogramovaná v jazyce PROLOG použitím technik mega-interpretů ([Bar93], [BaŠt95]) a zásuvných (plug-in) modulů. Ve druhé části je prezentován kód zásuvného modulu pro práci s podmínkami nad celými čísly. Na závěr jsou uvedeny dva příklady zadání problému do CSP.

Ohodnocovací část systémů pro splňování podmínek, tzv. *labelling*, je relativně jednoduchá procedura. Jejím úkolem je vybrat dosud neohodnocenou proměnnou, zvolit pro ní hodnotu z její domény (`assign_value`) a otestovat platnost podmínek při dosud vybraném ohodnocení proměnných (`test_cons`). Ohodnocovací část v programu realizuje procedura `labelling`, která pro jednoduchost vybírá pro ohodnocení vždy první proměnnou ze seznamu dosud neohodnocených proměnných[3].

```
labelling([V|Vs],Cons):-
     assign_value(V),
     test_cons(Cons,Vs,NewCons,NewVs),
     labelling(NewVs,NewCons).
labelling([],Cons).
```

Testování splnitelnosti podmínek (`test_cons`) se provádí jednoduše tak, že je brána jedna podmínka po druhé a v případě, že lze splnitelnost podmínky otestovat (`callable`), je vyvolána procedura pro test podmínky. Po případném otestování podmínky je rozhodnuto, zda je podmínka již navždy splněna (`finished`) a lze ji tedy z testování po přiřazení hodnot dalším proměnným vypustit. Procedura pro test splnitelnosti podmínky dostává kromě vlastních parametrů také seznam všech dosud neohodnocených proměnných, a může tak při dopředné kontrole nalézt hodnoty i pro další volné proměnné, případně omezit jejich domény[4].

```
test_cons([C|RCons],Vs,NewCons,NewVs):-
     C=..[P|Args],
     (callable(C) -> (CC=..[P,Vs,AuxVs|Args],call(CC))
                   ; AuxVs=Vs),
     (finished(C) -> NewCons=AuxCons
                   ; NewCons=[C|AuxCons]),
     test_cons(RCons,AuxVs,AuxCons,NewVs).
test_cons([],Vs,[],Vs).
```

Dosud prezentované procedury (`labelling` a `test_cons`) tvoří jádro obecného systému pro splňování podmínek. Toto jádro lze nyní doplnit extenzí (zásuvným modulem) realizující konkrétní systém splňování podmínek. Extenze tedy musí obsahovat procedury `assign_value`, `callable`, `finished` a procedury pro otestování splnitelnosti jednotlivých podmínek.

V následující části bude popsán kód jednoduché extenze realizující splňování podmínek nad konečnými množinami celých čísel metodou prohledávání s navracením (backtracking). Proměnné zde budou reprezentovány dvojicí `Proměnná::Doména`, kde `Proměnná` je klasická PROLOGovská proměnná a `Doména` je seznam hodnot,

[3] Proměnné je možné setřídit např. podle velikosti domény před vyvoláním procedury labelling.

[4] Při testu podmínky je také možné přeuspořádat pořadí volných proměnných pro ohodnocování.

kterých může tato proměnná nabývat. Aby nebylo nutné zadávat v doméně vždy všechny její prvky explicitně, je možné použít zkráceného intervalového zápisu `K…L`, kde `K` a `L` jsou celá čísla taková, že `K<L`. Korektní zápis domény potom může vypadat třeba takto: `[1,3,5…9,11]`.

Systém pracuje s běžnými podmínkami nad celými čísly: `eq` (=), `neq` (≠), `lt` (<), `gt` (>) a s podmínkou `diff`, která jako parametr dostává seznam proměnných. `diff([X_1,…,X_n])` je splněna právě tehdy, když jsou hodnoty přiřazené proměnným $X_1,\ldots,X_n$ navzájem různé. Přirozeně tak zastupuje sadu nerovností $X_1 \neq X_2$, …, $X_1 \neq X_n$, $X_2 \neq X_3$, …, $X_{n-1} \neq X_n$.

