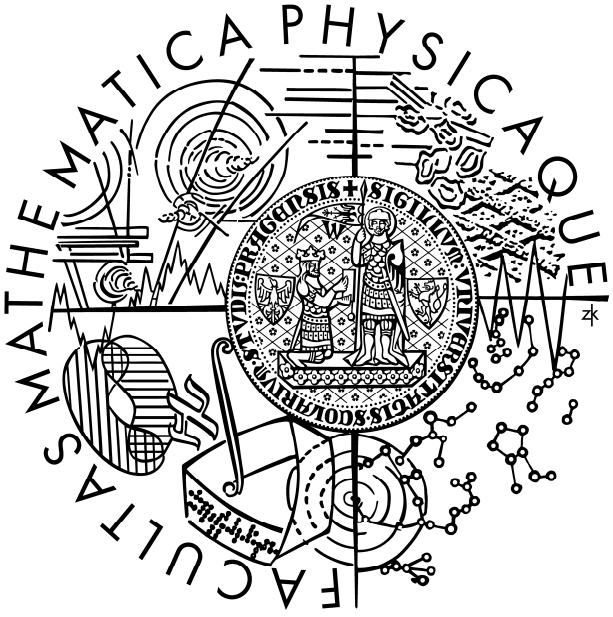
UNIVERZITA KARLOVA V PRAZE
MATEMATICKO-FYZIKÁLNÍ FAKULTA

RIGOROSNÍ PRÁCE



PAVEL SURYNEK

DYNAMICKÉ PROBLÉMY S PODMÍNKAMI

KATEDRA TEORETICKÉ INFORMATIKY A MATEMATICKÉ LOGIKY

PRAHA 2005

Děkuji všem, kteří mne při sepisování této práce podpořili, i když o tom třeba ani nevědí. Moje zvláštní poděkování pak patří Doc. RNDr. Romanu Bartákovi, Ph.D., s nímž jsem měl možnost spolupracovat na některých problémech, což pro mne bylo velmi inspirativní.

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

Ve Vlašimi dne 30. července 2005 *Pavel Surynek*

Obsah

Seznam algoritmů

Seznam obrázků

Seznam tabulek

Seznam grafů

Abstrakt

Název práce:	Dynamické problémy s podmínkami
Autor:	Pavel Surynek
Katedra (ústav):	Katedra teoretické informatiky a matematické logiky
E-mail autora:	pavel.surynek@seznam.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 několik nových algoritmů pro dynamickou hranovou konzistenci, které představují lepší kompromis mezi paměťovými a časovými požadavky v porovnání s podobnými existujícími algoritmy, jako jsou DnAC-4, DnAC-6 a AC|DC. Hlavním výsledkem práce je pak nový algoritmus, který překonává i dosud nejrychlejší existující algoritmus pro udržování hranové konzistence v dynamických problémech DnAC-6. Aby bylo možné provádět výkonnostní testy, vyvinuli jsme knihovnu `SPlan` napsanou v C++. Knihovna je určena pro řešení DCSP a nové algoritmy jakož i srovnatelné existující jsou implementovány jako její součást. Pomocí experimentálních testů na náhodně generovaných DCSP jsme demonstrovali efektivitu nových algoritmů při praktickém použití.

Klíčová slova: CSP, dynamický CSP, DCSP, dynamická hranová konzistence, DnAC-4, DnAC-6, AC|DC, AC3.1|DC

Abstract

Title: Dynamic Constraint Satisfaction Problems
Author: Pavel Surynek
Department: Department of Theoretical Computer Science and Mathematical Logic
Author's e-mail address: pavel.surynek@seznam.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 several new dynamic arc-consistency algorithms that yield a better compromise between time and space in comparison to similar existing algorithms like DNAC-4, DNAC-6 and AC|DC. The highlight of the work is a new algorithm that outperforms so far fastest algorithm for maintaining arc consistency in dynamic problems DNAC-6. In order to do performance experiments we have developed a library `SPlan` written in C++. This library solves DCSPs and the new algorithms as like as the comparable existing ones are implemented as a part of the library. Experimental results on randomly generated DCSPs demonstrate the practical efficiency of our new algorithms.

Keywords: CSP, dynamic CSP, DCSP, dynamic arc-consistency, AC|DC, DNAC-4, DNAC-6, AC|DC, AC3.1|DC

Úvod

Už v době, kdy jsem obhajoval svou diplomovou práci *Řešení dynamických problémů s podmínkami*, jsem měl v úmyslu výsledky získané při jejím řešení publikovat na mezinárodním vědeckém fóru. To se mi později skutečně podařilo a to hned na dvou mezinárodních konferencích, na *CP* v roce 2004 a na *FLAIRS* v roce 2005. V obou případech jsem na konferenčních příspěvcích úzce spolupracoval s Doc. RNDr. Romanem Bartákem, Ph.D., který byl vedoucím mé diplomové práce a který rozhodujícím způsobem přispěl ke kvalitě obou konferenčních příspěvků.

První konferenční příspěvek [SuBa04b] byl publikován na konferenci CP 2004 v kanadském Torontu jako krátký článek (poster). Příspěvek však nejprve vznikl jako plný článek a to, co nebylo publikováno v [SuBa04b], je dostupné jako technická zpráva [SuBa04a] v rámci ITI Series. Po publikaci výsledků na CP 2004 jsem dále pracoval na zlepšování přínosu a konkurenceschopnosti práce. Nové výsledky jsem pak publikoval na konferenci FLAIRS v americkém Clearwateru. Tentokrát byl příspěvek přijat jako plný článek [BaSu05] a bylo možné jej na konferenci také prezentovat, této příležitosti jsem využil.

Rigorosní práci *Dynamické problémy s podmínkami* jsem začal připravovat po uveřejnění výsledků na konferenci FLAIRS 2005. Vedla mne k tomu hlavně skutečnost, že do konferenčních článků se zdaleka nevešlo vše, co bych si přál. Přičemž tento nedostatek prostoru, podle mne, postihoval asi nejvíce empirické výsledky. Tato rigorosní práce je jakýmsi zúplněním původní diplomové práce, přičemž jsou zde navíc shrnuty a doplněny všechny výsledky, které byly získány při přípravě uvedených článků. Také se jedná o výrazné doplnění toho, co v článcích chybí. Ačkoli rigorosní práce vychází z původní diplomové práce, kterou jsem obhájil 2.února 2004, nelze ji chápat jako pouhé její rozšíření. Ty části, které jsem převzal z diplomové do rigorosní práce, jsem zásadním způsobem přepracoval.

Rigorosní práce se zabývá problematikou *dynamických problémů s podmínkami*. Výklad dané problematiky je veden v kontextu existujících technik pro práci s dynamickými problémy, na což je později, v hlavní části práce, navázáno vývojem a prezentací nových technik a výsledků získaných 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. V praxi 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. Novými původními výsledky práce je několik algoritmů pro udržování hranové konzistence vzhledem k operacím měnícím strukturu problému, které řadě ohledech zlepšují dosavadní přístupy. Vyvrcholením práce je pak algoritmus, který svou efektivitou při udržování hranové konzistence překonává i dosud nejrychlejší existující algoritmus.

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.

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 jsou pak rozvíjeny nové přístupy, jež vyúsťují návrhem nových algoritmů a příslušnou teo-

retickou analýzou, tj. důkazy korektnosti, rozbory časové a prostorové složitosti v nejhorším případě a dalšími tvrzeními o navržených algoritmech.

Čtvrtá kapitola je věnována empirickému srovnání nových algoritmů se srovnatelnými existujícími přístupy. Hlavním cílem této kapitoly je potvrdit přínos nových algoritmů 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 ukázku rozšíření jednoho z navržených algoritmů pro obecné 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*) je vzhledem ke své univerzálnosti a efektivitě velmi užitečným a silným nástrojem pro řešení celé řady reálných úloh 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 vyjadřujícími možnostmi formalizmu, který poskytují pro *popis problémů*, a existencí množství efektivních algoritmů pro *hledání řešení* těchto problémů.

Popis problémů pomocí omezujících podmínek je založen pouze na několika jednoduchých matematických pojmech (*proměnná*, *podmínka*) a je tak nezávislý na jakémkoli programovacím jazyce. Díky tomuto jednoduchému matematickému charakteru popisných prostředků je integrace omezujících podmínek do libovolného programovacího jazyka naopak značně usnadněna. Důležitou vlastností také 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 nějaký druh prohledávání prostoru všech možných ohodnocení ve spojení s tzv. *konzistenčními technikami*, které prohledávaný prostor různě prořezávají (zmenšují). Uživatel může pro řešení problémů zvolit některý z tzv. *systematických* algoritmů, které postupně (uspořádaně) 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 všech možných řešení prohledávají nesystematicky (neuspořádaně). Ačkoli algoritmy tohoto typu nemohou prokázat neexistenci řešení ani nalézt řešení všechna, jejich velkou předností je vysoká efektivita při řešení velmi rozsáhlých problémů, u nichž je použití systematických algoritmů pro značnou velikost prohledávaného prostoru problematické.

Prakticky všechny implementace řešících algoritmů nějakým způsobem aplikují nejrůznější heuristiky pro vytyčování správného směru, kudy se má algoritmus vydávat, když má možnost volby. Obecně lze říci, že omezující podmínky nabízejí pro nasazení heuristik rozsáhlé pole uplatnění. Jelikož se při hledání řešení problémů reálné velikosti bez užití heuris-

tik prakticky nelze obejít, je toto také jeden z důvodů, proč je zde programování s omezujícími podmínkami tak úspěšné.

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* (*Variables*) a *omezujících podmínek* (*Constraints*). 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* (*Domain*) 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. $(X < 3Y + 1)$, $(X \geq 4)$, $alldifferent(X,Y,Z) \equiv (X,Y,Z \text{ různé})$). 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 jednodušímu a přehlednějšímu zápisu algoritmů se často uvažují bez újmy na obecnosti podmínky nejvýše binární. To je umožněno také díky tomu, že podmínky vyšší arity než 2 lze reprezentovat pomocí binárních podmínek.

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. ■

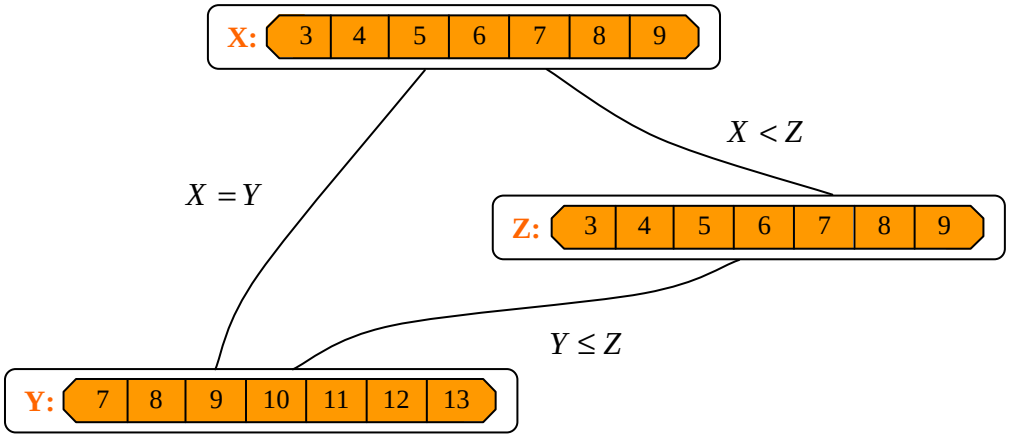
Binární podmínku svazující proměnné u a v budeme někdy též označovat symbolem $\{u,v\}$. Následující příklad ukazuje jednoduchý problém splňování podmínek obsahující tři binární podmínky.

Příklad 1.1.

