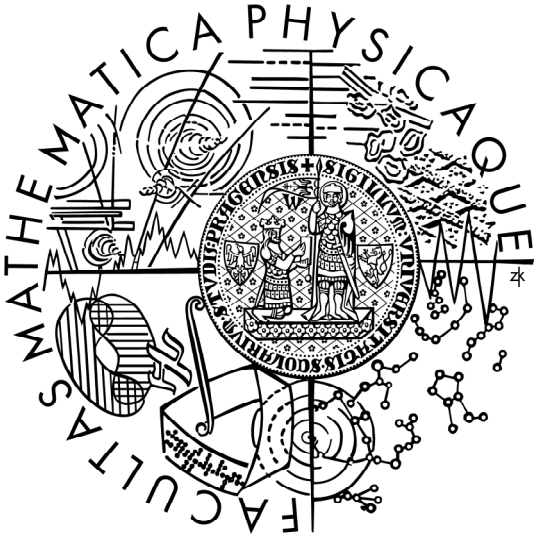
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Pavel Surynek

Řešení dynamických problémů s podmínkami

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

PRAHA 2003

Děkuji všem, kteří mne při sepisování této práce podpořili. Mé zvláštní poděkování patří RNDr. Romanu Bartákovi, Ph.D. za odborné vedení a množství cenných rad.

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

V Praze dne 18. listopadu 2003 *Pavel Surynek*

Obsah

Abstrakt **v**

Úvod **1**

1 Programování s omezujícími podmínkami **3**

1.1 Úvod 3

1.2 Problém splňování podmínek 3

1.3 Ekvivalence a binarizace problémů splňování podmínek 5

1.4 Globální podmínky 12

1.5 Řešení problémů s podmínkami 13

2 Vybrané konzistenční algoritmy **15**

2.1 Hranová konzistence 16

2.2 Konzistenční algoritmus AC-3 18

2.3 Hranová konzistence založená na podporách - algoritmus AC-4 21

2.4 Vylepšená hranová konzistence se seznamy podpor - algoritmus AC-6 24

3 Řešení dynamických problémů s podmínkami **28**

3.1 Dynamické problémy splňování podmínek 29

3.2 Udržování hranové konzistence v dynamických problémech 29

3.3 Inkrementální hranová konzistence 30

3.4 Plně dynamická hranová konzistence - algoritmus DnAC-4 31

3.5 Vylepšení algoritmu DnAC-4 - algoritmus DnAC-6 38

3.6 Dynamická hranová konzistence bez seznamů podpor - algoritmy AC|DC 44

3.7 Závěr 65

4 Empirická analýza algoritmů pro dynamickou hranovou konzistenci 66

4.1 Náhodný problém 66

4.2 Empirická analýza AC|DC-2 67

4.3 Zhodnocení výsledků 73

Závěr a možnosti dalšího vývoje 75

A Algoritmus AC|DC-2 pro nebinární podmínky 77

B Implementace experimentální knihovny 81

B.1 Objektová struktura knihovny 81

B.2 Reprezentace problému splňování podmínek 82

B.3 Služby pracující s reprezentací problému 85

B.4 Ukázkové a testovací programy 88

C Další empirické testy 89

Literatura 99

Abstrakt

Název práce:	Řešení dynamických problémů s podmínkami
Autor:	Pavel Surynek
Katedra (ústav):	Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce:	RNDr. Roman Barták, Ph.D.
e-mail vedoucího:	bartak@kti.mff.cuni.cz

Abstrakt: Problémy splňování podmínek (CSP) jsou s vzhledem k možnostem praktického použití velmi zkoumanou oblastí kombinatorického prohledávání (optimalizace). CSP je definován množinou proměnných, kterým mají být přiřazeny hodnoty, a množinou podmínek, jež omezují tato přiřazení. Jelikož mnoho praktických problémů vyžaduje dynamické prostředí, byl koncept CSP rozšířen na dynamický CSP (DCSP), kde se množina proměnných a/nebo podmínek může měnit během řešícího procesu. S ohledem na prohledávací algoritmy jsou problémy splňování podmínek obvykle nejprve zjednodušovány pomocí konzistenčních (filtračních) technik, jako je například hranová konzistence. V této práci jsme se zaměřili na studium otázky udržování hranové konzistence v DCSP. Navrhli jsme nový algoritmus pro dynamickou hranovou konzistenci, který představuje lepší kompromis mezi paměťovými a časovými požadavky v porovnání s podobnými existujícími algoritmy, jako jsou DnAC-4, DnAC-6 a AC|DC. Aby bylo možné provádět výkonnostní testy, vyvinuli jsme knihovnu `SPlan` napsanou v C++. Knihovna je určena pro řešení DCSP a nový algoritmus je implementován jako její součást. Pomocí experimentálních testů na náhodně generovaných DCSP jsme demonstrovali efektivitu nového algoritmu při praktickém použití.

Klíčová slova: dynamické CSP, dynamická hranová konzistence, AC|DC

Title:	Dynamic Constraint Satisfaction Problems
Author:	Pavel Surynek
Department:	Department of Theoretical Computer Science and Mathematical Logic
Supervisor:	RNDr. Roman Barták, Ph.D.
Supervisor's e-mail address:	bartak@kti.mff.cuni.cz

Abstract: Constraint satisfaction problems (CSPs) are a type of combinatorial (optimization) problems that invite much interest in many practical applications. CSP is defined by a set of variables, to which values must be assigned, and a set of constraints that restrict these assignments. Since many problems of practical interest require a dynamic environment, the model of CSP was extended to a dynamic CSP (DCSP), in which the set of variables and/or constraints can be modified during the constraint resolution process. To simplify the constraint satisfaction problem for search algorithms, consistency (filtering) techniques like arc-consistency are usually applied. In this thesis, we study the problem of maintaining arc-consistency in DCSPs. We propose a new dynamic arc-consistency algorithm that yields a better compromise between time and space in comparison to similar algorithms like DnAC-4, DnAC-6 and AC|DC. In order to do performance experiments we developed a library `SPlan` written in C++. This library solves DCSPs and the new algorithm is a part of this library. Experimental results on randomly generated DCSPs demonstrate the practical efficiency of our new algorithm.

Keywords: dynamic CSPs, dynamic arc-consistency, AC|DC

Úvod

Diplomová práce Řešení dynamických problémů s podmínkami si klade za cíl seznámit čtenáře s problematikou dynamických problémů a v kontextu existujících technik pro práci s dynamickými problémy prezentovat nové techniky a výsledky získané během vypracování.

Práce je v první řadě koncipována jako výzkumná zpráva informující o dosažených výsledcích, nicméně nepředpokládá žádné speciální znalosti z oblasti programování s omezujícími podmínkami, všechny užívané definice, tvrzení a algoritmy jsou v přiměřené míře popsány přímo v práci. Detailnější popis je pak věnován zejména těm existujícím přístupům, jež jsou v práci dále rozvíjeny a na nichž jsou založeny nové techniky.

Prozatím stručně naznačme, že dynamický problém splňování podmínek je problém, u něhož dochází v průběhu řešení ke změnám množiny proměnných a podmínek, tj. k přidávání a ubírání proměnných a stejně tak podmínek. Reálně se dynamické problémy hojně vyskytují v nejrůznějších oblastech. Asi nejvýznačnější oblastí aplikace technik z dynamických problémů je plánování a rozvrhování. Obecněji řečeno, techniky z dynamických problémů je možno uplatnit všude tam, kde konečná podoba problému není předem známa a nějakým způsobem závisí na průběhu řešení v předchozích krocích.

U dynamických problémů jsou studovány hlavně otázky *udržování řešení* a *udržování konzistence* (později budou pojmy přesně vymezeny) vzhledem k operacím měnícím strukturu problému (přidání, odebrání proměnné či podmínky). Tato práce se zabývá druhým jmenovaným tématem, tedy udržováním konzistence, konkrétně konzistence hranové. V této oblasti bylo vyvinuto již několik algoritmů, avšak stále zde existuje prostor k dalšímu zdokonalování. Novým původním výsledkem práce je algoritmus pro udržování hranové konzistence vzhledem k operacím měnícím strukturu problému, který hned v několika ohledech zlepšuje dosavadní přístupy.

Rozvržení práce je následující. První kapitola obecně uvádí čtenáře do problematiky splňování podmínek, zavádí základní pojmy a definice a obecným způsobem charakterizuje techniky používané v programování s omezujícími podmínkami. Poněkud podrobnější výklad je zde věnován otázkám binarizace, na které se pak odvolává kapitola popisující samotné algoritmy pro udržování hranové konzistence v dynamických problémech (zkráceně budeme dále tyto algoritmy označovat buď jen jako dynamické nebo jako algoritmy pro dynamickou konzistenci).

Druhá kapitola popisuje existující algoritmy pro výpočet hranové konzistence pro klasické problémy (nedynamické), na jejich základě jsou pak postaveny dynamické algoritmy v další kapitole. V druhé kapitole jsou rovněž zavedeny definice a operace, které později slouží jako stavební kameny pro dynamické algoritmy. V souvislosti s popisem algoritmů zde zavádíme programový zápis, který pak bude užíván v celém zbytku práce.

Třetí kapitola představuje jádro práce, zde jsou popsány, teoreticky zhodnoceny a srovnány existující algoritmy pro dynamickou hranovou konzistenci. V kontextu tohoto popisu je pak rozvíjen nový přístup, jenž vyúsťuje návrhem nového algoritmu a příslušnou analýzou, tj. důkazem korektnosti, rozborem časové a prostorové složitosti v nejhorším případě a dalšími tvrzeními o navrženém algoritmu.

Čtvrtá kapitola je věnována empirickému srovnání nového algoritmu se srovnatelnými existujícími přístupy. Hlavním cílem této kapitoly je potvrdit přínos nového algoritmu při praktickém použití (v průměrném případě).

Dále následují celkem tři přílohy, do kterých byly přenechány určité doplňující informace, které nebylo vhodné zařazovat do vlastního textu práce. Jedná se o rozšíření navrženého algoritmu pro nebinární podmínky (příloha A), dále o stručný popis experimentální knihovny `SPlan` napsané v jazyce C++, která byla implementována za účelem ověřování hypotéz a provádění empirických testů (příloha B) a nakonec o doplňující statistiku operací srovnávaných dynamických algoritmů (příloha C).

Součástí diplomové práce je i přiložené CD, na němž se nacházejí zdrojové texty knihovny `SPlan` a testovacích programů, které byly použity k provádění empirických testů, plus kompletní výsledky měření provedených v rámci empirických testů.

Kapitola 1

Programování s omezujícími podmínkami

1.1 Úvod

Programování s omezujícími podmínkami (*Constraint Programming*) představuje vzhledem ke své univerzálnosti a efektivitě velmi užitečný nástroj pro řešení celé řady reálných problémů z nejrůznějších oborů. Omezující podmínky jsou s úspěchem nasazovány v plánování, rozvrhování, umělé inteligenci, při řešení kombinatorických problémů, v počítačové grafice, uživatelských rozhraních a v dalších oblastech. Základ k širokým možnostem nasazení omezujících podmínek je dán zejména formalizmem, který poskytují pro popis problémů, a existencí efektivních algoritmů pro hledání řešení těchto problémů.

Popis problémů pomocí omezujících podmínek je založen pouze na matematických pojmech (*proměnná*, *podmínka*) a je tak nezávislý na jakémkoli programovacím jazyce. Integrace omezujících podmínek do libovolného programovacího jazyka je díky této vlastnosti naopak značně usnadněna. Důležitou vlastností rovněž je, že v popisu problému není nikterak určeno, jakým způsobem má být problém řešen. V tomto smyslu je popis problémů do značné míry oddělen od algoritmů pro hledání řešení.

V současnosti je k dispozici poměrně velké množství efektivních řešících algoritmů pro omezující podmínky. Základem většiny z nich je prohledávání prostoru všech možných ohodnocení ve spojení s tzv. *konzistenčními technikami*, které prohledávaný prostor různě omezují (zmenšují). Uživatel může pro řešení problémů zvolit některý z tzv. *systematických* algoritmů, které postupně prohledávají celý prostor možných řešení a mohou dokázat neexistenci řešení či nalézt všechna řešení. Nebo se uživatel může rozhodnout pro některý z tzv. *lokálních* algoritmů, jež prostor možných řešení prohledávají nesystematicky a nemohou prokázat neexistenci řešení ani nalézt řešení všechna, avšak dosahují dobrých výsledků u velmi rozsáhlých problémů, u nichž je použití systematických algoritmů pro značnou velikost prohledávaného prostoru problematické.

Prakticky všechny implementace řešících algoritmů určitým způsobem aplikují nejrůznější heuristiky pro vytyčení správného směru prohledávání. Obecně lze říci, že omezující podmínky nabízejí pro nasazení heuristik rozsáhlý prostor. Jelikož se při hledání řešení problémů reálné velikosti bez užití heuristik prakticky nelze obejít, je toto také jeden z důvodů, proč zde programování s omezujícími podmínkami nachází značné uplatnění.

1.2 Problém splňování podmínek (CSP)

Ústředním pojmem programování s omezujícími podmínkami je *problém splňování podmínek* (*Constraint Satisfaction Problem - CSP*).

Problém splňování podmínek sestává z *proměnných* a *omezujících podmínek*. Termín proměnná uvažujeme v běžném matematickém smyslu. Každá proměnná má přiřazenu konečnou množinu hodnot, kterých může nabývat (např. $X \in \{1,2,\ldots,10\}$). Tato množina hodnot se nazývá *doména* proměnné. Zpravidla se jedná o množinu celých čísel, neboť ve většině případů je snadné zakódovat pomocí celých čísel i jiné objekty, se kterými je třeba pracovat.

Omezující podmínka je libovolná relace nad proměnnými, které svazuje, tj. podmnožina kartézského součinu domén příslušných proměnných (např. $X < 3Y + 1$, $X \geq 4$, X, Y, Z *různé*). Podmínka vyjadřuje souvislost mezi proměnnými a určuje, které kombinace hodnot daných proměnných jsou přípustné a které ne. Jinými slovy, podmínka omezuje domény svých proměnných na přípustné kombinace hodnot. Poznamenejme, že v omezujících podmínkách neklademe žádná omezení na typy hodnot proměnných, které svazují (podmínka může svazovat např. celočíselnou a booleovskou proměnnou). Podle arity rozlišujeme podmínky unární, binární, ternární atd. Kvůli jednoduššímu a přehlednějšímu zápisu algoritmů se často uvažují bez újmy na obecnosti podmínky nejvýše binární. To je umožněno díky tomu, že podmínky vyšší arity než 2 lze reprezentovat pomocí binárních podmínek, přesný postup, jak to provést, bude ukázán později.

Formální vymezení problému splňování podmínek podává následující definice, podobnou definici lze též nalézt v práci [Tsa93].

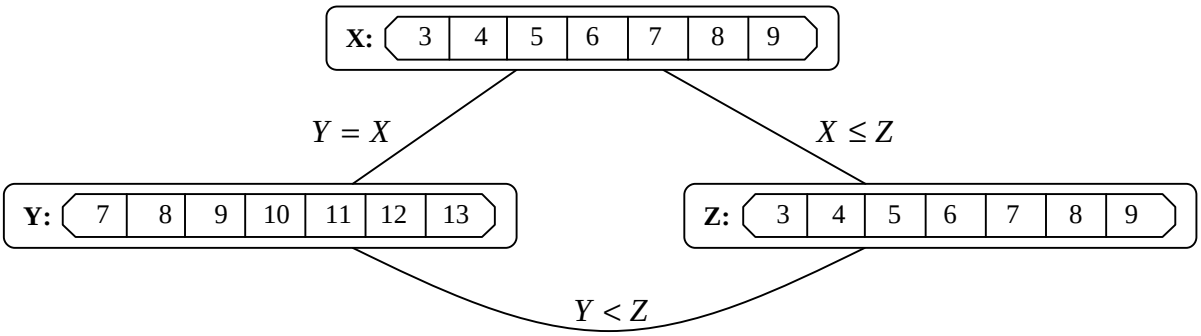
Definice 1.1 (PROBLÉM SPLŇOVÁNÍ PODMÍNEK)**.** *Problém splňování podmínek (Constraint Satisfaction Problem - CSP) je dvojice* (V, C)*, kde* V *je konečná množina proměnných, pro kterou platí, že každá proměnná* $v \in V$ *má přiřazenu konečnou doménu* D_v*, a* C *je konečná množina omezujících podmínek nad proměnnými z množiny* V*. Pro podmínku* $c \in C$ *bude symbol* V_c *označovat množinu proměnných svázaných podmínkou* c*.* ■

Binární podmínku svazující proměnné u a v budeme někdy též označovat symbolem (u, v). Následující příklad ukazuje jednoduchý problém splňování podmínek obsahující tři binární podmínky.

Příklad 1.1. $P = (V, C)$, kde $V = \{X, Y, Z\}$, přičemž
$D_X = \{3,4,5,6,7,8,9\}$; $D_Y = \{7,8,9,10,11,12,13\}$ a $D_Z = \{3,4,5,6,7,8,9\}$;
$C = \{(X = Y),\ (X \leq Y),\ (Y < Z)\}$. ■

Problém splňování podmínek je obvykle znázorňován jako neorientovaný graf, pokud obsahuje nejvýše binární podmínky, popřípadě jako neorientovaný hypergraf, pokud obsahuje i podmínky vyšší arity než 2. Proměnné jsou v grafu reprezentovány vrcholy. Podmínka mezi danými proměnnými je v grafu reprezentována hranou, resp. hyperhranou mezi příslušnými vrcholy. Takto zkonstruovaný graf někdy též označujeme jako *graf problému*.

Následující obrázek znázorňuje problém z příkladu 1.1 jako graf.



Obrázek 1.1: Znázornění problému splňování podmínek pomocí grafu

Uvědomme si, že unární podmínky je možné rovnou přenést do domén proměnných. Naopak na doménu proměnné můžeme nahlížet jako na unární podmínku.

K definici řešení problému splňování podmínek $P = (V, C)$ budeme užívat pojmu *ohodnocení*. Ohodnocení je přiřazení hodnot proměnným. Formálně můžeme ohodnocení proměnných z množiny V definovat jako substituci $\sigma = \{v_1/d_1, v_2/d_2, \ldots, v_m/d_m\}$, kde $\{v_1, v_2, \ldots, v_m\} \subseteq V$ a $d_i \in D_{vi}$ pro $i = 1,2,\ldots,m$. Proměnná v_i nabývá při ohodnocení σ hodnoty $\sigma(v_i) = d_i$. Přiřazuje-li ohodnocení σ hodnotu všem proměnným daného problému, nazýváme σ *úplným ohodnocením*. Pokud ohodnocení σ přiřazuje hodnotu pouze některým proměnným, hovoříme o σ jako o *částečném ohodnocení*.

Definice 1.2 (ŘEŠENÍ PROBLÉMU)**.** *Řešení problému splňování podmínek $P = (V, C)$ je úplné ohodnocení proměnných z množiny V, které splňuje všechny podmínky z množiny C. Množinu všech řešení problému P budeme označovat jako $Sol(P)$.* ■

Řešením problému z příkladu 1.1 je například přiřazení $X = 8$, $Y = 8$, $Z = 9$. Zapsáno užitím substituce tedy $\sigma = \{X/8, Y/8, Z/9\}$.

1.3 Ekvivalence a binarizace problémů splňování podmínek

Při modelování problémů splňování podmínek je často užitečné zvážit jiné ekvivalentní vyjádření problému, které by se lépe hodilo pro použité řešící nástroje. V této souvislosti dobře poslouží pojem ekvivalence problémů splňování podmínek. Ekvivalenci problémů předvedeme ve spojení s procesem tzv. *binarizace*.

Budeme-li z nějakých důvodů, například pro jednodušší zápis algoritmů, uvažovat v problémech splňování podmínek nejvýše binární podmínky, nedojde v důsledku tohoto omezení ke ztrátě vyjadřovacích schopností takto ochuzeného systému. Podmínky arity vyšší než 2 je totiž možné reprezentovat pomocí podmínek binárních. Příslušný převod zajišťuje zmiňovaný proces binarizace, který daný problém splňování podmínek transformuje na problém v určitém smyslu ekvivalentní, jenž bude obsahovat podmínky arity nejvýše 2.

Definice 1.3 (BINÁRNÍ PROBLÉM)**.** *Binární problém splňování podmínek je problém splňování podmínek, kde všechny podmínky jsou unární nebo binární.* ■

Zásadní otázkou při transformaci problémů splňování podmínek je způsob, jakým je pohlíženo na jejich ekvivalenci. Intuitivně požadujeme, aby ekvivalentní problémy obsahovaly ve svých řešeních stejnou informaci. Jednou z možností, které se pro definici ekvivalence nabízejí, je definovat ji pomocí rovnosti množin řešení, což v některých případech skutečně stačí. Avšak zde pouhým přejmenováním proměnných obdržíme neekvivalentní problém. Vzhledem k binarizaci, je tedy taková definice příliš omezující.

Ještě předtím než se budeme podrobněji zabývat ekvivalencí problémů, ukážeme standardní způsoby binarizace. Z toho bude poté jasnější, jakým způsobem má být ekvivalence problémů definována, aby postihovala zmíněnou intuitivní představu.

V zásadě existují dvě základní metody, jak daný problém binarizovat - *duální kódování* a *kódování se skrytou proměnnou*. Obě metody byly navrženy v práci [RDP90].

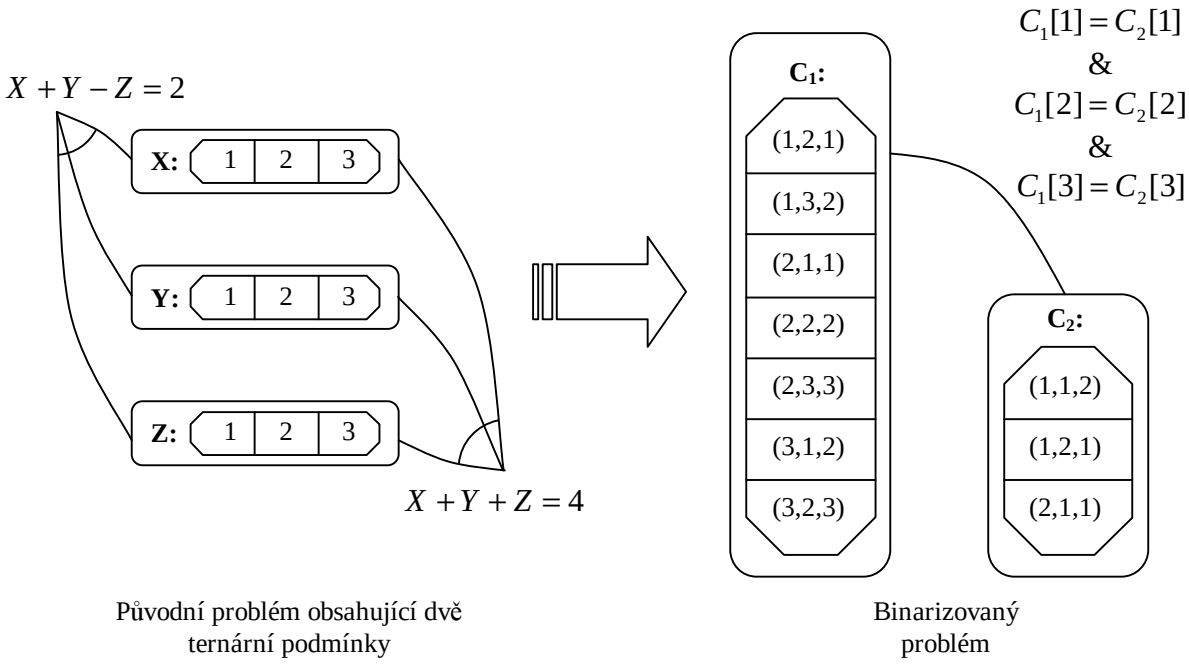
Duální kódování provádí, zjednodušeně řečeno, záměnu podmínek a proměnných. Pro každou k-ární ($k > 2$) podmínku původního problému zavádí proměnnou, jejíž doména obsahuje všechny uspořádané k-tice složené z hodnot proměnných, které daná k-ární podmínka svazuje a jež jsou vzhledem k této podmínce přípustné. Pro každou dvojici podmínek původního problému, které sdílejí proměnnou, zavádí binární podmínku mezi odpovídajícími proměnnými, která omezuje dvojice k-tic na ty, kde má daná složka stejnou hodnotu. Názorně je duální kódování ukázáno na následujícím příkladu.

Příklad 1.2. Původní problém splňování podmínek $P = (V, C)$:

$V = \{X, Y, Z\}$, kde
$D_X = \{1,2,3\}$; $D_Y = \{1,2,3\}$ a $D_Z \in \{1,2,3\}$;
$C = \{(X + Y - Z = 2),\ (X + Y + Z = 4)\}$

Pro podmínku $X + Y - Z = 2$, resp. $X + Y + Z = 4$ zavedeme proměnnou $C_1 \in \{(1,2,1);(1,3,2);(2,1,1);(2,2,2);(2,3,3);(3,1,2);(3,2,3)\}$, resp. $C_2 \in \{(1,1,2);(1,2,1);(2,1,1)\}$. Dvojice podmínek $X + Y - Z = 2$ a $X + Y + Z = 4$ sdílí proměnné X, Y i Z, musíme tedy zavést binární podmínky $C_1[1] = C_2[1]$, $C_1[2] = C_2[2]$ a $C_1[3] = C_2[3]$ nebo jejich konjunkci jakožto jedinou podmínku, symbol $[i]$ zde značí projekci i-té složky uspořádané k-tice. ■

Ilustraci k binarizaci z předchozího příkladu ukazuje obrázek 1.2.



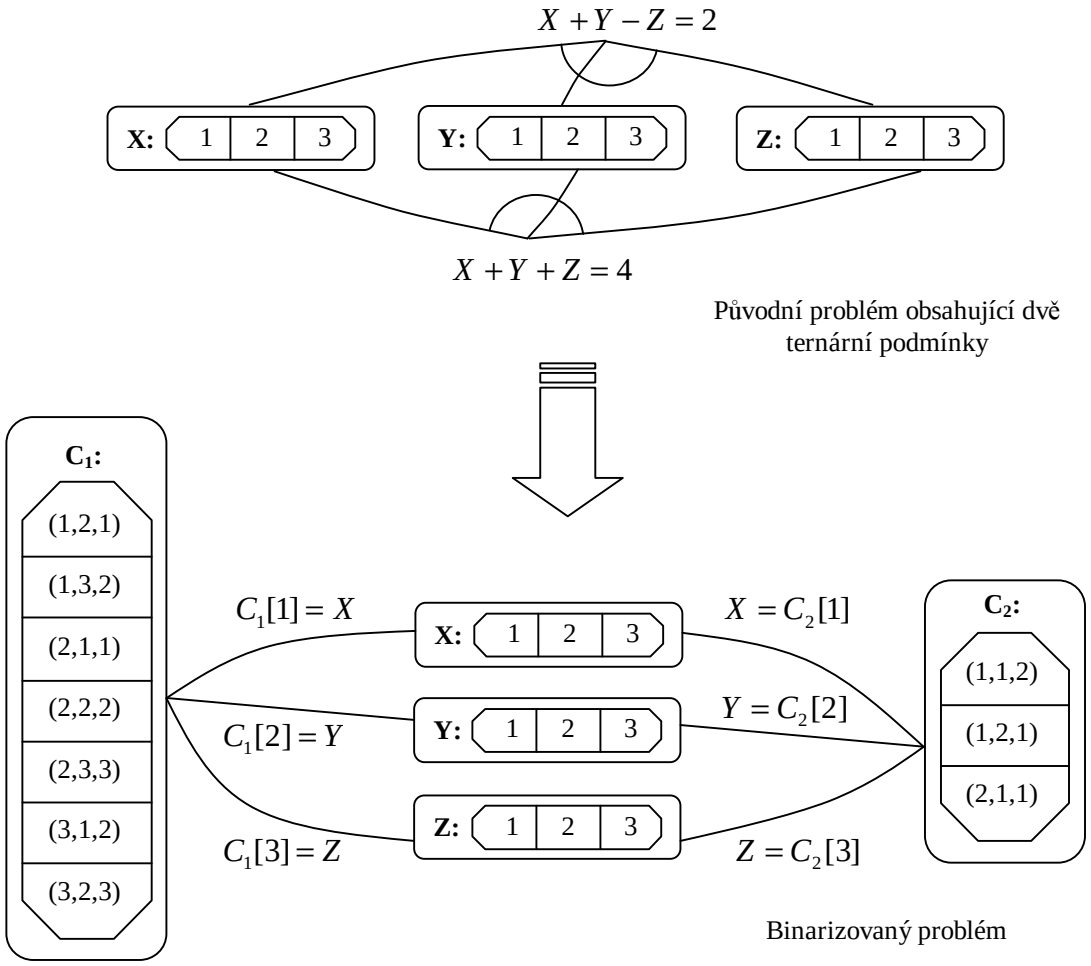
Obrázek 1.2: Proces binarizace metodou duálního kódování

Kódování se skrytou proměnnou zachovává původní proměnné. Pro každou k-ární podmínku ($k > 2$) zavádí duální proměnnou stejně jako v duálním kódování. Pro každou proměnnou v v původní k-ární podmínce zavádí binární podmínku mezi odpovídající duální proměnnou a proměnnou v, která omezuje dvojice hodnot na ty, kde má daná složka k-tice v duální proměnné stejnou hodnotu jako proměnná v. Názorně je toto kódování ukázáno na příkladě 1.3.

Příklad 1.3. Původní problém vezmeme z příkladu 1.2.

Pro podmínku $X + Y - Z = 2$, resp. $X + Y + Z = 4$ zavedeme duální proměnnou $C_1 \in \{(1,2,1);(1,3,2);(2,1,1);(2,2,2);(2,3,3);(3,1,2);(3,2,3)\}$, resp. $C_2 \in \{(1,1,2);(1,2,1);(2,1,1)\}$ stejně, jako je tomu u duálního kódování, proměnné X, Y a Z ale zůstávají. Poté přidáme podmínky $X = C_1[1]$, $Y = C_1[2]$, $Z = C_1[3]$, $X = C_2[1]$, $Y = C_2[2]$ a $Z = C_2[3]$, které svazují původní a přidané proměnné. ■

Ilustrace metody binarizace z příkladu 1.3 je ukázána na obrázku 1.3.



Obrázek 1.3: Proces binarizace metodou kódování se skrytou proměnnou

Poznamenejme, že kódování se skrytou proměnnou zachovává původní proměnné a není tedy nutné zpětně převádět získané řešení. Duální kódování naproti tomu poskytuje lepší vlastnosti vzhledem k prohledávacím algoritmům (z implementačního hlediska poskytuje explicitnější vyjádření podmínek). Oba typy binarizace lze podle potřeby kombinovat, což je podrobněji ukázáno například v [Bar98].

Binarizované problémy získané jako výsledek některé z uvedených metod mají až na určitý převod stejnou množinu řešení jako původní problém. U obou způsobů binarizace představuje dvojice původní problém a binarizovaný problém speciální případ ekvivalence. V tomto případě je u obou konstrukcí zřejmé, které proměnné a hodnoty si vzájemně odpovídají a jak získat z řešení jednoho problému řešení druhého (např. úplnému ohodnocení $X = 1$, $Y = 2$, $Z = 1$ původního problému z příkladu 1.2 odpovídá ohodnocení $C_1 = (1,2,1)$, $C_2 = (1,2,1)$ problému binarizovaného metodou duálního kódování, přičemž převod je následující

$X \leftrightarrow (C_1[1] = C_2[1])$, $Y \leftrightarrow (C_1[2] = C_2[2])$ a $Z \leftrightarrow (C_1[3] = C_2[3])$), obecně však situace tak jednoznačná není.

V obecné ekvivalenci problémů splňování podmínek půjde především o vystižení pojmu vzájemné korespondence proměnných a jejich hodnot. Právě tímto způsobem je navržena i obecná ekvivalence problémů publikovaná v práci [RDP90].

Vzájemná korespondence proměnných a hodnot se opírá o zobecněné uspořádané n-tice prvků (dále jen zobecněné n-tice prvků) popsané následující definicí.

Definice 1.4 (ZOBECNĚNÁ USPOŘÁDANÁ N-TICE)**.** *Pro danou množinu S definujeme množinu $t(S)$ následujícím způsobem:*

- $e \in t(S)$ pro každé $e \in S$,
- $(e_1, e_2, \ldots, e_n) \in t(S)$ pro všechna $n \geq 1$ a $e_i \in t(S)$. ■

Jinými slovy, množina $t(S)$ obsahuje všechny uspořádané n-tice prvků množiny S, dále všechny uspořádané n-tice uspořádaných n-tic prvků množiny S atd. (např. pro $S = \{1,2\}$ bude $t(S)$ obsahovat mimo jiné prvky 2, $(1,2)$, $(1,(1,2)), \ldots$).

Pro zjednodušení budeme na chvíli pracovat pouze s problémy splňování podmínek, kde všechny proměnné mají stejnou doménu D, takové problémy budeme označovat (V, C, D). Zobecnění pro problémy splňování podmínek s různými doménami je pouze technickou záležitostí (například můžeme D položit jako sjednocení domén jednotlivých proměnných).

Obecná ekvivalence je vybudována pomocí pojmu tzv. *převoditelnosti problémů*. Problém splňování podmínek P je převoditelný na problém Q, jestliže je možné získat množinu všech řešení problému P pomocí určitých syntaktických operací z množiny všech řešení problému Q. To se provádí přesně definovaným způsobem, který v následujícím textu stručně popíšeme. Nelze-li tímto způsobem množinu řešení problému P zrekonstruovat z množiny řešení problému Q, pak problém P není převoditelný na problém Q.

Uvažujme nyní dva problémy splňování podmínek $P = (V_P, D_P, C_P)$ a $Q = (V_Q, D_Q, C_Q)$. Vlastní nalezení rekonstrukce množiny řešení problému P z množiny řešení problému Q probíhá v několika úrovních.

Základní úlohou je nalézt ke každému řešení problému P odpovídající řešení problému Q, neboli řešení problému Q, ze kterého lze definovaným způsobem (což bude dále ještě upřesněno) dané řešení problému P zrekonstruovat. Tento krok můžeme formulovat jako hledání funkce $f_1 : Sol(P) \rightarrow Sol(Q)$, která splňuje uvedený požadavek.

Převod konkrétních řešení se provádí po jednotlivých proměnných. Hodnotu libovolné proměnné v řešení problému P musí být možné získat z korespondující zobecněné k-tice proměnných problému Q (způsob bude opět upřesněn v dalším kroku). Potřebujeme tedy určit dvojice proměnná problému P a korespondující zobecněná k-tice proměnných problému Q. Formálně tedy hledáme funkci $f_2 : V_P \rightarrow t(V_Q)$ splňující uvedený požadavek. Důležité je, že tato funkce je společná pro všechna převáděná řešení, nelze tedy uvažovat různou korespondenci proměnných v různých řešeních.

Konečně hodnota proměnné v v řešení σ problému P musí být určitelná pouze za pomoci dovelených prostředků z korespondujících proměnných problému Q, tedy ze

zobecněné k-tice proměnných $f_2(v)$ v odpovídajícím řešení problému Q, tedy v řešení $f_1(\sigma)$, ostatně takto byly funkce f_1 a f_2 konstruovány.

K rekonstrukci hodnoty proměnné v z hodnot zobecněné k-tice proměnných $f_2(v)$ je dovoleno používat pouze funkce sestavené z projekce, funkce t_n vytvářející uspořádané n-tice a operace substituce známých z teorie částečně rekurzivních funkcí, čímž je zajištěno, že převod probíhá pouze na syntaktické úrovni.

Projekce je funkce typicky označovaná jako π_i^n, pro kterou platí, že π_i^n vrací pro vstupní uspořádanou n-tici její i-tý prvek. Funkce t_n jednoduše vytváří ze svých n parametrů uspořádanou n-tici. Operace substituce nebo též skládání je definována nad funkcemi, přesnou definici zde uvádět nebudeme, pro účely ekvivalence problémů splňování podmínek postačí vědět, že užitím operace substituce lze z π_i^n a t_n sestavovat složené funkce tak, jak to ukazuje následující příklad. Mějme zobecněnou k-tici $\xi = (1,(2,3),(4,5))$, předpokládejme, že požadavkem je ξ transformovat na jinou zobecněnou k-tici, řekněme $\zeta = ((3,4),1)$. To lze provést například pomocí funkce g dané předpisem $g(\xi) = t_2(t_2(\pi_2^2(\pi_2^3(\xi)),\pi_1^2(\pi_3^3(\xi))),\pi_1^3(\xi)) = \zeta$, jež je sestavena pomocí substituce pouze z funkcí π_i^n a t_n. Naproti tomu dvojici $(6,7)$ uvedeným způsobem ze zobecněné k-tice ξ získat nelze.

Vrátíme-li se k původnímu úkolu, hledáme funkci $f_3 : (V_P \times t(D_Q)) \to D_P$, která pro danou proměnnou v problému P a hodnoty korespondujících proměnných problému Q určí hodnotu proměnné v, přičemž ke konstrukci f_3 smějí být použity pouze výše uvedené syntaktické prostředky. Opět platí, že funkce f_3 musí být společná pro všechna převáděná řešení a proměnné. Ještě je třeba upozornit na fakt, že f_3 zúžená na jednu konkrétní proměnnou není definována pro celé $t(D_Q)$, ale jen pro tu část $t(D_Q)$, která odpovídá struktuře zobecněné k-tice proměnných $f_2(v)$.

Shrneme-li celý proces převoditelnosti problémů, otázka, která nás zajímá je, zda existuje trojice funkcí f_1, f_2 a f_3, které splňují uvedené požadavky. Hledání těchto funkcí bylo popisováno postupně, nicméně ještě zdůrazněme, že ve skutečnosti jsou hledány najednou, neboť jsou vzájemně závislé.

Právě prezentovaný postup převádí jeden problém na druhý pouze pomocí syntaktických prostředků. Není tedy možné takto převádět například problémy, jejichž domény sestávají z úplně jiných objektů, ačkoli lze oba problémy interpretovat stejným způsobem (např. provedeme-li u nějakého problému s doménami přirozených čísel záměnu čísel za slova a podmínky příslušně upravíme, získáme problém, na nějž původní problém převést nelze).

Převoditelnost problémů ještě objasníme na následujícím příkladě.