Protože systém realizuje jednoduché prohledávání s navracením, lze splnitelnost podmínky testovat až tehdy, neobsahuje-li žádné volné (neohodnocené) proměnné. Je-li potom podmínka splněna, lze ji ze systému vyřadit. Výjimku z tohoto pravidla tvoří podmínka `diff`, jejíž splnitelnost lze testovat vždy. Důvodem je to, že se vlastně jedná o meta-podmínku, jejíž jednotlivé složky (nerovnosti, viz. předchozí odstavec) lze testovat dříve než jsou všechny proměnné podmínky `diff` ohodnoceny.

```
assign_value(X::[H|T]):-
     H=(A…B) -> gen_num(X,A,B)
              ; X=H.
assign_value(X::[_|T]):-
     assign_value(X::T).

gen_num(A,A,B):-
     A=<B.
gen_num(X,A,B):-
     A<B, A1 is A+1,
     gen_num(X,A1,B).

callable(C):-no_vars(C),!.
callable(diff(_)).
finished(C):-no_vars(C).

eq(Vs,Vs,A,B):-
     CA is A, CB is B, CA=CB.
neq(Vs,Vs,A,B):-
     CA is A, CB is B, CA\=CB.
lt(Vs,Vs,A,B):-
     CA is A, CB is B, CA<CB.
gt(Vs,Vs,A,B):-
     CA is A, CB is B, CA>CB.
diff(Vs,Vs,List):-
     diff(List).
diff([H|T]):-
     (nonvar(H),mem(H,T)) -> fail ; diff(T).
diff([]).
```

Podobným způsobem lze snadno naprogramovat i sofistikovanější extenze realizující třeba prohledávání s dopřednou kontrolou (forward checking). V takovýchto extenzích je možné testovat splnitelnost podmínky i když obsahuje jednu volnou proměnnou. Podle druhu podmínky lze potom dopočítat hodnotu této volné proměnné (`eq`), nebo alespoň omezit její doménu (`neq`, `lt`, `gt`, `diff`).

Příklady:

a) Kryptoaritmetika

Najděte řešení rovnice SEND + MORE = MONEY tak, aby každé písmeno odpovídalo jedné cifře a všechna písmena měla přiřazena navzájem různé cifry.

V řeči CSP by zadání problému mohlo vypadat třeba takto:

proměnné a jejich domény:

```
S,M::[1…9]; E,N,D,O,R,Y::[0…9]
```

omezující podmínky:

```
eq((1000*S+100*E+10*N+D)+(1000*M+100*O+10*R+E),
   10000*M+1000*O+100*N+10*E+Y),
diff([S,E,N,D,M,O,R,Y])
```

Z důvodů efektivity je vzhledem k charakteru systémů pro splňování podmínek vhodnější jednu „složitou" podmínku svazující větší množství proměnných nahradit větším počtem jednodušších podmínek. V tomto konkrétním příkladě by potom zadání problému mohlo vypadat takto:

proměnné a jejich domény:

```
S,M::[1…9]; E,N,D,O,R,Y::[0…9]; P1,P2,P3::[0,1]
```

omezující podmínky:

```
eq(D+E,10*P1+Y), eq(N+R+P1,10*P2+E),
eq(E+O+P2,10*P3+N), eq(S+M+P3,10*M+O),
diff([S,E,N,D,M,O,R,Y])
```

Systém řešení podmínek nalezl následující ohodnocení proměnných:

```
S/9,E/5,N/6,D/7,M/1,O/0,R/8,Y/2,P1/1,P2/1,P3/0
```

b) Problém N dam

Najděte rozmístění N dam na šachovnici N×N tak, aby se navzájem neohrožovaly. N je přirozené číslo větší než 1.