Problém splňování podmínek $P = (V, C)$, 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 $D[Z] = \{3,4,5,6,7,8,9\}$;

$C = \{(X = Y),\ (X < Z),\ (Y \leq Z)\}$. ■



Obrázek 1.1: Znázornění problému splňování podmínek pomocí grafu

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. Podmín-

ka mezi danými proměnnými je v grafu znázorňována hranou, resp. hyperhranou mezi příslušnými vrcholy. Takto zkonstruovaný graf někdy též označujeme jako *graf problému*. Příklad grafického znázornění problému splňování podmínek z příkladu 1.1 pomocí grafu ukazuje obrázek 1.1.

Často je užitečné na binární podmínku pohlížet směrově, tedy pro podmínku svazující například proměnné u a v rozlišovat směr od proměnné u k proměnné v a naopak. O obou směrech dané binární podmínky potom hovoříme jako o hranách a označujeme je zápisem (u,v), resp. (v,u).

Na závěr k definici problému splňování podmínek poznamenejme, že unární podmínky je možné rovnou přenést do domén proměnných. Naopak na doménu proměnné můžeme nahlížet jako na unární podmínku.

1.2.1 Řešení problému splňování podmínek

Při definování *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 $d_i \in D[v_i]$ pro $i = 1, 2, \ldots, m$. Tímto zápisem je určeno, že proměnná v_i nabývá při ohodnocení σ hodnoty $\sigma(v_i) = d_i$ pro $i = 1, 2, \ldots, m$. 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*.

Řešení problému splňování podmínek je s využitím pojmu ohodnocení vymezeno v definici 1.2.

Definice 1.2 (ŘEŠENÍ PROBLÉMU SPLŇOVÁNÍ PODMÍNEK)**.** *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 $Sol(P)$.* ■

Řešení problému z příkladu 1.1 je například následující přiřazení hodnot proměnným: $X = 8$, $Y = 8$, $Z = 9$. Užijeme-li substitučního zápisu, pak toto řešení odpovídá ohodnocení $\sigma = \{X/8, Y/8, Z/9\}$.

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 hovoříme o tzv. *ekvivalenci problémů* splňování podmínek. Zjednodušeně řečeno, dva ekvivalentní problémy mají až na jistou transformaci stejnou množinu řešení.

Jiné ekvivalentní vyjádření problému se využívá například v situaci, kdy určitý algoritmus neumí pracovat s podmínkami obecné arity a vyžaduje, aby všechny podmínky v problému byly nejvýše binární. Takové problémy označujeme jako binární, vymezeny jsou v následující definici.

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

Několik algoritmů pracujících s dynamickými problémy vyžadují jako vstup binární problém, podrobně se těmito algoritmy budeme zabývat v kapitole 3 (částečně i v kapitole 2) a na tuto skutečnost ještě upozorníme.

1.3.1 Ekvivalence problémů splňování podmínek

Ekvivalence problémů splňování podmínek 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.

Klíčovým momentem při budování převoditelnosti problémů je vystižení vzájemné korespondence proměnných a jejich hodnot. Předvedeme, jak tuto korespondenci navrhli Francesca Rossi, Vasant Dhar a Charles Petrie v článku [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 $e_i \in t(S)$. ■

Množina $t(S)$ v definici 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 nyní pracovat pouze s problémy splňování podmínek, kde mají všechny proměnné stejnou doménu D, takové problémy budeme označovat $P = (V, C, D)$. Zobecnění pro problémy splňování podmínek s různými doménami je pouze technickou záležitostí (například můžeme D položit jako sjednocení domén jednotlivých proměnných).

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 $f_1 : Sol(P) \to Sol(Q)$, 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 $f_2 : V_P \to t(V_Q)$ 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 $f_1(\sigma)$, takto ostatně 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 rekurzivních funkcí, čímž je zajištěno, že převod probíhá pouze na syntaktické úrovni.

Projekce je funkce obvykle označovaná jako π_i^n, pro kterou platí, že π_i^n vrací pro vstupní uspořádanou n-tici její i-tý prvek. Naproti tomu funkce t_n 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 například zobecněnou k-tici $\xi = (1,(2,3),(4,5))$, předpokládejme, že úkolem 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 $f_3 : (V_P \times t(D_Q)) \rightarrow D_P$, 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 takto možné 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 tímto způsobem nelze). Převoditelnost problémů ještě objasníme na následujícím příkladě.

Příklad 1.2.

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

$P : V_P = \{X, Y, Z\}$, kde	$Q : V_Q = \{W_1, W_2\}$, kde
$D_X = \{1,2,3\}$; $D_Y = \{1,2,3\}$ a	$D_{W_1} = \{(1,3);(2,2);(3,1)\}$ a
$D_Z = \{1,2,3\}$;	$D_{W_2} \in \{(1,3);(2,2);(3,1)\}$;
$C_P = \{(X + Y = 4),\ (Y + Z = 4)\}$	$C_Q = \{(W_1[2] = W_2[1])\}$

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 lze převést na problém Q, protože existují funkce f_1, f_2 a f_3 takové, že:

$f_1 : Sol(P) \to Sol(Q)$, přičemž
$$f_1(\sigma_1) = \theta_1 \text{ a } f_1(\sigma_2) = \theta_2;$$

$f_2 : \{X, Y, Z\} \to \{(W_1, W_2)\}$, přičemž
$$f_2(X) = f_2(Y) = f_2(Z) = (W_1, W_2) \text{ a}$$

$f_3 : (\{X, Y, Z\} \times \{((1,3),(3,1)),((2,2),(2,2))\}) \to \{1,2,3\}$, přičemž
$$f_3(X, ((1,3),(3,1))) = 1, \; f_3(Y, ((1,3),(3,1))) = 3,$$
$$f_3(Z, ((1,3),(3,1))) = 1, \; f_3(X, ((2,2),(2,2))) = 2,$$
$$f_3(Y, ((2,2),(2,2))) = 2 \text{ a } f_3(Z, ((2,2),(2,2))) = 2.$$

A přesně to je požadavek na převoditelnost. ■

Fakt, že lze jeden problém splňování podmínek 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 1.5 (ROZŠÍŘENÁ EKVIVALENCE PROBLÉMŮ)**.** *Mějme dva problémy splňování podmínek P a Q. Říkáme, že problémy P a Q jsou rozšířeně ekvivalentní,, 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 uveďme, že se skutečně jedná o rozšíření pojmu ekvivalence problémů definované pomocí rovnosti množin řešení.

1.3.2 Binarizace problémů splňování podmínek

Bude-li z nějakých důvodů, jak jsme už zmínili, například proto, že to nějaký algoritmus vyžaduje, 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 tzv. *binarizace*, který daný problém splňování podmínek transformuje na problém (rozšířeně) ekvivalentní, jenž bude obsahovat podmínky arity nejvýše 2.

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

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.

Naproti tomu kódování se skrytou proměnnou zachovává původní proměnné a pro každou k-ární podmínku ($k > 2$) zavádí duální proměnnou stejně jako v duálním kódování. Pro každou proměnnou v v původní k-ární podmínce dále 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.

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 už 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 byly výhodnější než 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 pomocný algoritmus, který pohlíží na globální podmínku jako na celek a plně využívá její sémantiky. Tento pomocný algoritmus je posléze využíván obecným řešícím algoritmem hledajícím řešení celé-

ho 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* [Chip05] od společnosti Cosytec či *ILOG* [Ilog05] od společnosti 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. To 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 další informace týkající se globálních podmínek odkazujeme na práci [Vil01] od Petra Vilíma, kde lze nalézt popis celé škály dalších druhů globálních podmínek.

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 obecného 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 jednoduše 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. Nicméně je třeba říci, že si tento stručný přehled nečiní nárok na to být nějakou klasifikací existujících řešících algoritmů.

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 možných 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] od Andrewa B. Bakera.

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 procházejí nesystematicky (neuspořádaně), 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 ale zaměřuje práce Williama D. Harveye [Har95].

1.5.1 Konzistenční techniky

Oba druhy prohledávacích algoritmů více či méně spoléhají ještě 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 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 těchto 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 obvykle pracují v polynomiálním čase vzhledem k velikosti problému (součtu velikosti domén proměnných). Konzistenční algoritmy jsou díky této vlastnosti 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 zpravidla zaplatí větším množstvím spotřebovaného času či paměti.

Přesný popis mnoha konzistenčních algoritmů lze nalézt v publikaci [Tsa93] od Edwarda Tsanga nebo v elektronické podobě v [Bar98] od Romana Bartáka. 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

2.1 Úvod: účel konzistenčních algoritmů

Velmi silnou stránkou programování s omezujícími podmínkami jsou konzistenční algoritmy někdy též označované termínem konzistenční procedury. Fungování a význam konzistenčních algoritmů jsme už nastínili v první kapitole. Jsou to speciální algoritmy, které mají za úkol vyřazovat z domén proměnných hodnoty, které jistě nemohou figurovat ve výsledném řešení problému. Lze dokonce odstraňovat z domén celé množiny proměnných množiny hodnot, resp. z uspořádané k-tice domén vyřazovat k-tice hodnot.

Existuje celá řada různých druhů konzistencí a algoritmů, které je realizují. Liší se hlavně silou konzistence, kterou poskytují, tj. jaké množství hodnot jsou schopny vyřadit z domén proměnných, a svou náročností na čas a prostor. Zpravidla platí, že silnější konzistenční algoritmus má větší nároky na čas a paměť.

O hodnotách vyřazených z domén proměnných konzistenčním algoritmem hovoříme jako o *nekonzistentních* vzhledem k danému druhu konzistence. Důležitou vlastností těchto algoritmů (a konzistence obecně) je, že odstraněním nekonzistentních hodnot je množina řešení u daného problému splňování podmínek zachována beze změny, tj. množina řešení původního problému a množina řešení problému po odstranění nekonzistentních hodnot jsou identické. Pro názornost uvažujme jednoduchý příklad, mějme dvě proměnné $A \in \{1,2\}$ a $B \in \{1\}$ a podmínku $(A \neq B)$, pak můžeme ihned říci, že se hodnota 1 v proměnné A nikdy nestane součástí žádného řešení a může proto být bez ztráty řešení odstraněna z domény proměnné A (později právě použitý druh konzistence přesně definujeme).

Je jasné, že aplikací konzistenční procedury na daný problém dojde ke zmenšení počtu všech možných ohodnocení proměnných. A tím k redukci prostoru, který je potřeba při hledání řešení prohledat. Právě v tom spočívá význam konzistenčních technik.

Hlavní pole působnosti tedy konzistenční procedury nacházejí při spoluprácí s obecnými řešícími algoritmy. V praktických implementacích systémů založených na omezujících podmínkách téměř vždy obecnému řešícímu algoritmu pomáhá nějaká konzistenční procedura, která má na starosti odstraňování nekonzistentních hodnot v průběhu hledání řešení. Včasným odstraněním nekonzistentních hodnot je zužován 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.

Je také třeba zvlášť zdůraznit, že konzistenční algoritmy pracují ve většině případů 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.

2.1.1 Realizace odstraňování hodnot z domén proměnných

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 $D_0[v]$. 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 $D_0[v] - D[v]$. Ve skutečné implementaci je ale vhodné chybějící hodnoty udržovat ve zvláštní datové struktuře.

2.2 Hranová konzistence

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 řešením dynamických problémů splňování podmínek, neboť řada algoritmů z této oblasti nějakým způsobem staví právě na konzistenčních algoritmech uvedených v této kapitole.