Příklad 1.4. Mějme dva problémy splňování podmínek $P = (V_P, C_P)$ a $Q = (V_Q, C_Q)$:

$P: V_P = \{X, Y, Z\}$, kde
$D_X = \{1,2,3\}$; $D_Y = \{1,2,3\}$ a
$D_Z = \{1,2,3\}$;
$C_P = \{(X + Y = 4),\ (Y + Z = 4)\}$

$Q: V_Q = \{W_1, W_2\}$, kde
$D_{W_1} = \{(1,3); (2,2); (3,1)\}$ a
$D_{W_2} \in \{(1,3); (2,2); (3,1)\}$;
$C_Q = \{(W_1[2] = W_2[1])\}$

Oba problémy mají dvě řešení a lze snadno nahlédnout jejich souvislost.

$Sol(P) = \{\sigma_1 = \{X/1, Y/3, Z/1\}; \sigma_2 = \{X/2, Y/2, Z/2\}\}$;
$Sol(Q) = \{\theta_1 = \{W_1/(1,3), W_2/(3,1)\}; \theta_2 = \{W_1/(2,2), W_2/(2,2)\}\}$.

Problém P je převoditelný na problém Q, jelikož existují funkce f_1, f_2 a f_3 takové, že:

$f_1 : Sol(P) \to Sol(Q)$, přičemž
$f_1(\sigma_1) = \theta_1$ a $f_1(\sigma_2) = \theta_2$,
$f_2 : \{X, Y, Z\} \to \{(W_1, W_2)\}$, přičemž
$f_2(X) = f_2(Y) = f_2(Z) = (W_1, W_2)$ a
$f_3 : (\{X, Y, Z\} \times \{((1,3),(3,1)),((2,2),(2,2))\}) \to \{1,2,3\}$, přičemž
$f_3(X, ((1,3),(3,1))) = 1$, $f_3(Y, ((1,3),(3,1))) = 3$, $f_3(Z, ((1,3),(3,1))) = 1$,
$f_3(X, ((2,2),(2,2))) = 2$, $f_3(Y, ((2,2),(2,2))) = 2$ a
$f_3(Z, ((2,2),(2,2))) = 2$.

Přesně to je požadavek na převoditelnost. ■

Fakt, že jeden problém splňování podmínek lze převést na jiný, vlastně znamená, že druhý problém (z hlediska převodu cílový) v sobě zahrnuje přinejmenším tolik informace, co problém první. Z tohoto důvodu se pro definici ekvivalence problémů ve shodě s intuitivní představou nabízí idea vzájemné převoditelnosti. Obvykle se ekvivalence zavedená tímto způsobem nazývá rozšířená ekvivalence, zatímco pojem ekvivalence je ponechán pro problémy se stejnou množinou řešení.

Definice 1.6 (ROZŠÍŘENÁ EKVIVALENCE PROBLÉMŮ)**.** *Mějme dva problémy splňování podmínek P a Q. Říkáme, že problémy P a Q jsou rozšířeně ekvivalentní; zapisujeme $P \equiv_e Q$, jestliže problém P je převoditelný na problém Q a zároveň problém Q je převoditelný na problém P.* ■

Příkladem dvojice rozšířeně ekvivalentních problémů mohou být problémy z příkladu 1.4. Následující věta ospravedlňuje pojmenování právě definovaného pojmu.

Věta 1.1 (ROZŠÍŘENÁ EKVIVALENCE PROBLÉMŮ)**.** *Relace rozšířené ekvivalence problémů je symetrická, reflexivní a tranzitivní. Rozšířená ekvivalence problémů je tedy relací ekvivalence.* ■

Podrobný důkaz věty, jakož i další podrobnosti je možné nalézt v [RDP90]. Závěrem k rozšířené ekvivalenci problémů poznamenejme, že se skutečně jedná o rozšíření pojmu ekvivalence problémů definované pomocí rovnosti množin řešení. Binarizované problémy získané oběma uvedenými metodami jsou rozšířeně ekvivalentní s původním problémem.

1.4 Globální podmínky

Významné postavení v programování s omezujícími podmínkami zaujímají vzhledem k praktickým aplikacím tzv. *globální podmínky*. Modelování určitých specifických vztahů mezi proměnnými je pomocí jednoduchých podmínek s nízkou a pevně danou aritou (binárních) často neefektivní a komplikované. Typickým příkladem takového vztahu mezi proměnnými je již zmiňovaná podmínka, která říká, že hodnoty proměnných, jež svazuje, musejí být různé (*alldifferent*). K namodelování podmínky *alldifferent* užitím binárních podmínek pro n proměnných $X_1, X_2, \ldots, X_n$ by bylo zapotřebí $n(n-1)/2$ podmínek $X_i \neq X_j$ pro $i, j = 1,2,\ldots,n$; $i \neq j$, což je s ohledem na relativní jednoduchost vztahu příliš mnoho.

Avšak hlavní neefektivita této reprezentace podmínky *alldifferent* a důvod, proč je v podobných případech volen jiný přístup, je, že se prakticky úplně vytratilo její explicitní vyjádření, neboť byla rozdrobena do mnoha jednoduchých podmínek, a s tím i možnost využití této explicitní informace k návrhu speciálních algoritmů řešících tuto podmínku, které by díky své specializaci předčily obecné řešící algoritmy.

K modelování tohoto a podobných vztahů se proto z uvedených důvodů používají globální podmínky. Globální podmínka je speciální druh podmínky, která modeluje určitý podproblém v rámci celého problému, tj. nahrazuje sadu několika jednoduchých podmínek, přičemž pro nalezení řešení tohoto podproblému poskytuje speciální algoritmus, který pohlíží na globální podmínku jako na celek a plně využívá její sémantiky. Tento algoritmus je posléze využíván obecným řešícím algoritmem hledajícím řešení celého problému. Globální podmínky tak přinášejí značné zefektivnění obecných řešících algoritmů. Je pochopitelné, že globální podmínky, resp. jejich speciální řešící algoritmy jsou hledány hlavně pro vztahy, které se často vyskytují jako podproblémy v modelech reálných problémů. Velké množství globálních podmínek bylo navrženo například pro rozvrhovací problémy.

Praktický význam globálních podmínek můžeme také demonstrovat faktem, že v existujících komerčních softwarových produktech založených na programování s omezujícími podmínkami, jako je například *CHIP* či *ILOG*, tvoří právě globální podmínky podstatnou část zdrojového kódu.

Jako příklad globální podmínky uveďme již zmiňovanou $alldifferent(X_1, X_2, \ldots, X_n)$, která určuje, že hodnoty proměnných $X_1, X_2, \ldots, X_n$ musejí být různé. Další často užívanou globální podmínkou je $atleast(k, y, \{X_1, X_2, \ldots, X_m\})$, která říká, že alespoň k proměnných z množiny $\{X_1, X_2, \ldots, X_m\}$ musí nabývat hodnoty y.

Na uvedených příkladech je dobře vidět, že daný typ globální podmínky dostává sadu několika parametrů: množinu proměnných, které svazuje a další dodatečné parametry. Tento fakt má význam pro konkrétní implementace, stačí totiž napsat obecnou parametrizovanou verzi globální podmínky, neboli parametrizovanou verzi jejího řešícího algoritmu. A poté lze již snadno různou volbou příslušných parametrů vytvářet různé konkrétní instance globální podmínky daného typu.

Pro detailní informace týkající se globálních podmínek odkazujeme na práci [Vil01].

1.5 Řešení problémů s podmínkami

Pod pojmem řešení problémů s podmínkami obecně rozumíme algoritmy, pomocí nichž jsou hledána řešení zadaného problému splňování podmínek. Uvědomme si, že nalezení řešení problému splňování podmínek je ve své nejobecnější podobě NP-úplný problém, což lze jednoduše ukázat tak, že nějaký známý NP-úplný problém namodelujeme jako problém splňování podmínek. Vyřešení problému splňování podmínek tedy není nikterak snadný úkol.

Jádro algoritmů pro řešení omezujících podmínek tvoří algoritmy založené na prohledávání prostoru všech možných ohodnocení daného problému. U těžkých problémů skutečně není známý způsob řešení, který by postupoval určitým, řekněme konstruktivním, způsobem a který by se obešel bez prohledávání prostoru všech možných ohodnocení. Vezměme například známý NP-úplný problém barvení grafu (obarvit vrcholy grafu tak, aby sousední vrcholy byly obarveny různými barvami) namodelovaný jako problém splňování podmínek. Ačkoli se zde podaří obarvit velkou část grafu, tj. vytvořit částečné ohodnocení, které splňuje podmínky v něm zahrnuté, není znám žádný konstruktivní způsob, jak dále pokračovat, čehož příčnou je zejména skutečnost, že není zaručena možnost toto částečné obarvení rozšířit na celý graf, a tedy nezbývá nic jiného než prohledávat všechna možná obarvení.

Základní rozdělení řešících algoritmů založených na prohledávání bychom mohli učinit podle způsobu, jakým prochází prostor všech možných ohodnocení.

První skupinu prohledávacích algoritmů tvoří tzv. *systematické* prohledávání, typickými reprezentanty těchto algoritmů jsou *backtracking* a *backjumping*. Tyto algoritmy postupně prochází celý prostor všech ohodnocení, přičemž žádné ohodnocení řešící problém není vynecháno a žádné ohodnocení řešící problém není uvažováno vícekrát. Většina systematických algoritmů je realizována jako postupné procházení alternativ, průběh algoritmu je pak možné interpretovat jako procházení stromu představujícího prostor všech ohodnocení. Výhodou těchto algoritmů je možnost prokázání neexistence řešení či nalezení všech řešení a jejich dobré vlastnosti pro teoretický výzkum. U velmi rozsáhlých reálných problémů, tedy u problémů s velmi velkým prostorem všech ohodnocení, mohou ale systematické algoritmy přes veškerá urychlení selhávat.

Velmi podrobný přehled základních systematických algoritmů včetně analýzy jejich chování na těžkých kombinatorických problémech podává práce [Bak95].

Druhou skupinu prohledávacích algoritmů představuje tzv. *nesystematické* prohledávání. Sem patří zejména algoritmy *lokálního* prohledávání (např. *metoda minimalizace konfliktů*) a nesystematické algoritmy založené na nesystematickém procházení stromu všech ohodnocení (např. *iterative sampling*). Označení těchto algoritmů jako nesystematické je odvozeno od

faktu, že prostor všech možných ohodnocení je procházen nesystematicky, kdy některé jeho části obsahující řešení mohou být vynechány.

Algoritmy lokálního prohledávání bývají často založeny na metodě největšího stoupání, konkrétně jde o zlepšování dosavadního ohodnocení ve směru gradientu určité objektivní funkce. Nesystematické prohledávání se osvědčuje hlavně na rozsáhlých problémech, kde systematické algoritmy selhávají kvůli příliš velkému prostoru ohodnocení a kde uživateli postačí jedno či několik řešení a nepotřebuje dokazovat existenci řešení či hledat všechna řešení.

Některé algoritmy lokálního prohledávaní jsou opět uvedeny v [Bak95]. Konkrétně na nesystematické algoritmy založené na myšlence backtrackingu se zaměřuje práce [Har95].

Oba druhy prohledávacích algoritmů více či méně spoléhají na tzv. *konzistenční techniky*. Nejpoužívanějšími konzistenčními technikami jsou hranová konzistence - algoritmy označované jako *AC (Arc Consistency)* a konzistence po cestě - algoritmy označované jako *PC (Path Consistency)*.

Konzistenční technika nebo též procedura je speciální algoritmus, který odstraňuje z aktuálních domén proměnných ty hodnoty nebo celé k-tice hodnot, které se za určitého daného částečného ohodnocení nemohou stát součástí řešícího úplného ohodnocení, jež by bylo rozšířením tohoto částečného ohodnocení. Konzistenční algoritmus tímto způsobem omezuje prohledávaný prostor. Odstraňování nekonzistentních hodnot z domén proměnných se nazývá *filtrace domén (Domain Filtering)*. Konzistenční algoritmus obvykle pracuje tak, že kdykoli dojde k odstranění nějaké hodnoty z domény určité proměnné, nastartuje tato změna pokus o odstranění dalších hodnot z domén proměnných, které nějak souvisí s proměnnou, kde došlo ke změně domény (touto souvislostí může být například podmínka svazující obě proměnné). Uvedený postup šíření změny při zmenšování domén proměnných je označován jako *propagace (Propagation)*.

Důležitou vlastností konzistenčních algoritmů je, že narozdíl od prohledávacích algoritmů pracují v polynomiálním čase vzhledem k velikosti problému (součtu velikosti domén proměnných). Konzistenční algoritmy jsou vzhledem k tomuto faktu velmi silnou stránkou programování s omezujícími podmínkami.

V současnosti existuje poměrně velké množství konzistenčních algoritmů, liší se především silou konzistence, kterou poskytují, tedy tím, kolik hodnot dokáží odstranit, a svými nároky na prostor a čas. Silnější konzistenční algoritmus odstraní více nekonzistentních hodnot a více zmenší prohledávaný prostor, ovšem na druhou stranu za to zaplatí větším množstvím spotřebovaného času či paměti.

Přesný popis mnoha konzistenčních algoritmů lze nalézt v publikaci [Tsa93] nebo v elektronické podobě v [Bar98]. Některé konzistenční algoritmy budou uvedeny v kapitole 2.

Nakonec uveďme, že kromě prohledávacích algoritmů lze k řešení problémů splňování podmínek použít i algoritmy z jiných oblastí, jako je třeba operační výzkum, typickým reprezentantem takového algoritmu je *simplexová metoda*.

Kapitola 2

Vybrané konzistenční algoritmy

Velmi silnou stránkou programování s omezujícími podmínkami jsou již zmiňované konzistenční algoritmy někdy též označované termínem konzistenční procedury. Z kapitoly 1 už víme, že konzistenční procedura je speciální algoritmus, který vyřazuje hodnoty nebo celé k-tice hodnot z domén proměnných, které jistě nemohou figurovat ve výsledném řešení problému. O takto vyřazených hodnotách hovoříme jako o *nekonzistentních* vzhledem k dané konzistenční proceduře. Je třeba zvlášť zdůraznit, že konzistenční algoritmy pracují vždy v polynomiálním čase vzhledem k velikosti problému, to je podstatný rozdíl oproti samotným řešícím algoritmům, jejichž časová složitost může dosáhnout až exponenciální velikosti, jelikož formalizmem omezujících podmínek lze modelovat i NP-úplné problémy.

Hlavní pole působnosti nacházejí konzistenční procedury v obecných řešících algoritmech. Ve skutečných implementacích systémů založených na omezujících podmínkách je téměř vždy do obecného řešícího algoritmu integrována nějaká konzistenční procedura za účelem odstraňování nekonzistentních hodnot v průběhu řešení. Včasným odstraněním nekonzistentních hodnot algoritmus zužuje prostor všech možných ohodnocení, který je nutné v následných krocích ještě prohledat. Řešící algoritmus tak ušetří značné množství času, který by jinak strávil neúspěšnými pokusy přiřadit proměnným nekonzistentní hodnoty.

V souvislosti s vyřazováním hodnot z domén proměnných je vhodné zavést pojem tzv. *aktuální domény* proměnné. Aktuální doména proměnné bude od této chvíle představovat jakousi pracovní doménu proměnné, nad kterou budou operovat konzistenční algoritmy. Konzistenční algoritmy budou modifikovat vždy aktuální domény proměnných, zatímco původní domény budou ponechány pro jiné účely. Pro aktuální doménu proměnné v budeme užívat symbolu D_v. Původní doménu proměnné v, která byla až dosud označována jako D_v a která vystupovala v popisu problému, budeme odteď označovat symbolem D_v^0. Někdy bude též vhodné pracovat s množinou hodnot z původní domény proměnné v, jež se v daném okamžiku nenacházejí v příslušné aktuální doméně, v programovém zápisu množinu chybějících hodnot obdržíme jako rozdíl $D_v^0 - D_v$, v reálné implementaci je ale vhodné chybějící hodnoty udržovat ve zvláštní struktuře.

Asi nejpoužívanějším druhem konzistence je pro svou jednoduchost a relativně značnou účinnost tzv. *hranová konzistence (Arc Consistency)*. V této kapitole popíšeme základní algoritmy pro hranovou konzistenci. Na tyto algoritmy se později budeme odvolávat v souvislosti s hranovou konzistencí v dynamických problémech splňování podmínek, neboť několik významných algoritmů z této oblasti staví právě na konzistenčních algoritmech uvedených v této kapitole.

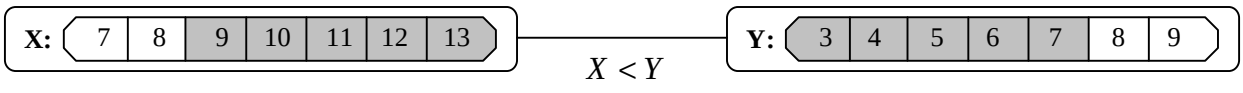
2.1 Hranová konzistence

Původní hranová konzistence je definována pro binární podmínky, na problém je přitom pohlíženo jako na graf, kde hrany znázorňují podmínky. Později ukážeme, jak hranovou konzistenci rozšířit též pro podmínky vyšší arity než 2. V souladu s literaturou (například [Tsa93]) zavedeme pojem hranové konzistence v následující definici.

Definice 1.1 (HRANOVÁ KONZISTENCE)**.** *Nechť jsou dány proměnné* u *a* v *s aktuálními doménami* D_u*, resp.* D_v *svázané binární podmínkou* c*. Hrana* (u,v) *je hranově konzistentní vzhledem k podmínce* c*, právě když platí, že pro každou hodnotu* $d_u \in D_u$ *existuje hodnota* $d_v \in D_v$ *taková, že ohodnocení* $u = d_u$ *a* $v = d_v$ *splňuje podmínku* c*. Řekneme, že binární podmínka* c *svazující proměnné* u *a* v *je hranově konzistentní, jestliže jsou hrany* (u,v) *a* (v,u) *hranově konzistentní vzhledem k podmínce* c*. Řekneme, že problém* $P = (V,C)$ *je hranově konzistentní, jestliže jsou všechny podmínky z množiny* C *hranově konzistentní.* ■

Jestliže pro danou hodnotu $d_u \in D_u$ existuje $d_v \in D_v$ taková, že ohodnocení $u = d_u$ a $v = d_v$ splňuje danou podmínku, říkáme také, že hodnota d_u má podporu d_v vzhledem k dané podmínce. Je zřejmé, že hodnoty, jež porušují definici hranové konzistence vůči nějaké podmínce, tedy takové hodnoty, jež nemají v proměnné, s níž jsou spojeny určitou podmínkou, odpovídající podporu, nikdy nemohou být součástí řešení. Proto platí, že vyřadíme-li takové hodnoty z aktuálních domén proměnných, nedojde tím ke ztrátě řešení. Automaticky se nyní nabízí otázka, jak učinit problém hranově konzistentní, tedy, jak odstranit nekonzistentní hodnoty a tím omezit prostor ohodnocení, který je třeba prohledat za účelem nalezení řešení. Samozřejmě jde o to, spočítat vzhledem k inkluzi maximální aktuální domény proměnných, pro které je problém hranově konzistentní. Jinak by triviálně stačilo aktuální domény jednoduše vyprázdnit.

Učinit hranově konzistentní jednu určitou podmínku je snadné, stačí postupovat přesně podle definice a z aktuálních domén odstraňovat nekonzistentní hodnoty. Následující obrázek ukazuje hranově konzistentní stav podmínky $X < Y$. Šedou barvou zvýrazněné prvky byly vyřazeny z aktuální domény proměnné.



Obrázek 3.1: Hranová konzistence binární podmínky

Pro výpočet hranové konzistence podmínky zavedeme speciální funkci FILTER, která určí hodnoty, jež je třeba vyřadit z aktuálních domén proměnných svázaných zadanou podmínkou. Pro binární podmínku c svazující proměnné u a v vrátí volání FILTER(c, D_u, D_v) hodnoty, které musí být odstraněny ze zadané aktuální domény D_u, resp. D_v proměnné u,

resp. v, aby byla splněna podmínka hranové konzistence nad podmínkou c. Návratovou hodnotou funkce je dvojice $(D_u^{remove}, D_v^{remove})$ množin hodnot k odstranění.

Jednoduše je možné hranovou konzistenci podmínky a tedy i funkci FILTER rozšířit na podmínky vyšší arity než 2. Aby byla n-ární podmínka c svazující proměnné $v_1, v_2, \ldots, v_n$ hranově konzistentní, musí být pro každou proměnnou v_i a každou hodnotu $d_{vi} \in D_{vi}$ splněno, že v aktuálních doménách ostatních proměnných $D_{v1}, D_{v2}, \ldots, D_{v(i-1)}, D_{v(i+1)}, \ldots, D_{vn}$ existují hodnoty $d_{v1}, d_{v2}, \ldots, d_{v(i-1)}, d_{v(i+1)}, \ldots, d_{vn}$ takové, že podmínka c je pro ohodnocení $v_1 = d_{v1}$, $v_2 = d_{v2}$, …, $v_n = d_{vn}$ splněna. V právě popsané situaci rovněž říkáme, že hodnota $d_{vi} \in D_{vi}$ má podpory v aktuálních doménách proměnných $v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_n$ vzhledem k podmínce c. Pro n-ární podmínku c vrátí volání FILTER$(c, D_{v1}, D_{v2}, \ldots, D_{vn})$ n-tici $(D_{v1}^{remove}, D_{v2}^{remove}, \ldots, D_{vn}^{remove})$ množin hodnot, které je potřeba odebrat ze zadaných aktuálních domén $D_{v1}, D_{v2}, \ldots, D_{vn}$ proměnných po řadě $v_1, v_2, \ldots, v_n$ za účelem uvedení podmínky do konzistentního stavu.

Kromě hodnot k vyřazení je v různých algoritmech také často potřeba vědět, zda má určitá vybraná hodnota podporu v ostatních proměnných, s nimiž je svázána určitou podmínkou. K tomu zavedeme další speciální funkci HAS-SUPPORT. Pro binární podmínku c svazující proměnné u a v vrátí volání funkce HAS-SUPPORT(c, u, d_u, v, D_v) booleovskou hodnotu **true**, jestliže hodnota d_u proměnné u má podporu v zadané doméně D_v proměnné v vzhledem k podmínce c (existuje $d_v \in D_v$ takové, že c je splněna pro $u = d_u$ a $v = d_v$), v opačném případě vrátí volání funkce hodnotu **false**. Podobně jako u funkce FILTER, i zde lze zavést zobecnění pro podmínky arity vyšší než 2. Je-li dána n-ární podmínka c svazující proměnné $v_1, v_2, \ldots, v_n$, pak funkci HAS-SUPPORT budeme pro podmínku c definovat následovně. Volání funkce HAS-SUPPORT$(c, v_i, d_{vi}, v_1, D_{v1}, v_2, D_{v2}, \ldots, v_{(i-1)}, D_{v(i-1)}, v_{(i+1)}, D_{v(i+1)}, \ldots, v_n, D_{vn})$ vrátí hodnotu **true**, jestliže hodnota d_{vi} proměnné v_i má podporu v zadaných doménách $D_{v1}, D_{v2}, \ldots, D_{v(i-1)}, D_{v(i+1)}, \ldots, D_{vn}$ proměnných po řadě $v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_n$ vzhledem k podmínce c, jinak vrátí volání funkce hodnotu **false**.

Zavedení funkcí FILTER a HAS-SUPPORT je užitečné zejména ve spojení s různými reprezentacemi podmínek. Obě funkce mohou být implementovány specializovaně, a tudíž efektivněji, pro každý druh či reprezentaci podmínky, se kterými se v problému pracuje.

Nejjednodušší avšak nejméně efektivní reprezentací n-ární podmínky je výčet všech n-tic tzv. *kompatibilních* hodnot, pro něž je podmínka splněna. Funkce FILTER a HAS-SUPPORT při této reprezentaci pracují přesně podle uvedených definic hranové konzistence.

Zajímavější a mnohem efektivnější reprezentaci umožňují například aritmetické podmínky definované matematickou formulí, tj. různé rovnosti a nerovnosti kombinované s běžnými matematickými funkcemi (např. $aX + bY^2 < cZ$, kde a, b a c jsou konstanty). Reprezentace takové aritmetické podmínky je obvykle dána definující formulí plus specializovanými instancemi funkcí FILTER a HAS-SUPPORT, přičemž funkce FILTER a HAS-SUPPORT jsou implementovány maximálně efektivně užitím intervalových výpočtů a monotonie. Pro další bližší informace týkající se reprezentace podmínek odkazujeme na publikaci [Tsa93] či elektronickou [Bar98].

Samostatnou kapitolou jsou v souvislosti konzistenčními technikami globální podmínky zmíněné v předchozí kapitole. Když jsme hovořili o speciálním algoritmu poskytovaném v

rámci reprezentace globální podmínky, měli jsme na mysli právě algoritmus realizující funkci FILTER popřípadě HAS-SUPPORT. Aniž bychom zacházeli do detailů, neboť globální podmínky nejsou v centru zájmu této práce, pro ilustraci šíře používaných technik uveďme, že funkce FILTER bývá například u globální podmínky *alldifferent* realizována pomocí algoritmů na hledání párování v bipartitním grafu.

Výpočet hranové konzistence celého problému nelze provést jednorázově, jako je tomu u jednotlivých podmínek. Uvést všechny podmínky problému jednou do konzistentního stavu nestačí, neboť odebrání hodnoty z aktuální domény určité proměnné mohlo způsobit, že již konzistentní podmínka se v důsledku této změny stala opět nekonzistentní. Z tohoto důvodu je u problému potřeba jednotlivé podmínky uvádět do hranově konzistentního stavu neboli revidovat opakovaně, a to tak dlouho, dokud dochází ke změnám aktuálních domén.

V následujícím textu ukážeme několik významných algoritmů pro řešení hranové konzistence, v další kapitole na jejich základě vybudujeme algoritmy pro dynamické problémy. Pro všechny algoritmy bude kvůli zjednodušení programových zápisů platit konvence, že každá podmnožina proměnných problému je svázána nejvýše jednou podmínkou, čili nepracujeme s násobnými podmínkami (graf problému je graf a nikoli multigraf). Toto zjednodušující opatření ve skutečnosti nepřináší žádné omezení, neboť místo násobných podmínek lze uvažovat jejich konjunkci v podobě jediné podmínky a navíc je možné algoritmy poměrně jednoduše rozšířit i pro skutečně násobné podmínky.

2.2 Konzistenční algoritmus AC-3

Velmi oblíbeným algoritmem řešícím hranovou konzistenci je pro svou jednoduchost AC-3 navržený v článku [Mac77]. Algoritmus popíšeme s využitím výše definované funkce FILTER. Jádrem algoritmu je opakované počítání hranové konzistence jednotlivých podmínek pomocí funkce FILTER až do okamžiku, kdy se obsah aktuálních domén proměnných ustálí. Přesněji, postupně jsou operací FILTER odstraňovány nekonzistentní hodnoty z aktuálních domén proměnných, přičemž kdykoli dojde ke zmenšení aktuální domény nějaké proměnné, jsou k revizi, tj. k další aplikaci operace FILTER, naplánovány všechny další podmínky, jež tuto proměnnou svazují. Obecně podobné procesy v souvislosti s konzistenčními algoritmy označujeme jako propagaci změn, jak již ostatně bylo zmíněno v předchozí kapitole.

Programový zápis konzistenčního algoritmu pro hranovou konzistenci AC-3 ukazuje algoritmus 2.1.

Ohledně programového zápisu platí následující konvence: jména procedur a funkcí jsou psána kapitálkami, klíčová slova jsou psána tučně, struktura programu je vyznačena pouze užitím odsazení, symbol "." slouží k odkazu na jednotlivé části složených datových struktur. Dále musíme dát na vědomí, že se jedná pouze o symbolický zápis programu, nikoli o plnou implementaci v určitém programovacím jazyce pro reálný počítač. Občas se tedy může, samozřejmě v souladu s ustálenými konvencemi, objevit obrat, pro nějž by při skutečné implementaci bylo zapotřebí udržovat další dodatečné informace.

Algoritmus 2.1. AC-3

PROPAGATE-AC-3 (P , C_{REVISE})
1 $Q_{REVISE} \leftarrow C_{REVISE}$
2 **while** $Q_{REVISE} \neq \emptyset$ **do**
3 $c \in Q_{REVISE}$ libovolná podmínka
4 $Q_{REVISE} \leftarrow Q_{REVISE} - \{c\}$
5 $\{v_1, v_2, \ldots, v_n\} \leftarrow V_c$
6 $(D_{v1}^{remove}, D_{v2}^{remove}, \ldots, D_{vn}^{remove}) \leftarrow$
7 $\leftarrow$ FILTER(c , $P.D_{v1}$, $P.D_{v2}$, … , $P.D_{vn}$)
8 **for each** $i \in \{1,2,\ldots n\}$ **do**
9 **if** $D_{vi}^{remove} \neq \emptyset$ **then**
10 $P.D_{vi} \leftarrow P.D_{vi} - D_{vi}^{remove}$
11 $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{c' \in P.C \mid v_i \in V_{c'} \,\&\, c' \neq c\}$
12 **return** P

Algoritmus AC-3 je reprezentován funkcí PROPAGATE-AC-3, která v parametrech obdržuje problém P, u nějž má být spočtena hranová konzistence, a množinu C_{REVISE} podmínek, které je nutno zrevidovat. Chceme-li učinit zadaný problém $P = (V, C)$ hranově konzistentní, spustíme algoritmus voláním PROPAGATE-AC-3(P , $P.C$), k revizi jsou tímto voláním dány všechny podmínky problému.

Podmínky aktuálně naplánované k revizi pomocí funkce FILTER se nacházejí ve frontě Q_{REVISE}. V průběhu algoritmu platí, že podmínka se nachází ve frontě Q_{REVISE} vždy, když existuje podezření, že není konzistentní. V každém kroku algoritmus vybere jednu podmínku z fronty Q_{REVISE} a na ni spustí funkci FILTER. Jestliže přitom dojde ke změně aktuální domény některé z proměnných, které svazovala revidovaná podmínka, jsou do fronty Q_{REVISE} naplánovány další podmínky, jichž se tato změna dotkla.

Časovou a prostorovou složitost budeme u algoritmů v souladu s konvencemi uvádět vždy jen pro binární problémy. Příslušnou analýzu lze rozšířit i pro podmínky vyšší arity, v takovém případě je ale navíc potřeba uvažovat případné rozdíly mezi aritami podmínek přítomných v problému.

Prostorová složitost v nejhorším případě algoritmu AC-3 je přesně úměrná počtu prvků fronty Q_{REVISE}, tedy $O(|C|)$. Časová složitost významně závisí na konkrétní implementaci funkce FILTER. Uvažujme proto nejméně příznivou implementaci, tedy implementaci, kdy jsou podmínky reprezentovány výčtem kompatibilních n-tic hodnot a odstraňování hodnot z aktuálních domén postupuje přesně podle definice hranové konzistence. Aplikace operace FILTER na binární podmínku svazující proměnné u a v spotřebuje za těchto okolností $O(|D_u||D_v|)$ času. Revize každé podmínky svazující proměnné u a v může proběhnout až $|D_u| + |D_v|$ krát, když uvažujeme, že každé volání funkce FILTER odstranilo pouze jedinou hodnotu.

Celkový čas algoritmu AC-3 je tedy v nejhorším případě $O(\sum_{(u,v)\in C}(|D_u|+|D_v|)(|D_u\|D_v|))$, což pro identické domény dává $O(|C\|D|^3)$. Přehled složitosti algoritmu AC-3 ještě shrnuje následující tabulka.

AC-3	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Propagace	$O(\|C\|)$	$O(\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u\|+\|D_v\|)(\|D_u\|\|D_v\|))$	$O(\|C\|D\|^3)$

Tabulka 2.1: Shrnutí časové a prostorové složitosti algoritmu AC-3

V rámci přípravy na algoritmy týkající se dynamických problémů zavedeme ještě funkci EXTEND, která bude představovat jakousi reverzi funkce FILTER a bude naopak konzistentně rozšiřovat aktuální domény proměnných. Opět platí, že funkce EXTEND může být implementována specializovaně pro různé druhy, resp. reprezentace podmínek.

Máme-li binární podmínku c svazující proměnné u a v, potom volání EXTEND(c, D_u, D_v) vrátí dvojici množin hodnot (D_u^{add}, D_v^{add}), jimiž je možné konzistentně rozšířit aktuální domény proměnných u a v vzhledem k zadaným aktuálním doménám D_u a D_v. Přesněji pro množiny D_u^{add} a D_v^{add} platí, že $d_u \in D_u^{add}$, jestliže d_u má podporu v zadané aktuální doméně D_v proměnné v, resp. $d_v \in D_v^{add}$, jestliže d_v má podporu v zadané aktuální doméně D_u proměnné u.

Pro podmínku c arity vyšší než 2 svazující proměnné $v_1, v_2, \ldots, v_n$ definujeme funkci EXTEND podobným způsobem. Volání EXTEND$(c, D_{v1}, D_{v2}, \ldots, D_{vn})$ vrátí uspořádanou n-tici množin hodnot $(D_{v1}^{add}, D_{v2}^{add}, \ldots, D_{vn}^{add})$, o něž je možné konzistentně rozšířit aktuální domény proměnných po řadě $v_1, v_2, \ldots, v_n$ vzhledem k zadaným aktuálním doménám $D_{v1}, D_{v2}, \ldots, D_{vn}$. Pro n-tici množin hodnot $(D_{v1}^{add}, D_{v2}^{add}, \ldots, D_{vn}^{add})$ tedy musí platit, že pro každé $i \in \{1,2,\ldots,n\}$ je $d_{vi} \in D_{vi}^{add}$, jestliže pro hodnotu d_{vi} proměnné v_i existují podpory v zadaných aktuálních doménách $D_{v1}, D_{v2}, \ldots, D_{v(i-1)}, D_{v(i-1)}, \ldots, D_{vn}$ proměnných po řadě $v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_n$.

Při bližším zkoumání algoritmu AC-3 zjistíme, že značné množství práce provádí algoritmus při opětovných revizích téže podmínky pomocí operace FILTER opakovaně, aniž by využil výsledků z předchozích testů (samozřejmě předpokládáme, že funkce FILTER si neudržuje žádný vnitřní stav, kterého by mohla využít k eliminaci opakujících se testů). Tento nedostatek se snaží řešit následující algoritmus.

2.3 Hranová konzistence založená na podporách: Algoritmus AC-4

Algoritmus nazvaný AC-4, který byl poprvé uveřejněn v článku [MH86], lépe využívá znalostí ze závislostí mezi hodnotami v doménách proměnných, čímž se pokouší eliminovat zbytečně opakované testy podmínek pro stejná ohodnocení svazovaných proměnných. Za tímto účelem používá algoritmus AC-4 speciální datové struktury - počítadla podpor a seznamy podporovaných hodnot.

Stručně se dá algoritmus popsat zhruba následovně. Vždy, když dojde odstranění nějaké hodnoty z aktuální domény proměnné, je všem hodnotám, pro než tato hodnota představovala podporu, sníženo počítadlo podpor o jedna, klesne-li přitom hodnota tohoto počítadla na 0, je i hodnota, jíž počítadlo patří, naplánována k odstranění. Tímto způsobem jsou testy zacíleny přímo na proměnné a hodnoty, jichž se provedená změna bezprostředně dotkla.

Algoritmus AC-4 formulovaný pomocí zavedeného programového zápisu ukazuje algoritmus 2.2. Vzhledem k použitým datovým strukturám je algoritmus AC-4 realizovatelný pouze pro binární podmínky.

Pro uchovávání zmiňovaných informací o podporách je v programu vyhrazena datová struktura *data*, která obsahuje položky *counter* a *S*. Položka *counter* je pole indexované dvojicí proměnné a její hodnoty (u, d_u) a další proměnnou v, která s u sousedí prostřednictvím podmínky c. Buňka $counter[(u, d_u), v]$ vždy obsahuje počet podpor hodnoty d_u z aktuální domény proměnné u v aktuální doméně proměnné v. Jak v průběhu algoritmu ubývají hodnoty z aktuálních domén proměnných, jsou tato počítadla aktualizována. Stane-li se, že některá z buněk pole *counter* náležející nějaké dvojici proměnné a její hodnoty, řekněme (u, d_u), (kterákoli z buněk $counter[(u, d_u), \_]$) nabude hodnoty 0, je i hodnota d_u odstraněna z aktuální domény proměnné u a dvojice (u, d_u) je naplánována ke kontrole.

Aby byl tento proces realizovatelný, potřebuje algoritmus ještě u každé hodnoty znát, pro které další hodnoty je tato hodnota podporou, neboť všem podporovaným hodnotám je třeba snížit počítadla podpor při případném odebrání této podporující hodnoty. K tomuto účelu algoritmus používá další datovou strukturu - seznamy podporovaných hodnot, v algoritmu reprezentovaných položkou *S* struktury *data*. Položka *S* představuje pole indexované proměnnou u a její hodnotou d_u, přičemž buňka $S[u, d_u]$ obsahuje všechny dvojice proměnných a jejich hodnot, pro něž je hodnota d_u v doméně proměnné u podporou. Zde je třeba poznamenat, že seznamy podporovaných hodnot nejsou budovány pro aktuální domény, nýbrž pro domény původní, což může u některých problémů znamenat neúnosné paměťové nároky.