Formalizaci problému do CSP si ukážeme na příkladě pro N=4:

proměnné a jejich domény:

```
N1,N2,N3,N4::[1…4]
```

proměnná N_i odpovídá dámě umisťované do řádku i, hodnota této proměnné určuje pořadové číslo sloupce, ve kterém bude dáma umístěna

omezující podmínky:

```
diff([N1,N2,N3,N4]),
neq(N1-N2,1), neq(N1-N2,-1), neq(N1-N3,2),
neq(N1-N3,-2), neq(N1-N4,3), neq(N1-N4,-3),
neq(N2-N3,1), neq(N2-N3,-1), neq(N2-N4,2),
neq(N2-N4,-2), neq(N3-N4,1), neq(N3-N4,-1)
```

podmínka vzájemného neohrožení dam N_i a N_j ($i<j$) je zde reprezentována trojicí omezujících podmínek: `neq(Ni,Nj)`-implicitně obsaženo v `diff([N1,N2,N3,N4])`, `neq(Ni-Nj,j-i)` a `neq(Ni-Nj,i-j)`

Systém řešení podmínek nalezl dvě různá ohodnocení proměnných:

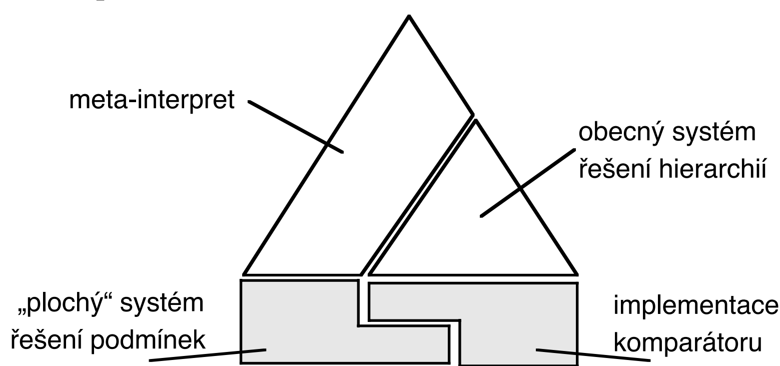
```
N1/2,N2/4,N3/1,N4/3
N1/3,N2/1,N3/4,N4/2.
```

Příloha B

Základní HCLP interpret

Příloha je věnována implementaci obecného interpretu HCLP programů vytvořeném v jazyce PROLOG použitím technik mega-interpretů ([Bar93], [BaŠt95]) a zásuvných (plug-in) modulů. Nejprve je popsán jednoduchý meta-interpret pro běh HCLP program[illegible] následovaný popisem obecného systému pro řešení hierarchií. Ve druhé části jsou uk[illegible] implementací zásuvného modulu realizujícího srovnávání locally-predicate-[illegible] komparátorem a zásuvného modulu pro řešení podmínek nad Herbrandovým un[illegible]

Strukturu navrženého obecného interpretu HCLP ukazuje násled[illegible] Jednotlivé složky systému mezi sebou komunikují „přes společné hrany“. [illegible] pravidlo, že výše umístěná složka používá služeb níže umístěné složky, tj. [illegible] volá služby systému řešení hierarchií a systému řešení podmínek, systém řeš[illegible] používá pouze implementaci komparátoru a implementace komparátor[illegible] systém řešení podmínek.



Navrhovaná architektura interpretu HCLP umožňuje snadnou kombinací modulů pr[illegible] řešení podmínek s moduly realizujícími různé komparátory dosáhnout vytvoř[illegible]í libovolného interpretu HCLP(D,Comp) nad různými doménami D a s použitím rů[illegible]ých komparátorů Comp. V praxi byly dosud realizovány moduly pro všechny b[illegible]né komparátory predikátového typu (viz. kapitola 1.2.2).

Meta-interpret pro interpretaci HCLP programu vychází z myšlenek jednoduchého interpretu HCLP (viz. kapitola 1.5.1). Jeho úkolem je zredukovat cíl zadaný jako seznam atomických cílů použitím pravidel programu. Nutné podmínky jsou při tom ihned předávány k vyřešení systému pro řešení podmínek, zatímco měkké podmínky jsou pouze shromažďovány. Jakmile se podaří cíl zredukovat, předá se nashromážděný zásobník podmínek s dosud nalezeným řešením nutných podmínek systému pro řešení hierarchií podmínek.