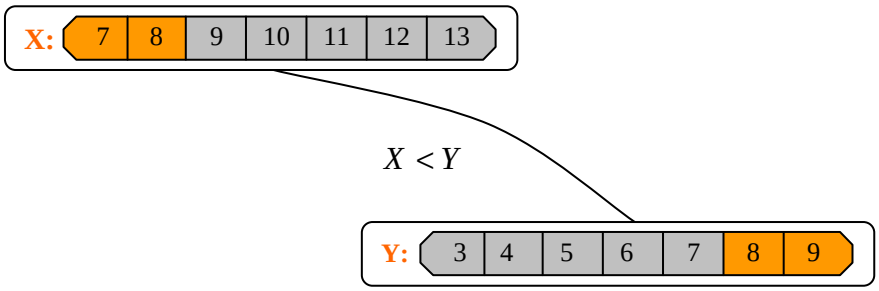
Původní hranová konzistence je definována pro binární podmínky, na problém je přitom pohlíženo jako na graf, kde hrany znázorňují podmínky. Později ukážeme, jak hranovou konzistenci rozšířit též pro podmínky vyšší arity než 2. V souladu s literaturou (například s [Tsa93]) zavedeme pojem hranové konzistence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

Nyní se automaticky 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. Oranžové prvky znázorňují aktuální doménu proměnných.



Obrázek 2.1: Hranová konzistence binární podmínky

Abychom zpřehlednili a zjednodušili zápis algoritmů, zavedeme nyní několik speciálních funkcí pro práci s hranovou konzistencí.

K výpočtu 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 zadaných aktuálních domén $D[u]$, resp. $D[v]$ proměnných u, resp. v, aby byla splněna podmínka hranové konzistence nad podmínkou c. Návratovou hodnotou funkce je dvojice (D_u^-, D_v^-) množin hodnot k odstranění z domén daných proměnných.

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 $d_{vi} \in D[v_i]$ splněno, že v aktuálních doménách ostatních proměnných $D[v_1], D[v_2], \ldots, D[v_{i-1}]$, $D[v_{i+1}], \ldots, D[v_n]$ 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 $v_1 = d_{v1}, v_2 = d_{v2}, \ldots, v_{i-1} = d_{v(i-1)}, v_i = d_{vi}, v_{i+1} = d_{v(i+1)}, \ldots,$ $\ldots, v_n = d_{vn}$ splněna. V právě popsané situaci rovněž říkáme, že hodnota $d_{vi} \in D[v_i]$ 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, $(D[v_1], D[v_2], \ldots, D[v_n])$) n-tici $(D^-_{v1}, D^-_{v2}, \ldots, D^-_{vn})$ množin hodnot, které je potřeba odebrat ze zadaných aktuálních domén $D[v_1], D[v_2], \ldots, D[v_n]$ 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 **pravda** (**true**), jestliže hodnota d_u proměnné u má podporu v zadané doméně $D[v]$ proměnné v vzhledem k podmínce c (existuje hodnota $d_v \in D[v]$ taková, že c je splněna pro ohodnocení $u = d_u$ a $v = d_v$), v opačném případě vrátí volání funkce hodnotu **nepravda** (**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($c, v_i, d_{vi}, (v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots v_n), (D[v_1], D[v_2], \ldots$ $\ldots, D[v_{i-1}], D[v_{i+1}], \ldots, D[v_n])$) vrátí hodnotu **pravda**, jestliže hodnota d_{vi} proměnné v_i má podporu v zadaných doménách $D[v_1], D[v_2], \ldots, D[v_{i-1}], D[v_{i+1}], \ldots, D[v_n]$ 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 **nepravda**.

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, tedy n-tic hodnot, pro něž je podmínka splněna. Funkce FILTER a HAS-SUPPORT při této reprezentaci pracují přesně podle uvedených definic hranové konzistence, s tím že podpory jsou hledány lineárním procházením domén.

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 od Edwarda Tsanga [Tsa93] či elektronickou od Romana Bartáka [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 v aktuálních doménách proměnných. Při hledání hranové konzistentního stavu je tedy vlastně hledán pevný bod operátoru, který uvádí do hranově konzistentního stavu všechny dosud hranově nekonzistentní podmínky.

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 skutečně 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.

2.2.1 Konzistenční algoritmus AC-3

Velmi oblíbeným algoritmem řešícím hranovou konzistenci je pro svou jednoduchost AC-3 navržený Alanem K. Mackworthem a publikovaný v článku [Mac77]. Jádro myšlenky algoritmu je velmi prosté. Hranová konzistence problému je počítána pomocí opakovaného uvádění jednotlivých podmínek do hranově konzistentního stavu až do okamžiku, kdy se obsah aktuálních domén proměnných ustálí (je nalezen pevný bod). Přitom celý proces hledání konzistentního stavu vypadá zhruba následovně. Kdykoli dojde v důsledku uvedení nějaké podmínky do konzistentního stavu k vyloučení některých hodnot z aktuální domény určité proměnné, jsou k *revizi*, tj. k dalšímu pokusu o uvedení do konzistentního stavu, naplánovány všechny podmínky, které jsou s touto proměnnou incidentní. Obecně tento proces šíření změn aktuálních domén v souvislosti s konzistenčními algoritmy označujeme jako *propagaci*, jak již ostatně bylo zmíněno v předchozí kapitole.

Algoritmus popíšeme s využitím výše definovaného primitiva - speciální funkce FILTER. Funkce FILTER bude užívána při revizi podmínky k určení nekonzistentních hodnot.

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

Ohledně programového zápisu platí následující konvence: jména procedur (nemají návratovou hodnotu) a funkcí (mají návratovou hodnotu) jsou psána kapitálkami, klíčová slova jsou psána tučně, struktura programu je vyznačena pouze užitím odsazení. Dále je třeba 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

procedure PROPAGATE-AC-3 (C_{REVISE})
1 $Q_{REVISE} \leftarrow C_{REVISE}$
2 **while** $Q_{REVISE} \neq \emptyset$ **do**
⋮ |

```
⋮
3     c ∈ Q_REVISE , libovolná podmínka
4     Q_REVISE ← Q_REVISE − {c}
5     {v_1, v_2, …, v_n} ← proměnné svázané podmínkou c
6     (D⁻_v1, D⁻_v2, …, D⁻_vn) ← FILTER ( c , (D[v_1], D[v_2], …, D[v_n]) )
7     for each i ∈ {1,2,…n} do
8         if D⁻_vi ≠ Ø then
9             D[v_i] ← D[v_i] − D⁻_vi
10            Q_REVISE ← Q_REVISE ∪ {e ∈ C | e svazuje v_i & e ≠ c}
```

Programový zápis algoritmu AC-3 předpokládá existenci několika globálních struktur definujících problém splňování podmínek - množiny proměnných V, množiny podmínek C, pole s aktuálními doménami proměnných D (indexy jsou proměnné z množiny V).

Algoritmus AC-3 je reprezentován funkcí PROPAGATE-AC-3, která ve svém jediném parametru obdržuje množinu C_{REVISE} podmínek, které je nutno zrevidovat. Chceme-li tedy učinit zadaný problém $P = (V, C)$ hranově konzistentní, spustíme algoritmus voláním PROPAGATE-AC-3(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. Algoritmus pracuje tak dlouho, dokud jsou ve frontě Q_{REVISE} obsaženy nějaké podmínky (cyklus na řádcích 2 až 10). V každém kroku algoritmus vybere jednu podmínku z fronty Q_{REVISE} (řádky 3 a 4) a na ni spustí funkci FILTER (řádek 6). 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 (podmínka na řádcích 8 až 10).

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.

Celková asymptotická časová složitost algoritmu AC-3 v nejhorším případě je tedy $O(\sum_{(u,v)\in C}(|D[u]|+|D[v]|)(|D[u]||D[v]|))$, což pro identické domény dává $O(|C||D|^3)$. Přehled složitosti algoritmu AC-3 ještě shrnuje následující tabulka.

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

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

V rámci přípravy na algoritmy pracujícími nad dynamickými problémy zavedeme ještě funkci EXTEND, která bude představovat jakousi reverzi funkce FILTER a bude naopak konzistentně rozšiřovat aktuální domény proměnných. Stejně jako u funkce FILTER, i funk-

ce 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^+, D_v^+), 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^+ a D_v^+ platí, že $d_u \in D_u^+$, jestliže d_u má nějakou podporu v zadané aktuální doméně $D[v]$ proměnné v, resp. $d_v \in D_v^+$, jestliže d_v má nějakou 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. Aplikace EXTEND(c, $(D[v_1], D[v_2], \ldots, D[v_n])$) vrátí uspořádanou n-tici množin hodnot $(D_{v1}^+, D_{v2}^+, \ldots, D_{vn}^+)$, o něž je možné konzistentně rozšířit aktuální domény proměnných po řadě $v_1, v_2, \ldots, v_n$ vzhledem k zadaným aktuálním doménám $D[v_1], D[v_2], \ldots, D[v_n]$. Pro n-tici množin hodnot $(D_{v1}^+, D_{v2}^+, \ldots, D_{vn}^+)$ tedy musí platit, že pro každé $i \in \{1,2,\ldots,n\}$ je $d_{vi} \in D_{vi}^+$, jestliže pro hodnotu d_{vi} proměnné v_i existují podpory v zadaných aktuálních doménách $D[v_1], D[v_2], \ldots, D[v_{i-1}], D[v_{i+1}], \ldots$ $\ldots, D[v_n]$ proměnných po řadě $v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$.

2.2.2 Hranová konzistence založená na seznamech podpor: Algoritmus AC-4

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.

Algoritmus nazvaný AC-4, jehož autory jsou Roger Mohr a Thomas C. Henderson, byl poprvé uveřejněn v článku [MoHe86]. Algoritmus AC-4 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 nulu, 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. Uveďme ale, že pro podmínky vyšší arity existuje zobecněná verze algoritmu AC-4 autorů Rogera Mohra a Géralda Masiniho nazvaná GAC-4. Na bázi GAC-4 žádný dynamický algoritmus budovat nebudeme, proto se jím zde zabývat nebudeme a pouze odkážeme na článek od jmenovaných autorů [MoMa88], kde je algoritmus popsán.

Pro uchovávání zmiňovaných informací o podporách jsou v programu vyhrazeny datová struktury $counter$ a S. Datová struktura $counter$ reprezentuje počítadla podpor, zatímco struktura S reprezentuje seznamy podporovaných hodnot. Struktura $counter$ je pole indexované dvojicí proměnné a její hodnoty (u, d_u) a další proměnnou v, která s u sousedí prostřednictvím nějaké 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), (tedy 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, resp. ke kontrole jsou naplánovány proměnné sousedící s u.