Algoritmus 2.2. AC-4

INITIALIZE-AC-4 (P , *data*, C_{REVISE})

1 $Q_{REVISE} \leftarrow \emptyset$
2 **for each** $c \in C_{REVISE}$ **do**
3 $\{u, v\} \leftarrow V_c$
4 **for each** $d_u \in P.D_u^0$ **do**
5 $data.counter[(u, d_u), v] \leftarrow 0$
6 $data.S[u, d_u] \leftarrow data.S[u, d_u] - \{(v, d_v) \mid d_v \in P.D_v^0\}$
7 **for each** $d_v \in P.D_v^0$ **do**
8 $data.counter[(v, d_v), u] \leftarrow 0$
9 $data.S[v, d_v] \leftarrow data.S[v, d_v] - \{(u, d_u) \mid d_u \in P.D_u^0\}$
10 **for each** $d_u \in P.D_u^0$ **do**
11 **for each** $d_v \in P.D_v^0$ **do**
12 **if** (d_u, d_v) splňuje podmínku c **then**
13 $data.counter[(u, d_u), v] \leftarrow$
14 $\leftarrow data.counter[(u, d_u), v] + 1$
15 $data.S[v, d_v] \leftarrow data.S[v, d_v] \cup \{(u, d_u)\}$
16 $data.counter[(v, d_v), u] \leftarrow$
17 $\leftarrow data.counter[(v, d_v), u] + 1$
18 $data.S[u, d_u] \leftarrow data.S[u, d_u] \cup \{(v, d_v)\}$
19 (P , *data*, Q_{REVISE}^{uv}) $\leftarrow$ INITIALIZE-ARC-AC-4(P , *data*, u , v)
20 (P , *data*, Q_{REVISE}^{vu}) $\leftarrow$ INITIALIZE-ARC-AC-4(P , *data*, v , u)
21 $Q_{REVISE} \leftarrow Q_{REVISE} \cup Q_{REVISE}^{uv} \cup Q_{REVISE}^{vu}$
22 **return** (P , *data*, Q_{REVISE})

INITIALIZE-ARC-AC-4 (P , *data*, u , v)

1 $Q_{REVISE} \leftarrow \emptyset$
2 **for each** $d_u \in P.D_u$ **do**
3 **if** $data.counter[(u, d_u), v] = 0$ **then**
4 $P.D_u \leftarrow P.D_u - \{d_u\}$
5 $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{(u, d_u)\}$
6 **return** (P , *data*, Q_{REVISE})

PROPAGATE-AC-4 (P , *data*, Q_{REVISE})
1 **while** $Q_{REVISE} \neq \emptyset$ **do**
2 $(v, d_v) \in Q_{REVISE}$ libovolná dvojice proměnné a hodnoty
3 $Q_{REVISE} \leftarrow Q_{REVISE} - \{(v, d_v)\}$
4 **for each** $(u, d_u) \in data.S[v, d_v]$ **do**
5 $data.counter[(u, d_u), v] \leftarrow data.counter[(u, d_u), v] - 1$
6 **if** $data.counter[(u, d_u), v] = 0$ **and** $d_u \in P.D_u$ **then**
7 $P.D_u \leftarrow P.D_u - \{d_u\}$
8 $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{(u, d_u)\}$
9 **return** (P , *data*)

Program algoritmu AC-4 sestává z funkcí INITIALIZE-AC-4, INITIALIZE-ARC-AC-4 a PROPAGATE-AC-4. Chceme-li uvést problém P do hranově konzistentního stavu, algoritmus spustíme voláním $result \leftarrow$ INITIALIZE-AC-4(P ,*data*, $P.C$) následovaným voláním PROPAGATE-AC-4(*result*. P , *result.data*, *result*. Q_{REVISE}) , kde struktura *data* je vynulovaná.

Funkce INITIALIZE-AC-4 má na starosti počáteční inicializaci datových struktur algoritmu AC-4 a počáteční odebrání nekonzistentních hodnot. Parametry funkce jsou řešený problém P, částečně vyplněná, tj. vyplněná jen pro některé podmínky, nebo nevyplněná (vynulovaná) datová struktura *data* a množina podmínek C_{REVISE}, jež uživatel požaduje algoritmem AC-4 zrevidovat. Návratovou hodnotou funkce INITIALIZE-AC-4 je trojice obsahující modifikovaný problém P po počátečním odebrání nekonzistentních hodnot, příslušně vyplněná datová struktura *data* a fronta Q_{REVISE} dvojic proměnná a její hodnota naplánovaných k revizi. Vyplněnou datovou strukturou *data* zde máme na mysli zkonstruované seznamy podporovaných hodnot a inicializovaná počítadla podpor. Funkce INITIALIZE-AC-4 používá ke své práci ještě pomocnou funkci INITIALIZE-ARC-AC-4, která provádí počáteční odstranění nekonzistentních hodnot.

Struktury obsažené v návratové hodnotě funkce INITIALIZE-AC-4 jsou pak očekávány jako parametry funkcí PROPAGATE-AC-4, která v podstatě realizuje hlavní smyčku algoritmu, pracující podle výše popsaného schématu. Běh funkce PROPAGATE-AC-4 se opírá o frontu Q_{REVISE}, ta obsahuje dvojice proměnná a její hodnota, která byla v minulosti odebrána a pro niž ještě nebyla aktualizována počítadla podporovaných hodnot. Výsledkem činnosti funkce PROPAGATE-AC-4 je hranově konzistentní problém P a odpovídajícím způsobem modifikovaná datová struktura *data*.

Oproti algoritmu AC-3 značně narostly u AC-4 paměťové nároky, přičemž ty má na svědomí hlavně položka *S* datové struktury *data*, která v nejhorším případě zabere prostor o velikosti $O(\sum_{(u,v)\in C} |D_u^0||D_v^0|)$, což je pro identické domény $O(|C||D|^2)$. Prostor potřebný k uchování počítadel podpor *counter* ve struktuře *data* je $O(\sum_{(u,v)\in C} |D_u^0| + |D_v^0|)$, pro identické domény $O(|C||D|)$, což vzhledem k paměťovým nárokům seznamů podporovaných hodnot *S* nepředstavuje další navýšení. K tomuto výsledku lze dojít tak, že spočteme, kolik záznamů o počtu podpor připadá na jednu podmínku.

Nakonec je nutné ještě zvážit prostor potřebný pro frontu Q_{REVISE}, který činí $O(\sum_{v \in V} |D_v|)$, i tato hodnota lze bez dalšího navýšení započítat k prostoru pro strukturu *S* (pokud ovšem nepracujeme s proměnnými, které nejsou svázány žádnou podmínkou, což ale za standardních okolností vždy platí).

K určení časové složitosti postačí pozorování, že na každou dvojici kompatibilních hodnot v aktuálních doménách proměnných svázaných podmínkou připadá konstantní množství práce. Tak obdržíme pro časovou složitost v nejhorším případě hodnotu $O(\sum_{(u,v) \in C} |D_u^0 \| D_v^0|)$, což je $O(|C| \|D|^2)$ pro identické domény. Následující tabulka shrnuje uvedené výsledky.

AC-4	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Propagace	$O(\sum_{(u,v) \in C} \|D_u^0 \| D_v^0\|)$	$O(\|C\| \|D\|^2)$	$O(\sum_{(u,v) \in C} \|D_u^0 \| D_v^0\|)$	$O(\|C\| \|D\|^2)$

Tabulka 2.2: Shrnutí časové a prostorové složitosti algoritmu AC-4

Časová složitost v nejhorším případě je u algoritmu AC-4 dokonce optimální. Avšak všimněme si, že ve výrazech udávající prostorovou a časovou složitost vystupují velikosti originálních domén nikoli aktuálních. V praktických implementacích se ukazuje, že v průměrném případě algoritmus tráví příliš mnoho času aktualizací datových struktur pro hodnoty, které se momentálně vůbec nenacházejí v aktuálních doménách proměnných. Vzhledem k příliš velké prostorové složitosti a nepříznivému chování v průměrném případě bývá algoritmus AC-4 často nahrazován, a to zejména u velmi rozsáhlých problémů, jednodušším AC-3.

Zmiňované neuspokojivé vlastnosti algoritmu AC-4 se pokouší vylepšit následující algoritmus.

2.4 Vylepšená hranová konzistence se seznamy podpor: Algoritmus AC-6

Algoritmus AC-6 popsaný v článku [BC94] částečně vychází z algoritmu AC-4, opět se opírá o ideu podpor, resp. podporovaných hodnot. Zásadní rozdíl je však v tom, že algoritmus AC-6 místo uchovávání seznamu všech možných podpor ukládá a aktualizuje vždy pouze podpory nejmenší vzhledem k nějakému zvolenému lineárnímu uspořádání hodnot v aktuálních doménách proměnných. Tím je dosaženo jednak o řád menší paměťové složitosti a jednak lepšího chování v průměrném případě oproti algoritmu AC-4.

Lineární uspořádání hodnot v aktuálních doménách proměnných bude definováno pomocí operací FIRST a NEXT. Operace FIRST bude pro zadanou doménu vracet její první prvek, případně hodnotu *NIL*, pokud doména nebude obsahovat žádné prvky. Operace NEXT

bude pro zadanou doménu D_v a hodnotu $d_v \in D_v$ vracet hodnotu bezprostředně následující po hodnotě d_v v D_v nebo NIL, pokud d_v byla hodnotou poslední v D_v. Poznamenejme, že toto uspořádání nemusí mít nic společného s běžnými uspořádáními definovanými nad prvky domén proměnných.

Stručně by se dal průběh algoritmu charakterizovat tak, že kdykoli dojde k odstranění nějaké hodnoty z aktuální domény proměnné, je třeba pro všechny další hodnoty, pro něž byla tato odebraná hodnota nejmenší podporou v dané proměnné vzhledem k určité podmínce, najít podporu jinou, vzhledem k definovanému lineárnímu uspořádání bezprostředně následující po odebrané hodnotě. Nepodaří-li se žádnou další podporu najít, je i tato hodnota naplánována k odstranění.

Algoritmus opět používá datovou strukturu *data*, ovšem ta nyní obsahuje pouze položku *S*, která opět reprezentuje pole. Význam pole S je zde podobný jako u algoritmu AC-4, buňky tohoto pole však již neobsahují seznam všech podporovaných hodnot, ale v každé buňce $S[v, d_v]$ je udržován pouze seznam všech dvojic proměnná a její hodnota (u, d_u), pro které je hodnota d_v nejmenší podporou v aktuální doméně proměnné v.

Programový zápis algoritmu AC-6 je reprezentován algoritmem 2.3. Kostra algoritmu se příliš neliší od AC-4. Hlavní odlišností samotného algoritmu je zpracování pole *S* v datové struktuře *data*.

Algoritmus 2.3. AC-6

```
INITIALIZE-AC-6 ( P , data, C_REVISE )
1     Q_REVISE ← Ø
2     for each c ∈ C_REVISE do
3           {u, v} ← V_c
4           ( P , data, Q^uv_REVISE ) ← INITIALIZE-ARC-AC-6( P , data, u , v )
5           ( P , data, Q^vu_REVISE ) ← INITIALIZE-ARC-AC-6( P , data, v , u )
6           Q_REVISE ← Q_REVISE ∪ Q^uv_REVISE ∪ Q^vu_REVISE
7     return ( P , data, Q_REVISE )

INITIALIZE-ARC-AC-6 ( P , data, u , v )
1     Q_REVISE ← Ø
2     for each d_v ∈ P.D_v do
3           data.S[v, d_v] ← data.S[v, d_v] − {(u, d_u) | d_u ∈ P.D^0_u}
```

```
4       for each d_u ∈ P.D_u do
5           d_v^next ← NEXT-SUPPORT-AC-6( P , u , d_u , v , NIL )
6           if d_v^next ≠ NIL then
7               data.S[v,d_v^next] ← data.S[v,d_v^next] ∪ {(u,d_u)}
8           else
9               P.D_u ← P.D_u − {d_u}
10              Q_REVISE ← Q_REVISE ∪ {(u,d_u)}
11      return ( P , data, Q_REVISE )

PROPAGATE-AC-6 ( P , data, Q_REVISE )
1       while Q_REVISE ≠ Ø do
2           (v,d_v) ∈ Q_REVISE  libovolná dvojice proměnné a hodnoty
3           Q_REVISE ← Q_REVISE − {(v,d_v)}
4           for each (u,d_u) ∈ data.S[v,d_v] do
5               data.S[v,d_v] ← data.S[v,d_v] − {(u,d_u)}
6               d_v^next ← NEXT-SUPPORT-AC-6( P , u , d_u , v , d_v )
7               if d_v^next ≠ NIL then
8                   data.S[v,d_v^next] ← data.S[v,d_v^next] ∪ {(u,d_u)}
9               else if d_u ∈ P.D_u then
10                  P.D_u ← P.D_u − {d_u}
11                  Q_REVISE ← Q_REVISE ∪ {(u,d_u)}
12      return ( P , data)

NEXT-SUPPORT-AC-6 ( P , u , d_u , v , d_v )
1       nechť u a v jsou svázány podmínkou c
2       if d_v ≠ NIL then
3           d_v ← NEXT( d_v , P.D_v )
4       else
5           d_v ← FIRST( P.D_v )
6       while d_v ≠ NIL and c není splněna pro (d_u,d_v) do
7           d_v ← NEXT( d_v , P.D_v )
8       return d_v
```

Program algoritmu AC-6 se skládá z funkcí INITIALIZE-AC-6, INITIALIZE-ARC-AC-6, PROPAGATE-AC-6 a NEXT-SUPPORT-AC-6. Význam prvních třech jmenovaných je prakticky totožný s odpovídajícími funkcemi v algoritmu AC-4. Novinkou oproti algoritmu AC-4 je funkce NEXT-SUPPORT-AC-6, která zadané hodnotě hledá další podporu (vzhledem k danému uspořádání bezprostředně následující) v proměnné sousedící skrze nějakou podmínku.

Je-li naším cílem učinit zadaný problém P hranově konzistentním, algoritmus spustíme voláním $result \leftarrow$ INITIALIZE-AC-6(P ,*data*, $P.C$) a PROPAGATE-AC-6(*result*. P , *result.data*, *result*. Q_{REVISE}) , kde struktura *data* je vynulovaná.

Počáteční vyplnění seznamů nejmenších podpor, jakož i počáteční odebrání nekonzistentních hodnot mají na starosti funkce INITIALIZE-AC-6 a INITIALIZE-ARC-AC-6. Parametry i návratové hodnoty těchto funkcí se shodují s odpovídajícími funkcemi u algoritmu AC-4.

Hlavní smyčka algoritmu je realizována funkcí PROPAGATE-AC-6, která opět pracuje s frontou Q_{REVISE} obsahující dvojice proměnná a její hodnota, jež byla v minulosti odebrána. Jestliže v minulosti odebraná hodnota byla nejmenší podporou jiným hodnotám, je nutné pro tyto hodnoty najít jinou nejmenší podporu, v daném uspořádání následující, nepodaří-li se to, jsou i ony odebrány a naplánovány do fronty Q_{REVISE} ke kontrole.

Seznamy nejmenších podpor v poli *S* zaberou v nejhorším případě prostor o velikosti $O(\sum_{(u,v)\in C}|D_u|+|D_v|)$, neboli $O(|C||D|)$ pro identické domény. K tomuto výsledku lze dojít tak, že každý záznam o nejmenší podpoře je započítán příslušné podmínce. Fronta Q_{REVISE} vystačí s prostorem $O(\sum_{v\in V}|D_v|)$, což lze bez navýšení započítat do prostoru potřebného pro seznamy nejmenších podpor v poli *S*, stejně jako jsme to učinili u předchozího algoritmu.

Ohledně časové složitosti v nejhorším případě je potřeba nahlédnout, že každá dvojice hodnot obsažených v aktuálních doménách sousedících skrze určitou podmínku je ve funkci NEXT-SUPPORT-AC-6 otestována nejvýše jedenkrát. To dává čas $O(\sum_{(u,v)\in C}|D_u||D_v|)$, čili $O(|C||D|^2)$ pro identické domény. Ostatní části programu již časovou složitost nenavýší. Uvedené výsledky ještě přehledně shrnuje následující tabulka.

AC-6	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Propagace	$O(\sum_{(u,v)\in C}(\|D_u\|+\|D_v\|))$	$O(\|C\|\|D\|)$	$O(\sum_{(u,v)\in C}\|D_u\|\|D_v\|)$	$O(\|C\|\|D\|^2)$

Tabulka 2.3: Shrnutí časové a prostorové složitosti algoritmu AC-6

Algoritmem AC-6 uzavřeme přehled konzistenčních procedur pro hranovou konzistenci. Na tomto místě ještě zmíníme, že kromě hranové konzistence existují silnější konzistenční techniky jako je například konzistence po cestě či *k*-konzistence, resp. silná *k*-konzistence, které odstraní více nekonzistentních hodnot než hranová konzistence. Samozřejmě platí, že algoritmy realizující tyto druhy konzistence mají větší nároky na čas i prostor. Blížeji o těchto algoritmech pojednávají například publikace [Tsa93] nebo elektronická [Bar98].

Kapitola 3

Řešení dynamických problémů s podmínkami

Dosud popisovaný formalizmus omezujících podmínek a algoritmy se zabývaly pouze problémy splňování podmínek, v nichž zůstává množina proměnných a podmínek po celou dobu práce s problémem neměnná. Tento přístup, kdy množina proměnných a podmínek v problému zůstává statická, je v řadě aplikací nedostatečný, o čemž svědčí například následující situace.

Uvažme stav, kdy se uživatelem zadaný problém nepodaří vyřešit, tj. řešící algoritmus odpoví, že řešení neexistuje. Pro uživatele by za těchto okolností bylo velmi užitečné vědět, co přesně bylo příčinou tohoto neúspěchu, či ještě lépe, jak problém modifikovat, aby nějaké řešení existovalo. Důvodem neexistence řešení problému, může být z pohledu uživatele příliš svazující množina podmínek. O problémech s touto vlastností říkáme, že jsou *příliš omezené*. Jako východisko z této situace se nabízí zjednodušení problému (zmírnění omezení) pomocí odebrání některých podmínek. Změna způsobená odebráním několika podmínek bývá v porovnání s velikostí celého problému celkem nepatrná, jedná se pouze o malou lokální změnu. Z tohoto hlediska je proto neefektivní řešit modifikovaný problém znovu, úplně od začátku bez využití jednou již vypočtených informací.

Snadno si lze představit, že interakce s uživatelem během řešení nabude ještě mnohem komplikovanější podoby. Uživatel může učinit rozhodnutí o změnách v řešeném problému již na základě dosavadního průběhu řešení a nečekat až na výsledné řešení či odpověď, že řešení neexistuje. Pro uživatele by bylo nepochybně velmi výhodné, aby daný systém uměl efektivně reagovat na jeho zásahy.

Navíc kromě změn vyvolaných zásahy uživatele je též často užitečné, aby modifikace problému prováděl samotný řešící algoritmus. Vhodná oblast, kde by automatická modifikace problému v průběhu řešení mohla nalézt své uplatnění, jsou například plánovací problémy řešené pomocí omezujících podmínek. Stručně naznačme, že u plánovacího problému je úkolem transformovat počáteční stav pomocí aplikace dovolených operací či pravidel na stav cílový. Jelikož ale předem není znám počet kroků aplikací daných pravidel vedoucích k řešení, bývá plánovací problém modelován pomocí podmínek vždy pro určitý omezený počet kroků aplikace dovolených operací. Není-li u takto namodelovaného plánovacího problému nalezeno řešení, je model rozšířen pomocí přidávání proměnných a podmínek o další kroky. Popsaný přístup aplikuje například algoritmus CPlan popsaný v [BC99]. Zde se opět ukazuje jako velmi neefektivní řešit modifikovaný problém úplně od začátku bez využití dosud získaných výsledků.

Problémy splňování podmínek, u nichž dochází k častým změnám množiny proměnných a podmínek, jsou označovány jako *dynamické*. V této kapitole popíšeme některé obecné metody pro práci s dynamickými problémy.

3.1 Dynamické problémy splňování podmínek

Než přejdeme ke konkrétním otázkám týkajících se dynamických problémů, musíme pojem dynamického problému nejprve přesně vymezit. Následující definice má původ v práci [DD88].

Definice 3.1 (DYNAMICKÝ PROBLÉM SPLŇOVÁNÍ PODMÍNEK)**.** *Dynamický problém splňování podmínek (Dynamic Constraint Satisfaction Problem - DCSP) je posloupnost problémů splňování podmínek* $P_0, P_1, \ldots, P_\alpha, \ldots$, *přičemž pro každý problém v posloupnosti kromě* P_0 *je dána informace, jakým způsobem jej získat z předchozího problému aplikací operací přidání proměnné, odebrání proměnné, přidání podmínky a odebrání podmínky.* ■

Obyčejné problémy splňování podmínek tvořící posloupnost dynamického problému jsou v kontextu dynamických problémů označovány jako *statické*. Informaci udávající způsob získání problému $P_{i+1} = (V^{i+1}, C^{i+1})$ v posloupnosti z předcházejícího problému $P_i = (V^i, C^i)$ lze reprezentovat pomocí množin čtyř množin V^i_{ADD}, V^i_{DEL}, C^i_{ADD} a C^i_{DEL}, kde V^i_{ADD} obsahuje proměnné, jež mají být přidány, V^i_{DEL} obsahuje proměnné, jež mají být odebrány, C^i_{ADD} představuje množinu podmínek k přidání a nakonec C^i_{DEL} obsahuje podmínky k odebrání. Samozřejmě musí být splněno, že $V^i_{ADD} \cap V^i = \emptyset$, $V^i_{DEL} \subseteq V^i$, $C^i_{DEL} \cap C^i = \emptyset$ a $C^i_{DEL} \subseteq C^i$. U operace odebírání proměnných je třeba navíc dodržet konvenci, že odebíraná proměnná již nesmí být svázána s žádnou podmínkou. Alternativně je možné operaci odebrání proměnné definovat tak, že s danou proměnou jsou odebrány i podmínky, ve kterých byla tato proměnná zahrnuta.

Přechod ke zvolenému problému v posloupnosti od problému předcházejícího je dán následujícím vztahem. Jestliže $P_i = (V^i, C^i)$, pak $P_{i+1} = (V^{i+1}, C^{i+1}) = = ((V^i \cup V^i_{ADD}) - V^i_{DEL}, (C^i \cup C^i_{ADD}) - C^i_{DEL})$.

3.2 Udržování hranové konzistence v dynamických problémech

Otázkou, kterou se budeme zabývat v této kapitole, je konzistence v dynamických problémech, přesněji její udržování vzhledem k operacím měnících strukturu problému. Otázka udržování konzistence v dynamických problémech představuje úkol vyřešit problém konzistence pro každý statický problém v posloupnosti dynamického problému. Jinak řečeno, úkolem je spočítat vzhledem k inkluzi maximální aktuální domény jednotlivých proměnných každého problému v posloupnosti tak, aby tyto problémy byly konzistentní vůči dané konzistenční proceduře. Z pohledu modifikujících operací požadujeme, aby byl vstupní konzistentní problém transformován na modifikovaný výstupní problém, který je opět konzistentní.

Nejvíce algoritmů zbývajících se touto otázkou řešilo konzistenci hranovou. Tímto směrem se budeme ubírat i my. V současnosti již existuje řada algoritmů řešících problém dynamické hranové konzistence, nejdůležitější z nich popíšeme v této kapitole. Prvním významným algoritmem z této kategorie byl DnAC-4 navržený 1991 Christianem Bessièrem v

práci [Bes91]. O pár let později, v roce 1996, na algoritmus DnAC-4 navázal Romuald Debruyne svým algoritmem DnAC-6, který popsal v práci [Deb96]. Oba algoritmy, DnAC-4 a DnAC-6, jak ostatně napovídají jejich názvy, ideově vycházejí algoritmů AC-4, resp. AC-6 pro nedynamickou hranovou konzistenci.

Poněkud odlišný postup pro řešení dynamické hranové konzistence uplatňuje algoritmus AC|DC, který navrhli Bertrand Neveu a Pierre Berlandier v roce 1994.

Po prostudování uvedených algoritmů, vybudujeme na základě získaných poznatků nový algoritmus řešící dynamickou hranovou konzistenci.

Ještě než přejdeme k vlastním algoritmů, je třeba zvlášť zdůraznit některá fakta. Operace přidání a odebrání proměnné, která není svázaná žádnou podmínkou, nepředstavují při udržování hranové konzistence žádnou překážku. Přidání či odebrání takové proměnné ponechá původní konzistentní problém opět konzistentním. Z tohoto důvodu se nadále budeme vzhledem k udržování konzistence zabývat pouze operacemi přidání, resp. odebrání podmínky. Jak uvidíme později, vyřešení efektivního odebrání podmínky představuje mnohem složitější úkol než její přidání.

Triviálně může být konzistence v dynamických problémech vyřešena aplikací konzistenční procedury na jednotlivé statické problémy. Takový přístup je ale pro svou náročnost zcela neuspokojivý. Přirozenou cestou, jak řešit konzistenci u dynamických problémů, je počítat konzistenci modifikovaného problému na základě již spočtené konzistence u problému předcházejícího. Přidání či odebrání podmínky je pouze lokální změna, proto se pro výpočet konzistence u modifikovaného problému nabízí postup, který lokálně obnovuje či opravuje konzistenci modifikovaného problému jen v určitém okolí zasaženém přidáním nebo odebráním podmínky.

Obecně způsobí operace přidání podmínky větší omezení daného problému neboli větší zúžení aktuálních domén některých proměnných. Přidaná podmínka přímo vyloučí nekonzistentní hodnoty z domén proměnných, které svazuje. Další hodnoty jsou nepřímo vyloučeny propagací přímo provedených změn, tj. aplikací konzistenční procedury na přidanou podmínku a její bezprostřední okolí.

Odebrání podmínky naopak problém ulehčí, což představuje nutnost znovu zařadit do aktuálních domén některých proměnných hodnoty, jež byly v minulosti odstraněny. Určení těchto proměnných a hodnot představuje hlavní komplikaci v udržování konzistence u dynamických problémů, podobné přímočaré zdůvodnění jako u opačné operace zde není k dispozici. Metody řešení tohoto problému uvidíme v konkrétních algoritmech.

Ohledně všech algoritmů uvedených v této kapitole opět přijmeme kvůli zjednodušení programových zápisů konvenci, že každá podmnožina proměnných problému je svázána nejvýše jednou podmínkou, stejně jako jsme to učinili v předchozí kapitole. Opět tu platí, že toto opatření ve skutečnosti není žádným reálným omezením.

3.3 Inkrementální hranová konzistence

První krok k dynamizaci hranové konzistence představuje tzv. *inkrementální hranová konzistence*, která ošetřuje případ přidávání podmínky. Inkrementální hranová konzistence může být založena na libovolném konzistenčním algoritmu pro hranovou konzistenci. Inkrementální přístup dovoluje postupné přidávání posloupnosti podmínek, odebírání podmínek není umožněno.

Operace přidání podmínky je realizována zapojením podmínky do struktury problému a následným spuštěním propagace pro přidanou podmínku. Přidání podmínky tímto způsobem využívající konzistenční proceduru AC-3 popisuje algoritmus 3.1.

Algoritmus 3.1. PŘIDÁNÍ PODMÍNKY INKREMENTÁLNĚ

ADD-CONSTRAINT-INCREMENTALLY (P , c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 $P \leftarrow$ PROPAGATE-AC-3 (P , $\{c\}$)
3 **return** P

Platí, že byl-li vstupní problém algoritmu hranově konzistentní, pak výstupní problém s přidanou podmínkou je rovněž hranově konzistentní. Poznamenejme, že uvedený způsob inkrementálního přidávání podmínek lze jednoduše adaptovat i pro jiné konzistenční procedury, než je hranová konzistence.

Jak již bylo řečeno, odebrání podmínky inkrementální hranová konzistence nijak neřeší. Nejsou-li tedy k dispozici jiné prostředky, je nutné přistoupit k aplikaci procedury hranové konzistence na celý problém s odebranou podmínkou.

3.4 Plně dynamická hranová konzistence: Algoritmus DnAC-4

První algoritmus, který řeší odebírání podmínek v dynamických problémech, byl navržen v práci [Bes91] pod názvem DnAC-4. Algoritmus implementuje plnou dynamickou hranovou konzistenci, kdy operace přidávání a odebírání podmínek mohou být volány v libovolném pořadí.

DnAC-4 vychází z konzistenčního algoritmu AC-4, který byl prezentován v předchozí kapitole. Postup při přidávání podmínky je v DnAC-4 prakticky totožný s přidáváním podmínky u inkrementální hranové konzistence, jde tedy o zařazení nové podmínky následované spuštěním propagace pomocí AC-4 na přidanou podmínku.

Operace odebírání podmínek je vybudována s využitím datových struktur konzistenčního algoritmu AC-4, tj. s využitím počítadel podpor a seznamů podporovaných hodnot. Výpočet hranové konzistence po odebrání podmínky je v algoritmu DnAC-4 realizován postupným přidáváním hodnot do aktuálních domén proměnných, jejichž odebrání v minulosti mohlo být způsobeno přímo či nepřímo právě odebíranou podmínkou. K efektivnímu stanovení hodnot, které mají být do aktuálních domén navráceny, však s počítadly podpor a seznamy podporovaných hodnot lze vystačit jen obtížně. Proto autoři algoritmu DnAC-4 obohacují konzistenční proceduru AC-4 o udržovaní dalších pomocných informací. Konkrétně jde o informaci udávající důvod odebrání dané hodnoty z aktuální domény proměnné, přičemž zmiňovaným důvodem je z pohledu algoritmu DnAC-4 proměnná, kde odebraná hodnota poprvé ztratila všechny podpory. Pro zavedený programový zápis to znamená rozšíření datové struktury *data* algoritmu AC-4 o další položku *justifications*, která bude reprezentovat pole

indexované proměnnou a její hodnotou, kde pro jednotlivé buňky bude platit, že když algoritmus AC-4 odebral z aktuální domény proměnné v hodnotu d_v, pak buňka $justifications[v,d_v]$ obsahuje proměnnou, v jejíž aktuální doméně hodnota d_v poprvé ztratila všechny podpory.

Jednoduchou úpravou adaptujeme program konzistenční procedury AC-4 tak, aby vyhovovala požadavkům algoritmu DnAC-4. Řádky, kde dochází k odstranění hodnoty z aktuální domény proměnné obohatíme o příslušnou aktualizaci buňky pole *justifications*. Formálně to znamená nahradit všechny výskyty řádku

$$\vdots$$
$$P.D_u \leftarrow P.D_u - \{d_u\}$$
$$\vdots$$

řádky

$$\vdots$$
$$P.D_u \leftarrow P.D_u - \{d_u\}$$
$$justifications[u,d_u] \leftarrow v$$
$$\vdots$$

V programovém zápisu algoritmu AC-4 se náhrada týká řádku 4 funkce INITIALIZE-ARC-AC-4 a řádku 7 funkce PROPAGATE-AC-4.

Mimo správnou obnovu odebraných hodnot je úkolem algoritmu DnAC-4 též udržovat uvedené datové struktury v korektním stavu v případě, že jsou nějakým způsobem modifikovány v důsledku změny ve struktuře problému.

Obě operace, přidání a odebrání podmínky, algoritmu DnAC-4 jsou popsány pomocí programového zápisu v algoritmu 3.2. Stejně jako u algoritmu AC-4, je i zde nutné omezit se kvůli použití datových struktur reprezentující závislosti mezi hodnotami pouze na binární podmínky.

Algoritmus 3.2. DNAC-4

ADD-CONSTRAINT-DNAC-4 (P , *data*, c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 (P , *data*, Q_{REVISE}) $\leftarrow$ INITIALIZE-AC-4(P , *data*, $\{c\}$)
3 (P , *data*) $\leftarrow$ PROPAGATE-AC-4(P , *data*, Q_{REVISE})
4 **return** (P , *data*)

RETRACT-CONSTRAINT-DNAC-4 (P , *data*, c)
1 $\{u, v\} \leftarrow V_c$
2 (P , *data*, $Q^{uv}_{RESTORE}$) $\leftarrow$
3 $\leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-4(P , *data*, u , v)
4 (P , *data*, $Q^{vu}_{RESTORE}$) $\leftarrow$
5 $\leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-4(P , *data*, v , u)
6 $C_{REVISE} \leftarrow \emptyset$
7 $Q_{RESTORE} \leftarrow Q^{uv}_{RESTORE} \cup Q^{vu}_{RESTORE}$
8 $P.C \leftarrow P.C - \{c\}$ a zruš struktury *data.S* a *data.counter* příslušející c
9 **while** $Q_{RESTORE} \neq \emptyset$ **do**
10 $(v, d_v) \in Q_{RESTORE}$ libovolná dvojice proměnné a hodnoty
11 $Q_{RESTORE} \leftarrow Q_{RESTORE} - \{(v, d_v)\}$
12 **for each** $(u, d_u) \in data.S[v, d_v]$ **do**
13 $data.counter[(u, d_u), v] \leftarrow data.counter[(u, d_u), v] + 1$
14 **if** $data.justifications[u, d_u] = v$ **and** $d_u \in (P.D^0_u - P.D_u)$ **then**
15 $P.D_u \leftarrow P.D_u \cup \{d_u\}$
16 $data.justifications[u, d_u] \leftarrow NIL$
17 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, d_u)\}$
18 $C_{REVISE} \leftarrow C_{REVISE} \cup \{c \in P.C \mid v \in V_c\}$
19 (P , *data*, Q_{REVISE}) $\leftarrow$ INITIALIZE-AC-4(P , *data*, C_{REVISE})
20 (P , *data*) $\leftarrow$ PROPAGATE-AC-4(P , *data*, Q_{REVISE})
21 **return** (P , *data*)

INITIALIZE-ARC-RETRACTION-DNAC-4 (P , *data*, u , v)
1 $Q_{RESTORE} \leftarrow \emptyset$
2 **for each** $d_u \in (P.D^0_u - P.D_u)$ **do**
3 **if** $data.justifications[u, d_u] = v$ **then**
4 $P.D_u \leftarrow P.D_u \cup \{d_u\}$
5 $data.justifications[u, d_u] \leftarrow NIL$
6 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, d_u)\}$
7 **return** (P , *data*, $Q_{RESTORE}$)

Algoritmus DnAC-4 sestává z funkcí ADD-CONSTRAINT-DNAC-4, RETRACT-CONSTRAINT-DNAC-4 a INITIALIZE-ARC-RETRACTION-DNAC-4.

Funkce ADD-CONSTRAINT-DNAC-4 realizující přidání podmínky dostává jako parametr řešený problém P, vyplněnou rozšířenou datovou strukturu algoritmu AC-4 *data* a přidávanou podmínku c. Před přidáním podmínky je nutné doplnit datovou strukturu *data* o odpovídající informace o nové podmínce. Po zařazení podmínky do struktur problému je spuštěna

konzistenční procedura AC-4, která propaguje změny v aktuálních doménách proměnných svázaných nově přidanou podmínku.

Snadno lze ověřit, že operace transformuje původně hranově konzistentní problém na hranově konzistentní problém rozšířený o novou podmínku.

Operace odebrání podmínky je prováděna funkcí RETRACT-CONSTRAINT-DNAC-4. Funkce opět obdržuje jako parametry řešený problém P, vyplněnou rozšířenou datovou strukturu procedury AC-4 *data* a odebíranou podmínku c. Při obnově aktuálních domén proměnných se algoritmus opírá o závislosti vypočtené modifikovanou konzistenční procedurou AC-4 obsažené v parametru *data*, přesně bude tento proces popsán v následujících odstavcích.

Na řádcích 1 až 8 funkce RETRACT-CONSTRAINT-DNAC-4 probíhá inicializace odebírání podmínky, při inicializaci jsou do aktuálních domén proměnných, jež odebíraná podmínka svazuje, navráceny hodnoty, jejichž odstranění bezprostředně tato podmínka způsobila. Tato počáteční obnova aktuálních domén je realizována funkcí INITIALIZE-ARC-RETRACTION-DNAC-4 volané na řádcích 2-3 a 4-5, která v parametrech obdrží řešený problém P, k němu příslušně vyplněnou datovou strukturu *data*, a proměnné u a v, jež jsou svázané odebíranou podmínkou. Funkce navrátí do aktuální domény proměnné u ty hodnoty, za jejichž odebrání v minulosti byla odpovědná proměnná v. O které konkrétní hodnoty se jedná, je určeno pomocí příčin odebrání obsažených v poli *justifications*. Pro inicializaci odebrání podmínky c svazující proměnné u a v je funkce INITIALIZE-ARC-RETRACTION-DnAC-4 volána dvakrát, poprvé pro hranu (u,v) (obnovena je aktuální doména proměnné u), podruhé pro hranu (v,u) (obnovena je aktuální doména proměnné v).

Kdykoli dojde k navrácení hodnot do aktuálních domén některých proměnných, je nutné naplánovat revizi sousedních proměnných, protože se může stát, že některá hodnota z domény proměnné sousedící s proměnnou, jíž byly navráceny hodnoty do aktuální domény, získala díky přidané hodnotě novou podporu a může být díky tomu rovněž navrácena. Dvojice proměnné a její hodnoty, jejichž sousedy je třeba zkontrolovat na získání podpory, jsou plánovány do fronty $Q_{RESTORE}$. Přípravu fronty činí rovněž funkce INITIALIZE-ARC-RETRACTION-DNAC-4. Problém modifikovaný rozšířením aktuální domény proměnné svázané odebíranou podmínkou a příslušně upravená datová struktura *data* algoritmu AC-4, jakož i fronta $Q_{RESTORE}$ jsou obsaženy v návratové hodnotě funkce INITIALIZE-ARC-RETRACTION-DNAC-4.

Na závěr inicializace operace odebrání podmínky proběhne ve funkci RETRACT-CONSTRAINT-DnAC-4 vyřazení odebírané podmínky ze struktury problému. Poté algoritmus pokračuje do hlavní smyčky představující jádro algoritmu.

Hlavní smyčka algoritmu DnAC-4 se nachází na řádcích 9 až 18. Z fronty $Q_{RESTORE}$ je vždy odebrána libovolná dvojice (v,d_v) proměnné a její hodnoty popisující změnu, která proběhla v minulosti (byla přidána hodnota d_v do aktuální domény proměnné v). Následně je zvýšeno o 1 počítadlo podpor u těch hodnot, pro které představuje hodnota d_v podporu. Hodnoty, jimž mají být aktualizována počítadla jsou obsaženy v příslušné buňce pole *S*. Jestliže je navíc zjištěno, že hodnota z domény proměnné u, které bylo zvýšeno počítadlo podpor, byla odebrána kvůli podmínce svazující proměnné u, v a stále v aktuální doméně proměnné u chybí, je tato hodnota také navrácena. Přitom je opět naplánována revize aktuálních domén sousedních proměnných, jelikož přidané hodnoty se opět mohly stát podporami pro jiné dosud chybějící hodnoty.