Omezující podmínky se zadávají ve tvaru `C@L`, kde `C` je vlastní podmínka[5] a `L` je její preference. Názvy preferencí jsou uživatelsky definované a určuje je fakt `lab(List)`, kde `List` je seznam názvů preferencí setříděných od nejsilnějších k nejslabším, např.:

```
lab([required,strong,prefer,weak]).
```

Při redukci atomického cíle generuje meta-interpret omezující podmínku `Cíl=Hlava`, kde `Cíl` je řešený atomický cíl a `Hlava` je hlava varianty klauzule použité pro redukci. Systém řešení podmínek tedy musí být schopen řešit minimálně rovnosti.

[5] Tvar omezující podmínky je určen systémem pro řešení podmínek.

```
solve([C@L|T],CurrAnsw,Hier,Answ):-!,
    add_constr(C,L,CurrAnsw,Hier,NewAnsw,NewHier),
    solve(T,NewAnsw,NewHier,Answ).
solve([G|T],CurrAnsw,Hier,Answ):-
    reduce(G,CurrAnsw,NewG,NewAnsw),
    add_subgoal(NewG,T,NewT),
    solve(NewT,NewAnsw,Hier,Answ).
solve([],CurrAnsw,ConstrHier,Answ):-
    solve_constr_hier(ConstrHier,[CurrAnsw],Answ).

reduce(SG,OldAnsw,true,NewAnsw):-
    system(SG),!,
    copy_term(SG,CopySG),
    call(CopySG),
    solve_constr(SG=CopySG,OldAnsw,NewAnsw).
reduce(G,OldAnsw,NewG,NewAnsw):-
    copy_term(G,CopyG),
    clause(CopyG,NewG),
    solve_constr(G=CopyG,OldAnsw,NewAnsw).

add_constr(Const,Lab,OldAnsw,OldHier,NewAnsw,NewHier):-
    lab(Labs),
    (Labs=[Lab|_]
        -> solve_constr(Const,OldAnsw,NewAnsw),
               NewHier=OldHier
               % nutné podmínky jsou řešeny ihned
         ; constr_to_hier(Const,Lab,OldHier,NewHier),
               NewAnsw=OldAnsw).
               % měkké podmínky se shromažďují

constr_to_hier(Const,Lab,[C@L|T],[C@L|NewT]):-
    stronger(L,Lab),!,  % L je silnější než Lab
    constr_to_hier(Const,Lab,T,NewT).
constr_to_hier(Const,L,[Cs@L|T],[[Const|Cs]@L|T]):-!.
constr_to_hier(Const,Lab,T,[[Const]@Lab|T]).
```

Jak je vidět z předchozího kódu, udržuje se během redukce cíle hierarchie měkkých podmínek ve tvaru seznamu, ve kterém jsou podmínky rozděleny do jednotlivých úrovní od nejsilnějších po nejslabší. Po úspěšné redukci cíle je tato hierarchie spolu s řešením nutných podmínek předána systému pro řešení hierarchií.

Systém pro obecné řešení hierarchií musí být dostatečně flexibilní, aby podporoval všechny druhy komparátorů. Proto v sobě implicitně obsahuje jedinou společnou vlastnost komparátorů a tou je respektování hierarchie podmínek (viz. kapitola 1.2.2). Respektování hierarchie podmínek se dosáhne jednoduše tak, že systém řeší hierarchii podmínek postupně po vrstvách od nejsilnější po nejslabší. Částečná řešení nalezená na preferovanějších vrstvách jsou potom předávána k dořešení na preferenčně slabší vrstvy. Pokud po vyřešení celé hierarchie stále zbyde více řešení, jsou jedno po druhém vracena uživateli (`select_answer`).

```
solve_constr_hier([TopLevel@L|WeakerLs],PartAnsws,Answ):-
   solve_level(TopLevel,PartAnsws,SubAnsws),
   solve_constr_hier(WeakerLs,SubAnsws,Answ).
solve_constr_hier([],AnswList,Answ):-
   select_answer(Answ,AnswList).
```

Dosud prezentovaný kód představuje jádro obecného interpretu HCLP, z kterého lze doplněním o modul pro řešení podmínek nad doménou D a modul realizující komparátor Comp získat interpret pro HCLP(D,Comp). V následující části je popsán zásuvný modul realizující vyřešení množiny podmínek se stejnou preferencí použitím locally-predicate-better (LPB) komparátoru. Tento komparátor zde byl zvolen z důvodů jeho širokého používání (viz. kapitola 1.5) a průhledné a přímočaré implementace.