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 druhou datovou strukturu - seznamy podporovaných hodnot S. Struktura S je 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

```
function INITIALIZE-AC-4 (C_REVISE)
1     Q_REVISE ← Ø
2     for each c ∈ C_REVISE do
3         {u,v} ← proměnné svázané podmínkou c
4         zruš záznamy ve struktury S a counter příslušející podmínce c
5         for each (d_u, d_v) ∈ D_0[u]×D_0[v] do
6             if (d_u, d_v) splňuje podmínku c then
7                 counter[(u,d_u),v] ← counter[(u,d_u),v]+1
8                 S[v,d_v] ← S[v,d_v] ∪ {(u,d_u)}
9                 counter[(v,d_v),u] ← counter[(v,d_v),u]+1
10                S[u,d_u] ← S[u,d_u] ∪ {(v,d_v)}
11        Q^uv_REVISE ← INITIALIZE-ARC-AC-4 ((u,v))
12        Q^vu_REVISE ← INITIALIZE-ARC-AC-4 ((v,u))
13        Q_REVISE ← Q_REVISE ∪ Q^uv_REVISE ∪ Q^vu_REVISE
14    return Q_REVISE

function INITIALIZE-ARC-AC-4 ((u,v))
1     Q_REVISE ← Ø
2     for each d_u ∈ D[u] do
⋮
```

```
⋮
3       if counter[(u, d_u), v] = 0 then
4           D[u] ← D[u] − {d_u}
5           Q_REVISE ← Q_REVISE ∪ {(u, d_u)}
6   return Q_REVISE
```

procedure PROPAGATE-AC-4 (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) ∈ S[v, d_v] do
5           counter[(u, d_u), v] ← counter[(u, d_u), v] − 1
6           if counter[(u, d_u), v] = 0 and d_u ∈ D[u] then
7               D[u] ← D[u] − {d_u}
8               Q_REVISE ← Q_REVISE ∪ {(u, d_u)}
```

Program algoritmu AC-4 sestává z funkcí INITIALIZE-AC-4, INITIALIZE-ARC--AC-4 a procedury PROPAGATE-AC-4. Předpokládá se existence globálních datových struktur tvořících problém - množiny proměnných V, množiny podmínek C, polí s aktuálními a původními doménami proměnných - D a D_0. Struktury *counter* a S jsou rovněž globální a předpokládá se jejich korektní inicializace.

Chceme-li uvést problém $P = (V, C)$ do hranově konzistentního stavu, algoritmus spustíme voláním $Q_{REVISE} \leftarrow$ INITIALIZE-AC-4(C) následovaným voláním PROPAGATE-AC-4(Q_{REVISE}). V tomto případě jsou podpůrné datové struktury *counter* a S budovány od začátku a postačí tedy, když jsou před prvním voláním vynulovány.

Funkce INITIALIZE-AC-4 má na starosti počáteční přípravu globálních datových struktur algoritmu AC-4 (*counter* a S) a počáteční odebrání nekonzistentních hodnot. Parametrem funkce je množina podmínek C_{REVISE}, jež uživatel požaduje algoritmem AC-4 uvést do konzistentního stavu. Návratovou hodnotou funkce INITIALIZE-AC-4 je fronta Q_{REVISE} dvojic proměnná a její hodnota naplánovaných k revizi, tj. proměnných a hodnot,

jejichž sousedy je třeba zkontrolovat, zda neztratily podporu. Funkce pro každou podmínku z množiny C_{REVISE} nejprve připraví struktury *counter* a S (cyklus na řádcích 5 až 10), přičemž postupuje přesně podle definice splňování podmínky, tzn., že pro každou dvojici hodnot z původních domén proměnných svázaných danou podmínkou (řádek 5) je otestováno zda je pro ně podmínka splněna a pokud ano, jsou provedeny příslušné aktualizace globálních datových struktur (řádky 6 až 10).

Funkce INITIALIZE-AC-4 používá ke své práci ještě pomocnou funkci INITIALIZE--ARC-AC-4, která provádí počáteční odstranění nekonzistentních hodnot pro zadanou hranu - pro uspořádanou dvojici (u,v) proměnných. Funkce rovněž provádí přípravu fronty Q_{REVISE} vzhledem k hraně (u,v). Činnost funkce je v podstatě jednoduchá, pro všechny hodnoty z aktuální domény proměnné u je zkontrolován počet podpor *counter* vůči podmínce svazující proměnné u a v v proměnné v (řádek 3) a je-li nulový, je daná hodnota vyřazena z aktuální domény a je naplánovaná kontrola sousedů do fronty Q_{REVISE} (řádky 4 a 5).

Fronta Q_{REVISE} obsažená v návratové hodnotě funkce INITIALIZE-AC-4 je pak očekávána jako parametr procedury PROPAGATE-AC-4, která v podstatě realizuje hlavní smyčku algoritmu pracující podle výše popsané myšlenky. 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 a odpovídajícím způsobem modifikované globální datové struktury *counter* a S. Procedura PROPAGATE-AC-4 postupně vybírá prvky z fronty Q_{REVISE} - dvojice proměnných a jejich hodnot (u,d_u) (řádek 2), pro které kontroluje všechny podporované hodnoty, resp. jim snižuje počítadla podpor (cyklus od řádku 4) . Jestliže některé počítadlo podpor klesne na 0 (řádek 6), je i jemu příslušná hodnota odebrána z aktuální domény proměnné (řádek 7) a opět je naplánována kontrola sousedů (řádek 8).

Oproti algoritmu AC-3 značně narostly u AC-4 paměťové nároky, přičemž ty má na svědomí hlavně globální datová struktura S, která v nejhorším případě zabere prostor o velikosti $O(\sum_{(u,v)\in C}|D_0[u]\|D_0[v]|)$, což je pro identické domény $O(|C\|D|^2)$. Prostor potřebný

k uchování počítadel podpor *counter* je $O(\sum_{(u,v)\in C}|D_0[u]|+|D_0[v]|)$, pro identické domény $O(|C||D|)$, což vzhledem k paměťovým nárokům seznamů podporovaných S hodnot 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 $O(\sum_{(u,v)\in C}|D_0[u]||D_0[v]|)$, což je $O(|C||D|^2)$ pro identické domény. Následující tabulka shrnuje uvedené výsledky.

AC-4		**Propagace**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$

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.

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

Algoritmus AC-6 popsaný v článku [BeCo94], jehož autory jsou Christian Bessière a Marie-Odile Cordier, částečně vychází z algoritmu AC-4. Opět se opírá o ideu seznamů podpor, resp. seznamů 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, čímž máme na mysli například uspořádání přirozených čísel.

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 podmín-

ce, 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 globální datovou strukturu S, která opět reprezentuje pole indexované proměnnými a jejich hodnotami. 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 jiný způsob zpracování pole S.

Algoritmus 2.3. AC-6

```
function INITIALIZE-AC-6 (C_REVISE)
1       Q_REVISE ← Ø
2       for each c ∈ C_REVISE do
3           {u,v} ← proměnné svázané podmínkou c
4           zruš záznamy ve struktuře S příslušející podmínce c
5           Q_REVISE^uv ← INITIALIZE-ARC-AC-6 ((u,v))
6           Q_REVISE^vu ← INITIALIZE-ARC-AC-6 ((v,u))
7           Q_REVISE ← Q_REVISE ∪ Q_REVISE^uv ∪ Q_REVISE^vu
8       return Q_REVISE

function INITIALIZE-ARC-AC-6 ((u,v))
1       Q_REVISE ← Ø
2       for each d_u ∈ D[u] do
3           d_v^↑ ← NEXT-SUPPORT-AC-6 ((u,d_u),(v,NIL))
4           if d_v^↑ ≠ NIL then
⋮           ⋮
```

```
⋮
5           S[v, d_v^↑] ← S[v, d_v^↑] ∪ {(u, d_u)}
6       else
7           D[u] ← D[u] − {d_u}
8           Q_REVISE ← Q_REVISE ∪ {(u, d_u)}
9   return Q_REVISE
```

procedure PROPAGATE-AC-6 (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) ∈ S[v, d_v] do
5           S[v, d_v] ← S[v, d_v] − {(u, d_u)}
6           d_v^↑ ← NEXT-SUPPORT-AC-6 ((u, d_u), (v, d_v))
7           if d_v^↑ ≠ NIL then
8               S[v, d_v^↑] ← S[v, d_v^↑] ∪ {(u, d_u)}
9           else if d_u ∈ D[u] then
10              D[u] ← D[u] − {d_u}
11              Q_REVISE ← Q_REVISE ∪ {(u, d_u)}
```

function NEXT-SUPPORT-AC-6 ($(u, d_u), (v, d_v)$)

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

Program algoritmu AC-6 se skládá z funkcí INITIALIZE-AC-6, INITIALIZE-ARC--AC-6 a NEXT-SUPPORT-AC-6 a procedury PROPAGATE-AC-6. Význam INITIALIZE-AC-6, INITIALIZE-ARC-AC-6 a PROPAGATE-AC-6 je prakticky totožný s odpovídajícími funkcemi a procedurou 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.

Program opět předpokládá existenci obvyklých globálních datových struktur definujících problém V, C, D a D_0, a pomocné datové struktury S. Je-li cílem učinit zadaný problém $P=(V,C)$ hranově konzistentním, algoritmus je třeba spustit voláním $Q_{REVISE} \leftarrow$ INITIALIZE-AC-6(C) a následně PROPAGATE-AC-6(Q_{REVISE}). Globální datová struktura S se na začátku předpokládá 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. Funkce INITIALIZE-AC-6 k tomuto účelu používá pomocnou funkci INITIALIZE-ARC-AC-6, která provádí tento úkol vzhledem k jedné zadané hraně. Ta pak ještě dále volá funkci NEXT-SUPPORT-AC-6 (řádek 3), pomocí níž hledá nejmenší podpory a podle toho, zda nějaké nalezne, aktualizuje datovou strukturu S (řádek 5) nebo zasaženou hodnotu odstraní (řádky 7 a 8).

Hlavní smyčka algoritmu je realizována funkcí PROPAGATE-AC-6, která opět pracuje s frontou Q_{REVISE}, kterou připravují inicializační funkce. Fronta Q_{REVISE} obsahuje dvojice proměnná a její hodnota, která byla v minulosti odebrána. Procedura pracuje tak, že postupně vybírá prvky z fronty (řádky 2 a 3) a když zjistí, že v minulosti odebraná hodnota byla nejmenší podporou jiným hodnotám (cyklus na řádcích 4 až 11), je nutné pro tyto hodnoty najít jinou nejmenší podporu, v daném uspořádání následující (řádek 6). Nepodaří-li se to (řádek 9), jsou i ony odebrány a naplánovány do fronty Q_{REVISE} ke kontrole (řádky 10 a 11).

Jak je to s nároky na časové a paměťové zdroje ? 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		**Propagace**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|C\|D\|)$
	Různé domény	$O(\sum_{(u,v)\in C}(\|D[u]\|+\|D[v]\|))$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|D[v]\|)$

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

Nyní, narozdíl od algoritmu AC-4, ve výsledcích figurují aktuální domény, nikoli původní. To je pozitivní vzhledem k chování v průměrném případě, kdy jsou typicky aktuální domény menší než původní. Pro nejhorší případ je algoritmus dokonce optimální, co se časových nároků týče.

Posledním algoritmem, který v této kapitole popíšeme, je konzistenční algoritmus AC-3.1. AC-3.1 vznikl vylepšením velmi oblíbeného algoritmu AC-3, jehož nepříznivou vlastností jsou příliš velké časové nároky v nejhorším případě. Právě tento nedostatek odstraňuje algoritmus AC-3.1, přičemž důraz je kladen na zachování jednoduchosti algoritmu. V následující kapitole využijeme algoritmus AC-3.1 k budování algoritmů pro dynamické problémy.

2.2.4 Vylepšený AC-3: konzistenční algoritmus AC-3.1

Autory algoritmu AC-3.1 jsou Yuanlin Zhang a Roland H. C. Yap, publikovali jej v článku [ZhYa01]. Klíč ke zlepšení výkonu algoritmu AC-3 autoři hledali ve zefektivnění operace FILTER. Již jsme zmiňovali, že v rámci algoritmu AC-3 provádí operace FILTER řadu testů opakovaně bez toho, aby bylo využito již spočtených informací. V algoritmu AC-3.1 je tomu jinak. Operace FILTER si zde pamatuje, v jaké situaci skončilo její poslední volání a při dalším vstupu do funkce pokračuje od tohoto místa. Tak je zajištěno, že každý test podmínky na splnění pro dané hodnoty proběhne nejvýše jedenkrát.

Algoritmus AC-3.1 vyžaduje, podobně jako AC-6, lineární uspořádání domén proměnných. Toto uspořádání je opět definováno operacemi FIRST a NEXT. Narozdíl od původního článku [ZhYa01] uvedeme algoritmus pro podmínky obecné arity. K tomu je nutné dodefinovat uspořádání prvků v doménách pro uspořádané n-tice hodnot (v uspořádaných n-ticích domén). Jako prostředek pro toto zobecnění poslouží lexikografické uspořádání.