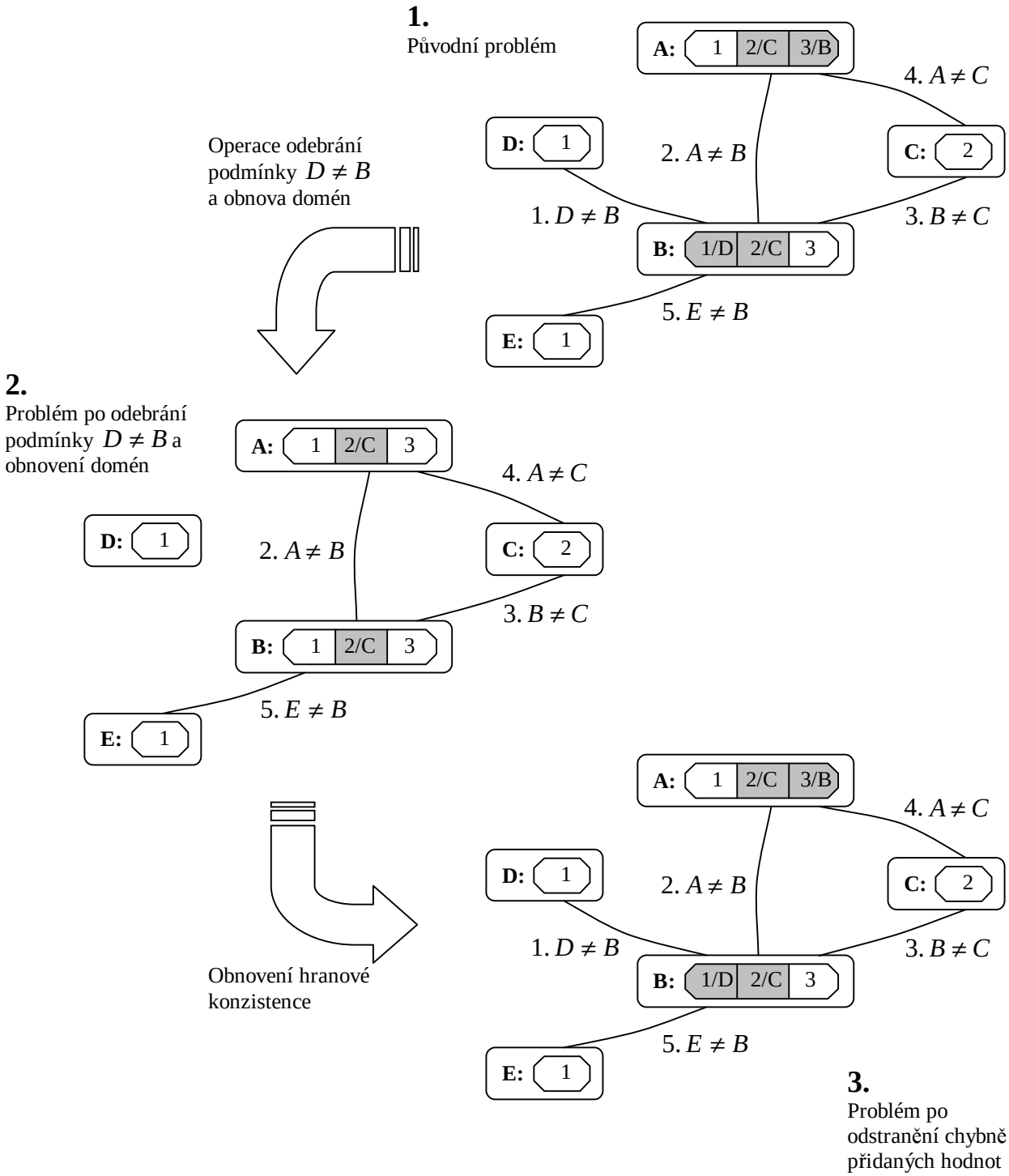
Dosud jsme nepopsali úlohu seznamu podmínek C_{REVISE}. Při navracení hodnot do aktuálních domén může dojít k tomu, že přidávaná hodnota není hranově konzistentní vůči

všem podmínkám, tj. nemá podpory ve všech sousedních proměnných. Navíc se může stát, že tyto podpory nezíská ani v hodnotách obnovených v následujících krocích. K tomu dochází v případě, že hodnota odebraná kvůli určité podmínce, resp. kvůli tomu, že ztratila všechny podpory v určité sousední proměnné, by se navíc později (kdyby zůstala v aktuální doméně přítomna) stala nekonzistentní vůči jiné podmínce. Neboli, ztratila by všechny podpory ještě u nějaké další sousední proměnné.

Hodnoty, které jsou tímto způsobem chybně navráceny, je nutné opět vyloučit. Právě k tomu slouží seznam podmínek C_{REVISE} . Tento seznam obsahuje podmínky, které mají být znovu prověřeny konzistenčním algoritmem AC-4, přitom případně dojde k odstranění chybně přidaných hodnot z aktuálních domén zainteresovaných proměnných. Tato závěrečná revize konzistenčním algoritmem probíhá na řádcích 19 a 20. Poznamenejme, že u skutečné implementace bychom spíše než celé podmínky dávali k revizi pouze obnovené hodnoty, zde byly celé podmínky voleny kvůli zvýšení přehlednosti algoritmu. Rovněž inicializační fáze závěrečné propagace probíhající na řádku 19 lze implementovat efektivněji (zde se znovu vyplňuje struktura *data* pro zasažené podmínky, což ve skutečnosti není třeba, jelikož nám jde pouze o počáteční konstrukci fronty Q_{REVISE}).

Výsledný modifikovaný problém a odpovídajícím způsobem upravená datová struktura algoritmu AC-4 jsou nakonec vráceny formou návratové hodnoty.

Průběh algoritmu DnAC-4 při odebírání podmínky ještě ukazuje obrázek 3.1. Původní problém je vytvořen postupným přidáváním podmínek (pořadí přidávání je vyznačeno pořadovým číslem u každé podmínky). Hodnoty odstraněné z aktuálních domén proměnných jsou označeny šedou barvou, u odstraněné hodnoty je vždy uvedena sousední proměnná, ve které chybí pro odstraněnou hodnotu podpora. Proces odebírání podmínky je znázorněn ve dvou fázích. V první fázi je odpojena podmínka z grafu problému a jsou obnoveny aktuální domény přímo i nepřímo zasažených proměnných. Druhá fáze pak opravuje hranovou konzistenci odstraněním některých chybně přidaných hodnot v první fázi.



Obrázek 4.1: Průběh odebírání podmínky algoritmem DnAC-4

Přesnou analýzu korektnosti algoritmu zde nebudeme provádět, pouze shrneme výsledek do následující věty. Pro podrobnou argumentaci dokazující korektnost algoritmu odkazujeme na práci [Bes91].

Věta 3.1 (KOREKTNOST ALGORITMU DNAC-4)**.** *Operace přidání a odebrání podmínky algoritmem DnAC-4 transformují vstupní hranově konzistentní problém na výstupní modifikovaný problém, který je opět hranově konzistentní.* ■

Algoritmus DnAC-4 dědí po konzistenční proceduře AC-4 všechny její výhody i nevýhody, a to jednak kvůli tomu, že konzistenční proceduru AC-4 algoritmus DnAC-4 přímo volá, a jednak kvůli tomu, že DnAC-4 pracuje podobným způsobem a přímo se opírá o datové struktury konzistenční procedury AC-4.

Složitost algoritmu DnAC-4 zachycuje následující tabulka.

DNAC-4	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Přidání podmínky	$O(\sum_{(u,v)\in C}\lvert D_u^0 \rVert D_v^0\rvert)$	$O(\lvert C\rVert D\rvert^2)$	$O(\sum_{(u,v)\in C}\lvert D_u^0 \rVert D_v^0\rvert)$	$O(\lvert C\rVert D\rvert^2)$
Odebrání podmínky	$O(\sum_{(u,v)\in C}\lvert D_u^0 \rVert D_v^0\rvert)$	$O(\lvert C\rVert D\rvert^2)$	$O(\sum_{(u,v)\in C}\lvert D_u^0 \rVert D_v^0\rvert)$	$O(\lvert C\rVert D\rvert^2)$

Tabulka 3.1: Shrnutí časové a prostorové složitosti algoritmu DnAC-4

Důkaz těchto výsledků pouze naznačíme, plné znění odůvodnění těchto výsledků lze opět nalézt v [Bes91]. Předně je třeba říci, že rozšíření datové struktury *data* o položku *justifications* nenavýšilo prostorovou složitost v nejhorším případě, neboť prostor $O(\sum_{v\in V}\lvert D_v^0\rvert)$ potřebný pro pole *justifications* lze bez navýšení započítat do prostorové složitosti procedury AC-4.

Časová i prostorová složitost operace přidání podmínky přímo odpovídá složitosti konzistenční procedury AC-4. Prostorová složitost operace odebrání podmínky je rovněž přímo dána prostorovou složitostí algoritmu AC-4.

K výsledkům ohledně časové složitosti se u operace odebrání podmínky dobereme tak, že spočítáme dvojice vzájemně se podporujících hodnot. K tomu přidáme tvrzení, že algoritmus každou z těchto dvojic zkoumá nejvýše jedenkrát. Celkový počet vzájemně se podporujících dvojic hodnot je v celém problému nejvýše $\sum_{(u,v)\in C}\lvert D_u^0 \rVert D_v^0\rvert$. Volání konzistenční procedury na závěr operace odebrání podmínky již nepředstavuje další navýšení složitosti.

Chování algoritmu DnAC-4 v nejhorším případě se řadí mezi jeho výhody, časová složitost v tomto případě nabývá optimální hodnoty. V průměrném případě ale výsledky algoritmu zdaleka nejsou optimální, zde je potřeba si uvědomit, že algoritmus DnAC-4, stejně jako AC-4, aktualizuje počítadla podpor i u hodnot, které se v daném okamžiku nenacházejí v aktuálních doménách proměnných.

Od AC-4 přebírá algoritmus DnAC-4 též jeho nepříznivou prostorovou složitost, kterou mají na svědomí seznamy podporovaných hodnot v položce *S* datové struktury *data*.

Potenciálním nedostatkem algoritmu je také jeho použitelnost pouze pro binární podmínky, potřebuje-li uživatel pracovat také s podmínkami vyšší arity, je nutné přistoupit k některé metodě binarizace, jak bylo uvedeno v kapitole 1. S tím souvisí i další problém, a to je závislost algoritmu DnAC-4 na poměrně speciálních datových strukturách, které značně znesnadňují jeho případné další adaptace a rozšiřování.

3.5 Vylepšení algoritmu DnAC-4: Algoritmus DnAC-6

Snaha vylepšit prostorovou složitost v nejhorším případě a časovou složitost v průměrném případě algoritmu DnAC-4 vyústila navržením algoritmu DnAC-6. Jádro algoritmu DnAC-6 spočívá v adaptaci postupů užívaných konzistenční procedurou AC-6, která byla popsána v předchozí kapitole, pro dynamické problémy. Algoritmus DnAC-6 byl publikován v práci [Deb96]. Opět se jedná o implementaci plné dynamické hranové konzistence, stejně jako je tomu u algoritmu DnAC-4.

Přidávání podmínky algoritmus DnAC-6 zpracovává obvyklým způsobem, tj. obdobně jako DnAC-4 či inkrementální přidávání podmínek - nová podmínka je zařazena do struktury problému a je pro ni spuštěna propagace. Podstatný rozdíl je však v operaci odebrání podmínky, ta je realizována s využitím datových struktur algoritmu AC-6 rozšířených o informace o příčinách odebrání hodnot (*justifications*). Obnovení hranové konzistence se provádí postupným přidáváním dříve odstraněných hodnot do aktuálních domén proměnných na základě informací o závislostech mezi hodnotami získaných z rozšířených datových struktur algoritmu AC-6. Opět je nutné příslušným způsobem tyto datové struktury udržovat v korektním stavu.

Základní myšlenkou algoritmu AC-6 bylo pohlížet na domény proměnných jakožto na lineárně uspořádané množiny čili seznamy. Každá hodnota z domény jakoby měla své pořadové číslo, které určovalo její pozici v doméně, resp. předchozí a následující hodnotu. V průběhu celé propagace zůstávalo pořadové číslo dané hodnoty konstantní a společné pro všechny typy domén (původní, aktuální a doménu odebraných hodnot). Algoritmus DnAC-6 přináší v tomto ohledu zobecnění. Uspořádání hodnot v doménách proměnných je dynamické a v průběhu operací přidávání a odebírání podmínek se mění. Přesněji, domény proměnných jsou implementovány jako seznamy a kdykoli má být doména rozšířena o nějakou hodnotu, je tato hodnota připojena na konec seznamu reprezentujícího doménu. Uspořádání určené tímto seznamem je poté respektováno při hledání dalších podpor dané hodnoty. Význam tohoto přístupu bude diskutován později.

Pro připojení prvku na konec seznamu reprezentujícího aktuální domény zavedeme novou operaci realizovanou funkcí APPEND. Funkce APPEND bude dostávat jako parametry hodnotu a aktuální doménu, volání APPEND(d_u, D_u) vrátí aktuální doménu D_u s připojenou hodnotou d_u na konci.

Stejně jako u algoritmu AC-4, je i zde potřeba rozšířit program výchozí konzistenční procedury AC-6. Jednak je nutné obohatit datovou strukturu *data* procedury AC-6 o pole *justifications*, jehož význam je naprosto totožný s tímtéž polem u algoritmu DnAC-4, v němž budou udržovány informace o příčinách odstranění daných hodnot.

A dále je potřeba nahradit všechny výskyty řádku

$$\vdots$$
$$P.D_u \leftarrow P.D_u - \{d_u\}$$
$$\vdots$$

řádky

$$\vdots$$
$$P.D_u \leftarrow P.D_u - \{d_u\}$$
$$justifications[u, d_u] \leftarrow v$$
$$\vdots$$

V programovém zápisu algoritmus AC-6 se tato náhrada týká řádku 9 funkce INITIALIZE-ARC-AC-6 a řádku 10 funkce PROPAGATE-AC-6. Touto úpravou bude konzistenční procedura AC-6 připravena na použití v algoritmu DnAC-6.

Programový zápis algoritmu DnAC-6 ukazuje algoritmus 3.3. DnAC-6 má na aritu podmínek stejné omezení jako algoritmus DnAC-4, pracuje tedy pouze s binárními podmínkami. Důvodem jsou opět použité datové struktury.

Algoritmus 3.3. DNAC-6

ADD-CONSTRAINT-DNAC-6 (P , *data*, c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 (P , *data*, Q_{REVISE}) $\leftarrow$ INITIALIZE-AC-6(P , *data*, $\{c\}$)
3 (P , *data*) $\leftarrow$ PROPAGATE-AC-6(P , *data*, Q_{REVISE})
4 **return** (P , *data*)

RETRACT-CONSTRAINT-DNAC-6 (P , *data*, c)
1 $\{u, v\} \leftarrow V_c$
2 (P , *data*, $Q^{uv}_{RESTORE}$) $\leftarrow$
3 $\leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-6(P , *data*, u , v)
4 (P , *data*, $Q^{vu}_{RESTORE}$) $\leftarrow$
5 $\leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-6(P , *data*, v , u)
6 $C_{REVISE} \leftarrow \emptyset$
7 $Q_{RESTORE} \leftarrow Q^{uv}_{RESTORE} \cup Q^{vu}_{RESTORE}$
8 $P.C \leftarrow P.C - \{c\}$ a zruš strukturu *data.S* příslušející c

```
9     while Q_RESTORE ≠ Ø do
10        (v, d_v) ∈ Q_RESTORE libovolná dvojice proměnné a hodnoty
11        Q_RESTORE ← Q_RESTORE − {(v, d_v)}
12        for each c ∈ C svazující proměnnou v do
13            {u, v} ← V_c
14            d_u^next ← NEXT-SUPPORT-AC-6( P , v , d_v , u , NIL )
15            if d_u^next ≠ NIL then
16                data.S[u, d_u^next] ← data.S[u, d_u^next] ∪ {(v, d_v)}
17            for each d_u ∈ (P.D_u^0 − P.D_u) do
18                if data.justifications[u, d_u] = v and
19                  c je splněna pro (d_u, d_v) then
20                    P.D_u ← APPEND( d_u , P.D_u )
21                    data.S[v, d_v] ← data.S[v, d_v] ∪ {(u, d_u)}
22                    data.justifications[u, d_u] ← NIL
23                    Q_RESTORE ← Q_RESTORE ∪ {(u, d_u)}
24        C_REVISE ← C_REVISE ∪ {c ∈ P.C | v ∈ V_c}
25    ( P , data, Q_REVISE ) ← INITIALIZE-AC-6( P , data, C_REVISE )
26    ( P , data) ← PROPAGATE-AC-6( P , data, Q_REVISE )
27    return ( P , data)
```

```
INITIALIZE-ARC-RETRACTION-DNAC-6 ( P , data, u , v )
1     Q_RESTORE ← Ø
2     for each d_u ∈ (P.D_u^0 − P.D_u) do
3         if data.justifications[u, d_u] = v then
4             P.D_u ← APPEND( d_u , P.D_u )
5             data.justification[u, d_u] ← NIL
6             Q_RESTORE ← Q_RESTORE ∪ {(u, d_u)}
7     return ( P , data, Q_RESTORE )
```

Algoritmus DnAC-6 se skládá z funkcí ADD-CONSTRAINT-DNAC-6, RETRACT-CONSTRAINT-DNAC-6 a INITIALIZE-ARC-RETRACTION-DNAC-6.

Přidání podmínky provádí funkce ADD-CONSTRAINT-DNAC-6, tato funkce obdrží v parametrech řešený problém P, modifikovanou datovou strukturu *data* algoritmu AC-6 vyplněnou informacemi o problému P a přidávanou podmínku c. Po zařazení nové podmínky c do struktury problému P a rozšíření struktury *data* tak, aby reflektovala modifikovaný problém, je spuštěna konzistenční procedura AC-6 pro podmínku c. Tímto procesem je původně hranově konzistentní problém transformován na hranově konzistentní problém obsahující navíc podmínku c.

Odebrání podmínky je prováděno funkcí RETRACT-CONSTRAINT-DNAC-6. Tato funkce dostává v parametrech řešený problém P, příslušně vyplněnou datovou strukturu *data* a odebíranou podmínku c. Celý proces obnovy domén zasažených proměnných probíhá podobně jako u algoritmu DnAC-4. Nejprve jsou u proměnných, které podmínka c svazovala, navráceny do aktuálních domén hodnoty, za jejichž odstranění přímo mohla podmínka c. To je realizováno pomocí funkce INITIALIZE-ARC-RETRACTION-DNAC-6, která dostává jako parametry řešený problém P, odpovídajícím způsobem vyplněnou datovou strukturu algoritmu AC-6 *data* a proměnné u, v, jež jsou svázané odebíranou podmínkou. Hodnoty, které je potřeba navrátit do aktuálních domén proměnných u a v, jsou zjištěny pomocí pole *justifications* , jež obsahuje příčiny odstranění. Funkce INITIALIZE-ARC-RETRACTION-DNAC-6 je volána dvakrát - zvlášť pro hrany (u,v) a (v,u), při každém volání jsou obnoveny hodnoty v aktuální doméně jedné z proměnných svázaných odebíranou podmínkou.

Dojde-li k rozšíření aktuální domény proměnné, je zároveň ve funkci INITIALIZE-ARC-RETRACTION-DNAC-6 naplánována revize sousedních proměnných, neboť obnovená hodnota se mohla stát podporou pro některou z dosud nepřítomných hodnot v sousedních proměnných. Po dokončení inicializace, která probíhá na řádcích 1 až 7 funkce RETRACT-CONSTRAINT-DNAC-6, je podmínka c odstraněna ze struktury problému P a přichází ke slovu hlavní smyčka algoritmu.

Jak již bylo řečeno, obnovená hodnota je připojena vždy na konec seznamu hodnot přítomných v aktuální doméně. Důvodem, proč se navracená hodnota připojuje právě na konec, je způsob jakým konzistenční algoritmus AC-6 postupuje v okamžiku, ztratí-li nějaká hodnota podporu. U algoritmu AC-6 se jedná vždy o ztrátu podpory nejmenší vzhledem k danému uspořádání hodnot v doméně. Nastane-li taková situace, algoritmus AC-6 hledá pro zasaženou hodnotu jinou podporu postupně mezi hodnotami bezprostředně následujícími vzhledem k danému uspořádání po odstraněné hodnotě. Připojením obnovené hodnoty na konec seznamu hodnot aktuální domény proměnné, řekněme v, je zajištěno, že na tuto obnovenou hodnotu algoritmus při následných propagacích narazí a posoudí, jestli se nejedná o další podporu jiných hodnot, které právě ztratily svoji nejmenší podporu v aktuální doméně proměnné v.

Stejně jako v algoritmu DnAC-4, je i zde využívána fronta $Q_{RESTORE}$, která obsahuje dvojice proměnných a jejich hodnot, jež byly navráceny do aktuálních domén, a je tedy nutné u nich provést revizi sousedů. Na řádcích 9 až 24 se nachází hlavní smyčka algoritmu řízená frontou $Q_{RESTORE}$. Zde je v každém kroku z fronty $Q_{RESTORE}$ odebrána libovolná dvojice (v,d_v). Následně je pro každou proměnnou u sousedící prostřednictvím podmínky c s proměnnou v uskutečněn pokus vrátit do její aktuální domény všechny hodnoty, které mohly být odstraněny kvůli podmínce c, přesněji kvůli tomu, že v aktuální doméně proměnné v chyběla hodnota d_v. Kandidáti na navrácení do aktuální domény jsou ty hodnoty z množiny $P.D_u^0 - P.D_u$, u nichž příslušná buňka pole *justifications* obsahuje hodnotu v. Skutečně navrácená hodnota d_u pak musí navíc splňovat, že podmínka c je pro ohodnocení (d_u,d_v) splněna. Po připojení hodnoty d_u na konec seznamu aktuálních hodnot v doméně proměnné u jsou naplánovány ke kontrole domény sousedních proměnných, zda i v nich nelze obnovit další hodnoty.

Zvláštní komentář si zaslouží zejména aktualizace datové struktury *data*, konkrétně struktury S na řádcích 14 až 16 a na řádku 21. Připomeňme, že struktura S představuje dvourozměrné pole, kde buňka $S[u,d_u]$ obsahuje dvojice proměnná a její hodnota (v,d_v), pro které platí, že d_u je nejmenší podporou vzhledem k danému uspořádání v aktuální doméně

proměnné u hodnotě d_v. Po navrácení hodnoty d_v do aktuální domény proměnné v jsou tedy nutné aktualizace pole *S*, které nyní popíšeme.

Obnovená hodnota d_v v aktuální doméně proměnné v obvykle má, pokud není přidána chybně, také podpory v proměnných sousedících s proměnnou v prostřednictvím podmínek a ty je třeba zahrnout do příslušných buněk pole *S*. Ve spojení s algoritmem DnAC-6 nás samozřejmě zajímají pouze podpory nejmenší vzhledem k danému uspořádání. Předpokládejme, že nějaká hodnota d_u v aktuální doméně proměnné u, která sousedí s proměnnou v, je nejmenší podporou pro obnovenou hodnotu d_v. To je ale přesně požadavek na to, aby byla dvojice (v, d_v) přidána do množiny v buňce $S[u, d_u]$. Aktualizace datové struktury *data* odpovídající této situaci se odehrává na řádcích 14 až 16 funkce RETRACT-CONSTRAINT-DNAC-6.

Aby však byly změny v poli *S* úplné, musíme ještě uvážit navrácení hodnoty d_v do aktuální domény proměnné v z opačného pohledu, tj. zda se sama obnovená hodnota d_v nestala nejmenší podporou nějakým jiným hodnotám v aktuálních doménách sousedních proměnných. Tato situace skutečně nastane v okamžiku, kdy jsou obnovovány hodnoty v aktuálních doménách proměnných sousedících s v, jejichž odstranění bylo v minulosti zapříčiněno proměnnou v tím, že v její aktuální doméně chyběla hodnota d_v. Za těchto okolností se tedy již dříve navrácená hodnota d_v do aktuální domény proměnné v stává nejmenší podporou pro nově obnovovanou hodnotu. Nechť d_u je touto navracenou hodnotou a nechť u je cílová proměnná, pak je třeba do buňky $S[v, d_v]$ přidat dvojici (u, d_u). Právě popsaná aktualizace struktury *S*, probíhá na řádku 21 funkce RETRACT-CONSTRAINT-DnAC-6.

Na tomto místě se projeví výhodnost přidávání obnovené hodnoty na konec seznamu hodnot reprezentujícího aktuální doménu. Tento přístup totiž zamezí tomu, aby se navrácená hodnota stala nejmenší podporou hodnot již přítomných v aktuálních doménách a v důsledku tak velmi složité aktualizaci pole *S* datové struktury *data*.

Množina podmínek C_{REVISE} má stejný význam jako v algoritmu DnAC-4, obsahuje podmínky, které je třeba znovu zrevidovat, neboť i algoritmus DnAC-6 může obnovit nekonzistentní hodnoty. Tato revize probíhá na řádcích 22 a 23.

Výsledný problém s odebranou podmínkou a odpovídajícím způsobem modifikovaná datová struktura *data* jsou nakonec funkcí RETRACT-CONSTRAINT-DNAC-6 vráceny formou návratové hodnoty.

Poznamenejme, že opětovná revize podmínek zasažených přidáním hodnot do aktuálních domén lze provádět efektivnějším způsobem. Místo množiny podmínek k revizi C_{REVISE} je možné uvažovat přímo množinu dvojic proměnná a hodna navrácená do její aktuální domény a konstruovat tak přímo frontu Q_{REVISE} pro funkci PROPAGATE-AC-6. Dále uveďme, že algoritmus DnAC-6 ve verzi uvedené v [Deb96] používá k opětovné revizi ještě o něco sofistikovanější mechanizmus, kdy je k dvojicím proměnná a hodnota ve frontě Q_{REVISE} přidána navíc ještě informace o naposledy testované hodnotě v aktuální doméně, u níž skončilo hledání podpory. Přesným popisem tohoto postupu se nebudeme zabývat, čtenáře odkážeme na práci [Deb96], zde alespoň uveďme, že uvedená dodatečná informace poslouží v následných propagacích k eliminaci některých testů, celkovou teoretickou časovou složitost to ale neovlivní.

Zajímá-li nás průběh algoritmu DnAC-6, můžeme se klidně podívat na obrázek 3.1 zobrazující odebírání podmínky algoritmem DnAC-4, neboť průběh samotných změn ve struktuře řešeného problému je u obou algoritmů stejný.

Korektnost algoritmu shrnuje věta 3.2.

Věta 3.2 (KOREKTNOST ALGORITMU DNAC-6)**.** *Algoritmus DnAC-6 je korektní, tj. operace přidání a odebrání podmínky algoritmem DnAC-6 zachovávají hranovou konzistenci problému.* ■

Důkaz tvrzení této věty je možné nalézt v [Deb96], my jsme jej ovšem v podstatě demonstrovali při popisu programu realizujícího algoritmus. Přehled časové a prostorové složitosti uvádí následující tabulka. Zdůvodnění výsledků opět nebudeme rozvádět do všech detailů, pouze upozorníme na nejdůležitější obraty. Podrobnosti může čtenář opět nalézt v práci [Deb96].

DNAC-6	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Přidání podmínky	$O(\sum_{(u,v)\in C}(\lvert D_u\rvert+\lvert D_v\rvert)+\sum_{v\in V}\lvert D_v^0\rvert)$	$O(\lvert C\rVert D\rvert)$	$O(\sum_{(u,v)\in C}\lvert D_u\rVert D_v\rvert)$	$O(\lvert C\rVert D\rvert^2)$
Odebrání podmínky	$O(\sum_{(u,v)\in C}(\lvert D_u^0\rvert+\lvert D_v^0\rvert))$	$O(\lvert C\rVert D\rvert)$	$O(\sum_{(u,v)\in C}\lvert D_u^0\rVert D_v^0\rvert)$	$O(\lvert C\rVert D\rvert^2)$

Tabulka 3.2: Shrnutí časové a prostorové složitosti algoritmu DnAC-6

Nejprve si všimněme, že ve výrazech pro složitost přidání podmínky figurují velikosti aktuálních domén, zatímco ve výrazech pro složitost operace odebrání podmínky se objevují velikosti původních domén. Je tomu tak proto, že při odebírání podmínky algoritmus přímo pracuje s původní doménou.

Oproti algoritmu DnAC-4 přináší algoritmus DnAC-6 mnohem příznivější paměťovou složitost, nicméně ta zůstává stejně pro řadu problémů neúnosná. Na každou podmínku a hodnotu v aktuálních doménách proměnných, které svazuje, připadá jedna nejmenší podpora v datových strukturách algoritmu AC-6, což dohromady dává $O(\sum_{(u,v)\in C}(\lvert D_u\rvert+\lvert D_v\rvert))$ u přidání podmínky, resp. $O(\sum_{(u,v)\in C}(\lvert D_u^0\rvert+\lvert D_v^0\rvert))$ u odebrání podmínky. U složitosti operace přidání podmínky je nutné ještě připočíst prostor $O(\sum_{v\in V}\lvert D_v^0\rvert)$ potřebný pro uchování pole *justifications* v datové struktuře *data*.

Časová složitost v nejhorším případě operace přidání podmínky přesně odpovídá časové složitosti konzistenční procedury AC-6. Zastavme se u časové složitosti odebrání podmínky. Inicializace odebrání podmínky svazující proměnné u a v na řádcích 1 až 8 funkce RETRACT-CONSTRAINT-DNAC-6 proběhne v čase $O(\lvert D_u^0\rvert+\lvert D_v^0\rvert)$. V hlavní smyčce je každá navracená hodnota posuzována nejvýše jednou, na hledání podpory u sousední proměnné je přinejhorším potřeba projít celou aktuální doménu. Na jednu podmínku svazující proměnné u a v tedy

připadá čas $O(|D_u^0||D_v^0|)$, celkem má tedy hlavní smyčka časovou složitost $O(\sum_{(u,v)\in C}|D_u^0||D_v^0|)$. Konzistenční procedura volaná na závěr má složitost stejnou jako hlavní smyčka, dohromady tedy dostáváme uvedený výsledek.

Co je však nejdůležitější, jelikož jsou vždy aktualizovány pouze nejmenší podpory, vykazuje algoritmus dobré chování i v průměrném případě. Experimenty na vybraných známých problémech ukazují, že algoritmus DnAC-6 v průměrném případě algoritmus DnAC-4 výrazně předčí a to jak počtu vykonaných operací tak v celkovém čase.

Nevýhoda algoritmu DnAC-4 ohledně omezení arity podmínek na hodnotu 2 přetrvává i zde. Je-li třeba pracovat s podmínkami vyšší arity, je nutné i u algoritmu DnAC-6 aplikovat některou z metod binarizace.

Shrneme-li vlastnosti algoritmů DnAC-4 a DnAC-6, můžeme prohlásit, že oba algoritmy se vyznačují dobrou (v případě DnAC-6 velmi dobrou) časovou složitostí, která je v nejhorším případě dokonce optimální. Dobrá časová složitost je výsledkem toho, že algoritmy mají k dispozici velmi přesné informace o svém dosavadním průběhu a o závislostech mezi jednotlivými hodnotami v podobě udržovaných datových struktur a konají tak minimum zbytečné práce. Tato výhoda je u algoritmu DnAC-4 zaplacena velkou paměťovou složitostí, za niž jsou odpovědné zmiňované datové struktury, avšak algoritmus DnAC-6 tuto nevýhodu do určité míry odstraňuje.

Nevýhodu představuje fakt, že oba algoritmy mohou pracovat nejvýše s binárními podmínkami, a hlavně pak jejich závislost na poměrně složitých a speciálních datových strukturách, které ve svém důsledku znesnadňují další modifikace a adaptace obou algoritmů pro jiné formalizmy omezujících podmínek. Bez bližšího vysvětlení uveďme například formalizmus podmínek s preferencemi. Jako velmi výhodná se také jeví například možnost přizpůsobit algoritmus či alespoň jeho ideu pro jiný druh konzistence než hranové, což je u algoritmů DnAC-4 a DnAC-6 prakticky neproveditelné.

3.6 Dynamická hranová konzistence bez seznamů podpor: Algoritmy AC|DC

Docela jiný přístup k řešení hranové konzistence u dynamických problémů volí algoritmus AC|DC, který byl navržen v práci [NB94]. Algoritmus opět implementuje operace přidání a odebrání podmínky pro plnou dynamickou hranovou konzistenci. Základní myšlenkou algoritmu AC|DC je užívat a udržovat co nejméně pomocných datových struktur. To se odráží ve skutečnosti, že narozdíl od algoritmů DnAC-4 a DnAC-6 zde nejsou udržovány seznamy podpor.

Přidání podmínky je řešeno standardním způsobem jako v předchozích algoritmech, nejprve je podmínka zařazena do struktury problému a poté je pro ni spuštěna propagace konzistenční procedurou hranové konzistence.

Při odstraňování podmínky algoritmus AC|DC pracuje podobně jako algoritmy DnAC-4 či DnAC-6, opět postupně obnovuje jednotlivé hodnoty v aktuálních doménách proměnných. Rozdíl oproti předchozím algoritmům však spočívá v tom, že k této obnově používá minimum dodatečných datových struktur, které by bylo třeba udržovat. Stručně by se dal celý proces při navracení hodnot do aktuálních domén proměnných algoritmem AC|DC charakterizovat jako

konzistenční procedura AC-3 spuštěná opačným směrem, jinými slovy, propagována nejsou zúžení aktuálních domén, nýbrž jejich rozšíření.

V této sekci popíšeme rovněž nový algoritmus, který ideově vychází z algoritmu AC|DC. Pracovně jsme nový algoritmus nazvali AC|DC-2. Oproti algoritmu AC|DC přináší nový algoritmus podstatně příznivější dobu trvání obnovovací fáze. Z důvodů srozumitelnější prezentace nového algoritmu a též z důvodů lepšího naznačení vývojového procesu vedoucímu k novému algoritmu předvedeme nejprve algoritmus pracovně nazvaný AC|DC-0, který v podstatě představuje méně efektivní verzi algoritmu AC|DC publikovaného v článku [NB94]. Původní algoritmus AC|DC budeme od této chvíle označovat jako AC|DC-1. Nyní ale již přejdeme k formálnímu popisu jednotlivých algoritmů.

Program algoritmu AC|DC-0 v zavedeném programovém zápisu ukazuje algoritmus 3.4. Operace přidání podmínky je realizována standardním způsobem s využitím konzistenční procedury AC-3. Odebrání podmínky je realizováno přímočarou aplikací myšlenky propagace spuštěné opačným směrem. K realizaci odebrání podmínky je použita funkce EXTEND, která provádí konzistentní rozšíření dané aktuální domény. Připomeňme, že popis funkce EXTEND je uveden v předchozí kapitole.

Program je, jako obvykle, formulován pouze pro binární podmínky. Jelikož ale nezávisí na žádných speciálních datových strukturách, není toto omezení při praktické implementaci nutné. Ideu rozšíření pro nebinární podmínky předvedeme až později na algoritmu AC|DC-2, přičemž toto rozšíření bude jednoduše uplatnitelné jak na algoritmus AC|DC-0, tak na algoritmus AC|DC-1.

Algoritmus 3.4. AC|DC-0

ADD-CONSTRAINT-AC|DC-0 (P , c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 $P \leftarrow$ PROPAGATE-AC-3 (P , $\{c\}$)
3 **return** P

RETRACT-CONSTRAINT-AC|DC-0 (P , c)
1 $Q_{RESTORE} \leftarrow \emptyset$
2 $\{u, v\} \leftarrow V_c$
3 $D_u^{restore} \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-0(P , c , u , v)
4 $D_v^{restore} \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-0(P , c , v , u)
5 **if** $D_u^{restore} \neq \emptyset$ **then**
6 $P.D_u \leftarrow P.D_u \cup D_u^{restore}$
7 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{u\}$
8 **if** $D_v^{restore} \neq \emptyset$ **then**
9 $P.D_v \leftarrow P.D_v \cup D_v^{restore}$
10 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{v\}$
11 $P.C \leftarrow P.C - \{c\}$

```
12    (P, C_REVISE) ← PROPAGATE-ARC-RETRACTION-AC|DC-0(P, Q_RESTORE)
13    P ← PROPAGATE-AC-3(P, C_REVISE)
14    return P

INITIALIZE-ARC-RETRACTION-AC|DC-0 (P, c, u, v)
1     D_u^restore ← Ø
2     for each d_u ∈ (P.D_u^0 − P.D_u) do
3           if not HAS-SUPPORT(c, u, d_u, v, P.D_v) then
4                 D_u^restore ← D_u^restore ∪ {d_u}
5     return D_u^restore

PROPAGATE-ARC-RETRACTION-AC|DC-0 (P, Q_RESTORE)
1     C_REVISE ← Ø
2     while Q_RESTORE ≠ Ø do
3           u ∈ Q_RESTORE libovolná proměnná
4           Q_RESTORE ← Q_RESTORE − {u}
5           for each c ∈ P.C takovou, že u ∈ V_c do
6                 {u, v} ← V_c
7                 (D_u^restore, D_v^restore) ← EXTEND(c, P.D_u, P.D_v)
8                 if D_v^restore ≠ Ø then
9                       Q_RESTORE ← Q_RESTORE ∪ {v}
10                      P.D_v ← P.D_v ∪ D_v^restore
11          C_REVISE ← C_REVISE ∪ {c ∈ P.C | u ∈ V_c}
12    return (P, C_REVISE)
```

Algoritmus realizují funkce ADD-CONSTRAINT-AC|DC-0, RETRACT-CONSTRAINT-AC|DC-0, INITIALIZE-ARC-RETRACTION-AC|DC-0 a PROPAGATE-ARC-RETRACTION-AC|DC-0.

Funkce ADD-CONSTRAINT-AC|DC-0 provádí přidání podmínky, přičemž jako parametry dostává řešený problém P a přidávanou podmínku c. Příchozí podmínka je nejprve zapojena do struktury problému, poté je spuštěn konzistenční algoritmus. Původně hranově konzistentní problém funkce transformuje opět na hranově konzistentní s přidanou podmínkou, který je vrácen formou návratové hodnoty.

Odebrání podmínky má v programu na starosti funkce RETRACT-CONSTRAINT-AC|DC-0. Ta dostává v parametrech řešený problém P a odebíranou podmínku c. Samotné odebrání podmínky zde probíhá ve dvou fázích, stejně jako tomu bylo u algoritmů DnAC-4 a DnAC-6. Nejprve jsou do aktuálních domén zasažených proměnných navráceny hodnoty, za jejichž odstranění mohla přímo či nepřímo odebíraná podmínka, poté je spuštěna konzistenční procedura, která odstraní případné chybně navrácené hodnoty.

Zajímavý je ale na algoritmu AC|DC-0 hlavně způsob (to se ale bude týkat i AC|DC-1 a AC|DC-2), jakým jsou určovány hodnoty, jež mají být navráceny do aktuálních domén

proměnných po odebrání podmínky. Algoritmus k tomu používá postup odvozený od konzistenční procedury AC-3, ovšem jakoby spuštěné opačným směrem. Pokaždé dojde-li k rozšíření aktuální domény nějaké proměnné, řekněme v, jsou naplánovány k obnově sousední proměnné. Jejich aktuální domény jsou v následujících krocích rozšířeny o hodnoty, které mohly získat v právě obnovené aktuální doméně proměnné v podporu. K určení hodnot, jež mají být navráceny do dané aktuální domény, je v algoritmu použito specializované funkce EXTEND. V tomto lze spatřovat určitou analogii ke konzistenčnímu algoritmu AC-3, kde byla užívána funkce FILTER, která provádí přesný opak toho, co funkce EXTEND.