Nalezení řešení množiny stejně preferovaných podmínek použitím LPB komparátoru odpovídá nalezení maximální podmožiny splnitelných podmínek, které zároveň nejsou ve sporu s dosud nalezených částečným řešením. Maximalita množiny zde znamená, že tuto množinu nelze rozšířit o další splnitelnou podmínku (viz. množinová definice LPB komparátoru). Hledání všech maximálních podmnožin splnitelných podmínek je realizováno procedurou `solve_level_constr` formou jednoduchého prohledávání s navracením.

```
solve_level(Cs,[PartA|_],[LevelAnsw]):-
    solve_level_constr(Cs,PartA,LevelAnsw).
solve_level(Cs,[_|PartAs],LevelAnsws):-
    solve_level(Cs,PartAs,LevelAnsws).

solve_level_constr([C|R],PartAnsw,Answ):-
    solve_constr(C,PartAnsw,SubAnsw),
    solve_level_constr(R,SubAnsw,Answ).
solve_level_constr([C|R],PartAnsw,Answ):-
    solve_level_constr(R,PartAnsw,Answ),
    not solve_constr(C,Answ,_).
      % řešení nelze rozšířit o řešení podmínky C
solve_level_constr([],Answ,Answ).
```

Pro otestování vlastností obecného interpretu HCLP byl vytvořen modul pro řešení podmínek nad Herbrandovým univerzem [Llo84]. Tento modul umí řešit rovnosti a nerovnosti nad libovolnými termy definované takto:

$$X=Y \Leftrightarrow_{def} \exists\theta\ (X\theta \equiv Y\theta)$$

$$X\neq Y \Leftrightarrow_{def} \forall\theta\ \neg(X\theta \equiv Y\theta),$$

kde θ znamená substituci a $\equiv$ znamená syntaktickou rovnost termů. Rovnost tak neznamená nic jiného než existenci unifikace.

Pro predikátové komparátory jako je LPB stačí, aby systém řešení podmínek uměl přidat řešení podmínky k dosud nalezenému řešení. V případě Herbrandova univerza a jazyka PROLOG použitého pro implementaci můžeme využít podkladového unifikačního algoritmu. Rovnosti lze potom řešit přímo, nerovnosti, o kterých systém není schopen rozhodnout, zda budou pořád platit nebo ne (např. $X\neq Y$), se budou shromažďovat a v případě přiřazení hodnoty nějaké proměnné se platnost těchto nerovností opět otestuje.

```
solve_constr(L=R,OldA,NewA):-
    L=R,   % test rovnosti
    solve_constr_list(OldA,[],NewA).
          % opakování testu nerozhodnutých nerovností
solve_constr(L\=R,OldA,NewA):-
    L\=R  -> NewA=OldA        % nerovnost je splněna
          ; (L==R -> fail ; NewA=[L\=R|OldA]) .

solve_constr_list([H|T],OldA,NewA):-
    solve_constr(H,OldA,SubA),
    solve_constr_list(T,SubA,NewA).
solve_constr_list([],Answ,Answ).
```

Příklady:

Následující dvojice příkladů ukazuje řešení hierarchie podmínek nalezená použitím systému popsaného v předchozí části této přílohy.

1)

```
solve_constr_hier([[X=Y,Z\=Y,X=b,Z=b]@strong,[Y=c]@weak,
                  [[]],A).

X = b
Y = b
A = [Z\=b] ;

X = c
Y = c
Z = b
A = [] ;

X = b
Y = b
Z = b
A = [] ;

X = b
Y = c
Z = b
A = [] ;

no
```

2)

```
solve_constr_hier([[X=a,Y=b,Y\=b]@strong],[[X=Y]],A).

X = a
Y = a
A = [] ;

X = b
Y = b
A = [] ;

no
```