Nechť jsou dány domény proměnných $D[v_1], D[v_2], \ldots, D[v_n]$, předpokládejme, že prvky v doménách jsou uspořádány pomocí relací lineárního uspořádání po řadě $<_{v1}, <_{v2}, \ldots, <_{vn}$. Pro uspořádané n-tice $(d_{v1}, d_{v2}, \ldots, d_{vn}) \in D[v_1] \times D[v_2] \times \ldots \times D[v_n]$ a $(h_{v1}, h_{v2}, \ldots, h_{vn}) \in D[v_1] \times D[v_2] \times \ldots \times D[v_n]$ definujeme uspořádání $\prec_{v1\ldots vn}$ následovně. $(d_{v1}, d_{v2}, \ldots, d_{vn}) \prec_{v1\ldots vn} (h_{v1}, h_{v2}, \ldots, h_{vn})$, právě tehdy když pro nejmenší $i \in \{1,2,\ldots,n\}$ takové, že $d_{vi} \neq h_{vi}$, je $d_{vi} <_{vi} h_{vi}$. Pro uspořádání nad n-ticemi hodnot příslušně dodefinujeme i operace FIRST a NEXT. Aplikace FIRST($(D[v_1], D[v_2], \ldots, D[v_n])$) vrátí nejmenší prvek z $D[v_1] \times D[v_2] \times \ldots \times D[v_n]$ vzhledem k upořádání $\prec_{v1\ldots vn}$, případně hodnotu NIL, pokud je některá ze zadaných domén $D[v_1], D[v_2], \ldots, D[v_n]$ prázdná. Volání operace NEXT($(d_{v1}, d_{v2}, \ldots, d_{vn}), (D[v_1], D[v_2], \ldots, D[v_n])$) vrátí n-tici hodnot $(h_{v1}, h_{v2}, \ldots, h_{vn})$ bezprostředně následující po n-tici hodnot $(d_{v1}, d_{v2}, \ldots, d_{vn})$ v kartézském součinu zadaných domén $D[v_1] \times D[v_2] \times \ldots \times D[v_n]$ vzhledem k uspořádání $\prec_{v1\ldots vn}$. Je-li uspořádaná n-tice hodnot $(d_{v1}, d_{v2}, \ldots, d_{vn})$ největším prvkem v $D[v_1] \times D[v_2] \times \ldots \times D[v_n]$ vzhledem k $\prec_{v1\ldots vn}$, vrátí uvedené volání hodnotu NIL.

Algoritmus AC-3.1 používá speciální verzi funkce FILTER, která si udržuje vnitřní stav. Pro každou hodnotu v aktuálních doménách proměnných a vzhledem ke každé podmínce si funkce FILTER pamatuje, kde skončilo hledání další podpory pro tuto hodnotu při posledním volání funkce FILTER vzhledem k uspořádání $\prec_{v1\ldots vn}$. Tato informace je ukládána do pomocné globální datové struktury *resume*. Datová struktura *resume* je pole indexované podmínkou c a dvojicí proměnná a její hodnota (v_i, d_{vi}), přičemž proměnná v_i je jednou z proměnných svázaných podmínkou c. Jestliže podmínka c svazuje proměnné $v_1, v_2, \ldots, v_n$, pak buňka $resume[c, (v_i, d_{vi})]$ obsahuje $(n-1)$-tici hodnot $(d_{v1}, d_{v2}, \ldots$ $\ldots, d_{v(i-1)}, d_{v(i+1)}, \ldots, d_{vn})$ z (aktuálních) domén proměnných $v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$, která je poslední nalezenou podporou pro hodnotu d_{vi} v aktuální doméně proměnné v_i vůči podmínce c vzhledem k uspořádání $\prec_{v1\ldots v(n-1)}$. Záznam v buňce $resume[c, (v_i, d_{vi})]$ je pak použit v okamžiku, když hodnota d_{vi} v aktuální doméně proměnné v_i ztratí podporu $(d_{v1}, d_{v2}, \ldots, d_{v(i-1)}, d_{v(i+1)}, \ldots, d_{vn})$ vzhledem k podmínce c. Další podpory pro hodnotu d_{vi} jsou hledány mezi $(n-1)$-ticemi následujícími po $(d_{v1}, d_{v2}, \ldots, d_{v(i-1)}, d_{v(i+1)}, \ldots, d_{vn})$ vzhledem k uspořádání $\prec_{v1\ldots v(n-1)}$. Možnost začít hledat další podpory až od poslední nalezené a nikoli od začátku je umožněno právě díky udržování informací ve struktuře *resume*. Postupuje-li algoritmus tímto způsobem je zajištěno, že je každá podpora vzhledem ke každé podmínce posuzována nejvýše jedenkrát (u algoritmu AC-3 tomu tak nebylo). Časové nároky algoritmu tedy jsou pro nejhorší případ optimální.

Symbolický zápis algoritmu AC-3.1 je uveden jako algoritmus 2.4. Oproti originálnímu článku [ZhYa01] jsme algoritmus zobecnily i pro podmínky arity vyšší než 2.

Algoritmus 2.4. AC-3.1

procedure PROPAGATE-AC-3.1 (C_{REVISE})
1 $Q_{REVISE} \leftarrow C_{REVISE}$
2 **while** $Q_{REVISE} \neq \emptyset$ **do**
⋮

```
⋮
3        c ∈ Q_REVISE , libovolná podmínka
4        Q_REVISE ← Q_REVISE − {c}
5        {v_1, v_2, …, v_n} ← proměnné svázané podmínkou c
6        (D⁻_v1, D⁻_v2, …, D⁻_vn) ←
7                ← FILTER-AC-3.1 ( c , (D[v_1], D[v_2], …, D[v_n]) )
8        for each i ∈ {1,2,…n} do
9            if D⁻_vi ≠ Ø then
10               D[v_i] ← D[v_i] − D⁻_vi
11               Q_REVISE ← Q_REVISE ∪ {e ∈ C | e svazuje v_i & e ≠ c}
```

function FILTER-AC-3.1 (c , $(D[v_1], D[v_2], \ldots, D[v_n])$)

```
1    D^n ← (D[v_1], D[v_2], …, D[v_n])
2    for each i ∈ {1,2,…,n} do
3        D^(n−1) ← (D[v_1], D[v_2], …, D[v_(i−1)], D[v_(i+1)], …, D[v_n])
4        D⁻_vi ← Ø
5        for each d_vi ∈ D[v_i] do
6            if resume[c, (v_i, d_vi)] = NIL then
7                resume[c, (v_i, d_vi)] ← FIRST( D^(n−1) )
8            l^(n−1) ← resume[c, (v_i, d_vi)]
9            if l^(n−1) je zahrnuto v D^(n−1) then
10               continue
11           else
12               while l^(n−1) ≠ NIL do
13                   l^(n−1) ← NEXT( l^(n−1), D^(n−1) )
14                   l^n ← INSERT( i , d_vi , l^(n−1) )
15                   if l^(n−1) = NIL then
16                       D⁻_vi ← D⁻_vi ∪ {d_vi}
17                   else if c je splněna pro l^n then
18                       resume[c, (v_i, d_vi)] ← l^(n−1)
19                       break
20   return (D⁻_v1, D⁻_v2, …, D⁻_vn)
```

Program algoritmu AC-3.1 se skládá z procedury PROPAGATE-AC-3.1 a funkce FILTER-AC-3.1. Program opět předpokládá existenci globálních struktur V, C, D a D_0. Dále je předpokládána existence globálního pole *resume*, které uchovává naposledy nalezené podpory. Pole *resume* je na začátku vyplněno hodnotami *NIL*.

Fungování procedury PROPAGATE-AC-3.1 se liší od stejné procedury u algoritmu AC-3 pouze v jednom detailu, místo základní funkce FILTER je na řádcích 6 a 7 volána upravená funkce FILTER-AC-3.1 s ukládáním stavů do pole *resume*.

Myšlenka algoritmu je zakódována ve funkci FILTER-AC-3.1, která dostává jako parametry podmínku c a n-tici domén proměnných. Výsledek činnosti funkce FILTER--AC-3.1 je naprosto totožný s funkcí FILTER, opět vrací n-tici množin nekonzistentních hodnot, které je potřeba vyloučit po řadě ze zadaných domén. Funkce pracuje tak, že pro každou hodnotu ze zadaných domén (cykly od řádku 2, resp. 5) zkoumá, zda má podporu. S hledáním podpory se začíná u naposledy nalezené podpory uložené v poli *resume* (řádky 6 až 8). Hledání pokračuje vzestupně vzhledem k uspořádání $\prec_{v1\ldots v(n-1)}$ (cyklus na řádcích 12 až 19) a když se podaří další podporu najít, je tato uložena do pole *resume* a hledání je ukončeno (řádky 17 až 19). Při neúspěchu je hodnota, pro kterou je podpora hledána, prohlášena za nekonzistentní přidána do návratové hodnoty (řádky 15 a 16).

AC-3.1		**Propagace**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|)$
	Různé domény	$O(\sum_{(u,v)\in C}(\|D[u]\|+\|D[v]\|))$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|\|D[v]\|)$

Tabulka 2.4: Shrnutí časové a prostorové složitosti algoritmu AC-3.1

Časová a prostorová složitost v nejhorším případě algoritmu AC-3.1 pro binární podmínky je shrnuta v tabulce 2.4. Díky tomu, že je každá podpora vzhledem ke každé hodnotě a každé podmínce testována nejvýše jednou a celkem je takových podpor v binárním problému $O(\sum_{(u,v)\in C}|D[u]\|D[v]|)$, resp. $O(|C\|D|^2)$, když jsou všechny domény stejně velké, dostáváme uvedený výsledek. Prostorová složitost je dána hlavně nároky dodatečné struktury *resume*. V binárním problému je třeba pro každou podmínku svazující proměnné u a v uchovávat v poli *resume* $|D[u]|+|D[v]|$ záznamů. V celém problému to tedy znamená $\sum_{(u,v)\in C}(|D[u]|+|D[v]|)$, resp. $|C\|D|$ pro identické domény, záznamů. Další paměťové nároky již asymptotickou prostorovou složitost nenavýší.

2.2.5 Síla hranové konzistence

Algoritmem AC-3.1 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

Dynamické problémy s podmínkami

3.1 Úvod: proč dynamizovat CSP

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. V tomto smyslu statický přístup se ale ukazuje v některých aplikacích jako příliš svazující či úplně nevhodný. Primárním požadavkem na formalismy a techniky pro řešení problémů, ať už jakéhokoliv typu, je vždy možnost modelovat a řešit jejich použitím skutečné praktické problémy. Reálné problémy nejsou ovšem svou povahou vždy statické, často je třeba modelovat úlohy vycházející z dynamicky se měnícího prostředí. Při hledání řešení takových problémů je pak vhodné, aby řešící systém uměl odpovídajícím způsobem reagovat na skutečnosti, které jsou v plném rozsahu známy až v průběhu řešení a v původním zadání problému vůbec nefigurovaly (Dechter a Dechter v [DeDe88]).

Jak uvádí například Codognet, Diaz a Rossi v [CDR96], klíčovým prvkem moderních systémů programování s omezujícími podmínkami je také interakce systému s uživatelem a ta si sama o sobě vynucuje nestatický přístup. Navíc kromě dynamičnosti řešené úlohy nebo změn vyvolaných interakcí s uživatelem je též často užitečné, aby samotný řešící algoritmus prováděl určité zásahy do řešeného problému. Algoritmy pracující tímto způsobem popsují mimo jiné Jussien a Boizumault v [JuBo97] nebo titíž autoři společně s Boizumaultem v [JDB00] či Verfaillie a Schiex v [VeSch94a].

Tato kapitola je věnována technikám dynamizace problémů splňování podmínek. Úvodní části kapitoly jsou věnovány obeznámení se s problematikou a souvisejícími více či méně otevřenými otázkami. Poté se pozornost obrátí k jedné vybrané otázce, které bude věnován zbytek kapitoly až do konce - otázce udržování hranové konzistence v dynamických problémech.