Funkcí INITIALIZE-ARC-RETRACTION-AC|DC-0 je zajišťována inicializace procesu odebírání podmínky odehrávajícího se na řádcích 1 až 10 funkce RETRACT-CONSTRAINT-AC|DC-0. Odebíráme-li podmínku c, jež svazuje proměnné u a v, pak tato funkce vrátí hodnoty, které mají být obnoveny v aktuální doméně proměnné u, resp. v, jejichž odebrání mohlo být přímo způsobeno podmínkou c. Množinu hodnot, jež mají být v aktuální doméně dané proměnné obnoveny, funkce INITIALIZE-ARC-RETRACTION-AC|DC-0 vrací formou návratové hodnoty. Funkce je během inicializace volána dvakrát, pro obnovu aktuálních domén obou proměnných zvlášť, neboli zvlášť pro hrany (u,v) a (v,u). Na tomto místě musíme zdůraznit, že množina hodnot k navrácení je pouze aproximována nadmnožinou hodnot, jež by měly být skutečně obnoveny. Připomeňme, že množinu hodnot k obnově nemůžeme určit žádným jednoduchým způsobem přesně, neboť ta je obvykle ovlivněna ještě aktuálními doménami jiných proměnných, jejichž konečná podoba v tomto okamžiku (při inicializaci) ještě není známa. Nejlepší aproximací, kterou lze získat z informačních zdrojů, jež má algoritmus k dispozici, je obnova všech hodnot, pro které v aktuální doméně proměnné sousedící s obnovovanou proměnnou skrze odebíranou podmínku chybí podpora. Jestliže jsou obnoveny všechny hodnoty z právě nepřítomných, kterým v okamžiku odebírání podmínky chybí podpory, pak jsou mezi těmito obnovenými hodnotami jistě i ty, které byly v minulosti odstraněny přímo kvůli odebírané podmínce c. Hlubší formální odůvodnění tohoto postupu je uvedeno v rámci důkazu tvrzení 3.1. Poznamenejme, že kromě hodnot obnovených při této inicializaci mohou být během dalšího procesu obnov do aktuální domény proměnné u, resp. v navráceny i další hodnoty.

Kvalitu aproximace v tomto případě posuzujeme podle množství navracených hodnot, platí, že čím méně hodnot k navrácení, tím kvalitnější aproximace. Výhodnost použití co nejlepší aproximace oceníme v následných, časově velmi náročných krocích algoritmu. Nezřídka může dokonce nastat situace, že inicializační obnova aktuálních domén proměnných svázaných odebíranou podmínkou pomocí funkce INITIALIZE-ARC-RETRACTION-AC|DC-0 nenavrátí žádnou hodnotu, v takovém případě na další kroky algoritmu ani nedojde.

Funkce PROPAGATE-ARC-RETRACTION-AC|DC-0, která dostává jako parametry řešený problém P a frontu $Q_{RESTORE}$, nastupuje ihned po inicializaci, jejím úkolem je postupně obnovovat aktuální domény proměnných ovlivněných předchozími obnovami. Funkce tak vlastně realizuje zmiňovanou obrácenou propagaci. Pro tento proces opět platí to, co bylo řečeno v předchozím textu, kdykoli jsou navráceny do aktuální domény nějaké proměnné nové hodnoty, je posouzeno, zda nelze díky této změně rozšířit aktuální domény sousedních proměnných. Proměnné naplánované k revizi svých sousedů jsou vkládány do fronty $Q_{RESTORE}$. Na začátku fronta obsahuje ty proměnné z dvojice proměnných, které svazovala odebraná podmínka, jimž byly při inicializaci rozšířeny aktuální domény.

Ukažme nyní detailněji, jakým způsobem funguje hlavní smyčka funkce PROPAGATE-ARC-RETRACTION-AC|DC-0 operující s frontou $Q_{RESTORE}$. Předpokládejme, že byla z fronty $Q_{RESTORE}$ právě vybrána proměnná u, v tomto okamžiku platí, že proměnné u byla v minulosti

rozšířena aktuální doména. Změna aktuální domény proměnné u mohla ovlivnit sousední proměnné a tudíž je potřeba posoudit, zda není možné navrátit další hodnoty do aktuálních domén těchto sousedních proměnných. Zkoumání sousedních proměnných probíhá na řádcích 5 až 10 funkce PROPAGATE-ARC-RETRACTION-AC|DC-0. Pro každou podmínku, která svazuje proměnnou u a nějakou další proměnnou, označme ji třeba v, je zavolána operace EXTEND. Funkce vrátí množiny hodnot, o které je možné konzistentně rozšířit aktuální domény proměnných u a v, přičemž algoritmus pro skutečné rozšíření použije pouze množinu hodnot k navrácení do aktuální domény proměnné v. Jestliže je tato množina neprázdná, je i proměnná v naplánována do fronty $Q_{RESTORE}$.

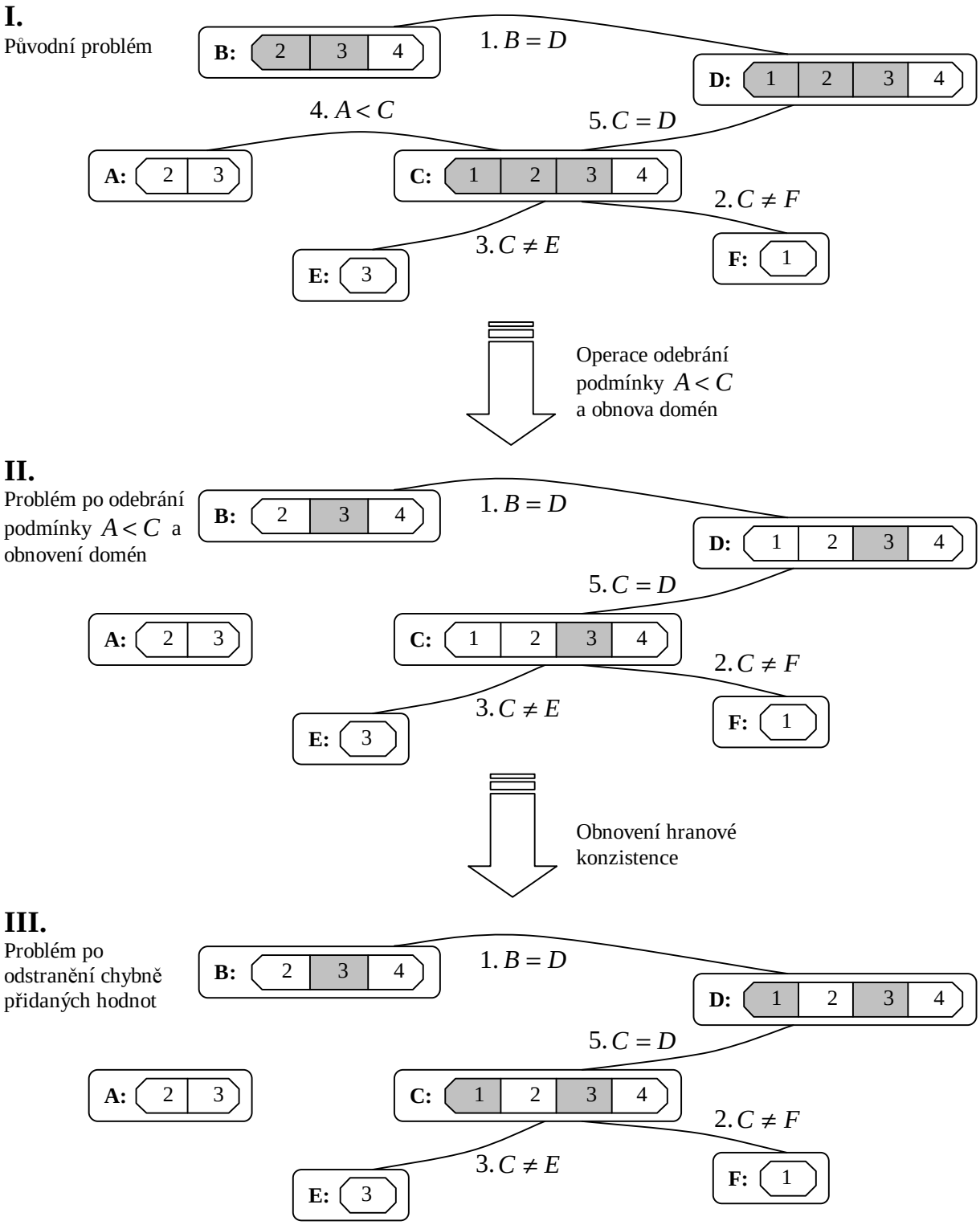
Během celého průběhu obnov aktuálních domén proměnných jsou s ohledem na možnost chybného obnovení hodnot plánovány k opětovné revizi podmínky svazující modifikované proměnné, k tomuto účelu slouží seznam C_{REVISE}.

Problém s rozšířenými aktuálními doménami proměnných a stejně tak i podmínky k revizi jsou funkcí PROPAGATE-ARC-RETRACTION-AC|DC-0 nakonec vráceny jako návratová hodnota.

Funkce RETRACT-CONSTRAINT-AC|DC-0 je následně zakončena spuštěním konzistenční procedury AC-3 pro podmínky obsažené v seznamu C_{REVISE}. Výsledný hranově konzistentní problém s odebranou podmínkou je nakonec vrácen v návratové hodnotě.

Při závěrečné filtraci chybně obnovených hodnot je možné jednoduchým opatřením algoritmus AC-3 přizpůsobit tak, aby se pokoušel odfiltrovat pouze hodnoty, které byly obnoveny v bezprostředně předcházející obnovovací fázi operace odebrání podmínky. Jak bude vidět u časové složitosti, může toto opatření ušetřit významné množství práce (ke snížení teoretické časové složitosti však tato úprava nepovede).

Ilustraci k průběhu odebírání podmínky algoritmem AC|DC-0 ukazuje obrázek 3.2. Obrázek znázorňuje obě fáze odebírání, tj. obnovu aktuálních domén a následnou opravu chyb pomocí konzistenční procedury.



Obrázek 4.2: Průběh odebrání podmínky algoritmem AC|DC-0

Korektnost algoritmu AC|DC-0 zachycuje věta 3.3, pro důkaz tvrzení této věty lze přímo aplikovat argumentaci k podložení téhož tvrzení o algoritmu AC|DC-1, což je uvedeno v práci [NB94]. Vzhledem k tomu, že uvedený popis programu algoritmu obsahuje podrobné odůvodnění pro jednotlivé kroky, nebudeme větu explicitně dokazovat.

Věta 3.3 (KOREKTNOST ALGORITMU AC|DC-0)**.** *Algoritmus AC|DC-0 je korektní, tj. operace přidání a odebrání podmínky algoritmem AC|DC-0 zachovávají hranovou konzistenci problému.* ■

Následující tabulka shrnuje složitost algoritmu AC|DC-0.

AC|DC-0	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Přidání podmínky	$O(\|C\|)$	$O(\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u\|+\|D_v\|)(\|D_u\|\|D_v\|))$	$O(\|C\|\|D\|^3)$
Odebrání podmínky	$O(\|V\|+\|C\|)$	$O(\|V\|+\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u^0\|+\|D_v^0\|)(\|D_u^0\|\|D_v^0\|))$	$O(\|C\|\|D\|^3)$

Tabulka 3.3: Shrnutí časové a prostorové složitosti algoritmu AC|DC-0

Prostorová složitost přidání podmínky algoritmem AC|DC-0 je rovna prostorové složitosti použitého konzistenčního algoritmu. O časové složitosti operace přidání podmínky lze prohlásit totéž.

Během odebírání podmínky je nutné udržovat frontu $Q_{RESTORE}$ a seznam C_{REVISE}, to dává celkem nároky na prostor $O(|V|+|C|)$. Společně s následnou aplikací konzistenční procedury AC-3 dostáváme prostorovou složitost odebrání podmínky algoritmem AC|DC-0 $O(|V|+|C|)$.

Nyní se podívejme na časovou složitost operace odebrání podmínky svazující proměnné u a v. Časová složitost funkce INITIALIZE-ARC-RETRACTION-AC|DC-0 je $O(|D_u^0||D_v|+|D_u||D_v^0|)$ za předpokladu, že používáme výchozí implementaci operace HAS-SUPPORT, která testuje přímo dvojice kompatibilních hodnot. Na každou z chybějících hodnot připadá prověřit všechny dvojice hodnot, jež tato chybějící hodnota utváří s hodnotami v aktuální doméně sousední proměnné, což je $|D_u|$, resp. $|D_v|$ dvojic hodnot.

K časové složitosti funkce PROPAGATE-ARC-RETRACTION-AC|DC-0 je zapotřebí učinit pozorování, že na každou podmínku svazující proměnné u a v může být funkce EXTEND zavolána nejvýše $|D_u^0|+|D_v^0|$ krát, jelikož každé její vyvolání musí být způsobeno navrácením alespoň jedné chybějící hodnoty do aktuální domény proměnné u nebo v. Vezmeme-li v potaz, že časová složitost v nejhorším případě je pro výchozí implementaci operace EXTEND (založené na dvojicích kompatibilních hodnot) $O(|D_u^0||D_v^0|)$ pro podmínku (u,v), dostaneme pro funkci PROPAGATE-ARC-RETRACTION-AC|DC-0 čas v nejhorším případě $O(\sum_{(u,v)\in C}(|D_u^0|+|D_v^0|)(|D_u^0||D_v^0|))$, což je pro identické domény $O(|C||D|^3)$.

Včetně následného spuštění konzistenční procedury AC-3 obdržíme pro časovou složitost operace odebírání podmínky výraz $O(|D_u^0||D_v| + |D_u||D_v^0| + \sum_{(u,v)\in C}(|D_u^0| + |D_v^0|)(|D_u^0||D_v^0|))$, což je $O(\sum_{(u,v)\in C}(|D_u^0| + |D_v^0|)(|D_u^0||D_v^0|))$, neboli $O(|C||D|^3)$ pro identické domény.

Prozkoumáme-li blíže činnost algoritmu AC|DC-0, brzy zjistíme, jak jednoduchou úpravou snížit časovou náročnost fáze, ve které jsou obnovovány aktuální domény proměnných. Tíživým místem je v algoritmu volání funkce EXTEND, ta provádí obnovu zainteresovaných domén vzhledem k jejich stavu v daném okamžiku a to bez ohledu na to, které hodnoty do nich byly navráceny v předchozích krocích (a mohly se stát novými podporami dosud nepřítomných hodnot) a které se v nich nacházely již předtím. Platí, že ke korektní obnově hodnot v aktuálních doménách sousedních proměnných jsou podstatné pouze ty hodnoty, vůči nimž obnova sousedních proměnných ještě neproběhla.

Aby bylo možné odpovídajícím způsobem algoritmus upravit, je třeba u každé hodnoty v aktuální doméně proměnné rozlišovat, zda vůči ní již proběhla obnova sousedů, či ještě ne. V okamžiku, kdy dochází k navracení hodnot do aktuálních domén sousedních proměnných, jsou jako výchozí hodnoty brány pouze hodnoty s neobnovenými sousedními proměnnými. Po dokončení navracení hodnot do sousedních proměnných je i těmto výchozím hodnotám nastaven příznak, že aktuální domény sousedních proměnných jsou již obnoveny.

Právě popsaný způsob přesně odpovídá tomu, jak postupuje publikovaný algoritmus AC|DC. Program algoritmu AC|DC, který však budeme uvádět pod označením AC|DC-1, formálně popisuje v zavedeném programovém zápise algoritmus 3.5. Podotkněme, že zde uvedená symbolická implementace algoritmu se drobně liší od programu uvedeném v článku [NB94]. Autoři algoritmu AC|DC používají jakoby zdvojené aktuální domény proměnných, přičemž jedna z těchto domén slouží k uchovávání hodnot, které už byly navráceny, ale dosud nebyly obnoveny hodnoty v aktuálních doménách sousedních proměnných vůči těmto již navráceným hodnotám. Po uskutečnění obnov sousedních proměnných se pak tyto hodnoty přesunou do klasické aktuální domény proměnné.

V algoritmu AC|DC-1 užíváme s ohledem na co nejjednodušší zápis programu standardně jednu aktuální doménu proměnné. Hodnoty, vůči nimž má proběhnout obnova sousedů, jsou místo do zdvojené aktuální domény plánovány přímo do řídící fronty algoritmu. Funkčně je uvedený algoritmus AC|DC-1 ekvivalentní s algoritmem popsaným v práci [NB94].

Algoritmus 3.5. AC|DC-1

ADD-CONSTRAINT-AC|DC-1 (P , c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 $P \leftarrow$ PROPAGATE-AC-3(P , $\{c\}$)
3 **return** P

RETRACT-CONSTRAINT-AC|DC-1 (P , c)
1 $Q_{RESTORE} \leftarrow \emptyset$
2 $\{u, v\} \leftarrow V_c$
3 $D_u^{restore} \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-1(P , c , u , v)
4 $D_v^{restore} \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-1(P , c , v , u)

5 **if** $D_u^{restore} \neq \emptyset$ **then**
6 $P.D_u \leftarrow P.D_u \cup D_u^{restore}$
7 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, D_u^{restore})\}$
8 **if** $D_v^{restore} \neq \emptyset$ **then**
9 $P.D_v \leftarrow P.D_v \cup D_v^{restore}$
10 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v, D_v^{restore})\}$
11 $P.C \leftarrow P.C - \{c\}$
12 $(P, C_{REVISE}) \leftarrow$ PROPAGATE-ARC-RETRACTION-AC|DC-1$(P, Q_{RESTORE})$
13 $P \leftarrow$ PROPAGATE-AC-3(P, C_{REVISE})
14 **return** P

INITIALIZE-ARC-RETRACTION-AC|DC-1 (P, c, u, v)
1 $D_u^{restore} \leftarrow \emptyset$
2 **for each** $d_u \in (P.D_u^0 - P.D_u)$ **do**
3 **if not** HAS-SUPPORT$(c, u, d_u, v, P.D_v)$ **then**
4 $D_u^{restore} \leftarrow D_u^{restore} \cup \{d_u\}$
5 **return** $D_u^{restore}$

PROPAGATE-ARC-RETRACTION-AC|DC-1 $(P, Q_{RESTORE})$
1 $C_{REVISE} \leftarrow \emptyset$
2 **while** $Q_{RESTORE} \neq \emptyset$ **do**
3 $(u, D_u^{restored}) \in Q_{RESTORE}$ libovolná proměnná
4 $Q_{RESTORE} \leftarrow Q_{RESTORE} - \{(u, D_u^{restored})\}$
5 **for each** $c \in P.C$ takovou, že $u \in V_c$ **do**
6 $\{u, v\} \leftarrow V_c$
7 $D_v^{restore} \leftarrow \emptyset$
8 **for each** $d_v \in (P.D_v^0 - P.D_v)$ **do**
9 **if** HAS-SUPPORT$(c, v, d_v, u, D_u^{restored})$ **then**
10 $D_v^{restore} \leftarrow D_v^{restore} \cup \{d_v\}$
11 **if** $D_v^{restore} \neq \emptyset$ **then**
12 $P.D_v \leftarrow P.D_v \cup D_v^{restore}$
13 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v, D_v^{restore})\}$
14 $C_{REVISE} \leftarrow C_{REVISE} \cup \{c \in P.C \mid u \in V_c\}$
15 **return** (P, C_{REVISE})

Program algoritmu AC|DC-1 sestává z funkcí ADD-CONSTRAINT-AC|DC-1, RETRACT-CONSTRAINT-AC|DC-1, INITIALIZE-ARC-RETRACTION-AC|DC-1 a PROPAGATE-ARC-

RETRACTION-AC|DC-1. Jejich význam přesně odpovídá obdobným funkcím z programu algoritmu AC|DC-1, proto zde nebudeme podávat jejich detailní popis.

Co však musíme podrobněji rozebrat je samotná realizace navracení hodnot do aktuálních domén proměnných. Vždy, když jsou do aktuální domény nějaké proměnné navráceny dříve odstraněné hodnoty, je tato proměnná společně s množinou jejích obnovených hodnot naplánována do fronty $Q_{RESTORE}$. Jako obvykle, ve frontě $Q_{RESTORE}$ se proměnná v plus navíc dodatečný seznam $D_v^{restored}$ hodnot obnovených v její aktuální doméně nachází právě tehdy, když je potřeba zkontrolovat, zda díky této změně nezískaly podpory další chybějící hodnoty v proměnných sousedících s v.

V hlavní smyčce algoritmu je pak pokaždé z fronty $Q_{RESTORE}$ odebrán jeden libovolný záznam, pro tuto chvíli jej označme $(u, D_u^{restored})$, a následně je uskutečněn pokus o navrácení dosud chybějících hodnot do všech aktuálních domén proměnných sousedících s u. Navrácení hodnoty je testováno pouze vzhledem k hodnotám v množině $D_u^{restored}$, jinými slovy, jestliže je v tomto kroku obnovena nějaká hodnota, pak je to díky tomu, že získala podporu mezi hodnotami v $D_u^{restored}$. Testování, zda má daná hodnota podporu v množině hodnot vzhledem k určité podmínce, je přenecháno na specializovanou operaci HAS-SUPPORT.

Korektnost algoritmu shrnuje následující věta.

Věta 3.4 (KOREKTNOST ALGORITMU AC|DC-1)**.** *Algoritmus AC|DC-1 je korektní, tj. operace přidání a odebrání podmínky algoritmem AC|DC-1 transformují původní hranově konzistentní problém na modifikovaný problém, který je opět hranově konzistentní.* ■

Ohledně počtu kroků algoritmu AC|DC-1 již z pouhého programového zápisu plyne, že algoritmus provede nejvýše tolik kroků, co AC|DC-0. V tomto případě je však situace ještě příznivější, neboť dokonce platí, že algoritmus AC|DC-1 má lepší teoretickou časovou složitost obnovovací fáze v nejhorším případě. Prostorovou a časovou v nejhorším případě algoritmu AC|DC-1 shrnuje následující tabulka.

AC|DC-1	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Přidání podmínky	$O(\|C\|)$	$O(\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u\|+\|D_v\|)(\|D_u\|\|D_v\|))$	$O(\|C\|\|D\|^3)$
Odebrání podmínky	$O(\sum_{v\in V}\|D_v^0\|+\|C\|)$	$O(\|V\|\|D\|+\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u^0\|+\|D_v^0\|)(\|D_u^0\|\|D_v^0\|))$	$O(\|C\|\|D\|^3)$

Tabulka 3.4: Shrnutí časové a prostorové složitosti algoritmu AC|DC-1

Prostorová i časová složitost v nejhorším případě opět přesně odpovídá použité konzistenční proceduře AC-3.

U operace odebrání podmínky došlo k navýšení prostorové složitosti o výraz $\sum_{v\in V}|D_v^0|$. Toto navýšení má na svědomí rozšíření položek řídící fronty o množiny obnovených hodnot.

Poznamenejme, že ke stejnému navýšení by došlo, kdyby byly použity zdvojené domény, jež vystupují v původním návrhu algoritmu AC|DC-1.

Časová složitost v nejhorším případě operace odebrání podmínky zůstává stejná. Avšak obnovovací fáze operace odebrání podmínky nyní nepotřebuje čas $O(\sum_{(u,v)\in C}(|D_u^0|+|D_v^0|)(|D_u^0\|D_v^0|))$, ale vystačí pouze s $O(\sum_{(u,v)\in C}|D_u^0\|D_v^0|)$. K odůvodnění tohoto výsledku je nezbytné učinit pozorování, že každá dvojice hodnot v doménách sousedících proměnných je testována nejvýše jedenkrát. Tento výsledek opět platí pro výchozí implementaci operace HAS-SUPPORT, jež pracuje přímo s dvojicemi kompatibilních hodnot. Celkovou časovou složitost operace odebrání podmínky nakonec ale pokazí konzistenční procedura AC-3 volaná na závěr.

Jak sami autoři algoritmu AC|DC-1 podotýkají, mezi pozitivní vlastnosti algoritmu patří jeho nízká paměťová složitost. Dalším pozitivem je nezávislost algoritmu na jakýchkoli speciálních datových strukturách, což otevírá cestu dalším adaptacím. Autoři například zmiňují systémy podmínek s preferencemi. Další výhodou je díky absenci složitých datových struktur jistě i možnost implementace algoritmu pro podmínky vyšší arity.

Nedostatek algoritmu představuje jeho poměrně značná teoretická časová složitost v nejhorším případě způsobená konzistenční procedurou AC-3. Ovšem zde je třeba podotknout, že se jedná o nejhorší a nikoli o průměrný případ, který podle experimentů vyznívá příznivěji. Za cenu zvýšení paměťových nároků můžeme nahradit proceduru AC-3 efektivnějšími konzistenčními algoritmy pro hranovou konzistenci popsanými v předchozí kapitole. Nicméně taková náhrada jde zásadně proti myšlence algoritmu AC|DC-1, která klade důraz na jednoduchou implementaci a na nezávislost na seznamech podpor. Proto je mnohem přirozenější poohlédnout se například po algoritmu AC-2001 popsaném v [BR01] či AC-3.1, o kterém pojednává práce [ZY01]. Oba algoritmy ideově vycházejí z AC-3, ale zlepšují jeho složitost v nejhorším případě na optimální hodnotu $O(|C\|D|^2)$ pro problémy s identickými doménami.

Praktické experimenty s algoritmem AC|DC-1 ukazují, že výkon algoritmu v obnovovací fázi velmi významnou měrou závisí na přesnosti obnov aktuálních domén. Uvědomme si, že chybně navrácená hodnota může zapříčinit až celý řetěz chybně obnovených hodnot. Tento jev se na výkonu algoritmu odráží negativně hned dvojím způsobem. Za prvé, algoritmus stráví nezanedbatelný čas zbytečnou obnovou aktuálních domén zapříčiněnou původně chybně navrácenou hodnotou, za druhé, je pak ještě nutné chybně přidané hodnoty vyloučit, což je další práce navíc pro konzistenční proceduru volanou na závěr.

Autoři algoritmu AC|DC navrhli alternativní a řekněme poněkud opatrnější strategii pro obnovovací fázi, čímž se zřejmě snažili o eliminaci výše uvedeného jevu. Aniž bychom zacházeli do detailů uveďme, že při této strategii algoritmus pro každou navrácenou hodnotu ještě navíc testuje, zda tato hodnota v každé sousední proměnné podporuje aspoň jednu hodnotu. Jestliže tomu tak není, algoritmus si tuto hodnotu označí a v dalších krocích s ní již nepracuje. Praktické testy, které autoři algoritmu provedli, ale nakonec ukázaly, že časové nároky na dodatečné testy převyšují čas ušetřený obnovou menšího počtu hodnot.

My jsme zvolili k řešení uvedené neefektivity obnovovací fáze algoritmu AC|DC docela odlišný přístup. Jednou z hlavních myšlenek našeho nového přístupu je nějakým způsobem využít při obnově aktuálních domén informací, které by během své činnosti mohla vydedukovat konzistenční procedura, v našem případě procedura AC-3. Přesněji, při propagaci konzistenční procedurou AC-3 budeme zaznamenávat určité užitečné informace o jejím průběhu, které později pomohou ke značnému zefektivnění obnovovací fáze operace odebrání podmínky.

V průběhu konzistenční procedury AC-3 budeme pro každou odebranou hodnotu zaznamenávat příčinu jejího odstranění, tj. sousední proměnnou, ve které odstraněná hodnota poprvé ztratila všechny podpory. Další informací, kterou budeme uchovávat, je čas odstranění každé chybějící hodnoty, přičemž tímto časem míníme nějaké globální počítadlo kroků programu. Tyto dvě informace budeme, podobně jako je tomu u algoritmů DnAC-4 a DnAC-6, uchovávat v poli *justifications*, jež bude indexováno proměnnými a jejich hodnotami. Každá buňka tohoto pole bude mít dvě složky: *variable* reprezentující proměnnou, která zapříčinila odebrání hodnoty, jíž daná buňka patří, a *time* uchovávající čas odebrání dané hodnoty. Ještě než přejdeme k vlastnímu algoritmu pro dynamickou hranovou konzistenci, je nutné upravit konzistenční proceduru AC-3 tak, aby odrážela uvedené požadavky, tedy, aby odpovídajícím způsobem aktualizovala pole *justifications*.

Konzistenční proceduru AC-3 doplníme o datovou strukturu *data*, která bude obsahovat pole *justifications* a globální čas (celočíselný čítač kroků algoritmu) *time*. Abychom upravený algoritmus AC-3 odlišili od původní verze uvedené v předchozí kapitole, budeme jej označovat AC-3'.

Programový zápis algoritmu AC-3' pro binární podmínky ukazuje algoritmus 3.6. Omezení na binární podmínky zde ale není podmínkou, později v příloze ukážeme odpovídající rozšíření zahrnující též podmínky arity vyšší než 2.

Algoritmus 3.6. AC-3'

PROPAGATE-AC-3' (P , *data*, C_{REVISE})

1 $Q_{REVISE} \leftarrow C_{REVISE}$

2 **while** $Q_{REVISE} \neq \emptyset$ **do**

3 $c \in Q_{REVISE}$ libovolná podmínka

4 $Q_{REVISE} \leftarrow Q_{REVISE} - \{c\}$

5 $\{u, v\} \leftarrow V_c$

6 $(D_u^{remove}, D_v^{remove}) \leftarrow$ FILTER(c , $P.D_u$, $P.D_v$)

7 (P , *data*, Q_{REVISE}^{uv}) $\leftarrow$ FILTER-ARC-AC-3'(P , *data*, u , v , D_u^{remove})

8 (P , *data*, Q_{REVISE}^{vu}) $\leftarrow$ FILTER-ARC-AC-3'(P , *data*, v , u , D_u^{remove})

9 $Q_{REVISE} \leftarrow Q_{REVISE} \cup Q_{REVISE}^{uv} \cup Q_{REVISE}^{vu}$

10 **return** (P , *data*)

FILTER-ARC-AC-3' (P , *data*, u , v , D_u^{remove})

1 $Q_{REVISE} \leftarrow \emptyset$
2 nechť u a v jsou svázány podmínkou c
3 **if** $D_u^{remove} \neq \emptyset$ **then**
4 **for each** $d_u \in D_u^{remove}$ **do**
5 $P.D_u \leftarrow P.D_u - \{d_u\}$
6 $data.justifications[u, d_u].variable = v$
7 $data.justifications[u, d_u].time = data.time$
8 $data.time \leftarrow data.time + 1$
9 $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{c' \in P.C \mid u \in V_{c'} \,\&\, c' \neq c\}$
10 return (P , *data*, Q_{REVISE})

Z důvodů lepší čitelnosti jsme algoritmus rozdělili do dvou funkcí PROPAGATE-AC-3', jejíž význam odpovídá přesně funkci PROPAGATE-AC-3 z algoritmu AC-3, a pomocné FILTER-ARC-AC-3', která zpracovává odebrání hodnot z aktuální domény proměnné a aktualizuje dodatečné informace v datové struktuře *data*. Algoritmus funguje naprosto stejným způsobem jako standardní konzistenční procedura AC-3. Veškeré provedené úpravy mají za cíl pouze udržovat navíc informace o běhu procedury, tj. informace charakterizující okolnosti, za kterých došlo k odebrání dané hodnoty.

Ohledně časové složitosti algoritmu nedošlo k žádné změně. Prostorová složitost narostla o paměť potřebnou k uložení pole *justifications* v datové struktuře *data*, konkrétně se jedná o navýšení úměrné $O(\sum_{v \in V} |D_v|)$, neboli $O(|V||D|)$ pro identické domény.

S využitím informací udržovaných v poli *justifications* jsme vybudovali nový algoritmus pro dynamickou hranovou konzistenci, který jsme pracovně označili AC|DC-2. Nový algoritmus vychází z AC|DC-1, při obnovách aktuálních domén proměnných ale oproti AC|DC-1 postupuje opatrněji. Dodatečná režie opatrnějšího postupu je však v tomto případě nevýznamná a je bohatě vyvážena menším počtem navracených hodnot, tj. větším množstvím ušetřené práce.

Operace přidání podmínky je v algoritmu AC|DC-2 prováděna standardním způsobem. Nejprve je nová podmínka přidána do struktury problému a následně je spuštěna propagace konzistenční procedurou pro nově přidanou podmínku. V případě algoritmu AC|DC-2 se však nejedná o proceduru AC-3, nýbrž o AC-3', která navíc generuje informace do pole *justifications*.

Základní schéma postupu při odebírání podmínky je prakticky stejné jako u všech ostatních popsaných algoritmů. Po odebrání podmínky ze struktury problému jsou postupně do aktuálních domén přímo či nepřímo zasažených proměnných navraceny hodnoty, přičemž základní myšlenkou celého tohoto procesu je opět jakási obrácená propagace změn. Operace odebrání podmínky je nakonec uzavřena voláním konzistenční procedury AC-3', která vyloučí chybně přidané hodnoty z předchozí fáze.

Nyní se podívejme podrobněji na samotný postup při obnovách aktuálních domén proměnných. V první řadě jsou obnoveny hodnoty v aktuálních doménách proměnných, které svazovala odebíraná podmínka. Jestliže odebíraná podmínka svazovala proměnné u a v, pak do aktuální domény proměnné u jsou navráceny všechny hodnoty, jež nemají v aktuální

doméně proměnné v žádnou podporu a jejichž odebrání zapříčinila proměnná v, tj. v poli *justifications* měla buňka náležející hodnotě testované na navrácení složku *variable* rovnu v. Analogicky je celý postup aplikován i na proměnnou v. Pokaždé dojde-li ke změně aktuální domény nějaké proměnné, je naplánován pokus o obnovu jejích sousedů, neboť hodnoty dosud chybějící v aktuálních doménách těchto proměnných mohly díky provedené změně získat podporu.

Předpokládejme nyní, že do aktuální domény proměnné u byla v některém z minulých kroků navrácena množina hodnot $D_u^{restored}$. V tomto okamžiku je třeba uskutečnit pokus o navrácení dalších hodnot do aktuálních domén všech sousedních proměnných. Kandidáti pro navrácení do aktuální domény proměnné v sousedící s u jsou ty dosud chybějící hodnoty, jejichž odebrání zapříčinila proměnná u, tj. složka *variable* příslušné buňky pole *justifications* obsahuje proměnnou u, a jež byly odstraněny až po odstranění v minulosti nejdříve odstraněné hodnoty z množiny $D_u^{restored}$, formálně řečeno, čas ve složce *time* buňky pole *justifications* náležející hodnotě kandidující na navrácení musí být větší než nejmenší čas uložený v buňkách pole *justifications* patřící hodnotám z množiny $D_u^{restored}$. Nakonec skutečně navrácená hodnota do aktuální domény proměnné v musí mít ještě navíc alespoň jednu podporu mezi hodnotami z množiny $D_u^{restored}$.

Formálně je nový algoritmus AC|DC-2 popsán pomocí zavedeného programového zápisu jako algoritmus 3.7. Tato verze algoritmu pracuje pouze s binárními podmínkami, rozšíření pro podmínky vyšší arity uvedeme později v příloze.

Algoritmus 3.7. AC|DC-2

ADD-CONSTRAINT-AC|DC-2 (P , *data*, c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 $P \leftarrow$ PROPAGATE-AC-3'(P , *data*, $\{c\}$)
3 **return** (P , *data*)

RETRACT-CONSTRAINT-AC|DC-2 (P , *data*, c)
1 $Q_{RESTORE} \leftarrow \emptyset$
2 $\{u, v\} \leftarrow V_c$
3 $(time_u, D_u^{restore}) \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2(P , c , u , v)
4 $(time_v, D_v^{restore}) \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2(P , c , v , u)
5 **if** $D_u^{restore} \neq \emptyset$ **then**
6 $P.D_u \leftarrow P.D_u \cup D_u^{restore}$
7 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, time_u, D_u^{restore})\}$
8 **if** $D_v^{restore} \neq \emptyset$ **then**
9 $P.D_v \leftarrow P.D_v \cup D_v^{restore}$
10 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v, time_v, D_v^{restore})\}$
11 $P.C \leftarrow P.C - \{c\}$

12 $(P,\ C_{REVISE}) \leftarrow$ PROPAGATE-ARC-RETRACTION-AC|DC-2(P, *data*, $Q_{RESTORE}$)
13 $P \leftarrow$ PROPAGATE-AC-3'(P, C_{REVISE})
14 **return** (P, *data*)

INITIALIZE-ARC-RETRACTION-AC|DC-2 (P, c, u, v)
1 $time_u \leftarrow \infty$
2 $D_u^{restore} \leftarrow \emptyset$
3 **for each** $d_u \in (P.D_u^0 - P.D_u)$ **do**
4 **if** $data.justifications[u,d_u].variable = v$ **then**
5 **if not** HAS-SUPPORT(c, u, d_u, v, $P.D_v$) **then**
6 $D_u^{restore} \leftarrow D_u^{restore} \cup \{d_u\}$
7 $time_u \leftarrow \min(time_u, data.justifications[u,d_u].time)$
8 $data.justifications[u,d_u] \leftarrow NIL$
9 **return** ($time_u$, $D_u^{restore}$)

PROPAGATE-ARC-RETRACTION-AC|DC-2 (P, *data*, $Q_{RESTORE}$)
1 $C_{REVISE} \leftarrow \emptyset$
2 **while** $Q_{RESTORE} \neq \emptyset$ **do**
3 $(u, time_u, D_u^{restored}) \in Q_{RESTORE}$ libovolná proměnná
4 $Q_{RESTORE} \leftarrow Q_{RESTORE} - \{(u, time_u, D_u^{restored})\}$
5 **for each** $c \in P.C$ takovou, že $u \in V_c$ **do**
6 $\{u, v\} \leftarrow V_c$
7 $time_v \leftarrow \infty$
8 $D_v^{restore} \leftarrow \emptyset$
9 **for each** $d_v \in (P.D_v^0 - P.D_v)$ **do**
10 **if** $data.justifications[v,d_v].variable = u$ **and**
11 $data.justifications[v,d_v].time > time_u$ **then**
12 **if** HAS-SUPPORT(c, v, d_v, u, $D_u^{restored}$) **then**
13 $D_v^{restore} \leftarrow D_v^{restore} \cup \{d_v\}$
14 $time_v \leftarrow$
15 $\leftarrow \min(time_v, data.justifications[v,d_v].time)$
16 $data.justifications[v,d_v] \leftarrow NIL$
17 **if** $D_v^{restore} \neq \emptyset$ **then**
18 $P.D_v \leftarrow P.D_v \cup D_v^{restore}$
19 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v, time_v, D_v^{restore})\}$
20 $C_{REVISE} \leftarrow C_{REVISE} \cup \{c \in P.C \mid u \in V_c\}$
21 **return** (P, *data*, C_{REVISE})

Přidání podmínky má na starosti funkce ADD-CONSTRAINT-AC|DC-2, jejímiž parametry jsou řešený problém P, datová struktura *data* a přidávaná podmínka c. K obnovení hranové konzistence problému funkce uplatňuje konzistenční proceduru AC-3' voláním funkce PROPAGATE-AC-3'. Výsledkem činnosti funkce ADD-CONSTRAINT-AC|DC-2, je hranově konzistentní problém P s přidanou podmínkou c a datová struktura *data* s příslušně aktualizovaným polem *justifications*.