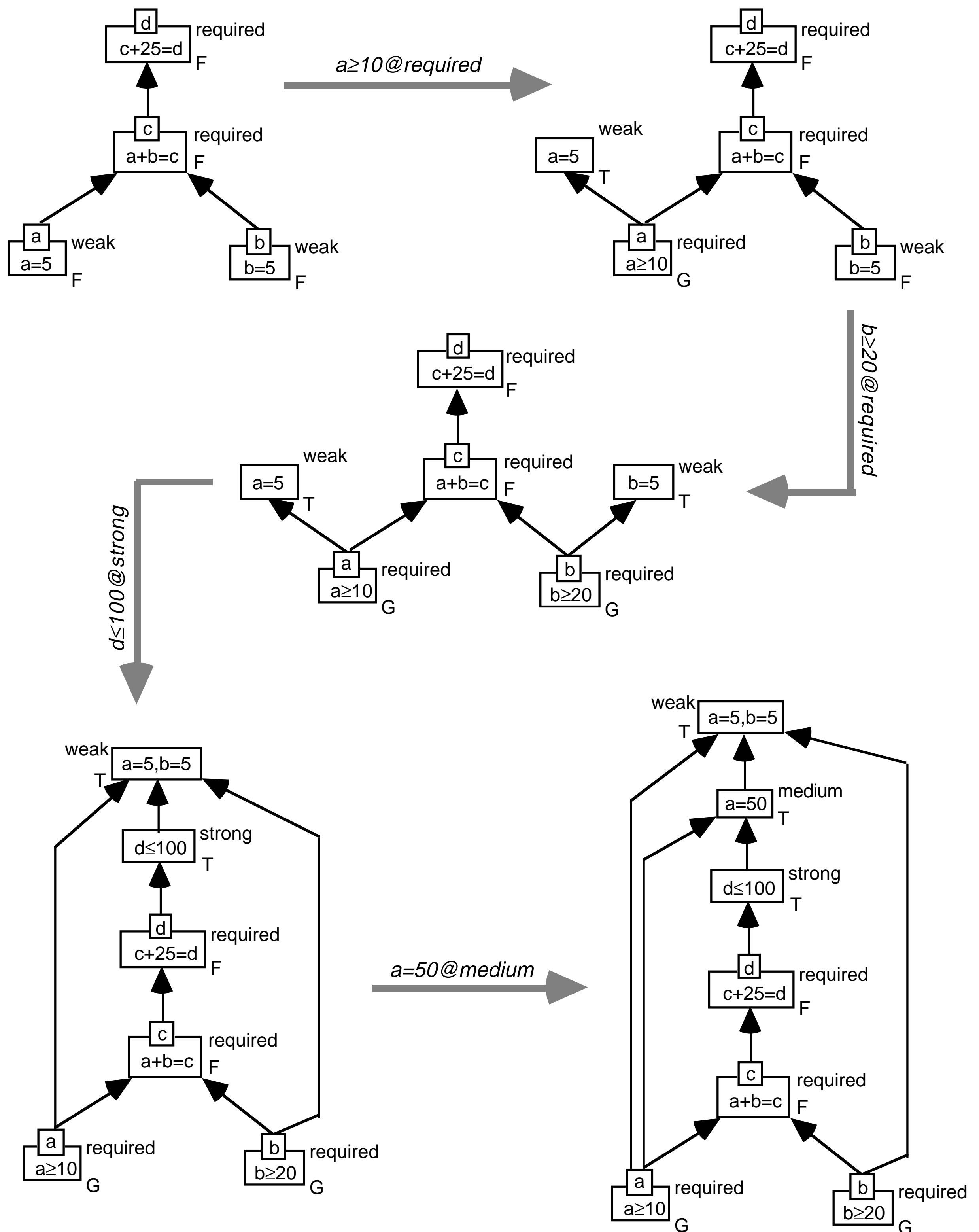
Příloha C

Plánovací algoritmus zjemňovací metody

```
add_constraint(c@l,(CC,E))
%% přidá označenou podmínku c@l do sítě podmínek (CC,E)
%% vrací novou síť podmínek

  %% zařazení podmínky do buňky
  if ∃Cell∈CC & i_strength(Cell)=l then
    % Cell=(Cs,In,Out)
    Cs:=Cs∪{c@l}         % přidej c@l do buňky Cell
  else
    Cell:=({c@l},{},{})  % vytvoř novou buňku
    CC:=CC∪{Cell}
  end if

  %% rozdělení proměnných a úprava hran tak, aby platila podmínka 5)
     definice 2.12
  V:=vars(c)
  for v∈V do
    if ∃Cell'∈CC & Cell'=(Cs',In',Out') & v∈Out' then
    % proměnná v je již určena nějakou buňkou sítě
      case of
        :(i_strength(Cell')<l)
        % proměnná v je určována silnější buňkou
          % Cell=(Cs,In,Out)
          In:=In∪{v}  % přidej v mezi vstupní proměnné buňky Cell
          E:=E ∪ (Cell',Cell)  % přidej příslušnou hranu
        :(i_strength(Cell')>l)
        % proměnná v je určována slabší buňkou
          % Cell'=(Cs',In',Out')
          % přeřaď v v buňce Cell' z výstupních do vstupních proměnných
          Out':=Out'-{v}
          In':=In'∪{v}
          % vyřaď hrany, které přeřazením v pozbyly platnost
          E:=E-{(Cell',C2) | (Cell',C2)∈E & C2=(Cs2,In2,Out2) &
                              Out'∩In2=∅}
          % Cell=(Cs,In,Out)
          Out:=Out∪{v}  % přidej v mezi výstupní proměnné buňky Cell
          % přidej příslušné hrany
          E:=E ∪ {(Cell,C2) | C2∈CC & C2=(Cs2,In2,Out2) & v∈In2}
        % pokud i_strength(Cell')=l, potom se nic menění
      end case
    else
    % proměnná v dosud v síti není
      % Cell=(Cs,In,Out)
      Out:=Out∪{v}  % přidej v mezi výstupní proměnné buňky Cell
    end if
    V:=V-{v}
  end for

  %% přidání hran tak, aby platila podmínka 7) definice 2.12
  E:=E∪{(Cell,C2) | C2∈CC & i_strength(C2)=min{i_strength(C)|C∈CC &
                  i_strength(C)>l}}
  E:=E∪{(C2,Cell) | C2∈CC & i_strength(C2)=max{i_strength(C)|C∈CC &