V souvislosti s udržováním hranové konzistence představíme v této kapitole všechny význačné existující algoritmy. Popíšeme a zdůvodníme jejich časovou a prostorovou složitost. U každého algoritmu rovněž shrneme jeho pozitivní i negativní vlastnosti a provedeme stručné srovnání existujících algoritmů. Od popisu existujících algoritmů pak v dalších částech kapitoly přejdeme k vlastním návrhům nových algoritmů, které budeme ve stejném duchu rozebírat a analyzovat. Kapitolu uzavře několik tvrzení podporujících dobré vlastnosti nově navržených algoritmů.

3.2 Dynamické úlohy

S úlohami dynamického charakteru se lze setkat v nejrůznějších oblastech a situacích. Příkladem úlohy, u které nefunguje statický přístup programování s omezujícími podmínkami, jsou plánovací problémy. Stručně naznačme, že u plánovacího problému je úkolem transformovat počáteční stav prostředí pomocí aplikace dovolených operací či pravidel na požadovaný cílový stav prostředí. Jelikož ale předem není znám počet kroků aplikací daných pravidel vedoucích k řešení, bývá plánovací problém modelován pomocí podmínek vždy jen pro určitý omezený počet kroků aplikace dovolených operací. Není-li u takto namodelovaného plánovacího problému nalezeno řešení (počet kroků, pro které je model vytvořen, nestačí k vyřešení úlohy), je model rozšířen o další kroky (pomocí přidávání proměnných a podmínek). Právě popsaný postup požívají Van Beek a Chen v systému *CPlan*, kterým se zabývá práce [VBChe99]. Zde se ukazuje jako velmi neefektivní řešit každý modifikovaný problém (s přidanými kroky) staticky, tj. úplně od začátku bez využití dosud získaných výsledků.

Podívejme se na jiný příklad. Uvažujme nyní situaci, 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 naopak 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 vzhledem k velikosti celého problému vcelku nepatrná - jedná se pouze o malou lokální změnu. Z tohoto hlediska je proto opět neefektivní hledat řešení změněného problému úplně od začátku bez využití jednou již vypočtených informací.

Snadno si lze představit, že interakce řešícího systému s uživatelem bude v průběhu hledání řešení problému ještě intenzivnější. Uživatel může učinit rozhodnutí o změnách v řešeném problému již na základě dosavadního průběhu hledání řešení a nečekat až na výsledné řešení či odpověď, že řešení neexistuje. Co nejrychlejší odezva daného řešícího systému na tento druh zásahů ze strany uživatele je tedy rozhodující vlastností.

3.3 Dynamický problém splňování podmínek (DCSP)

Dynamický charakter reálných problémů vedl Rinu Dechter a Aviho Dechtera v [DeDe88] k rozšíření formalizmu omezujících podmínek o pojmem tzv. *dynamického problému splňování podmínek*. Jednoduše řečeno, dynamický problém je problém splňování podmínek, u nějž dochází k častým změnám množiny proměnných a podmínek. Formálně je měnící se problém splňování podmínek modelován jako posloupnost klasických problémů splňování podmínek (splňujících definici 1.1), přičemž proběhnuvší změna je zde zachycena v odlišnosti sousedících členů posloupnosti, resp. jako přechod od jednoho problému k dalšímu. Dynamickým problémem budeme tedy dále rozumět posloupnost problémů, jak je uvedeno v následující definici.

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 často označovány jako *statické*. Informaci udávající způsob získání problému $P_i = (V^i, C^i)$ v posloupnosti z předcházejícího problému $P_{i-1} = (V^{i-1}, C^{i-1})$ 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. Množiny musí ještě splňovat omezení, aby přidávání a odebírání proměnných a podmínek bylo smysluplné, musí tedy platit, že $V^i_{ADD} \cap V^{i-1} = \emptyset$, $V^i_{DEL} \subseteq V^{i-1}$, $C^i_{ADD} \cap C^{i-1} = \emptyset$ a $C^i_{DEL} \subseteq C^{i-1}$. 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. Podobně u operace přidání podmínky musí být zajištěno, že daná podmínka svazuje pouze proměnné už přítomné v problému popřípadě proměnné právě přidávané.

Příklad 3.1.

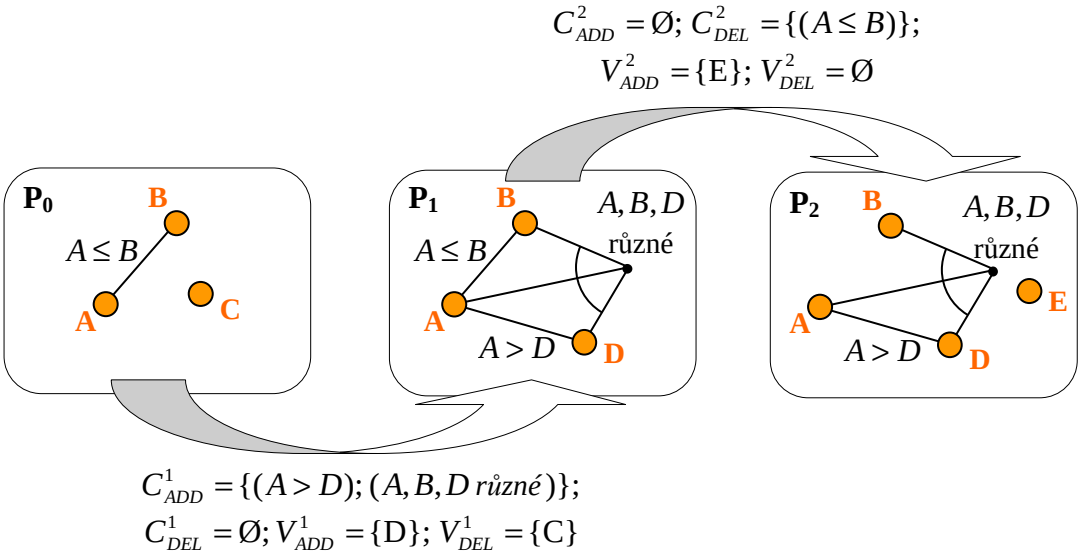
Původní problém: $P_0 = (V_0, C_0)$, kde $V_0 = \{A, B, C\}; C_0 = \{(A \leq B)\}$

Přechod od P_0 k P_1: $V^1_{ADD} = \{\mathrm{D}\}; V^1_{DEL} = \{\mathrm{C}\};$

$C^1_{ADD} = \{(A > D); (A, B, D \textit{ různé})\}; C^1_{DEL} = \emptyset;$

Přechod od P_1 k P_2: $V^2_{ADD} = \{\mathrm{E}\}; V^2_{DEL} = \emptyset;$

$C^2_{ADD} = \emptyset; C^2_{DEL} = \{(A \leq B)\}$. ■



Obrázek 3.1: Dynamický problém splňování podmínek sestávající ze tří statických problémů

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

Malý dynamický problém sestávající ze tří statických problémů P_0, P_1 a P_2 je ukázán v příkladu 3.1 a na obrázku 3.1.

Někdy se kvůli zjednodušení uvažuje první problém posloupnosti dynamického problému P_0 prázdný, tj. neobsahující žádné proměnné ani podmínky.

Kromě popsané formulace dynamického problému splňování podmínek se používá ještě jeden způsob vyjádření dynamičnosti v úlohách postavených na splňování podmínek, kterým se však nebudeme zabývat. Navrhli jej Sanjay Mittal a Brian Falkenhainer v práci [MiFa90], tento formalismus je vhodný zejména pro *konfigurační* úlohy, kde je úkolem například zvolit komponenty určitého výrobku tak, aby co nejlépe vyhovoval daným požadavkům.

3.4 Otázky řešené u dynamických problémů

U dynamických problémů splňování podmínek jsou intenzivně zkoumány úlohy *udržování řešení* a *udržování konzistence*. Tato práce se zabývá prakticky výhradně druhou jmenovanou otázkou.

Úloha udržování řešení v dynamickém problému představuje úkol nalézt řešení každého statického problému tvořícího posloupnost dynamického problému. Pro řešení této úlohy byla navržena řada technik. Thomas Schiex a Gérard Verfaillie navrhují v [SchVe93a] a [VeSch94b] techniku rekonstrukce řešení daného statického problému z řešení předchozího statického problému pomocí lokálních oprav. Další technika od stejných autorů popsaná v článcích [SchVe93a] a [SchVe93b] zaznamenává informace z dosavadního průběhu hledání řešení a ty pak používá k omezování prohledávaného prostoru v dalších fázích. Jiný přístup založený na hledání tzv. robustního řešení, tj. řešení, které zůstává platné i po malých změnách provedených v problému, popisují Richard Wallace a Eugene Freuder v článku [WaFr98]. Podobně navrhují Roman Barták, Tomáš Müller a Hana Rudová v [BMR04]

hledat co nejrobustnějšího řešení v tom smyslu, že toto řešení lze upravit na řešení nového problému (v posloupnosti následujícího) co nejmenším počtem změn.

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 $X \ in \ 0 \ldots \max(Y)$, $Y \ in \ \min(X) \ldots \infty$. Abychom jmenovali aspoň některé autory, otázkami týkajícími se dynamických problémů modelovanými v tomto formalizmu se podrobněji zabývají například Philippe Codognet, Daniel Diaz, Francesca Rossi v práci [GCR99].

Úloha udržování konzistence spočívá ve vyřešení konzistence u každého statického problému v posloupnosti dynamického problému. Pro udržování konzistence v dynamických problémech bylo rovněž navrženo množství technik. Podrobně o této úloze a technikách jejího řešení pojednávají následující části kapitoly.

Obě úlohy, udržování řešení a udržování konzistence, spolu úzce souvisejí. Mnoho algoritmů pro udržování řešení v dynamických problémech používá ke své činnosti techniky pro udržování konzistence. Techniky pro udržování konzistence z dynamických problémů nachází také uplatnění v algoritmech pro hledání řešení statických problémů splňování podmínek. Tímto tématem se zabývají například Narendra Jussien, Romuald Debruyne a Patrice Boizumault v článcích [JuBo97] a [JDB00].

Významnou oblastí aplikace algoritmů pro udržování konzistence je interaktivní řešení problémů splňování podmínek. Typickým představitelem takového interaktivního problému je tvorba rozvrhu (například školního), kdy uživatel postupně upřesňuje omezení kladená na rozvrh nebo naopak omezení zmírňuje pokud řešení neexistuje (nebo jej řešící systém nenalezl ve stanoveném čase), přičemž tento proces probíhá plně interaktivně a jakoby v reálném čase.

S interaktivními problémy se lze dále setkat třeba také v chemii či biologii. Ne příliš odlišným způsobem probíhá například i tvorba plánů na syntézu peptidů, čímž se zabývá článek [JJNVC90] od Janssena, Jégoua, Nouguiera, Vilarema a Castra. Nebo určování sekundárních struktur RNA řetězců, jak je popsují Gaspin a Westhof v [GaWe94]. V obou případech byl k interaktivnímu řešení použit algoritmus pro udržování konzistence.

3.5 Udržování hranové konzistence v dynamických problémech

Hlavní náplní celé této kapitoly bude otázka konzistence v dynamických problémech, přesněji otázka jejího udržování vzhledem k operacím měnících strukturu problému. Jak už bylo řečeno, úloha udržování konzistence v dynamických problémech představuje úkol vyřešit problém konzistence pro každý statický problém v posloupnosti dynamického problému. Jinými slovy, úkolem je vypočítat vzhledem k inkluzi maximální aktuální domény jednotlivých proměnných každého problému v posloupnosti tak, aby tyto problémy byly konzistentní vůči danému druhu konzistence. Z pohledu modifikujících operací je tedy vyžadováno, aby byl vstupní konzistentní problém transformován pomocí dané operace na změněný výstupní problém, který bude opět konzistentní.