Funkce RETRACT-CONSTRAINT-AC|DC-2 společně s funkcemi INITIALIZE-ARC-RETRACTION-AC|DC-2 a PROPAGATE-ARC-RETRACTION-AC|DC-2 uskutečňují odebrání podmínky. Funkce RETRACT-CONSTRAINT-AC|DC-2 v parametrech očekává řešený problém P, datovou strukturu *data* s vyplněným polem *justifications* a odebíranou podmínku c. Jako výsledek funkce vrátí problém s odebranou podmínkou a odpovídajícím způsobem modifikovanou datovou strukturu *data*.

Počáteční navrácení chybějících hodnot do aktuálních domén proměnných svázaných odebíranou podmínkou je, jako obvykle, prováděno funkcí INITIALIZE-ARC-RETRACTION-AC|DC-2. Funkce je opět volána dvakrát, zvlášť pro každý směr, tj. zvlášť pro hrany (u,v) a (v,u) za předpokladu, že odebíraná podmínka svazovala proměnné u a v. Narozdíl od algoritmů AC|DC-0 a AC|DC-1 je počáteční obnova prováděna ještě s ohledem na příčinu odebrání chybějící hodnoty.

Funkce PROPAGATE-ARC-RETRACTION-AC|DC-2 zajišťuje postupnou obnovu hodnot v aktuálních doménách proměnných v minulosti zasažených odebíranou podmínkou, přičemž hlavní smyčka algoritmu je řízena frontou $Q_{RESTORE}$. Pokaždé, když je rozšířen obsah aktuální domény nějaké proměnné o určitou množinu hodnot, je zároveň vyhledána hodnota z této množiny s nejmenším časem odebrání. Byla-li do aktuální domény proměnné u navrácena množina hodnot $D_u^{restored}$ a $time_u$ je čas odebrání hodnoty, jež byla z hodnot v množině $D_u^{restored}$ v minulosti odstraněna nejdříve, je do fronty $Q_{RESTORE}$ vložena trojice $(u, time_u, D_u^{restored})$.

Dokud je fronta $Q_{RESTORE}$ neprázdná, v hlavní smyčce algoritmu AC|DC-2 je z ní pokaždé vyjmuta jedna libovolná trojice, označme ji $(u, time_u, D_u^{restored})$, a následně je uskutečněn pokus o navrácení dalších dosud chybějících hodnot do aktuálních domén proměnných sousedících s proměnnou u. Kandidáti na navrácení jsou pouze ty hodnoty, za jejichž odstranění je odpovědná proměnná u (příslušná buňka pole *justifications* ve struktuře *data* obsahuje ve složce *variable* proměnnou u) a jejichž čas odstranění je větší než $time_u$ (odpovídající buňka v poli *justifications* má složku *time* větší než $time_u$). Skutečně navrácená hodnota pak navíc musí mít ještě aspoň jednu podporu v množině $D_u^{restored}$. K hledání podpor je použita specializovaná funkce HAS-SUPPORT.

Během celého obnovovacího procesu jsou shromažďovány podmínky svazující obnovené proměnné do seznamu C_{REVISE}, který je nakonec dán jako parametr konzistenční proceduře AC-3', jež v závěru funkce RETRACT-CONSTRAINT-AC|DC-2 vyloučí případné chybně navrácené hodnoty. Výsledný problém, tj. problém s odebranou podmínkou a znovu spočtenou hranovou konzistencí, společně s příslušně upravenou strukturou *data* jsou funkcí RETRACT-CONSTRAINT-AC|DC-2 vráceny v návratové hodnotě.

Korektnost algoritmu musíme teprve ukázat, proto ji uvedeme ve formě tvrzení, které dokážeme.

Tvrzení 3.1 (KOREKTNOST ALGORITMU AC|DC-2)**.** *Algoritmus AC|DC-2 je korektní, tj. operace přidání a odebrání podmínky algoritmem AC|DC-2 zachovávají hranovou konzistenci problému.* ■

Pravdivost tvrzení pro operaci přidání podmínky přímo vyplývá z definice hranové konzistence a nevyžaduje tedy žádnou zvláštní argumentaci.

K důkazu tvrzení pro operaci odebrání podmínky postačí, když ukážeme, že algoritmus AC|DC-2 obnoví všechny hodnoty, které je třeba obnovit. Pokud navíc kromě těchto hodnot obnoví algoritmus ještě nějaké další, nevadí to, neboť na závěr operace odebrání podmínky je spuštěna konzistenční procedura AC-3', která případné navíc přidané hodnoty vyloučí. Pro potřeby důkazu však musíme tyto požadavky zformulovat přesněji.

Mějme problém $P=(V,C)$, kde $C=\{c_1,c_2,\ldots,c_k\}$, předpokládejme, že tento problém vznikl postupným přidáváním podmínek $c_1,c_2,\ldots,c_k$ pomocí operace přidání podmínky algoritmu AC|DC-2. Dále předpokládejme, že byla operací odebrání podmínky algoritmu AC|DC-2 z problému P odebrána podmínka c_i. Korektní operace odebrání podmínky uvede problém P do stejného stavu, jako kdyby byl vytvořen postupným přidáváním podmínek $c_1,c_2,\ldots,c_{(i-1)},c_{(i+1)},\ldots,c_k$ pomocí operace přidání podmínky. Pomocný problém vzniklý postupným přidáním podmínek $c_1,c_2,\ldots,c_{(i-1)},c_{(i+1)},\ldots,c_k$ označme jako Q.

Formálně řečeno tedy od operace odebrání podmínky požadujeme, aby obnovila všechny hodnoty, které chybí v aktuálních doménách proměnných u problému P, ale jsou naopak přítomny v aktuálních doménách proměnných u problému Q.

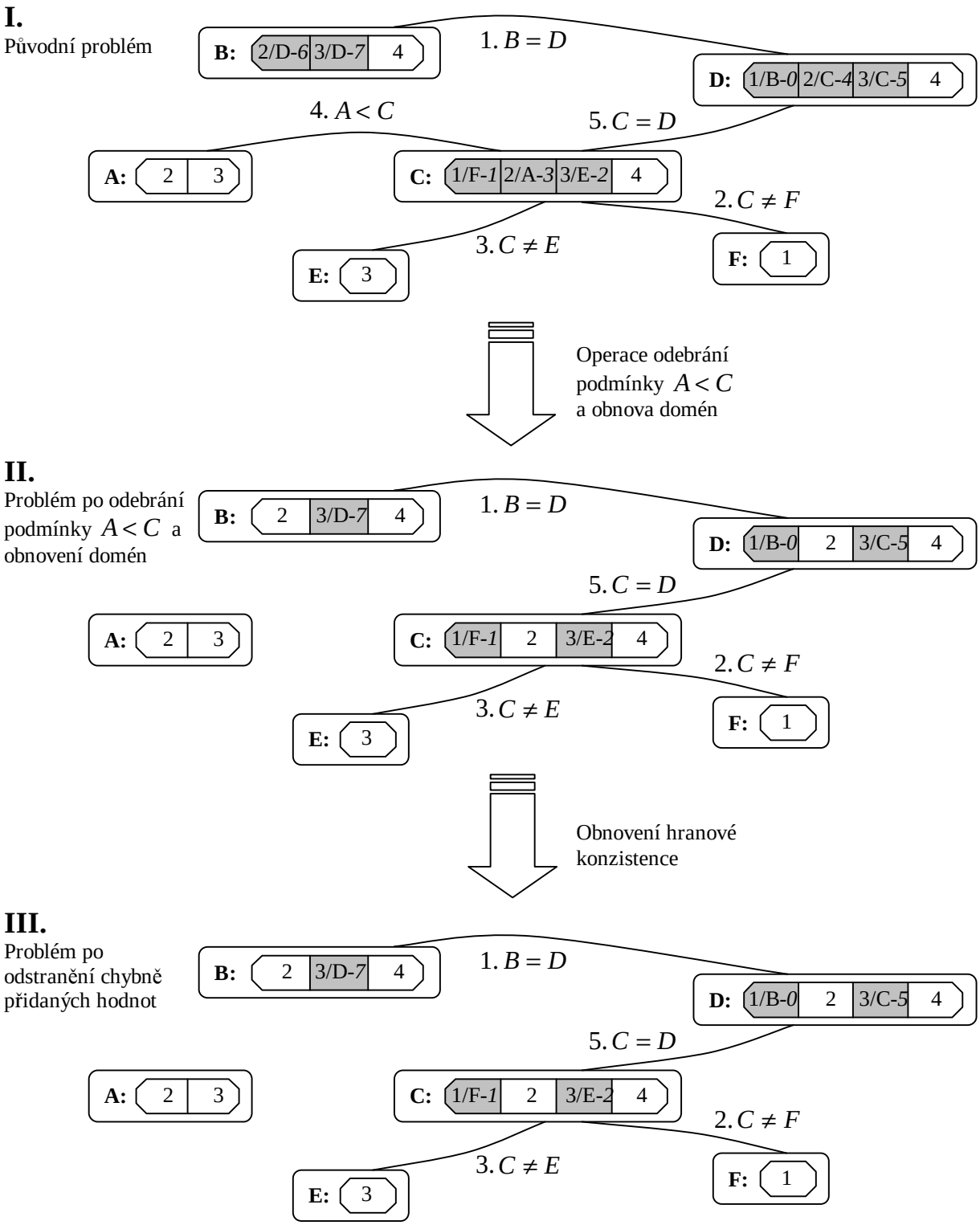
Budeme postupovat matematickou indukcí podle času odebrání jednotlivých hodnot. Nechť podmínka c_i byla do problému zařazena v čase t_0. Hodnoty, které byly odebrány z aktuálních domén proměnných svázaných podmínkou c_i od času t_0 až do okamžiku, kdy byla poprvé odebrána hodnota z aktuální domény jiné proměnné, jsou navráceny při počáteční obnově hodnot funkcí INITIALIZE-ARC-RETRACTION-AC|DC-2, což nyní dokážeme. Čas odebrání poslední z těchto hodnot označme t_0+t_1. Pro všechny hodnoty odstraněné v čase od t_0 do t_0+t_1 platí, že byly původně nekonzistentní vzhledem k podmínce c_i a obsahu aktuálních domén jejích proměnných v čase t_0 (a proto byly odstraněny). Každá z těchto hodnot má příčinu svého odstranění jednu z proměnných svázaných podmínkou c_i. Dále, aktuální domény proměnných svázaných s podmínkou c_i jsou v okamžiku volání funkce INITIALIZE-ARC-RETRACTION-AC|DC-2 vzhledem k inkluzi menší nebo rovny než tomu bylo v čase t_0. Jelikož funkce INITIALIZE-ARC-RETRACTION-AC|DC-2 navrací do aktuálních domén všechny nekonzistentní hodnoty vůči podmínce c_i a zmenšeným aktuálním doménám jejích proměnných, jsou jistě navráceny i hodnoty, jež byly v minulosti odebrány v čase od t_0 do t_0+t_1. Tím jsme ověřili počáteční krok důkazu matematickou indukcí.

Nyní učiníme indukční krok. Předpokládejme, že je třeba navrátit hodnotu d_u do aktuální domény proměnné u, u problému Q je tedy hodnota d_u v aktuální doméně proměnné u přítomna, zatímco u problému P ne. Nechť hodnota d_u byla odstraněna v čase t_2, k provedení indukčního kroku předpokládejme, že $t_2>t_0+t_1$. Z indukčního předpokladu dále platí, že všechny hodnoty, jež měly být obnoveny a jejichž čas odebrání byl menší než t_2, již

obnoveny byly. Byla-li hodnota d_u odebrána z aktuální domény proměnné u, pak nutně musela nejdřív ztratit všechny své podpory vůči nějaké jiné proměnné, řekněme, že to byla proměnná v. Všechny podpory pro hodnotu d_u musely být z aktuální domény odstraněny v čase menším než t_2. Jelikož ale hodnota d_u v aktuální doméně proměnné u je u problému Q přítomna, znamená to, že u problému Q jsou přítomny ještě také nějaké podpory pro hodnotu d_u v aktuální doméně proměnné v, které u problému P chybí. Z indukčního předpokladu vyplývá, že tyto podpory byly do aktuální domény proměnné v již navráceny a ve funkci PROPAGATE-ARC-RETRACTION-AC|DC-2 tedy také byla naplánována obnova sousedů proměnné v vzhledem k obnoveným podporám. Jakmile funkce PROPAGATE-ARC-RETRACTION-AC|DC-2 dospěje k obnově proměnné u, zjistí, že hodnota d_u splňuje všechny podmínky pro obnovu (byla odebrána kvůli proměnné v v čase t_2, který je větší než u obnovených podpor, a získala podporu v proměnné v), a navrátí ji do aktuální domény proměnné u.

Tím jsme dokázali, že během operace odebrání podmínky algoritmus skutečně obnoví všechny potřebné hodnoty, a můžeme důkaz uzavřít. ■

Průběh algoritmu AC|DC-2 je znázorněn na obrázku 3.3, příčiny a čas odebrání jednotlivých hodnot jsou vyznačeny. Jedná se o stejný problém, který byl použit pro ilustraci algoritmů AC|DC-0 a AC|DC-1. Všimněme si, že v obnovovací fázi algoritmus obnovil méně hodnot, něž tomu bylo u algoritmů AC|DC-0 a AC|DC-1.



Obrázek 4.3: Průběh odebrání podmínky algoritmem AC|DC-2

Na základě tvrzení o korektnosti a z programového zápisu nového algoritmu můžeme vyslovit následující tvrzení.

Tvrzení 3.2 (POČET OBOVENÝCH HODNOT ALGORITMEM AC|DC-2)**.** *Operace odebrání podmínky algoritmem AC|DC-2 obnoví nejvýše tolik chybějících hodnot, kolik jich obnoví operace odebrání podmínky algoritmem AC|DC-1 (AC|DC).* ■

K odůvodnění tvrzení stačí pozorování, že navracená hodnota musí u algoritmu AC|DC-2 splňovat více kritérií, než je tomu u algoritmu AC|DC-1, a tudíž má menší šanci na obnovu. Celkem tedy dostáváme uvedené tvrzení. Korektnost algoritmu AC|DC-2 samozřejmě zajišťuje, že hodnoty, které je nutné navrátit, skutečně navráceny jsou. ■

Uvedené tvrzení prozatím obhajuje nový algoritmus AC|DC-2 v tom smyslu, že jeho obnovovací fáze, a tím pádem i závěrečná opravná fáze, vykonají přinejhorším zhruba (aditivní a multiplikativní konstanty v souladu s konvencemi zanedbáváme) tolik kroků, kolik odpovídající fáze v algoritmu AC|DC-1. Neboli, existuje šance, že v průměrném případě algoritmus AC|DC-2 vykoná kroků méně. Oprávněnost této domněnky podložíme empirickými testy v následující kapitole.

Následující tabulka shrnuje časovou a prostorovou složitost v nejhorším případě algoritmu AC|DC-2.

AC|DC-2	Prostorová složitost (nejhorší případ)		Časová složitost (nejhorší případ)	
	Různé domény	Identické domény	Různé domény	Identické domény
Přidání podmínky	$O(\sum_{v\in V}\|D_v^0\|+\|C\|)$	$O(\|V\|\|D\|+\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u\|+\|D_v\|)(\|D_u\|\|D_v\|))$	$O(\|C\|\|D\|^3)$
Odebrání podmínky	$O(\sum_{v\in V}\|D_v^0\|+\|C\|)$	$O(\|V\|\|D\|+\|C\|)$	$O(\sum_{(u,v)\in C}(\|D_u^0\|+\|D_v^0\|)(\|D_u^0\|\|D_v^0\|))$	$O(\|C\|\|D\|^3)$

Tabulka 3.5: Shrnutí časové a prostorové složitosti algoritmu AC|DC-2

Oproti algoritmu AC|DC-1 zaznamenala nárůst prostorová složitost přidání podmínky o výraz $\sum_{v\in V}|D_v^0|$, to má na svědomí pole *justifications* datové sturktury *data*, které je třeba aktualizovat v průběhu používané konzistenční procedury AC-3'. Konzistenční algoritmus potřebuje dále ještě prostor o velikosti $O(|C|)$ pro uchovávání řídící fronty.

Do prostorových nároků operace odebrání podmínky je potřeba v první řadě započítat prostor požadovaný algoritmem AC-3', což je $O(\sum_{v\in V}|D_v^0|+|C|)$. Dále je potřeba uvážit prostor potřebný k reprezentaci řídící fronty, ten ale již výraz $O(\sum_{v\in V}|D_v^0|+|C|)$ nenavýší.

S teoretickou časovou složitostí v nejhorším případě jsme si oproti algoritmu AC|DC-1 nijak nepolepšili. Časová složitost přidání podmínky je opět přímo dána složitostí použité konzistenční procedury, zde AC-3'. Ohledně operace odebrání podmínky opět platí, že v obnovovací fázi je každá dvojice hodnot v doménách sousedících proměnných testována

nejvýše jedenkrát, to dává čas obnovovací fáze $O(\sum_{(u,v)\in C}|D_u^0\|D_v^0|)$, opět za předpokladu, že je použita výchozí implementace operace HAS-SUPPORT. Zde však nesmíme zapomenout, že se jedná o nejhorší čas a že již máme k dispozici tvrzení 3.2. Celkový čas operace odebrání podmínky nakonec pokazí konzistenční procedura AC-3'. To ve skutečnosti není tak závažný nedostatek, jak by se na první pohled mohlo zdát, neboť máme jednak algoritmy AC-2001 a AC-3.1, které lze též upravit tak, aby generovaly informace požadované algoritmem AC|DC-2, jako to činí procedura AC-3', a navíc chování algoritmu AC-3' není v průměrném případě zdaleka tak špatné.

Všimněme si, že v algoritmu je dovoleno, aby fronta $Q_{RESTORE}$ obsahovala více záznamů pro stejnou proměnnou. S použitím vhodné datové struktury není příliš těžké algoritmus upravit tak, aby záznamy pro stejné proměnné ve frontě $Q_{RESTORE}$ slučoval. Tímto opatřením zůstane algoritmus funkčně zcela ekvivalentní k uvedené verzi, avšak může dojít k určitému urychlení celkového času běhu. Navíc platí, že funkce HAS-SUPPORT by po této úpravě dostávala větší aktuální domény, vůči nimž se hledají podpory, což by mohlo být výhodné pro některé speciální implementace této funkce.

Na závěr k algoritmu AC|DC-2 uveďme, že na rozdíl od DnAC-4 či DnAC-6 jej lze rozšířit též pro podmínky vyšší arity než 2. Jelikož se jedná víceméně o technickou záležitost ponecháme programový zápis příslušného rozšíření do přílohy A. Uvedený rozbor časové a prostorové složitosti pochopitelně pro toto rozšíření algoritmu platit nebude.

Z důvodů lepší přehlednosti uvádíme ještě jednou složitosti jednotlivých algoritmů pro dynamickou hranovou konzistenci v jediné shrnující tabulce.

Algoritmus	**Operace**	Prostorová složitost (nejhorší případ)	Časová složitost (nejhorší případ)
DnAC-4	Přidání podmínky	$O(\|C\|\|D\|^2)$	$O(\|C\|\|D\|^2)$
	Odebrání podmínky	$O(\|C\|\|D\|^2)$	$O(\|C\|\|D\|^2)$
DnAC-6	Přidání podmínky	$O(\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
	Odebrání podmínky	$O(\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
AC|DC-0	Přidání podmínky	$O(\|C\|)$	$O(\|C\|\|D\|^3)$
	Odebrání podmínky	$O(\|V\|+\|C\|)$	$O(\|C\|\|D\|^3)$
AC|DC-1	Přidání podmínky	$O(\|C\|)$	$O(\|C\|\|D\|^3)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
AC|DC-2	Přidání podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$

Tabulka 3.6: Shrnutí časové a prostorové složitosti algoritmů pro dynamickou hranovou konzistenci

3.7 Závěr

Pro úplnost pojednání o dynamické hranové konzistenci podotkněme, že mnoho výsledků bylo získáno také v oblasti logického programování s podmínkami nad konečnými doménami - clp(FD) (*Finite Domain Constraint Logic Programming*). Aniž bychom se pouštěli do větších podrobností, uveďme, že v tomto formalizmu jsou podmínky vyjádřeny v explicitní formě určující přímo výpočet aktuální domény určité proměnné z aktuálních domén jiných proměnných, s nimiž je prostřednictvím podmínky svázána. Například klasická podmínka $X \leq Y$ je v tomto formalizmu modelována dvojicí podmínek $X \ in \ 0 \ldots \max(Y)$, $Y \ in \ \min(X) \ldots \infty$. Dynamické algoritmy pro udržování hranové konzistence v tomto formalizmu jsou uvedeny například v práci [GCR99].

Jiná otázka, která stojí u dynamických problémů za podrobnější zkoumání, je efektivní nalezení řešení dynamického problému neboli řešení každého jednotlivého statického problému tvořícího posloupnost řešeného dynamického problému. S ohledem na podobnost s udržováním konzistence se též někdy hovoří o udržování řešení vzhledem k operacím měnícím strukturu problému. Detailním zkoumáním této otázky se zabývat nebudeme.

Přesto ale uvedeme, že jednou z hlavních technik pro hledání řešení u dynamických problémů je metoda generování a zaznamenávání tzv. *nogoodů*. Stručně řečeno, nogood je podmínka, která popisuje kombinaci přiřazení hodnot proměnným, jež nikdy nemůže být součástí řešení problému. Cenné jsou pochopitelně především nogoody netriviální, tj. takové, které nejsou na první pohled vidět a k jejichž nalezení je třeba provést určité prohledávání prostoru všech možných ohodnocení. Nogoody nalezené při řešení určitého statického problému v posloupnosti dynamického problému jsou pak užívány k omezení prohledávaného prostoru u následujících statických problémů. Podrobně se těmito metodami zabývá například práce [SV94].

Bez bližšího upřesnění ještě uvedeme, že jiné metody hledající řešení u dynamických problémů aplikují dosud nalezená řešení jako heuristiky pro určení nejvýhodnějšího směru dalšího prohledávání, dalším typem metod jsou například algoritmy založené na lokálních opravách řešení předchozího statického problému.

Kapitola 4

Empirická analýza algoritmů pro dynamickou hranovou konzistenci

Dosud jsme teoreticky dokázali, že nový algoritmus AC|DC-2 provede nejvýše tolik kroků, co AC|DC-1. To však ještě neznamená, že algoritmus AC|DC-2 skutečně představuje nějaký praktický přínos, tedy, že dosahuje lepších časů při řešení hranové konzistence na reálných dynamických problémech než srovnatelný AC|DC-1. K tomu, abychom důkladněji prozkoumali chování nového algoritmu, přesněji, abychom potvrdili či vyvrátili jeho kvality, je nezbytné provést empirické testy.

V ideálním případě by bylo nejlepší otestovat nový algoritmus na co největším počtu reálných problémů, pro něž je algoritmus určen. To samozřejmě není z technických ani kapacitních důvodů možné. Proto se běžně algoritmy testují a srovnávají na určitých oblíbených testovacích problémech či na tzv. *náhodných problémech*, které se pro tento účel hodí o něco lépe. Pro testování a srovnávání algoritmu AC|DC-2 jsme zvolili náhodné problémy, neboť ty je možné oproti konkrétním testovacím problémům snadněji parametrizovat a vygenerovat tak větší množství dostatečně různorodých pokusných instancí problému.

Cílem testů bude ověřit chování algoritmu AC|DC-2 v průměrném případě a porovnat jej se srovnatelným AC|DC-1. Rovněž se budeme snažit na základě testů zjistit, za jakých okolností se případně projeví výhody nového algoritmu.

Testování algoritmu AC|DC-2 vůči algoritmům opírajících se o seznamy podpor, tj. DnAC-4 a DnAC-6 nebylo prováděno, jelikož tyto algoritmy jsou založeny na zcela odlišné myšlence a z tohoto důvodu se nehodily pro zařazení do experimentální knihovny (viz. příloha B), v jejímž rámci byly testované algoritmy implementovány. Nicméně srovnání algoritmů DnAC-4 a DnAC-6, s algoritmem AC|DC-1 lze nalézt například v článcích [NB94] a [Deb96]. Odtud vyplývá, že v průměrném případě si nejhůře vede DnAC-4, poté následuje AC|DC-1 a nejlepších výsledků dosahuje DnAC-6.

Dříve než přejdeme k samotným testům, zavedeme přesně pojem náhodného problému.

4.1 Náhodný problém

Nejběžněji je náhodný binární problém splňování podmínek zadán čtveřicí parametrů (n, d, p_c, p_s), kde n určuje počet proměnných, d udává velikost původních domén proměnných, p_c určuje pravděpodobnost přítomnosti podmínky mezi dvojicí proměnných a nakonec p_s je pravděpodobnost splnění přítomných podmínek. Podle potřeby lze pochopitelně používat rozšířené náhodné problémy zadávané dalšími dodatečnými parametry.

Význam parametrů počet proměnných a velikost původních domén netřeba zvlášť komentovat. Pravděpodobnost přítomnosti podmínky p_c lze interpretovat tak, že každá ze všech $n(n-1)/2$ možných podmínek je přítomna s pravděpodobností p_c. Hodnotu $100\,p_c$ někdy též označujeme jako hustotu podmínek a udáváme ji v procentech.

Jestliže je určitá podmínka přítomna, pak parametr p_s udává pravděpodobnost jejího splnění, jinak řečeno, podmínka je splněna pro každou z d^2 možných dvojic s pravděpodobností p_s.

Uvedené parametry dohromady určují jakousi složitost problému, tj., jak obtížný bude daný problém pro určitý algoritmus. Pochopitelně platí, že různé algoritmy jsou více citlivé na různé skupiny parametrů. V souvislosti s dynamickými problémy splňování podmínek a udržováním hranové konzistence jsou zajímavé zejména parametry udávající hustotu podmínek a pravděpodobnost splnění podmínek, což dokládají práce zabývající se touto tematikou, jako je například [VS94]. V podobném smyslu jsme navrhli empirickou analýzu i pro otestování algoritmu AC|DC-2.

Vlastní empirická analýza pomocí náhodných problémů probíhá tak, že testovaný algoritmus je vyzkoušen na množině vygenerovaných náhodných problémů s různými parametry.

Pro testování algoritmu AC|DC-2 jsme použili náhodné problémy zadanými mírně rozšířenou sadou parametrů. Problémy budeme generovat pro pětice parametrů $(n, d, p_c, p_{s1}, p_{s2})$, kde význam parametrů n, d a p_c zůstává stejný. Jediným drobným rozdílem je, že pravděpodobnost splnění přítomných podmínek je generována rovnoměrně z intervalu $[p_{s1}, p_{s2}]$.

4.2 Empirická analýza AC|DC-2

Algoritmy AC|DC-0, AC|DC-1 a AC|DC-2 jsme implementovali v jazyce C++ jako součást experimentální knihovny. Pro ověřování korektnosti implementace algoritmů AC|DC-0, AC|DC-1 a AC|DC-2 byl též implementován dynamický algoritmus založený čistě na proceduře AC-3, který počítá hranovou konzistenci po změně struktury problému pokaždé úplně od začátku. O detailech implementace testovaných algoritmů podrobněji pojednává příloha B, proto ji zde nebudeme zvlášť rozebírat. Zde jenom zdůrazněme samozřejmý fakt, a to, že skutečná implementace přesně odpovídá, co se funkčnosti týče, programovému zápisu uvedenému v kapitole 3. Užitím pojmu simulace známého z teorie algoritmů můžeme říci, že implementace simuluje programový zápis uvedený v kapitole 3.

Každý z algoritmů AC|DC-0, AC|DC-1 a AC|DC-2 jsme otestovali na třech sadách náhodných problémů.

- První sada náhodných problémů obsahovala problémy $(50,15,0.20, p_{s1}, p_{s2})$, kde $p_{s2} - p_{s1} = 0.2$, přičemž rozmezí hodnot parametrů p_{s1} a p_{s2} se pohybovalo od $[0.21,0.41]$ do $[0.48,0.68]$.

- Druhá sada náhodných problémů obsahovala problémy $(50,15,0.40, p_{s1}, p_{s2})$, kde $p_{s2} - p_{s1} = 0.2$, přičemž rozmezí hodnot parametrů p_{s1} a p_{s2} se zde pohybovalo od $[0.25,0.45]$ do $[0.48,0.68]$.

- Poslední sada náhodných problémů obsahovala problémy $(50,15,0.60, p_{s1}, p_{s2})$, kde $p_{s2} - p_{s1} = 0.2$, přičemž rozmezí hodnot parametrů p_{s1} a p_{s2} se zde pohybovalo od $[0.27,0.47]$ do $[0.48,0.68]$.

První sada problémů tedy sestává z problémů, kde je hustota podmínek poměrně řídká, zatímco problémy v poslední sadě problémů již mají velmi hustou síť podmínek.

Každý algoritmus dynamické hranové konzistence při testech pracoval tak, že nejprve pomocí operace přidání podmínky do problému (který na začátku neobsahoval žádnou podmínku) postupně zařadil všechny podmínky vygenerované podle zadaných parametrů popisujících náhodný problém. Po dokončení této fáze přišlo na řadu odebírání podmínek, kdy bylo z každé instance problému náhodným výběrem odebráno pomocí operace odebrání podmínky 10% všech přítomných podmínek. Měřena přitom byla dohromady jak fáze přidávání, tak fáze odebírání podmínek. Uvedený poměr 10% mezi počtem přidaných a odebraných podmínek byl zvolen na základě předběžných testů tak, aby se přiměřeným způsobem projevila jak fáze odebírání podmínek, tak fáze přidávání podmínek (platí, že odebírání podmínek je časově mnohem náročnější).

Rozsah parametrů p_{s1} a p_{s2} ($p_{s2} - p_{s1} = 0.2$) byl inspirován testy algoritmů uváděných v článcích týkajících se této problematiky (např. [VS94]). Abychom se vyhnuli náhodným fluktuacím, bylo od každé pětice parametrů vygenerováno 10 různých náhodných instancí problému. Výsledky pro každou pětici parametrů pak byly získány jako aritmetický průměr z výsledků celé sady 10-ti instancí náhodných problémů.

Dolní mez rozmezí hodnot parametrů p_{s1} a p_{s2} (u první skupiny problémů tedy $[0.21,0.41]$) je dána výskytem vyprázdnění aktuální domény proměnné při přidání podmínky. Implementované algoritmy pro dynamickou hranovou konzistenci v okamžiku, kdy dojde k vyprázdnění aktuální domény nějaké proměnné, nepokračují v další propagaci. V obvyklých aplikacích těchto algoritmů (je-li například dynamický algoritmus použit jako součást obecného řešícího algoritmu) značí vyprázdněná doména nemožnost dále pokračovat a následně zpravidla návrat do předchozího hranově konzistentního stavu. Z těchto důvodů jsme algoritmy implementovali a testovali jen pro hranově konzistentní stavy, neboť k ošetření hranově nekonzistentních stavů uvedeného typu lze použít jiné efektivnější techniky (například ukládání informací pro provedení zpětného kroku). Uvedené dolní meze parametrů p_{s1} a p_{s2} jsou nejnižší hodnoty, kdy ještě z větší části nedochází k vyprázdnění aktuálních domén.

Naopak u horní meze parametrů p_{s1} a p_{s2} (u první skupiny problémů tedy u $[0.48,0.68]$) již nedochází k žádnému odstraňování hodnot z aktuálních domén proměnných při přidání podmínky a dynamické algoritmy zde tak ztrácejí své opodstatnění.

Implementované algoritmy při svém běhu shromažďují řadu statistických údajů. Konkrétně se jedná o následující údaje:

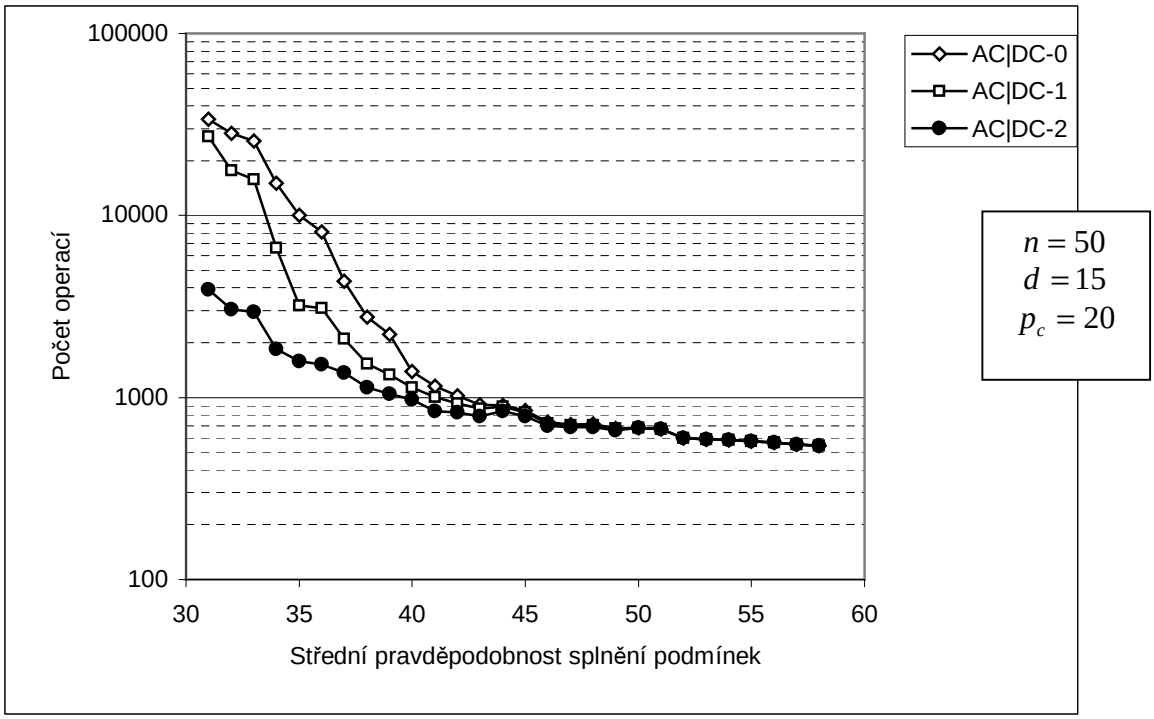
počet volání operace *přidání hodnoty* do aktuální domény
počet volání operace *odstranění hodnoty* z aktuální domény
počet *testů splnění* dané podmínky pro dvojici hodnot
počet volání operace *HAS-SUPPORT*
počet volání operace *FILTER*
volání operace *EXTEND*
celkový *čas běhu* v sekundách

Stěžejní informaci přitom představuje počet testů splnění podmínky pro dvojici hodnot. Počet těchto operací nejlépe vystihuje časovou složitost v průměrném případě nezávisle na kvalitě konkrétní implementace. Poznamenejme, že další operace, konkrétně tedy HAS-SUPPORT, FILTER a EXTEND, jsou ve své výchozí implementaci, která ostatně byla použita pro tuto empirickou analýzu, realizovány právě pomocí testů podmínky pro dvojice hodnot.

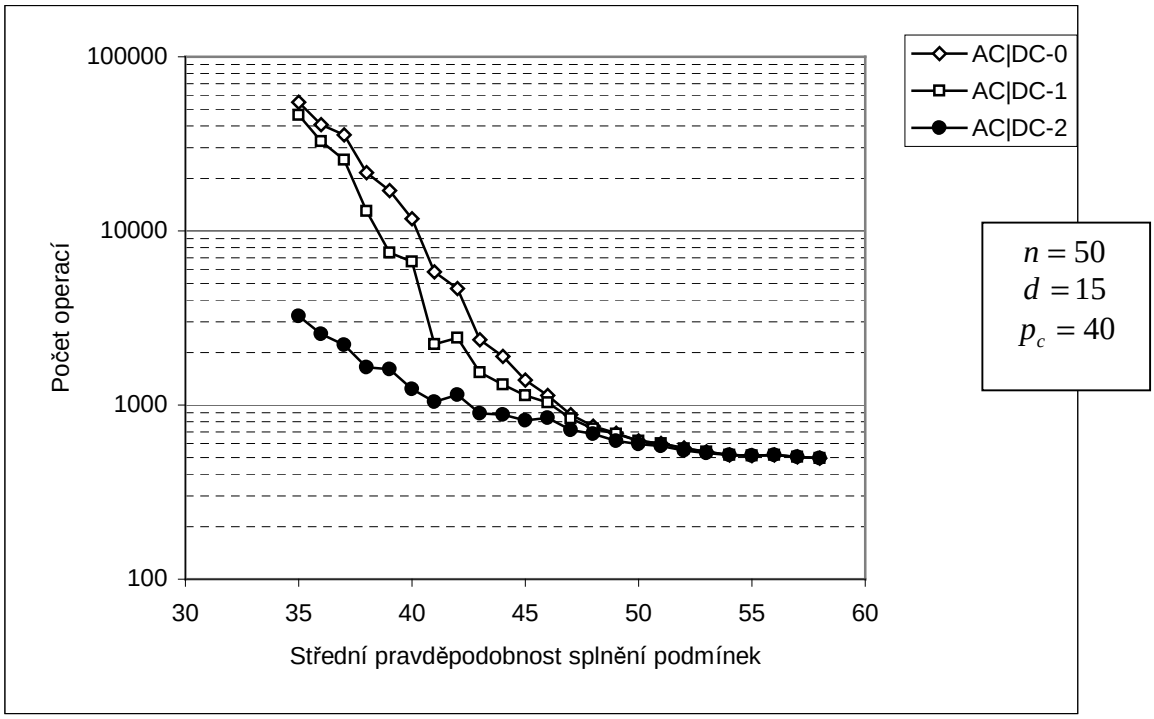
Výsledkem provedených testů jsou uvedené statistické údaje přepočítané na jednu odebíranou podmínku pro všechny výše specifikované náhodné problémy. Přesněji řečeno, jedno měření zhruba odpovídá přidání deseti podmínek a odebrání jedné podmínky pomocí operací poskytovaných příslušným dynamickým algoritmem.

V této kapitole uvedeme grafy závislosti počtu testů podmínek pro dvojici hodnot a celkového času běhu na střední hodnotě pravděpodobnosti splnění podmínky, jež se rovná hodnotě $(p_{s1} + p_{s2})/2$. Ostatní statistické výsledky přenecháme do přílohy C. Kompletní soubory dat získaných během měření, tj. dat, z nichž byly počítány výše zmiňované aritmetické průměry, jsou uloženy na přiloženém CD.