                  i_strength(C)<l}}

  return (CC,E)
end add_constraint
```

Příloha D

Tvorba sítě podmínek sofistikovaným plánovacím algoritmem



Příloha E

Schéma algoritmu pro HCLP s mezi-hierarchickým porovnáním

```
solve_goal(Goal,Program)
%% řeší cíl Goal použitím HCLP programu Program
%% vrací seznam řešení

  % vytvoř seznam cílů k vyřešení obsahující prvotní cíl Goal
  list:=insert_to_list(Goal,new_list)
  % vytvoř prázdný seznam pro řešení
  solved:=new_list

  while not empty(list)
    GoalToReduce:=delete_first(list)
    % redukuj cíl Goal použitím všech možných pravidel z programu Program
    GoalList:=reduce(Goal,Program)
    % zařaď seznam cílů po redukci
    distribute_list(GoalList,list,solved)
  end while

  return solved

end solve_goal



distribute_list(A,list,solved)
%% zařadí redukované cíle ze seznamu A do seznamů list resp. solved
   podle toho, zda je daný cíl již vyřešen; v seznamu solved nechá
   pouze nejlepší řešení a v seznamu list pouze cíle, které mohou vést
   k nejlepšímu řešení

  while not empty(A)
    H:=delete_first(A)
    if solved(H) then
      % cíl je již vyřešen
      case of
        :better(H,best_of(solved))
        % nové řešení je lepší než všechna stará řešení
        % -> vytvoř nový seznam solved obsahující pouze nové řešení
          solved:=insert_to_list(H,new_list)
          list:=delete_worst(H,list)   % vyřaď cíle, o kterých víš,
                                         že vedou k horšímu řešení
                                         než je současné řešení
        :not better(best_of(solved),H)
        % nové řešení není horší (je stejně dobré) než stará řešení
        % -> přidej nové řešení do seznamu řešení
          solved:=insert_to_list(H,solved)
        % pokud je nové řešení horší než stará řešení, zapomene se
      end case
    else
      % cíl ještě není vyřešen
      list:=insert_to_list(H,list)
    end if
  end while

end distribute_list
```