Nejvíce prací zbývajících se touto otázkou pojednává o hranové konzistenci. Stejným směrem se budeme ubírat i zde. Zmiňované oblasti praktického použití technik pro udržování konzistence (součást řešících algoritmů pro statické a dynamické problémy, interaktivní problémy) pracovaly také právě s hranovou konzistencí.

V současnosti již existuje řada algoritmů řešících problém dynamické hranové konzistence, nejdůležitější z nich budou popsány v této kapitole. Prvním významným algoritmem z této kategorie byl *DnAC-4* navržený Christianem Bessièrem v práci [Bes91]. Později na algoritmus *DnAC-4* navázal Romuald Debruyne svým podstatně lepším 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, avšak ne méně účinný, pohled na řešení dynamické hranové konzistence uplatňuje algoritmus *AC|DC*, který navrhli Bertrand Neveu a Pierre Berlandier a publikovali jej v [NeBe94].

Po prostudování uvedených algoritmů vybudujeme na základě získaných poznatků nový algoritmus řešící dynamickou hranovou konzistenci. Ten v dalších oddílech teoreticky zanalyzujeme a teoreticky srovnáme jej s existujícími algoritmy. V dalších částech kapitoly se pak vydáme cestou zdokonalování nového algoritmu s tím, že výsledky opět příslušně teoreticky zanalyzujeme.

3.5.1 Operace měnící strukturu problému a dynamická hranová konzistence

Dříve než přejdeme k vlastním algoritmům, je třeba zvlášť zdůraznit některá fakta týkající se operací měnících strukturu problému. 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, efektivní provedení odebrání podmínky vzhledem k hranové konzistenci 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 relaxuje (ulehčí - zmírní omezení), 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 hranové konzistence u dynamických problémů, podobné přímočaré zdůvodnění jako u opačné operace zde není k dispozici. Metody řešení tohoto problému uvidíme v konkrétních algoritmech.

Shrneme-li to, pak přidání podmínky implikuje odebrání hodnot z aktuálních domén, zatímco odebrání podmínek implikuje přidání hodnot do aktuálních domén proměnných.

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 a při skutečné implementaci lze algoritmy snadno upravit, aby nevyžadovaly platnost tohoto předpokladu.

3.5.2 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. Fungování inkrementální hranové konzistence je založeno na faktu, že přidávání podmínek je *monotónní* proces, při kterém jsou hodnoty z domén proměnných pouze vyřazovány.

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. Proměnné definující problém jsou globální (množina proměnných V, množina podmínek C a pole s aktuálními doménami proměnných D), stejná konvence byla užívána i při výkladu konzistenčních algoritmů.

Algoritmus 3.1. PŘIDÁNÍ PODMÍNKY INKREMENTÁLNĚ

procedure ADD-CONSTRAINT-INCREMENTALLY (c)
1 $C \leftarrow C \cup \{c\}$
2 PROPAGATE-AC-3 ($\{c\}$)

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 od začátku.

3.6 Algoritmy pro dynamickou hranovou konzistenci

V této kapitole popíšeme dva algoritmy - DNAC-4 a DNAC-6 - pro plnou dynamickou hranovou konzistenci. Tedy algoritmy, které zvládají i odebírání podmínek. Význačným rysem obou algoritmů je to, že jsou založeny na využití počítadel podpor a seznamů podporovaných hodnot.

3.6.1 Algoritmus DNAC-4

První algoritmus, který řeší odebírání podmínek v dynamických problémech, byl navržen Christianem Bessièrem v práci [Bes91] pod názvem DNAC-4. Algoritmus implementuje plnou dynamickou hranovou konzistenci. To znamená, že jsou dovoleny operace jak přidávání tak odebírání podmínek, přičemž obě operace mohou být nad daným problémem prováděny 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 o zařazení nové podmínky do struktury problému následované spuštěním propagace pomocí AC-4 na přidanou podmínku.

Operace odebrání podmínky je vybudována s využitím datových struktur konzistenčního algoritmu AC-4, tj. s využitím počítadel podpor a seznamů podporovaných hodnot. Vý-

počet 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, ale nelze vystačit pouze s počítadly podpor a seznamy podporovaných hodnot. Proto autor algoritmu DNAC-4 obohacuje konzistenční proceduru AC-4 o udržovaní dalších pomocných informací. Konkrétně jde o informaci udávající důvod odebrání dané hodnoty z aktuální domény proměnné, přičemž tímto důvodem je z pohledu algoritmu DNAC-4 proměnná, kde odebraná hodnota poprvé ztratila všechny podpory. V zavedeném programovém zápise bude tato informace uchovávána v nově zavedené datové struktuře $justification$, která bude reprezentována polem indexovaným proměnnou a její hodnotou. Pro jednotlivé buňky pole $justification$ bude platit, že když algoritmus AC-4 odebral z aktuální domény proměnné v hodnotu d_v, pak buňka $justification[v, d_v]$ bude obsahovat 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 aktualizaci příslušné buňky pole $justification$. Formálně to znamená nahradit všechny výskyty řádku:

$$\vdots$$
$$D[u] \leftarrow D[u] - \{d_u\}$$
$$\vdots$$

řádky

$$\vdots$$
$$D[u] \leftarrow D[u] - \{d_u\}$$
$$justification[v, d_v] \leftarrow v$$
$$\vdots$$

V programovém zápisu algoritmu AC-4 se tato 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

procedure ADD-CONSTRAINT-DNAC-4 (c)
1 $C \leftarrow C \cup \{c\}$
2 $Q_{REVISE} \leftarrow$ INITIALIZE-AC-4 ($\{c\}$)
3 PROPAGATE-AC-4 (Q_{REVISE})

procedure RETRACT-CONSTRAINT-DNAC-4 (c)
1 $\{u,v\} \leftarrow$ proměnné svázané podmínkou c
2 $Q^{uv}_{RESTORE} \leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-4 ((u,v))
3 $Q^{vu}_{RESTORE} \leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-4 ((v,u))
4 $Q_{RESTORE} \leftarrow Q^{uv}_{RESTORE} \cup Q^{vu}_{RESTORE}$
5 $L_{CHECK} \leftarrow \emptyset$
6 $C \leftarrow C - \{c\}$ a zruš záznamy v S a *counter* příslušející podmínce c
7 **while** $Q_{RESTORE} \neq \emptyset$ **do**
8 $(v, d_v) \in Q_{RESTORE}$, libovolná dvojice proměnné a hodnoty
9 $Q_{RESTORE} \leftarrow Q_{RESTORE} - \{(v, d_v)\}$
10 **for each** $(u, d_u) \in S[v, d_v]$ **do**
11 $counter[(u, d_u), v] \leftarrow counter[(u, d_u), v] + 1$
12 **if** $justification[u, d_u] = v$ **and** $d_u \in (D_0[u] - D[u])$ **then**
13 $D[u] \leftarrow D[u] \cup \{d_u\}$
14 $justification[u, d_u] \leftarrow NIL$
15 $Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, d_u)\}$
16 $L_{CHECK} \leftarrow L_{CHECK} \cup \{(u, d_u)\}$
17 RESTORE-ARC-CONSISTENCY-DNAC-4 (L_{CHECK})

```
procedure RESTORE-ARC-CONSISTENCY-DNAC-4 ( L_CHECK )
1       Q_REVISE ← Ø
2       while L_CHECK ≠ Ø do
3           (v, d_v) ∈ L_CHECK , libovolná dvojice proměnné a hodnoty
4           L_CHECK ← L_CHECK − {(v, d_v)}
5           if existuje u takové, že counter[(v, d_v), u] = 0 then
6               D[v] ← D[v] − {d_v}
7               Q_REVISE ← Q_REVISE ∪ {(v, d_v)}
8       PROPAGATE-AC-4 ( Q_REVISE )

function INITIALIZE-ARC-RETRACTION-DNAC-4 ( (u, v) )
1       Q_RESTORE ← Ø
2       for each d_u ∈ (D_0[u] − D[u]) do
3           if justification[u, d_u] = v then
4               D[u] ← D[u] ∪ {d_u}
5               justification[u, d_u] ← NIL
6               Q_RESTORE ← Q_RESTORE ∪ {(u, d_u)}
7       return Q_RESTORE
```

Algoritmus DNAC-4 sestává z procedur ADD-CONSTRAINT-DNAC-4, RETRACT-CONSTRAINT-DNAC-4, RESTORE-ARC-CONSISTENCY-DNAC-4 a funkce INITIALIZE-ARC-RETRACTION-DNAC-4.

Funkce ADD-CONSTRAINT-DNAC-4 realizující přidání podmínky dostává jako parametr přidávanou podmínku c. Kromě toho předpokládá existenci korektně inicializovaných globálních struktur, konkrétně tedy polí V a C obsahující proměnné, resp. podmínky tvořící problém. Dále globálních polí D a D_0 s aktuálními, resp. původními doménami proměnných a nakonec existenci globálních struktur *counter*, S a *justification* s počítadly a seznamy podpor, resp. příčinami odebrání hodnot.

Při přidávání podmínky je kromě samotného zařazení podmínky do problému také nutné vyplnit datové struktury s informacemi o nové podmínce, to se děje voláním procedury

INITIALIZE-AC-4 na řádku 2 procedury ADD-CONSTRAINT-DNAC-4. Po zařazení podmínky do struktur problému je spuštěna konzistenční procedura AC-4 (řádek 3), 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 procedurou RETRACT-CONSTRAINT-DNAC-4. Procedura opět obdržuje jediný parametr a to odebíranou podmínku c. Stejně jako v případě procedury ADD-CONSTRAINT-DNAC-4 pro přidávání podmínky, i zde předpokládáme existenci korektně inicializovaných globálních datových struktur obsahujících podmínky, proměnné, domény atd..

Při obnově aktuálních domén proměnných se algoritmus opírá o závislosti vypočtené modifikovanou konzistenční procedurou AC-4 obsažené ve strukturách $counter$, S a $justification$. Přesně bude proces obnovy aktuálních domén popsán v následujících odstavcích.

Na řádcích 1 až 6 procedury 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 a 3. Funkce v parametru dostává dvojici proměnných (u,v), které svazuje odebíraná podmínka. Výsledkem činnosti funkce INITIALIZE-ARC-RETRACTION-DNAC-4 je navrácení hodnot do aktuální domény proměnné u, za jejichž odebrání byla v minulosti bezprostředně odpovědná proměnná v (cyklus na řádcích 2 až 6). O které konkrétní hodnoty se jedná, je určeno pomocí příčin odebrání obsažených v poli $justification$ (řádek 3).

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. Dvo-

jice proměnné a její hodnoty, jejichž sousedy je třeba zkontrolovat na získání podpory, jsou plánovány do fronty $Q_{RESTORE}$. Přípravu fronty činí již funkce INITIALIZE-ARC-RETRACTION-DNAC-4 (na řádku 6), která ji vrací jako návratovou hodnotu.

Na závěr inicializace operace odebrání podmínky proběhne ve funkci RETRACT--CONSTRAINT-DNAC-4 vyřazení odebírané podmínky z množiny podmínek a pomocných datových struktur. Poté algoritmus pokračuje do hlavní smyčky představující jádro jeho činnosti.

Hlavní smyčka algoritmu DNAC-4 se nachází na řádcích 7 až 16 procedury RETRACT-CONSTRAINT-DNAC-4. Proces obnovy hodnot v hlavní smyčce probíhá tak, že 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 je hodnota d_v podporou (řádek 11). Hodnoty, jimž mají být aktualizována počítadla jsou obsaženy v příslušné buňce pole S (řádek 10). Jestliže je navíc zjištěno, že hodnota z domény proměnné u, které bylo zvýšeno počítadlo podpor, byla odebrána kvůli podmínce svazující proměnné u a v a stále v aktuální doméně proměnné u chybí (řádek 12), je tato hodnota také navrácena. Přitom je opět naplánována revize aktuálních domén sousedních proměnných (řádek 15), jelikož přidané hodnoty se opět mohly stát podporami pro jiné dosud chybějící hodnoty.