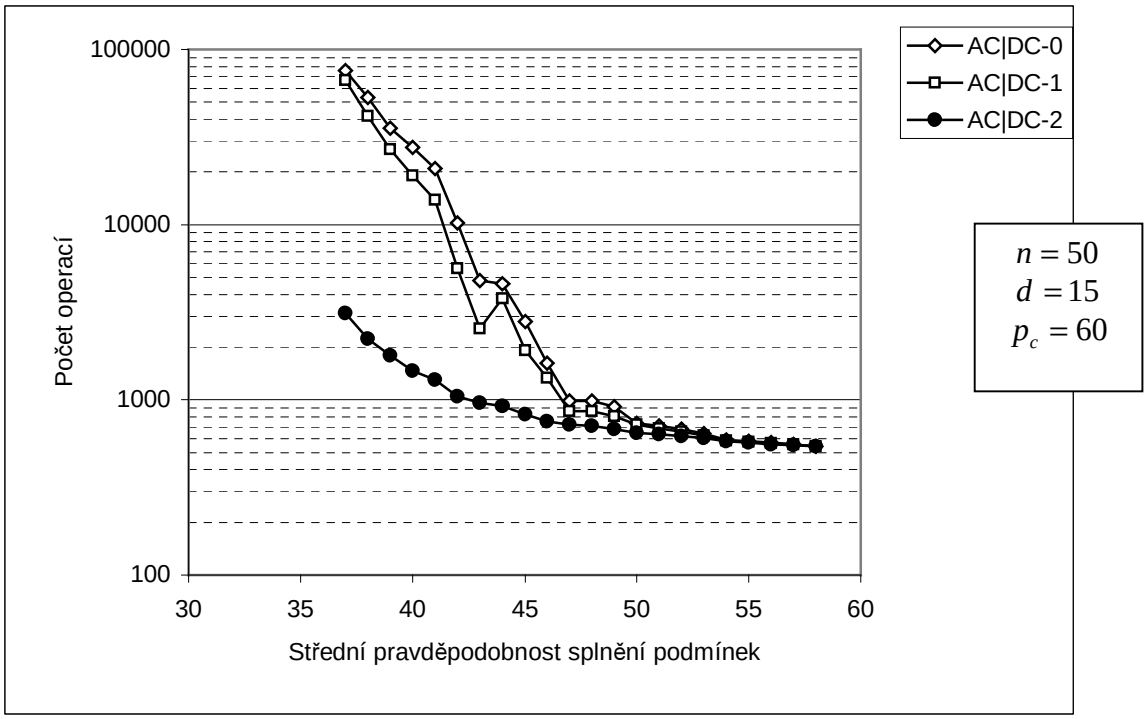
Každý z grafů 4.1 až 4.3 vždy srovnává počet testů podmínek u algoritmů AC|DC-0, AC|DC-1 a AC|DC-2 vzhledem ke měnícím se parametrům náhodného problému (střední pravděpodobnosti splnění podmínek). Osa znázorňující počet testů podmínek používá logaritmickou stupnici.



Graf 4.1: Závislost počtu testů podmínky na střední pravděpodobnosti splnění podmínek pro hustotu podmínek 20%



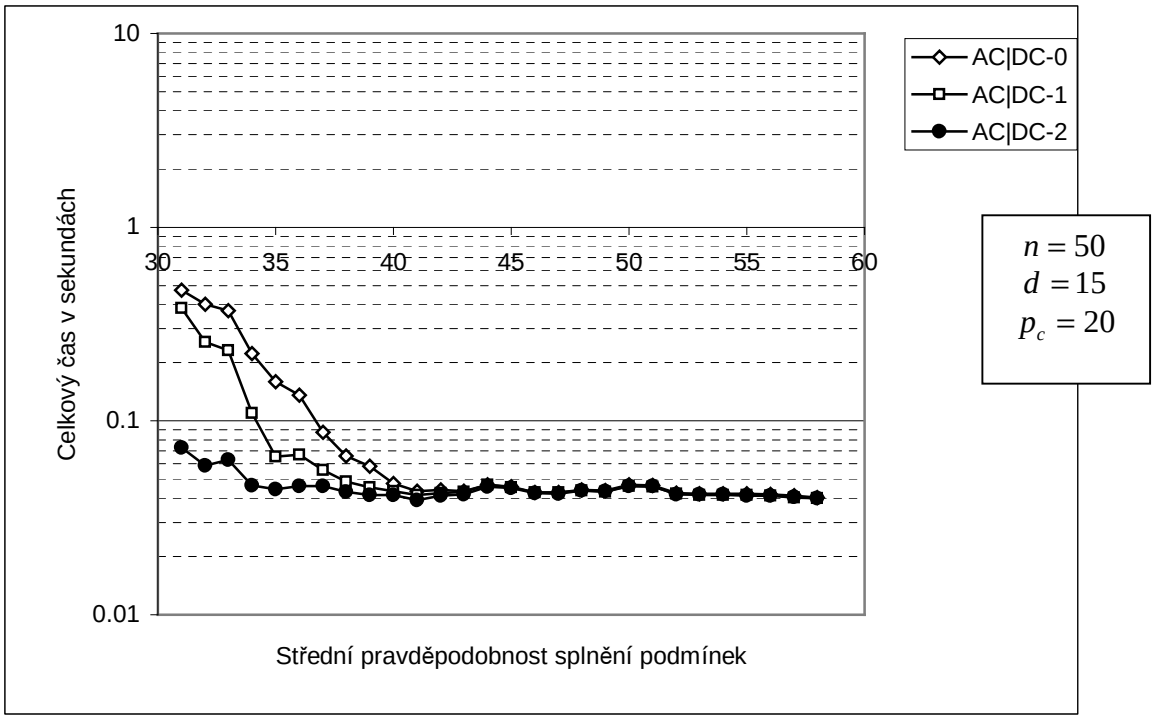
Graf 4.2: Závislost počtu testů podmínky na střední pravděpodobnosti splnění podmínek pro hustotu podmínek 40%



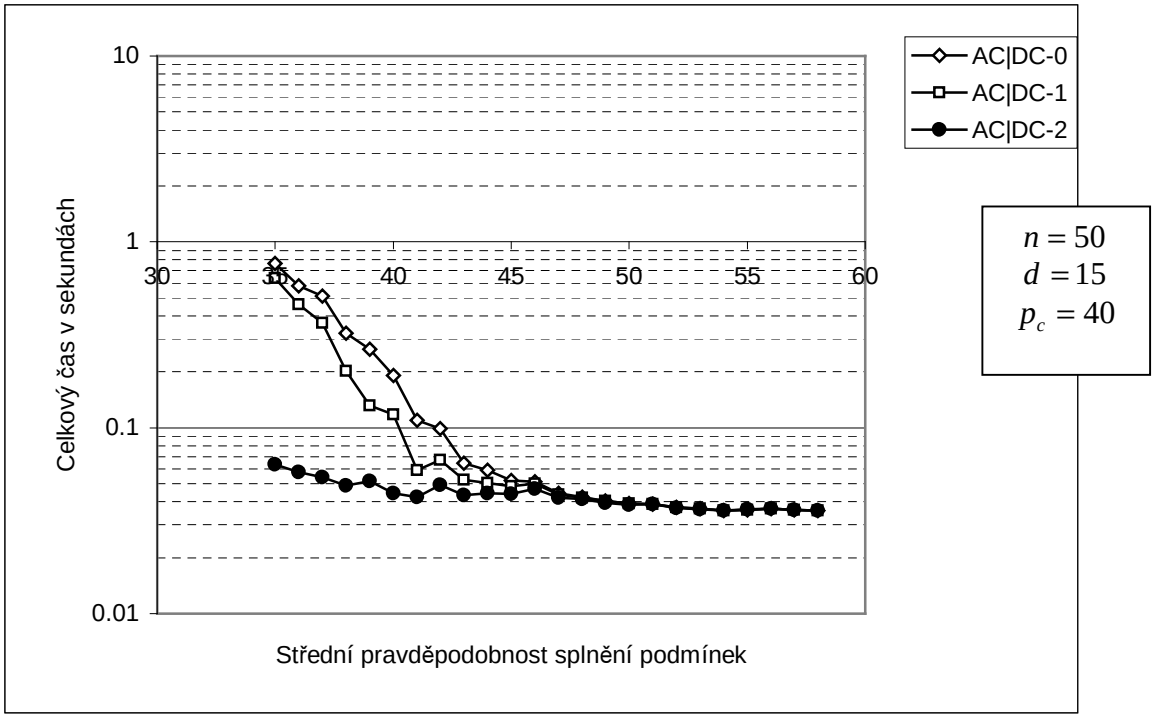
Graf 4.3: Závislost počtu testů podmínky na střední pravděpodobnosti splnění podmínek pro hustotu podmínek 60%

Grafy 4.4 až 4.6 ukazují, že nový algoritmus přináší též výraznou úsporu celkového času běhu[1]. Každý graf opět srovnává všechny tři testované algoritmy, stejně jako je tomu v grafech pro počet testů podmínek. Osa znázorňující celkový čas běhu používá opět logaritmickou stupnici.

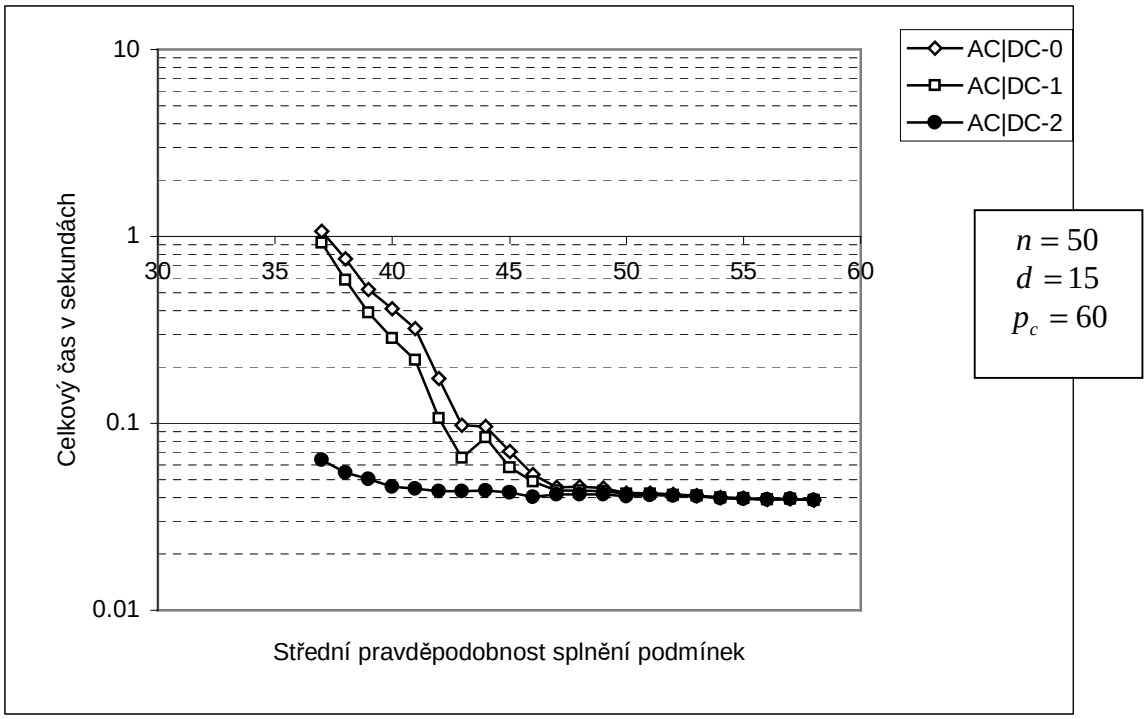
[1] Měření probíhala na počítači vybaveném procesorem Pentium II 233 Mhz, 128 MB operační paměti, pod operačním systémem Red Hat Linux 6.2. Testovací programy byly přeloženy kompilátorem egcs-2.91.66 se stupněm optimalizace O2.



Graf 4.4: Závislost celkového času běhu na střední pravděpodobnosti splnění podmínek pro hustotu podmínek 20%



Graf 4.5: Závislost celkového času běhu na střední pravděpodobnosti splnění podmínek pro hustotu podmínek 40%



Graf 4.6: Závislost celkového času běhu na střední pravděpodobnosti splnění podmínek pro hustotu podmínek 60%

4.3 Zhodnocení výsledků

Z uvedených grafů je vidět, že oproti algoritmu AC|DC-1 přináší AC|DC-2 výraznou úsporu počtu testů podmínek. Odpovídajícím způsobem se příznivější počet testů podmínek na splnění odrazí i na celkovém čase běhu algoritmu, a to i přesto, že experimentální implementace nebyla nijak speciálně optimalizována.

Převaha algoritmu AC|DC-2 se projevuje zejména, pokud je třeba do aktuálních domén zasažených proměnných navrátit větší množství chybějících hodnot. Za těchto okolností algoritmus AC|DC-1 evidentně navrací i značné množství hodnot, které ve skutečnosti navráceny být nemají. Jak již bylo v předchozích kapitolách zmíněno, chybně navrácená hodnota může vyvolat až řetěz dalších chybně navrácených hodnot, které je pak potřeba stejně vyloučit. Provedené testy ukazují, že snaha o maximální eliminaci tohoto jevu u se algoritmu AC|DC-2 skutečně vyplácí (neboli naopak, že algoritmus AC|DC-1 na tento jev výrazně doplácí).

Z testů je dále vidět, že algoritmus AC|DC-2 vykazuje výrazně lepší výsledky v celé pro uživatele zajímavé škále složitosti problému, tj. tam, kde dochází k netriviálním obnovám aktuálních domén. Přičemž platí, že čím je obnova hodnot složitější, tím více se projeví přínos algoritmu AC|DC-2 oproti AC|DC-1. Na druhou stranu v situaci, kdy je obnova jednoduchá, tedy, když se obnovuje malé množství hodnot, jsou výkony všech tří testovaných algoritmů srovnatelné. Tento fakt lze interpretovat tak, že při menším množství obnovovaných hodnot je

menší pravděpodobnost, že algoritmus AC|DC-1 (a tedy i AC|DC-0) provede řetěz chybných obnov a obnova se tak více blíží ideálnímu průběhu.

Na základě uvedených grafů můžeme též vyslovit tvrzení, že počet provedených testů podmínek velmi dobře odpovídá celkovému času běhu, což v důsledku znamená, že ušetření počtu testů podmínek není u algoritmu AC|DC-2 zaplaceno výraznějším navýšením jiné práce.

Jako domněnku založenou na empirických testech srovnávající algoritmy DnAC-4, DnAC-6 a AC|DC-1 uveřejněných v článcích [NB94] a [Deb96] uveďme, že algoritmus AC|DC-2 se výkonnostně nachází mezi algoritmy AC|DC-1 a DnAC-6, pravděpodobně blíže algoritmu DnAC-6. Jako podklad k této domněnce slouží skutečnost, že počet obnovovaných hodnot je u algoritmu AC|DC-2 již velmi nízký a lze se domnívat, že algoritmus DnAC-6 již nedosáhne dalšího výrazného snížení tohoto počtu.

Nakonec poznamenejme, že náhodné problémy sice neodrážejí přesné chování algoritmů na reálně řešených problémech, nicméně poskytují k tomu poskytují poměrně slušné vodítko.

Program pro testování algoritmů AC|DC-0, AC|DC-1 a AC|DC-2 na náhodných problémech se nachází na přiloženém CD.

Závěr a možnosti dalšího vývoje

V práci jsme podali přehled nejdůležitějších existujících algoritmů řešících dynamickou hranovou konzistenci. Algoritmy jsou uváděny v kompletním znění, je tedy uveden celý programový zápis s příslušným komentářem. U každého algoritmu je vysvětlena jeho základní idea, resp. její realizace ve vlastním programovém zápisu. Pojednání o každém jednotlivém algoritmu je vždy uzavřeno konstatováním jeho složitosti a větším či menším náznakem důkazu tohoto výsledku.

Speciální zájem je věnován algoritmům pro udržování dynamické hranové konzistence bez uchovávání seznamů podpor. Algoritmy tohoto typu jsou užitečné zejména pro řešení velmi rozsáhlých problémů. Existující algoritmus AC|DC ale například v porovnání s DnAC-6 značně zaostává, jak dokládá práce [Deb96]. Efektivita algoritmu DnAC-6 je ale na druhou stranu vykoupena na úkor prostorové složitosti, neboť algoritmus musí uchovávat rozsáhlé seznamy podpor.

Na základě poznatků z existujících algoritmů jsme proto navrhli zcela nový algoritmus AC|DC-2 pro řešení dynamické hranové konzistence, který se obejde bez udržování seznamů podpor a řeší tak problém algoritmů DnAC-4 a DnAC-6 a který zároveň nabízí oproti AC|DC výrazně vyšší úsporu počtu operací i celkového času, což dokládají dokázaná tvrzení a provedená empirická analýza.

Nový algoritmus je v práci popsán pomocí symbolického programového zápisu. Následují nezbytná tvrzení o korektnosti navrženého přístupu a analýza prostorové a časové složitosti. V práci je teoreticky dokázáno, že nový algoritmus přinejhorším provede nejvýše tolik kroků, co existující AC|DC.

Přínos nového algoritmu je rovněž ukázán v možnostech jeho dalších rozšíření, z nichž konkrétně rozšíření pro podmínky vyšší arity než 2 formulováno jako program v uvedený příloze A.

Algoritmus AC|DC-2 byl za účelem ověření a získání výsledků o chování v průměrném případě implementován jako součást experimentální knihovny `SPlan` pro práci s omezujícími podmínkami v jazyce C++. Knihovna `SPlan` je přiměřeným způsobem stručně popsána v příloze B.

Dalším vývoj může směřovat například k dynamizaci složitějších konzistenčních technik, jako je třeba konzistence po cestě či k-konzistence, resp. silná k-konzistence. Rovněž je možné pokračovat ve vývoji dalších ještě efektivnějších algoritmů pro dynamickou hranovou konzistenci.

Nesmíme ale opomenout, že hlavním smyslem dynamických konzistenčních algoritmů je jejich spolupráce s obecnými řešícími algoritmy. Proto se nabízí pokračovat ve vývoji integrace těchto algoritmů do existujících řešících algoritmů. O této problematice pojednávají například články [JB97] a [JDB00]. Prozatím však v této oblasti mnoho výsledků není a zdá se tedy být vhodná pro další výzkum.

Nakonec podotkněme, že návrhu nových efektivních konzistenčních a řešících algoritmů pro dynamické problémy by nepochybně pomohly také kvalitní a hlavně dostupné vizualizační nástroje. Podobný software dosud není k dispozici a z širšího hlediska lze i zde určitou měrou přispět k výzkumu v oblasti dynamických problémů.

Příloha A

Algoritmus AC|DC-2 pro nebinární podmínky

Tato příloha je věnována slibované verzi algoritmu AC|DC-2 pro podmínky vyšší arity než 2. Programový zápis verze algoritmu konzistenční procedury AC-3' pro nebinární podmínky představuje algoritmus A.1. Programový zápis algoritmu AC|DC-2 ukazuje algoritmus A.2. Připomeňme, že oba algoritmy opět předpokládají přítomnost nejvýše jedné podmínky na každou k-tici proměnných.

V následujícím odstavci pouze stručně popíšeme změny, jež odlišují nebinární verzi od verze uvedené v kapitole 3. Pro podrobný popis funkce celého algoritmu odkazujeme uvedenou kapitolu.

V případě nebinárních podmínek platí, že jestliže má být určitá hodnota odstraněna z aktuální domény proměnné, pak je tomu tak proto, že tato hodnota ztratila všechny své podpory v $(k-1)$-tici proměnných sousedících s danou proměnnou prostřednictvím k-ární podmínky, vzhledem k níž jsou podpory počítány. V tomto smyslu je tedy potřeba příslušně upravit původní položku *variable* v buňkách pole *justifications*, neboť ta nyní musí udržovat celé zmiňované $(k-1)$-tice, jelikož příčinami odebrání hodnoty může být více proměnných najednou. S ohledem na fakt, že původní položka *variable* má nyní obsahovat množinu proměnných, budeme ji zde označovat jako *variables*. Čas odebrání hodnot z aktuální domény proměnné, tj. položka *time* v buňkách pole *justifications*, zůstává beze změny.

Algoritmus A.1. AC-3'

PROPAGATE-AC-3' (P , *data*, C_{REVISE})
1 $Q_{REVISE} \leftarrow C_{REVISE}$
2 **while** $Q_{REVISE} \neq \emptyset$ **do**
3 $c \in Q_{REVISE}$ libovolná podmínka
4 $Q_{REVISE} \leftarrow Q_{REVISE} - \{c\}$
5 $\{v_1, v_2, \ldots, v_n\} \leftarrow V_c$
6 $(D_{v1}^{remove}, D_{v2}^{remove}, \ldots, D_{vn}^{remove}) \leftarrow$
7 $\leftarrow$ FILTER(c , $P.D_{v1}$, $P.D_{v2}$, … , $P.D_{vn}$)
8 **for each** $i \in \{1,2,\ldots n\}$ **do**
9 (P , *data*, Q_{REVISE}^{vi}) $\leftarrow$
10 $\leftarrow$ FILTER-ARC-AC-3'(P , *data*, v_i , D_{vi}^{remove} ,
11 $\{v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_n\}$)
12 $Q_{REVISE} \leftarrow Q_{REVISE} \cup Q_{REVISE}^{vi}$
13 **return** (P , *data*)

FILTER-ARC-AC-3' (P , *data*, v_i , D_{vi}^{remove} , *variables*)
1 $Q_{REVISE} \leftarrow \emptyset$
2 nechť $v_1, v_2, \ldots, v_n$ jsou svázány podmínkou c
3 **if** $D_{vi}^{remove} \neq \emptyset$ **then**
4 **for each** $d_{vi} \in D_{vi}^{remove}$ **do**
5 $P.D_{vi} \leftarrow P.D_{vi} - \{d_{vi}\}$
6 $data.justifications[v_i, d_{vi}].variables = variables$
7 $data.justifications[v_i, d_{vi}].time = data.time$
8 $data.time \leftarrow data.time + 1$
9 $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{c' \in P.C \mid v_i \in V_{c'} \,\&\, c' \neq c\}$
10 return (P , *data*, Q_{REVISE})

Algoritmus A.2. AC|DC-2

ADD-CONSTRAINT-AC|DC-2 (P , *data*, c)
1 $P.C \leftarrow P.C \cup \{c\}$
2 $P \leftarrow$ PROPAGATE-AC-3'(P , *data*, $\{c\}$)
3 **return** (P , *data*)

RETRACT-CONSTRAINT-AC|DC-2 (P , *data*, c)

1 $Q_{RESTORE} \leftarrow \emptyset$
2 $\{v_1, v_2, \ldots, v_n\} \leftarrow V_c$
3 **for each** $i \in \{1,2,\ldots n\}$ **do**
4 $(time_{vi}, D_{vi}^{restore}) \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2(P , c , v_i ,
5 $\{v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_n\}$)
6 **if** $D_{vi}^{restore} \neq \emptyset$ **then**
7 $P.D_{vi} \leftarrow P.D_{vi} \cup D_{vi}^{restore}$
8 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v_i, time_{vi}, D_{vi}^{restore})\}$
9 $P.C \leftarrow P.C - \{c\}$
10 $(P, C_{REVISE}) \leftarrow$ PROPAGATE-ARC-RETRACTION-AC|DC-2(P , c , u , v)
11 $P \leftarrow$ PROPAGATE-AC-3' (P , C_{REVISE})
12 **return** (P , *data*)

INITIALIZE-ARC-RETRACTION-AC|DC-2 (P , c , v_i , *variables*)

1 $\{v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_n\} \leftarrow$ *variables*
2 $time_{vi} \leftarrow \infty$
3 $D_{vi}^{restore} \leftarrow \emptyset$
4 **for each** $d_{vi} \in (P.D_{vi}^0 - P.D_{vi})$ **do**
5 **if** $data.justifications[v_i, d_{vi}].variables = variables$ **then**
6 **if not** HAS-SUPPORT(c , v_i , d_{vi} , v_1 , $P.D_{v1}$, v_2 , $P.D_{v2}$, …
7 …, $v_{(i-1)}$, $P.D_{v(i-1)}$, $v_{(i+1)}$, $P.D_{v(i+1)}$, …, v_n , $P.D_{vn}$) **then**
8 $D_{vi}^{restore} \leftarrow D_{vi}^{restore} \cup \{d_{vi}\}$
9 $time_{vi} \leftarrow \min(time_{vi}, data.justifications[v_i, d_{vi}].time)$
10 $data.justifications[v_i, d_{vi}] \leftarrow NIL$
11 **return** ($time_{vi}$, $D_{vi}^{restore}$)

PROPAGATE-ARC-RETRACTION-AC|DC-2 (P , *data*, $Q_{RESTORE}$)

1 $C_{REVISE} \leftarrow \emptyset$
2 **while** $Q_{RESTORE} \neq \emptyset$ **do**
3 $(v_i, time_{vi}, D_{vi}^{restored}) \in Q_{RESTORE}$ libovolná proměnná
4 $Q_{RESTORE} \leftarrow Q_{RESTORE} - \{(v_i, time_{vi}, D_{vi}^{restored})\}$
5 **for each** $c \in P.C$ takovou, že $v_i \in V_c$ **do**
6 $\{v_1, v_2, \ldots, v_n\} \leftarrow V_c$
7 **for each** $j \in \{1,2,\ldots,i-1,i+1,\ldots,n\}$ **do**
8 $time_{vj} \leftarrow \infty$
9 $D_{vj}^{restore} \leftarrow \emptyset$
10 **for each** $d_{vj} \in (P.D_{vj}^0 - P.D_{vj})$ **do**
11 **if** $v_i \in data.justifications[v_j, d_{vj}].variables$ **and**
12 $data.justifications[v_j, d_{vj}].time > time_{vi}$ **then**
13 **if** HAS-SUPPORT($c, v_i, d_{vi}, v_1, P.D_{v1},$
14 $v_2, P.D_{v2}, \ldots, v_{(i-1)}, P.D_{v(i-1)}, v_i, D_{vi}^{restored},$
15 $v_{(i+1)}, P.D_{v(i+1)}, \ldots, v_n, P.D_{vn}$) **then**
16 $D_{vj}^{restore} \leftarrow D_{vj}^{restore} \cup \{d_{vj}\}$
17 $time_{vj} \leftarrow \min(time_{vj},$
18 $data.justifications[v_j, d_{vj}].time$)
19 $data.justifications[v_j, d_{vj}] \leftarrow NIL$
20 **for each** $j \in \{1,2,\ldots,i-1,i+1,\ldots,n\}$ **do**
21 **if** $D_{vj}^{restore} \neq \emptyset$ **then**
22 $P.D_v \leftarrow P.D_{vj} \cup D_{vj}^{restore}$
23 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v_j, time_{vj}, D_{vj}^{restore})\}$
24 $C_{REVISE} \leftarrow C_{REVISE} \cup \{c \in P.C \mid u \in V_c\}$
25 **return** (P , *data*, C_{REVISE})

Rozbor časové a paměťové složitosti uvedený v kapitole 3 nelze bez patřičného zobecnění přenést na tuto verzi algoritmu pro nebinární podmínky. Nicméně lze při tomto rozboru postupovat naprosto analogicky, jako jsme to činili v kapitole 3.

Závěrem podotkněme, že velmi podobným způsobem, přesněji řečeno, o něco jednodušeji, lze upravit za účelem zpracování podmínek vyšší arity též algoritmy AC|DC-0 a AC|DC-1. Algoritmy DnAC-4 a DnAC-6 však tímto způsobem upravit nelze.

Příloha B

Implementace experimentální knihovny

K ověřování teoretických hypotéz a k různým praktickým testům byla implementována vlastní knihovna pro práci s omezujícími podmínkami. Experimentální knihovnu jsme zdánlivě nelogicky nazvali `SPlan`[1], knihovna se s veškerými svými zdrojovými texty nachází na přiloženém CD. Vzhledem k tomu, že mezi dostupnými knihovnami prakticky chybí univerzální a dostatečně robustní knihovna, ve které by se daly přiměřeným způsobem implementovat studované otázky týkající se dynamických problému, bylo přistoupeno k implementaci knihovny vlastní. Sekundární motivací k tomuto kroku byl také fakt, že přizpůsobování některé z existujících knihoven by nakonec mohlo být technicky náročnější než napsání vlastní knihovny, která bude od základu počítat s aplikací, pro niž je určena.

Knihovna `SPlan` byla implementována v jazyce C++, základní literaturu k tomuto jazyku je publikace [Str86]. Při implementaci byla významným způsobem též uplatněna knihovna STL (Standard Template Library), pomocí níž jsou realizovány veškeré datové struktury pro uchovávání dat hromadného charakteru. Vhodným informačním zdrojem ke knihovně STL je například elektronická dokumentace [SGI01]. Jako systémové prostředí pro experimentální testy byl zvolen operační systém Linux. Překlad knihovny je na Linuxu, resp. na unixových systémech zajišťován nástroji pro automatickou kompilaci (automake, autoconf, …).

Zdůrazněme, že cílem implementace knihovny `SPlan` nebyla ona samotná, nýbrž ověření teoretických výsledků empirickými testy. Tudíž od knihovny nelze očekávat všechny náležitosti, jimiž se má vyznačovat softwarové dílo (nicméně knihovna byla od začátku koncipována jako šířeji použitelné softwarové dílo a při implementaci byl brán zřetel i na možnou přenositelnost).

V této příloze se soustředíme hlavně na významné a zajímavé vlastnosti knihovny `SPlan`, které vyložíme v kontextu objektového modelu knihovny.

B.1 Objektová struktura knihovny

Objekty jazyka C++ vyskytující se v implementaci knihovny by bylo možné rozdělit zhruba do třech kategorií. První kategorie objektů obsahuje vše, co nějak souvisí s reprezentací problémů splňování podmínek, sem tedy patří mimo jiné hodnoty, domény, podmínky apod.

[1] Cílem knihovny je časem dospět do podoby plánovacího systému, jehož výkonné jádro bude tvořeno algoritmy pro splňování podmínek, zejména pak v této práci prezentovanými algoritmy pro dynamickou hranovou konzistenci. Tento cíl byl částečně inspirován existencí systému CPlan, o němž je zmínka v kapitole 3.

Do druhé kategorie by spadaly všechny objekty, které operují nad reprezentací problému či jejími částmi, takové objekty v rámci knihovny označujeme jako služby. Sem lze zařadit například propagační algoritmy, heuristiky určující pořadí proměnných, řešící algoritmy atd.

Do poslední kategorie patří všechny ostatní objekty, jež nespadají ani do první ani do druhé kategorie objektů. Příkladem takového objektu je třeba objekt zajišťující šíření událostí.

Do této stručné kategorizace nezahrnujeme části knihovny a objekty, které mají na starosti různé systémové záležitosti. Tedy například kód realizující výstup, analýzu příkazové řádky atp. Rovněž sem nezahrnujeme ani vlastní testovací programy, pomocí nichž byla prováděna empirická analýza studovaných algoritmů.

B.2 Reprezentace problému splňování podmínek

V této sekci stručně popíšeme objekty realizující reprezentaci problémů splňování podmínek. Obrázek B.1 charakterizuje hierarchii konkrétních objektů (tříd) jazyka C++, jež zajišťují tuto reprezentaci.

Obyčejné šipky na obrázku znázorňují dědičnost mezi objekty, resp. specializace objektů, šipky začínající zvýrazněným bodem znázorňují prvky objektů. Všimněme si, že všechny objekty mající něco společného s reprezentací problému jsou potomky objektu Element. Objekt Element slouží k uchovávání určitých společných vlastností, například umožňuje vložit do odvozeného objektu různé pomocné informace. Dále objekt EventHandler slouží k odchytávání událostí různého typu generovaných jinými objekty. Jestliže je nějaký objekt potomkem objektu EventHandler, je pak schopen reagovat na výskyty událostí. Příkladem takové generované události může být třeba změna aktuální domény proměnné.

Hodnoty v doménách proměnných jsou realizovány objektem Value, objekt Value představuje jakýsi abstraktní prvek, jehož specializací lze získat prvky domén libovolného typu (celá čísla, slova atd.). Konkrétní specializaci objektu Value pro celá čísla představuje odvozená třída IntegerValue. Požadavkem na objekty typu Value je, že mezi sebou musí být vzájemně porovnatelné a tato relace musí tvořit uspořádání, což je důležité pro efektivní reprezentaci domény proměnné.

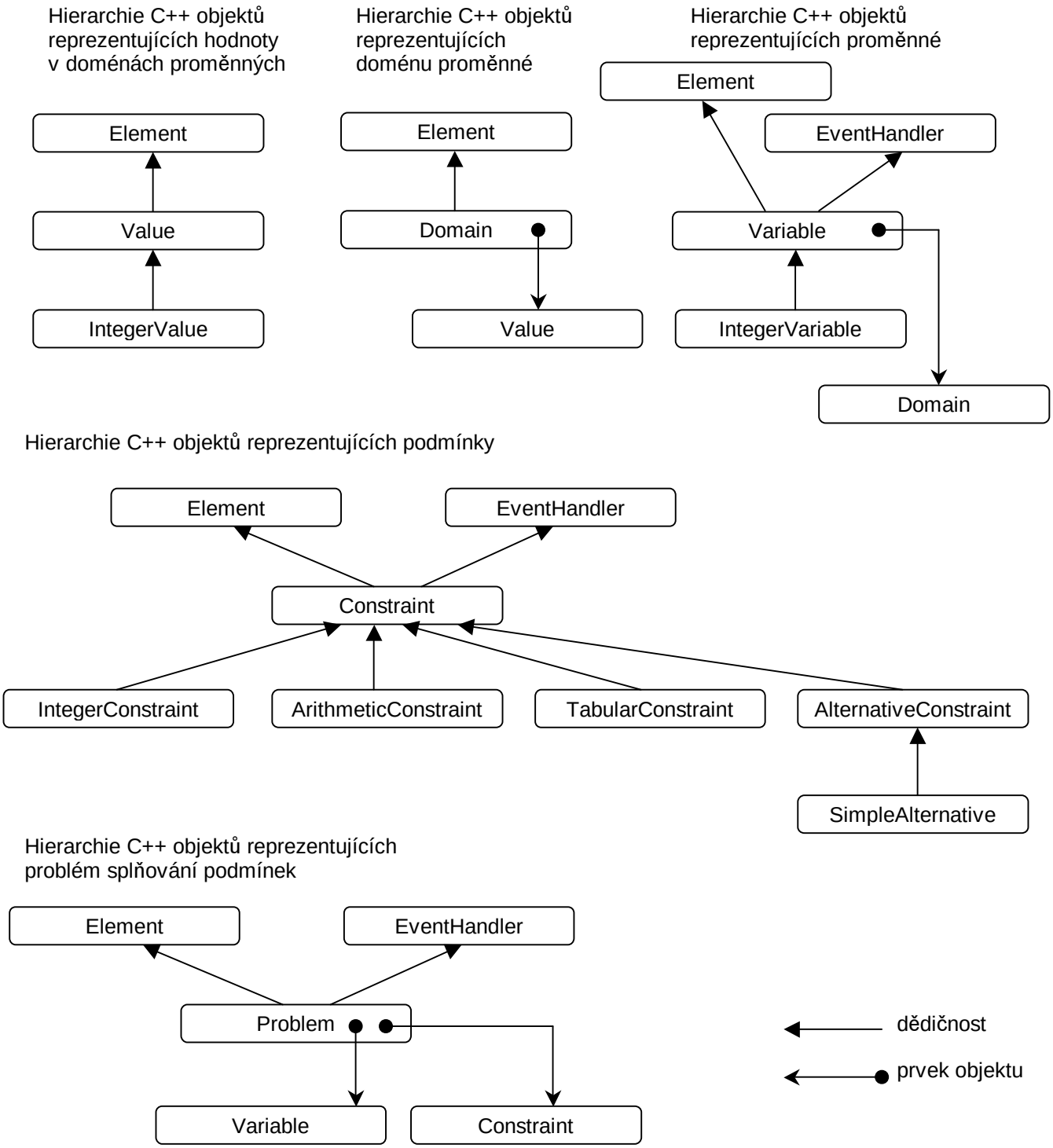
Dalším důležitým objektem je třída Domain reprezentující doménu proměnné. Prvky objektu Domain jsou tři uspořádané seznamy proměnných reprezentující původní a aktuální doménu a seznam hodnot momentálně nepřítomných v aktuální doméně. Objekt Domain poskytuje sadu operací nad doménou proměnné, jmenujme například operace přidání či odstranění hodnoty z aktuální domény. Významnou vlastností je, že tyto operace automaticky aktualizují všechny tři uvedené seznamy.

Reprezentaci proměnných má na starosti objekt odvozený od základové třídy Variable. Objekt Variable poskytuje operace přidání a odebrání hodnoty z aktuální domény proměnné, přičemž eventuálně může provádět i záznam informací pro případné provedení operace undo, tj. návrat do některého z předchozích stavů. Tato vlastnost je užitečná hlavně v souvislosti s řešícími algoritmy, které často potřebují při nemožnosti pokračovat dál provést návrat do některého z předchozích stavů. Specializaci objektu Variable realizující celočíselné proměnné představuje třída IntegerVariable.

Omezující podmínka je vždy odvozena od základového objektu Constraint. Knihovna `SPlan` pracuje v aktuální verzi pouze s unárními a binárními podmínkami. Každá podmínka musí poskytovat sadu operací probíraných v kapitole 2, tj. test, zda je splněna danou pro

hodnotu či dvojici hodnot, operaci HAS-SUPPORT, operaci FILTER a operaci EXTEND. Specializovaná podmínka může poskytovat efektivní verze všech těchto operací (založených například na intervalových výpočtech a monotonii). Nicméně povinný je pouze test na splnění pro hodnotu, resp. dvojici hodnot. Jestliže konkrétní podmínka neposkytne další operace jsou použity výchozí implementace, jež zajišťuje sama základová třída Constraint. Objekt Constraint může rovněž, pokud to uživatel požaduje, zaznamenávat informace pro návrat do předchozích stavů (což se zde týká stavů svazovaných proměnných).

Objekt IntegerConstraint realizuje rovnosti a nerovnosti mezi dvojicemi proměnných. Tato třída poskytuje specializované efektivní verze operací HAS-SUPPORT, FILTER a EXTEND založené na intervalových výpočtech. Podmínka implementovaná objektem ArithmeticConstraint realizuje lineární aritmetické relace mezi dvojicemi proměnných, nicméně na rozdíl od IntegerConstraint využívá pouze výchozích implementací operací HAS-SUPPORT, FILTER a EXTEND. Podmínka realizovaná třídou TabularConstraint představuje podmínku reprezentovanou výčtem všech kompatibilních dvojic. Pro generování náhodných podmínek v náhodných problémech jsou využívány podmínky právě typu TabularConstraint. Nakonec podmínka AlternativeConstraint slouží k ohodnocování proměnných pomocí podmínek, o čemž bude ještě zmínka. Specializace této podmínky reprezentovaná objektem SimpleAlternative pak realizuje ohodnocující podmínky založené na rovnostech a nerovnostech. Důležitou vlastností těchto podmínek je lepší podpora pro jejich negaci, což je dáno potřebou procházet při prohledávání prostoru všech možných ohodnocení sadu vzájemně se vylučujících alternativ.

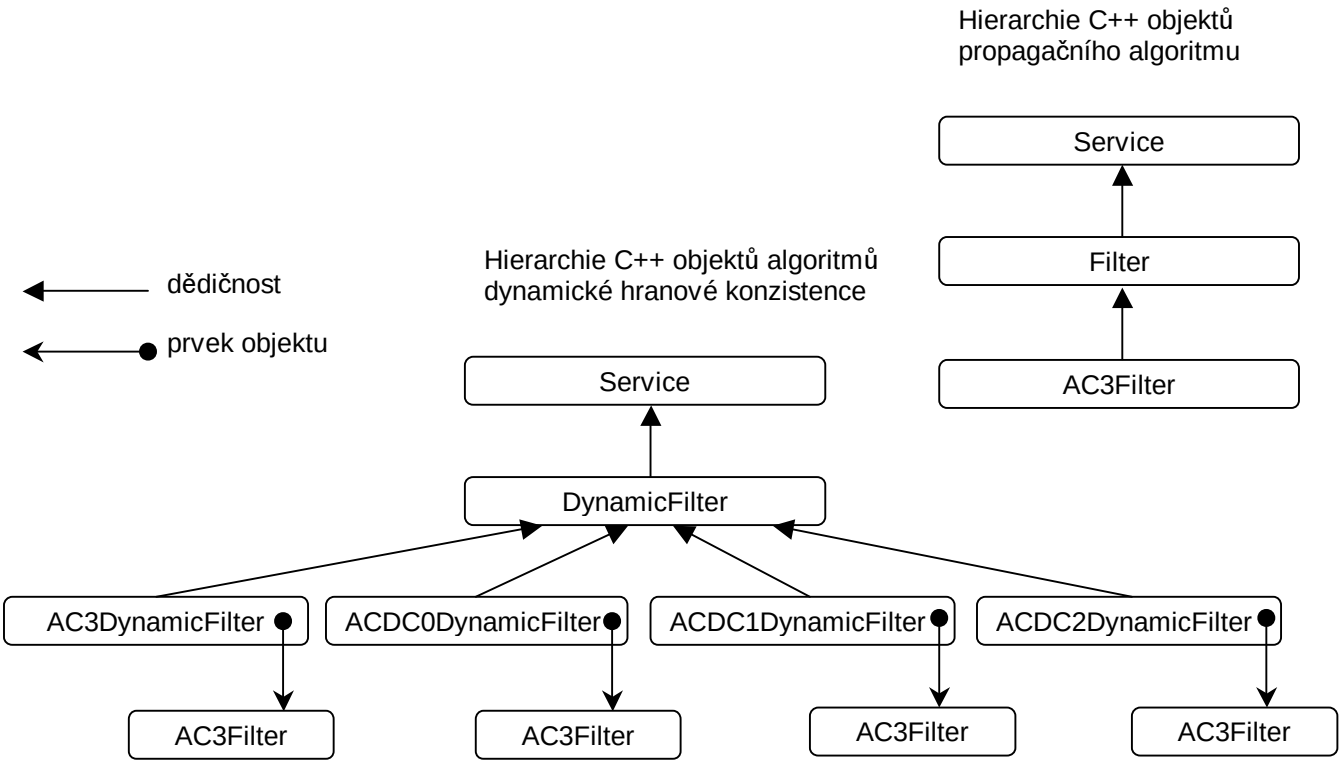


Obrázek B.1: Hierarchie C++ objektů reprezentujících problém splňování podmínek

Posledním objektem, který v této sekci zmíníme, je třída Problem. Objekt tohoto typu má na starosti udržování struktury problému jako celku, což v podstatě znamená udržovat seznam přítomných podmínek a proměnných. Objekt Problem též poskytuje operace měnící strukturu problému, tj. operace přidání a odebrání podmínky (tyto operace ale nezajišťují hranovou konzistenci) a přidání a odebrání proměnné. Stejně jako proměnné a podmínky, poskytuje i objekt reprezentující problém ukládání informací pro snadný návrat do předchozích stavů vzhledem k modifikujícím operacím.

B.3 Služby pracující s reprezentací problému

V této sekci popíšeme objekty realizující služby knihovny `SPlan`, tedy objekty, které narozdíl od objektů reprezentujících problém vykazují sami od sebe určitou činnost. Hierarchii služeb knihovny `SPlan` stručně charakterizují obrázky B.2 a B.3.



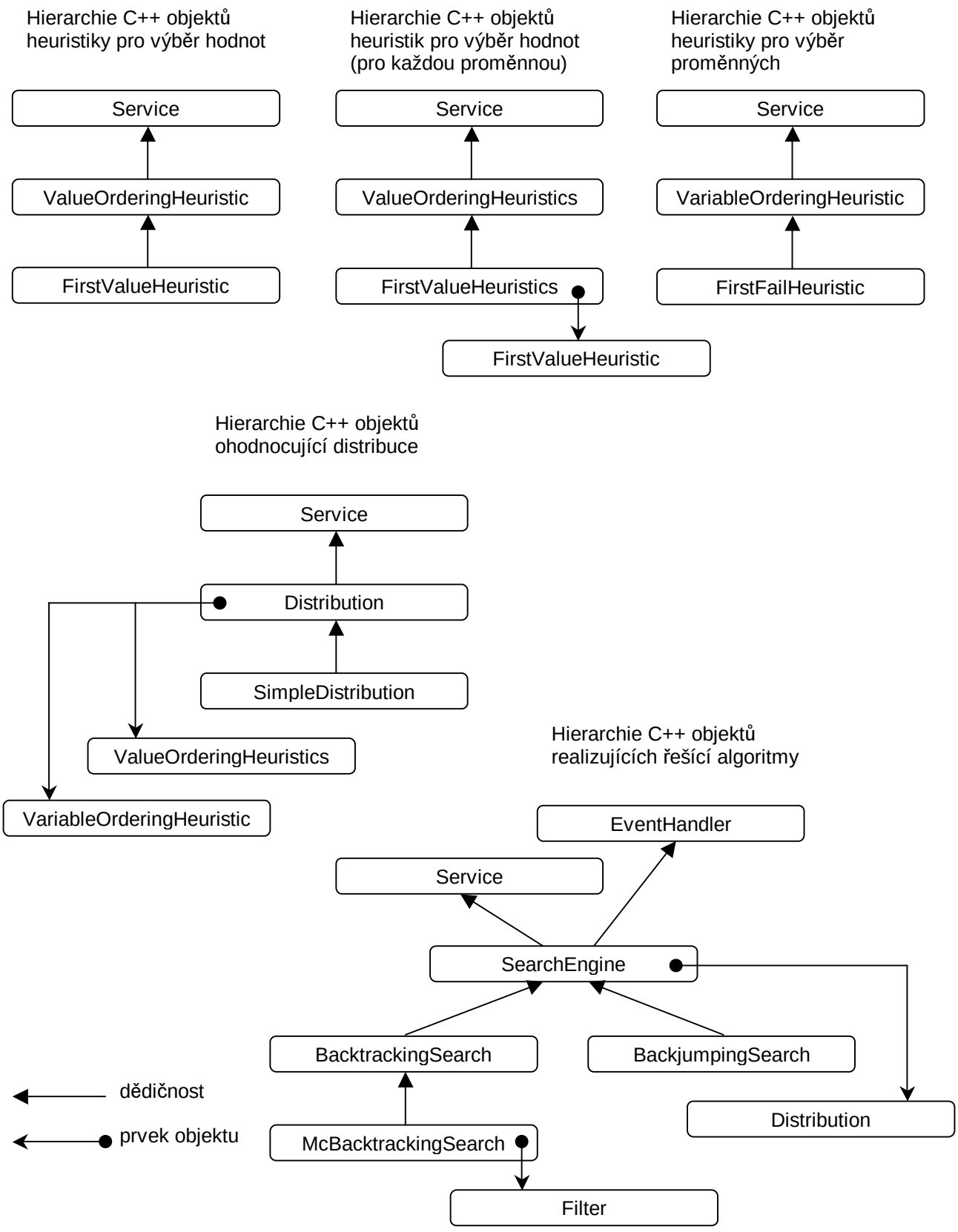
Obrázek B.2: Hierarchie objektů týkajících se algoritmů dynamické hranové konzistence

Bázovým objektem, tj. objektem, jenž je výchozím bodem pro odvozování, všech aktivních tříd je třída Service. Nejprve popíšeme objekty realizující dynamické algoritmy, k nimž se vztahuje obrázek B.2.

Třída Filter představuje základ všech konzistenčních algoritmů, konkrétní specializací tohoto objektu je pak třída AC3Filter, která implementuje konzistenční algoritmus AC-3, přesněji řečeno, jeho doplněnou verzi AC-3', jež byla popsána v kapitole 3.

Dynamické konzistenční algoritmy jsou všechny odvozeny od objektu DynamicFilter, který poskytuje operace přidání a odebrání podmínky s udržováním konzistence. Specializacemi této třídy jsou pak objekty AC3DynamicFilter, ACDC0DynamicFilter, ACDC1DynamicFilter a ACDC2DynamicFilter, které po řadě reprezentují algoritmy pro dynamickou hranovou konzistenci AC3 (konzistenční procedura AC-3 je pro modifikovaný problém spouštěna pokaždé od začátku), AC|DC-0, AC|DC-1 a AC|DC-2. Všechny dynamické konzistenční algoritmy pro svou funkci užívají propagační algoritmus AC-3 (AC-3'), který uchovávají jako svůj vnitřní prvek.

Implementace dynamických algoritmů prakticky přesně odpovídá programovému zápisu uvedenému v kapitole 3.



Obrázek B.3: Hierarchie objektů týkajících se řešících algoritmů

Ačkoli předmětem výzkumu byly především algoritmy pro dynamickou hranovou konzistenci, knihovna `SPlan` v rámci možnosti širšího uplatnění implementuje též služby pro hledání řešení problémů splňování podmínek. Objekty, které se nějakým způsobem týkají hledání řešení problémů s podmínkami, ukazuje obrázek B.3. K pochopení přesného významu všech těchto tříd je třeba znát základní techniky řešení problémů s podmínkami tak, jak je vyloženo například v publikaci [Tsa93] či elektronické [Bar98].

Implementovány jsou heuristiky pro výběr hodnot z aktuálních domén proměnných, jež je výhodné v daném kroku řešícího algoritmu vybrat pro ohodnocení. Bázové třídy těchto heuristik jsou ValueOrderingHeuristic, resp. ValueOrderingHeuristics, kde první jmenovaná je svázána s jednou konkrétní proměnnou problému, zatímco druhá třída poslouží v případě, že pro všechny proměnné v problému používáme stejnou heuristiku pro výběr hodnot. Konkrétními specializacemi uvedených tříd jsou pak objekty FirstValueHeuristic, resp. FirstValueHeuristics, které jednoduše vybírají vždy první hodnotu z aktuální domény. Dále jsou implementovány heuristiky pro výběr nejvhodnější proměnné k ohodnocení. Bázovým objektem těchto heuristik je třída VariableOrderingHeuristic. Odvozená třída FirstFailHeuristic pak implementuje heuristiku, jež vždy vybírá proměnnou s nejmenší aktuální doménou.

Knihovna `SPlan` používá místo klasického ohodnocování popsaného v první kapitole obecnější metodu - rozklad problému pomocí přidávání podmínek, tzv. *dvojic alternativ*. O tomto způsobu ohodnocování pojednává například publikace [Sch02]. Třída Distribution slouží ke generování ohodnocujících podmínek, k čemuž používá pro dosažení co nejvyšší efektivity heuristiky pro výběr hodnoty a proměnné vhodné k ohodnocení. Specializovaná třída SimpleDistribution pak generuje jednoduché unární podmínky pro rozklad problému (rovnosti a nerovnosti typu $v = d_v$, $v \neq d_v$).

Vlastní obecné řešící algoritmy jsou odvozovány od objektu SearchEngine, který poskytuje základní operace používané řešícími algoritmy. Všechny implementované řešící algoritmy jsou parametrizovány distribucí v podobě instance objektu Distribution. Z konkrétních řešících algoritmů jsou implementovány klasický backtracking realizovaný objektem BacktrackingSearch, dále backtracking s udržováním konzistence, jenž je implementován třídou McBacktrackingSearch, a nakonec backjumping realizovaný objektem BackjumpingSearch.

B.4 Ukázkové a testovací programy

Empirické testy byly prováděny pomocí přiložených ukázkových programů, které demonstrují a zároveň testují možnosti knihovny `SPlan`. Všechny ukázkové programy jsou ovládány z příkazové řádky, každý program disponuje poměrně podrobnou elektronickou nápovědou k parametrům očekávaným na příkazové řádce, takže ji zde nebudeme duplikovat dalším popisem.

Empirická měření byla prováděna pomocí programu `sprandom_ddemo`. Možnosti knihovny `SPlan` dále prezentují ukázkové programy `spqueens_demo`, jenž knihovnu testuje řešením problému, který asi nesmí chybět v žádném softwarovém produktu zabývajícím se programováním s omezujícími podmínkami, tedy známého problému N-dam (viz. například [Bar98]). Programem `spcoloring_demo` si uživatel může vyzkoušet fungování knihovny při řešení dalšího známého těžkého problému - barvení grafu. Nakonec program `sprandom_demo` testuje knihovnu `SPlan` při řešení náhodných problémů.

Příloha C

Další empirické testy

Do této přílohy byl přenechán kompletní přehled výsledků provedených testů. Následující tabulky obsahují hodnoty jednotlivých naměřených veličin připadajících na jednu odebíranou podmínku. Grafy v kapitole 4 jsou vytvořeny právě na základě zde uvedených tabulek. Informace o problémech, na nichž byly prováděny testy jsou detailně popsány v kapitole 4.

K celkovému času běhu zdůrazněme, že je uváděn pouze pro srovnání jednotlivých dynamických algoritmů a nikoli jako absolutní veličina, neboť u celkového času vše závisí na míře optimalizace dané implementace a samozřejmě i na použitém testovacím stroji.

Zdrojová data, na jejichž základě byly vytvořeny tabulky v této příloze, jsou uložena na přiloženém CD.

AC|DC-0

$n = 50$
$d = 15$
$p_c = 20$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
31	75.57143	80.4	33899.72	9543.733	354.4	409.2952	0.474
32	59.89773	63.09848	28333.62	8362.341	307.9053	334.2576	0.401212
33	41.02252	43.88739	25727.6	7813.068	280.5946	283.4414	0.369865
34	18.43307	20.02756	14992.18	4694.614	164.685	146.3661	0.223583
35	12.28226	13.60484	10056.3	3276.746	114.0726	95.70968	0.15996
36	8.205761	9.255144	8104.333	2710.926	93.88889	69.34979	0.13572
37	3.402597	4.25974	4362.333	1510.797	52.00433	30.61905	0.087229
38	1.83682	2.472803	2775.046	989.1423	33.87029	16.29707	0.066192
39	1.346457	1.866142	2231.402	820.7874	28.1063	11.93701	0.058386
40	0.564	0.936	1387.12	527.924	17.948	5.076	0.04768
41	0.353383	0.590226	1151.898	451.0564	15.2406	3.259398	0.043383
42	0.233871	0.423387	1025.234	411.6855	13.85887	2.052419	0.044113
43	0.118367	0.257143	909.5592	377.8163	12.68571	1.285714	0.043551
44	0.085586	0.216216	906.1667	387.8468	13.00901	0.833333	0.046982
45	0.074561	0.175439	848.0263	370.5658	12.41228	0.75	0.045833
46	0.038627	0.098712	730.176	326.412	10.90987	0.356223	0.042876
47	0.034188	0.089744	712.1538	326.047	10.89744	0.354701	0.04265
48	0.035398	0.084071	714.1018	334.6239	11.18142	0.367257	0.044115
49	0.022026	0.061674	676.1454	324.63	10.84141	0.211454	0.043612
50	0	0.014218	683.6066	336.2512	11.21327	0	0.046825
51	0	0.014218	669.3981	336.2512	11.21327	0	0.046493
52	0	0.008734	599.7948	307.6376	10.25764	0	0.042445
53	0	0.008734	589.4192	307.6376	10.25764	0	0.042183
54	0	0.004425	584.3274	310.9469	10.36726	0	0.042212
55	0	0.004425	574.7566	310.9469	10.36726	0	0.042035
56	0	0.004425	563.7389	310.9469	10.36726	0	0.041814
57	0	0.004425	554.0929	310.9469	10.36726	0	0.041062
58	0	0.004425	544.7522	310.9469	10.36726	0	0.04031

Tabulka C.1: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-0 pro náhodné problémy s hustotou podmínek 20%

AC|DC-1

$n = 50$
$d = 15$
$p_c = 20$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
31	55.39048	60.21905	27119.68	8532.41	311.0762	0	0.381143
32	34.43182	37.63258	17668.73	5685.295	209.0455	0	0.254886
33	20.00901	22.87387	15807.77	5167.248	185.2072	0	0.231892
34	5.976378	7.570866	6650.413	2183.965	76.51969	0	0.109803
35	1.842742	3.165323	3192.738	1076.371	37.45161	0	0.065363
36	1.621399	2.670782	3106.745	1068.202	36.92593	0	0.066914
37	0.753247	1.61039	2093.554	735.7316	25.26407	0	0.055758
38	0.422594	1.058577	1530.377	556.6736	18.98326	0	0.048577
39	0.318898	0.838583	1334.189	497.3504	16.94882	0	0.045394
40	0.248	0.62	1131.748	435.164	14.752	0	0.04352
41	0.229323	0.466165	1008.56	397.5188	13.41729	0	0.041579
42	0.153226	0.342742	930.4194	375.125	12.61694	0	0.042581
43	0.089796	0.228571	870.8041	362.0898	12.15102	0	0.04302
44	0.067568	0.198198	893.4189	382.6126	12.83333	0	0.046757
45	0.052632	0.153509	829.0307	362.5746	12.14035	0	0.045526
46	0.030043	0.090129	721.9871	322.8927	10.7897	0	0.042661
47	0.025641	0.081197	704.094	322.5427	10.77778	0	0.042607
48	0.026549	0.075221	705.9513	330.9956	11.05752	0	0.043805
49	0.022026	0.061674	676.0837	324.63	10.84141	0	0.043216
50	0	0.014218	683.6066	336.2512	11.21327	0	0.046303
51	0	0.014218	669.3981	336.2512	11.21327	0	0.045829
52	0	0.008734	599.7948	307.6376	10.25764	0	0.042009
53	0	0.008734	589.4192	307.6376	10.25764	0	0.041441
54	0	0.004425	584.3274	310.9469	10.36726	0	0.041637
55	0	0.004425	574.7566	310.9469	10.36726	0	0.041327
56	0	0.004425	563.7389	310.9469	10.36726	0	0.04115
57	0	0.004425	554.0929	310.9469	10.36726	0	0.04031
58	0	0.004425	544.7522	310.9469	10.36726	0	0.039912

Tabulka C.2: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-1 pro náhodné problémy s hustotou podmínek 20%

AC|DC-2

$n = 50$
$d = 15$
$p_c = 20$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
31	0.761905	5.590476	3933.61	1204.248	45.75238	0	0.072762
32	0.859848	4.060606	3040.405	966.6364	36.86364	0	0.058939
33	0.472973	3.337838	2961.482	947.7342	34.57207	0	0.062928
34	0.208661	1.80315	1844.724	600.7874	21.11811	0	0.046575
35	0.193548	1.516129	1588.173	535.4435	18.65323	0	0.044395
36	0.160494	1.209877	1524.346	526.358	18.15226	0	0.046173
37	0.08658	0.943723	1370.355	483.7749	16.57576	0	0.046061
38	0.041841	0.677824	1133.464	414.0084	14.09623	0	0.043138
39	0.03937	0.559055	1046.591	393.1102	13.36614	0	0.041339
40	0.064	0.436	973.66	375.508	12.7	0	0.0414
41	0.045113	0.281955	842.9549	334.1278	11.25188	0	0.039173
42	0.024194	0.21371	826.9556	335.2581	11.25403	0	0.04129
43	0.016327	0.155102	790.5143	330.0816	11.06122	0	0.041837
44	0.018018	0.148649	841.3694	360.3784	12.07658	0	0.045901
45	0.013158	0.114035	785.7018	344.3553	11.52193	0	0.045
46	0.004292	0.064378	700.7983	313.897	10.48498	0	0.042446
47	0.008547	0.064103	686.7778	314.8419	10.51709	0	0.042265
48	0.00885	0.057522	688.2522	323.0265	10.78761	0	0.043761
49	0.004405	0.044053	659.7137	316.9824	10.5815	0	0.043304
50	0	0.014218	683.564	336.218	11.21327	0	0.046019
51	0	0.014218	669.3555	336.218	11.21327	0	0.046114
52	0	0.008734	599.7074	307.607	10.25764	0	0.041878
53	0	0.008734	589.3275	307.607	10.25764	0	0.041659
54	0	0.004425	584.3186	310.9425	10.36726	0	0.041903
55	0	0.004425	574.7478	310.9425	10.36726	0	0.041239
56	0	0.004425	563.7301	310.9425	10.36726	0	0.041195
57	0	0.004425	554.0885	310.9425	10.36726	0	0.040442
58	0	0.004425	544.7478	310.9425	10.36726	0	0.04

Tabulka C.3: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-2 pro náhodné problémy s hustotou podmínek 20%

AC|DC-0

$n = 50$
$d = 15$
$p_c = 40$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
35	69.34571	71.25143	54619.7	17838.14	660.6743	736.02	0.767686
36	44.84665	46.28942	40834.54	13805.31	498.311	553.3348	0.579827
37	33.28049	34.40447	35521.09	12391.95	441.935	453.374	0.50998
38	16.54565	17.32272	21534.3	7764.83	271.9002	254.3142	0.322357
39	11.70901	12.37875	17049.56	6303.076	218.9954	197.0554	0.265497
40	7.355691	7.802846	11757.52	4463.425	154.4837	126.2744	0.190549
41	3.229839	3.538306	5807.692	2276.103	78.37903	57.03427	0.109294
42	2.206573	2.492958	4663.061	1872.103	64.19249	39.94601	0.09885
43	0.903846	1.094017	2378.291	982.3568	33.50427	16.38034	0.064316
44	0.631004	0.775109	1899.522	798.2598	27.10044	11.44541	0.058974
45	0.330435	0.447826	1386.122	596.6652	20.17826	6.017391	0.052239
46	0.181395	0.281395	1131.723	501.4	16.88605	3.353488	0.051093
47	0.108108	0.176715	882.738	403.447	13.55717	2.049896	0.044387
48	0.050505	0.09697	758.804	353.6869	11.85051	0.989899	0.042444
49	0.039139	0.074364	691.1292	328.2935	10.98826	0.739726	0.040352
50	0.017208	0.043977	621.7094	301.631	10.08031	0.3174	0.038987
51	0.009615	0.023077	601.7096	298.1135	9.951923	0.194231	0.038962
52	0.009294	0.020446	566.1617	286.6115	9.566914	0.19145	0.037286
53	0.003683	0.009208	538.9945	278.9724	9.305709	0.088398	0.036593
54	0	0.005484	516.1152	271.9196	9.067642	0	0.035941
55	0	0.001866	512.4198	275.0672	9.169776	0	0.036175
56	0	0	515.0115	281.9084	9.396947	0	0.036756
57	0	0	505.0935	281.9084	9.396947	0	0.036164
58	0	0	495.979	281.9084	9.396947	0	0.035744

Tabulka C.4: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-0 pro náhodné problémy s hustotou podmínek 40%

AC|DC-1

$n = 50$
$d = 15$
$p_c = 40$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
35	54.88857	56.79429	46149.22	15925.73	581.7371	0	0.6374
36	32.74082	34.18359	32611.27	11533.35	412.27	0	0.461058
37	21.49187	22.61585	25486.41	9209.606	325.876	0	0.366768
38	8.388535	9.165605	13041.88	4840.527	168.9554	0	0.202399
39	3.623557	4.293303	7509.4	2834.46	97.98152	0	0.131524
40	3.455285	3.902439	6639.907	2559.384	88.3313	0	0.117907
41	0.685484	0.993952	2244.157	893.3569	30.63306	0	0.058871
42	0.833333	1.119718	2430.695	982.993	33.61033	0	0.067066
43	0.405983	0.596154	1546.436	642.6368	21.8547	0	0.052329
44	0.253275	0.39738	1311.413	554.8603	18.75983	0	0.050611
45	0.184783	0.302174	1136.989	492.2283	16.61087	0	0.048587
46	0.134884	0.234884	1036.1	460.1674	15.4814	0	0.049837
47	0.081081	0.149688	835.8274	382.1019	12.82952	0	0.043701
48	0.038384	0.084848	732.1212	341.7313	11.44444	0	0.04202
49	0.033268	0.068493	680.3894	323.5284	10.82583	0	0.040294
50	0.017208	0.043977	621.7075	301.631	10.08031	0	0.03914
51	0.009615	0.023077	601.6808	298.1135	9.951923	0	0.038904
52	0.007435	0.018587	563.4201	285.3067	9.522305	0	0.037268
53	0.003683	0.009208	538.9687	278.9724	9.305709	0	0.036593
54	0	0.005484	516.1152	271.9196	9.067642	0	0.035923
55	0	0.001866	512.4198	275.0672	9.169776	0	0.036231
56	0	0	515.0115	281.9084	9.396947	0	0.036718
57	0	0	505.0935	281.9084	9.396947	0	0.036202
58	0	0	495.979	281.9084	9.396947	0	0.035744

Tabulka C.5: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-1 pro náhodné problémy s hustotou podmínek 40%

AC|DC-2

$n = 50$
$d = 15$
$p_c = 40$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
35	0.36	2.265714	3232.569	1103.117	41.28	0	0.063657
36	0.24838	1.691145	2564.227	897.4363	32.52484	0	0.057905
37	0.189024	1.313008	2223.904	797.5142	28.55691	0	0.054106
38	0.082803	0.859873	1648.476	606.7537	21.18684	0	0.04896
39	0.08776	0.757506	1613.242	607.642	21.0254	0	0.051824
40	0.038618	0.485772	1240.386	478.3638	16.4248	0	0.044451
41	0.02621	0.334677	1042.583	414.256	14.10282	0	0.042359
42	0.030516	0.316901	1148.155	467.2512	15.85211	0	0.049343
43	0.008547	0.198718	894.1346	373.3034	12.59615	0	0.043397
44	0.019651	0.163755	880.8144	374.6245	12.60262	0	0.044541
45	0.008696	0.126087	818.8804	357.2391	11.99783	0	0.043935
46	0.016279	0.116279	841.8488	376.0907	12.61395	0	0.047047
47	0.010395	0.079002	720.4927	329.4262	11.03326	0	0.042058
48	0.006061	0.052525	681.2283	318.4283	10.65051	0	0.041333
49	0	0.035225	623.3092	297.0763	9.925636	0	0.039511
50	0.001912	0.028681	596.5468	290.0554	9.686424	0	0.038623
51	0.001923	0.015385	583.8423	289.4923	9.659615	0	0.038692
52	0	0.011152	547.5911	277.5223	9.258364	0	0.036989
53	0	0.005525	529.081	273.9134	9.134438	0	0.036483
54	0	0.005484	516.1042	271.9104	9.067642	0	0.035832
55	0	0.001866	512.4179	275.0653	9.169776	0	0.036287
56	0	0	515.0115	281.9084	9.396947	0	0.03666
57	0	0	505.0935	281.9084	9.396947	0	0.036145
58	0	0	495.979	281.9084	9.396947	0	0.035763

Tabulka C.6: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-2 pro náhodné problémy s hustotou podmínek 40%

AC|DC-0

$n = 50$
$d = 15$
$p_c = 60$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
37	69.29284	70.52928	75973.34	26808.17	995.9588	1097.707	1.066009
38	38.43129	39.2517	53300.4	19231.49	689.5592	753.9279	0.758476
39	21.15746	21.73204	35638.77	13306.51	468.605	464.3577	0.519475
40	16.07067	16.536	27599.83	10529.93	369.4667	362.764	0.41088
41	11.56493	11.93708	20972.47	8209.933	286.2423	270.9786	0.321593
42	4.523288	4.749315	10224.08	4109.596	141.7726	118.2014	0.173301
43	1.656904	1.8159	4782.764	1973.139	67.52022	44.27615	0.097755
44	1.637883	1.778552	4595.003	1938.326	66.3649	43.60446	0.096379
45	0.8241	0.922438	2796.506	1204.198	41.00277	22.24238	0.070914
46	0.385224	0.456464	1625.347	714.9327	24.26517	10.25066	0.053087
47	0.107923	0.162568	982.8074	446.6148	15.03689	2.968579	0.045519
48	0.116373	0.173207	988.9378	460.1407	15.49662	3.179973	0.045683
49	0.103967	0.146375	916.1327	434.9781	14.63064	2.807114	0.045021
50	0.036536	0.066306	736.4384	356.9499	11.9594	1.014885	0.042179
51	0.030096	0.051984	707.29	350.2955	11.7223	0.800274	0.042189
52	0.02585	0.04898	683.2816	345.1796	11.54966	0.689796	0.041701
53	0.015068	0.032877	643.1493	331.4521	11.07808	0.40411	0.04111
54	0.004093	0.015007	590.719	311.8554	10.40928	0.115962	0.039959
55	0.004093	0.015007	579.5853	311.8554	10.40928	0.115962	0.039686
56	0.004093	0.015007	569.0177	311.8554	10.40928	0.115962	0.039195
57	0.001399	0.005594	557.5888	311.5804	10.39161	0.043357	0.039343
58	0	0.002797	542.4979	308.8979	10.2993	0	0.038699

Tabulka C.7: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-0 pro náhodné problémy s hustotou podmínek 60%

AC|DC-1

$n = 50$
$d = 15$
$p_c = 60$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
37	59.84165	61.07809	66596.29	24241.39	892.9284	0	0.920933
38	27.61224	28.43265	41789.7	15508.8	552.6354	0	0.587116
39	15.69061	16.26519	26997.39	10269.88	360.7251	0	0.392307
40	10.048	10.51333	19012.68	7389.493	258.3733	0	0.284947
41	7.243641	7.615797	13902.33	5506.704	191.4351	0	0.217992
42	2.09726	2.323288	5631.786	2289.827	78.83836	0	0.106753
43	0.641562	0.800558	2558.833	1062.901	36.25384	0	0.065523
44	1.286908	1.427577	3803.379	1608.841	55.04457	0	0.084178
45	0.439058	0.537396	1917.931	831.2839	28.2133	0	0.057936
46	0.255937	0.327177	1331.739	588.6201	19.9314	0	0.04872
47	0.056011	0.110656	861.2049	391.75	13.1612	0	0.043648
48	0.062246	0.11908	862.5507	401.5683	13.49526	0	0.043802
49	0.05472	0.097127	809.0739	384.8618	12.91655	0	0.043379
50	0.02977	0.05954	720.548	349.3302	11.69959	0	0.041962
51	0.021888	0.043776	685.5171	339.71	11.36252	0	0.041956
52	0.017687	0.040816	660.9823	334.1333	11.17415	0	0.041388
53	0.009589	0.027397	627.5027	323.5877	10.81096	0	0.04089
54	0.002729	0.013643	587.4775	310.1337	10.35061	0	0.039973
55	0.002729	0.013643	576.4052	310.1337	10.35061	0	0.039632
56	0.002729	0.013643	565.9195	310.1337	10.35061	0	0.039168
57	0.001399	0.005594	557.5888	311.5804	10.39161	0	0.039357
58	0	0.002797	542.4979	308.8979	10.2993	0	0.038699

Tabulka C.8: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-1 pro náhodné problémy s hustotou podmínek 60%

AC|DC-2

$n = 50$
$d = 15$
$p_c = 60$

$\frac{p_{s1} + p_{s2}}{2}$	Přidání hodnoty	Odstranění hodnoty	Testy podmínek	Operace HASSUPPORT	Operace FILTER	Operace EXTEND	Celkový čas v sekundách
37	0.258134	1.494577	3112.291	1125.215	42.05206	0	0.063666
38	0.159184	0.979592	2230.038	823.0231	29.72789	0	0.05449
39	0.099448	0.674033	1787.424	677.6243	23.80801	0	0.050235
40	0.058667	0.524	1464.712	567.228	19.796	0	0.045733
41	0.052209	0.424364	1305.145	517.4859	17.86345	0	0.044859
42	0.027397	0.253425	1046.571	425.4918	14.50137	0	0.043288
43	0.013947	0.172943	961.3013	400.0098	13.52999	0	0.043389
44	0.01532	0.155989	918.1421	390.8343	13.2117	0	0.043579
45	0.012465	0.110803	825.9612	360.6925	12.14127	0	0.042562
46	0.009235	0.080475	751.9881	334.8456	11.2467	0	0.040383
47	0.001366	0.056011	722.026	329.1557	11.02869	0	0.041639
48	0.002706	0.05954	712.7821	332.2828	11.13667	0	0.04161
49	0.001368	0.043776	681.1724	325.0834	10.87962	0	0.041532
50	0	0.02977	645.2869	314.0419	10.49797	0	0.040812
51	0.001368	0.023256	635.1149	315.2476	10.53078	0	0.0413
52	0.001361	0.02449	623.0136	315.2762	10.53333	0	0.040898
53	0.00137	0.019178	607.4603	313.4	10.46575	0	0.04063
54	0	0.010914	579.9795	306.2551	10.21965	0	0.039905
55	0	0.010914	569.0205	306.2551	10.21965	0	0.039495
56	0	0.010914	558.719	306.2551	10.21965	0	0.039031
57	0	0.004196	553.9524	309.6028	10.32448	0	0.039287
58	0	0.002797	542.4881	308.8923	10.2993	0	0.038699

Tabulka C.9: Výsledky testů pro měřené veličiny dosažené algoritmem AC|DC-2 pro náhodné problémy s hustotou podmínek 60%

Literatura

[Bak95] Andrew B. Baker: Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.
Disertační práce, University of Oregon, 1995.

[Bar98] Roman Barták: On-line Guide to Constraint Programming.
http://kti.ms.mff.cuni.cz/constraints/, 1998.

[BC94] Christian Bessière, Marie-Odile Conrdier: Arc-Consistency and Arc-Consistency Again.
In *Artificial Intelligence 65*, 179-190, 1994.

[BC99] Peter Van Beek, Xinguang Chen: A Constraint Programming Approach to Planning.
In *Proceedings of AAAI-99*, 585-590, 1999.

[Bes91] Christian Bessière: Arc-Consistency in Dynamic Constraint Satisfaction Problems.
In *Proceedings of AAAI-91*, 221-226, 1991.

[BR01] Christian Bessière, Jean Charles Régin: Refining the Basic Constraint Propagation Algorithm.
In *17th International Joint Conference in Artificial Intelligence*, 309-315, Seattle, 2001.

[DD88] Rina Dechter, Avi Dechter: Belief Maintenance in Dynamic Constraint Networks.
In *Proceedings of AAAI-88*, 37-42, 1988.

[Deb96] Romuald Debruyne: Arc-Consistency in Dynamic CSPs Is No More Prohibitive.
In *8th Conference on Tools with Artificial Intelligence*, 299-306, Toulouse, 1996.

[GCR99] Yan Georget, Philippe Codognet, Francesca Rossi: Constraint Retraction in clp(FD): Formal Framework and Performance Results.
In *Constraints, an International Journal*, 4(1):5-42, 1999.

[Har95] William D. Harvey: Nonsystematic Backtracking Search.
Disertační práce, Stanford University, 1995.

[JB97] Narendra Jussien, Patrice Boizumault: Dynamic Backtracking with Constraint Propagation - Application to Static and Dynamic CSPs.
In *CP97 Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Schloss Hagenberg, Austria, 1997.

[JDB00] Narendra Jussien, Romuald Debruyne, Patrice Boizumault: Maintaining Arc-Consistency within Dynamic Backtracking.
In *Sixth International Conference on Principles and Practice of Constraint Programming (CP'2000)*, 249-261, Springer-Verlag, 2000.

[Mac77] Alan K. Mackworth: Consistency in Networks of Relations.
In *Artificial Intelligence 8*, 99-118, 1977.

[MH86] R. Mohr, T. Henderson: Arc and Path Consistency Revised.
In *Artificial Intelligence 28*, 225-233, 1986.

[NB94] Bertrand Neveu, Pierre Berlandier: Maintaining Arc-Consistency Through Constraint Retraction: a RMS-free Approach.
In *6th Conference on Tools with Artificial Intelligence*, New Orleans, 1994.

[RDP90] Francesca Rossi, Vasant Dhar, Charles Petrie: On the Equivalence of Constraint Satisfactions Problems.
In *Proceedings of European Conference on Artificial Intelligence*, Stockholm, 1990.

[SGI01] STL - Standard Template Library.
Silicon Graphics Inc., http://www.sgi.com/tech/stl, 2001.

[Sch02] Christian Schulte: Programming Constraint Services: High-Level Programming of Standard and New Constraint Services.
Springer-Verlag, 2002.

[Str86] Bjarne Stroustrup: The C++ Programming Language.
Addison-Wesley Publishing Comp., 1986.

[SV94] Thomas Schiex, Gérard Verfaillie: Nogood Recording for Static and Dynamic Constraint Satisfaction Problems.
In *International Journal of Artificial Intelligence Tools*, 3(2):187-207, 1994.

[VS94] Gérard Verfaillie , Thomas Schiex: Solution Reuse in Dynamic Constraint Satisfaction Problems.
In *Proceedings of AAAI-94*, 585-590, 1994.

[Tsa93] Edward Tsang: Foundations of Constraint Satisfaction.
Academic Press, 1993.

[Vil01] Petr Vilím: Globálních podmínky.
Diplomová práce, Matematicko-fyzikální fakulta Univerzity Karlovy, 2001.

[ZY01] Yuanlin Zhang, Roland H. C. Yap: Making AC-3 an Optimal Algorithm.
In *17th International Joint Conference in Artificial Intelligence*, 316-321, Seattle, 2001.