Zbývá popsat úlohu seznamu L_{CHECK}, který obsahuje dvojice proměnných a jejích hodnot a procedury RESTORE-ARC-CONSISTENCY-DNAC-4. 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 L_{CHECK}. Tento seznam obsahuje proměnné a jejich hodnoty, které byly obnoveny a mají být znovu prověřeny na hranovou konzistenci algoritmem AC-4.

Tato závěrečná kontrola hranové konzistence probíhá v proceduře RESTORE-ARC-CONSISTENCY-DNAC-4 volané na konci procedury RETRACT-CONSTRAINT-DNAC-4. Nejprve proběhne na řádcích 2 až 7 test všech obnovených hodnot, zda jim nechybí podpora v některé ze sousedních proměnných. Je-li zjištěno, že některé z obnovených hodnot podpora chybí (řádek 5), je tato hodnota odebrána z aktuální domény příslušné proměnné a je naplánována kontrola hodnot v sousedních proměnných. Hodnoty naplánované k další revizi jsou vkládány do fronty Q_{REVISE} na řádku 6. Nakonec je fronta Q_{REVISE} předána algoritmu AC-4 pomocí volání PROPAGATE-AC-4 (Q_{REVISE}) na řádku 8. Toto volání dokončí obnovu maximální hranové konzistence.

Činnost algoritmu DNAC-4 při odebírání podmínky ještě ilustrujeme na příkladě 3.2 a na obrázku 3.2. 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, přítomné hodnoty jsou vyznačeny oranžově. U odstraněné hodnoty je vždy uvedena sousední proměnná, ve které chybí pro odstraněnou hodnotu podpora, resp. první sousední proměnná, kde podpora začala chybět poprvé.

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.

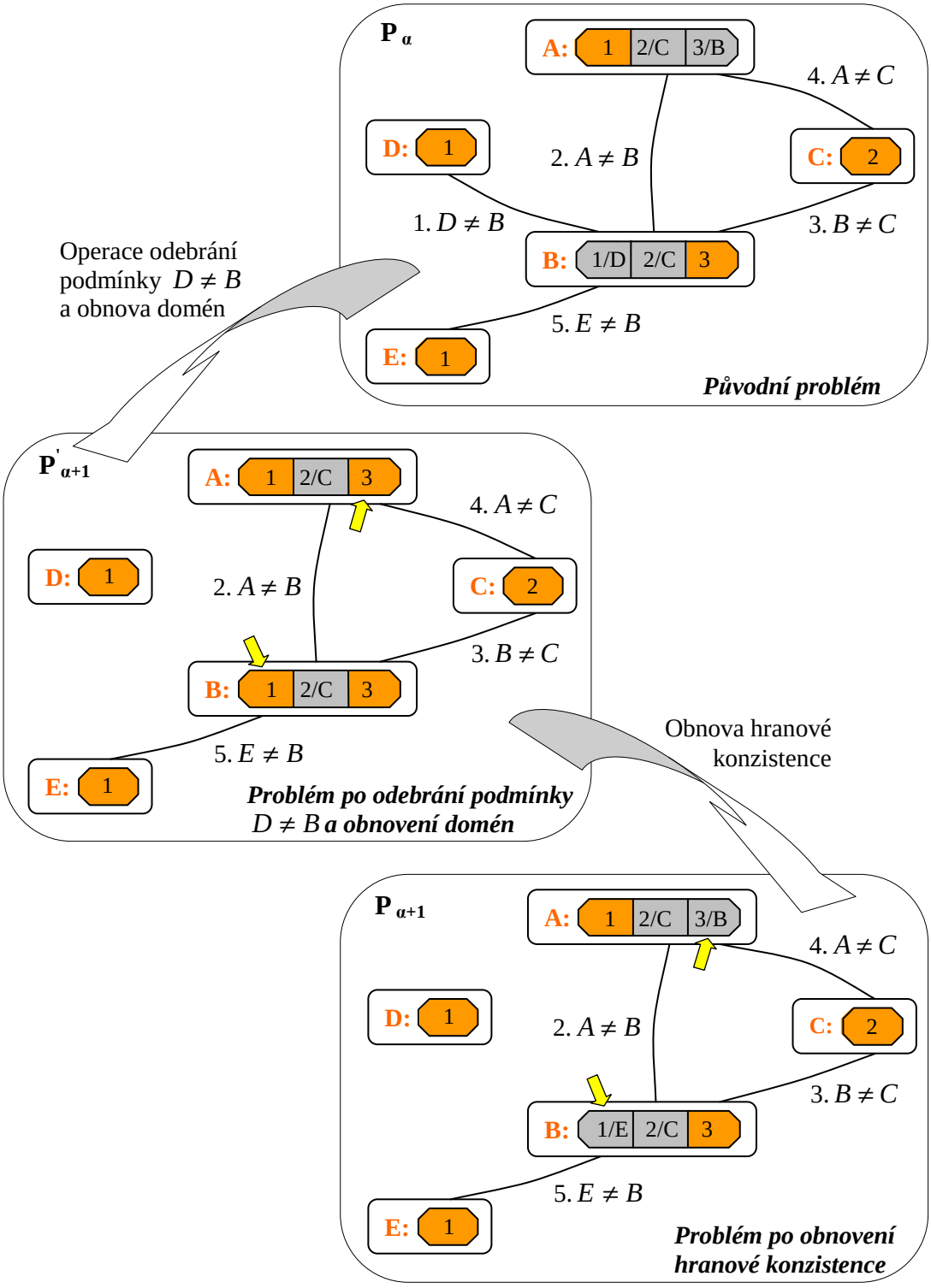
Příklad 3.2.

Původní problém: $P_\alpha = (V_\alpha, C_\alpha)$, kde $V_\alpha = \{A, B, C, D, E\}$;

$C_\alpha = \{(D \neq B); (A \neq B); (B \neq C); (A \neq C); (E \neq B)\}$,

Přechod od P_α k $P_{\alpha+1}$: $V_{ADD}^{\alpha+1} = \emptyset$; $V_{DEL}^{\alpha+1} = \emptyset$;

$C_{ADD}^{\alpha+1} = \emptyset; C_{DEL}^{\alpha+1} = \{(D \neq B)\}$, kde $\alpha \in \mathrm{N}$.■



Obrázek 3.2: 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 maximálně hranově konzistentní problém na výstupní modifikovaný problém, který je opět maximálně 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		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$

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 přidání datové struktury *justification* nenavýšilo asymptotickou prostorovou složitost v nejhorším případě, neboť prostor

$O(\sum_{v \in V} |D_0[v]|)$ potřebný pro pole *justification* lze bez navýšení započítat do asymptotických prostorových nároků konzistenčního algoritmu 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 $\sum_{(u,v) \in C} |D_0[u]| |D_0[v]|$. 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 tak příznivé. 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 datové struktuře S.

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, nebo použít jiný algoritmus.

Jako další nedostatek algoritmu DnAC-4 může být nahlížena i jeho závislost na poměrně speciálních datových strukturách, které znesnadňují jeho případné další adaptace a rozšiřování.

Na tomto místě je asi vhodné uvést, že pro práci s podmínkami vyšší arity než 2 navrhl Christian Bessière v článku [Bes92] algoritmus DnGAC-4. Opět se jedná o algoritmus udržující plnou hranovou konzistenci v dynamických problémech. Jak název napovídá algoritmus vychází ze zobecněného konzistenčního algoritmu GAC-4 od autorů Rogera Mohra a Géralda Masiniho, který je popsán v [MoMa88]. Konzistenční algoritmus GAC-4 navíc oproti AC-4 pracuje i s podmínkami vyšší arity než 2.

Algoritmus DnGAC-4 zde nebudeme rozebírat do podrobností, pro detaily o algoritmu odkážeme čtenáře na zmiňované články. Poznamenejme jenom, že stěžejní část úsilí o zo-

becnění pro nebinární podmínky spočívá ve zobecnění pomocných datových struktur se seznamy a počítadly podpor, resp. příčinami odebrání hodnot.

Zastavme se ale ještě u složitosti algoritmu DNGAC-4. Prostorová složitost je stejně nepříznivá jako u algoritmu DNAC-4, zde je ovšem navíc umocněna vyšší aritou podmínek. Prostorová složitost algoritmu DNGAC-4 v nejhorším případě je $O(\sum_{(v1,v2,..,vn)\in C}\prod_{i=1}^{n}|D_0[v_i]|)$, resp. $O(|C||D|^a)$ pro identické domény, kde a je arita podmínek vyskytující se v daném problému (předpokládáme stejnou aritu všech podmínek, jinak by bylo nutné jako a vzít aritu největší podmínky). Stejně jako u DNAC-4, mohou za vysokou paměťovou náročnost pomocné datové struktury.

Časová složitost přidání a odebrání podmínky algoritmem DNGAC-4 je $O(\sum_{(v1,v2,..,vn)\in C}\prod_{i=1}^{n}|D_0[v_i]|)$, resp. $O(|C||D|^a)$ pro identické domény, kde a opět značí aritu podmínek. To je relativně příznivá hodnota, ovšem i zde platí, že průměrný případ vychází hůře.

3.6.2 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, jehož autorem je Romuald Debruyne. Jádro algoritmu DNAC-6 spočívá v adaptaci postupů užívaných konzistenční procedurou AC-6 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 oproti DNAC-4 je však v operaci odebrání podmínky, ta je realizována s využitím datových struktur algoritmu AC-6 rozšířených o informace o příčinách odebrání hodnot (*justification*). 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. Úkolem algoritmu je také mimo jiné příslušným způsobem tyto datové struktury udržovat v korektním stavu.

Základní myšlenkou algoritmu AC-6 bylo pohlížet na domény proměnných jako na lineárně uspořádané množiny (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 při mění (jedna konkrétní hodnota se může v průběhu práce algoritmu objevit v doméně na různých místech), přičemž jsou dodržována určitá pravidla.

Popišme tento koncept 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 jiných hodnot. Význam tohoto přístupu bude podrobně diskutován později. Prozatím uveďme, že takto lze ušetřit výrazné množství kroků při hledání další podpory.

Vzhledem k tomu, že s doménami nyní bude zacházeno jako se seznamy, je třeba k tomu zavést novou operaci. Připojení prvku na konec seznamu reprezentujícího aktuální doménu bude realizované funkcí APPEND. Funkce APPEND bude dostávat jako parametry hodnotu a aktuální doménu proměnné, přičemž volání APPEND ($d_u, D[u]$) vrátí aktuální doménu $D[u]$ s připojenou hodnotou d_u na konci.

I zde je, stejně jako u minulého algoritmu, nutné mírně adaptovat program výchozí konzistenční procedury AC-6. Aby bylo možné používat AC-6 v dynamickém algoritmu, je třeba do algoritmu AC-6 integrovat datovou strukturu *justification* a její údržbu, v ní budou zase udržovány informace o příčinách odstranění daných hodnot. Datová struktura *justification* je v AC-6 opět reprezentována jako pole a jeho význam je naprosto totožný s tímtéž polem u algoritmu DNAC-4.

Prakticky to znamená nahradit všechny výskyty řádku:

$$\vdots$$
$$D[u] \leftarrow D[u] - \{d_u\}$$
$$\vdots$$

řádky

$$\vdots$$
$$D[u] \leftarrow D[u] - \{d_u\}$$
$$justifications[u, d_u] \leftarrow v$$
$$\vdots$$

V programovém zápisu algoritmu AC-6 se tato náhrada týká řádku 9 funkce INITIALIZE-ARC-AC-6 a řádku 10 procedury 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

procedure ADD-CONSTRAINT-DNAC-6 (c)
1 $C \leftarrow C \cup \{c\}$
2 $Q_{REVISE} \leftarrow$ INITIALIZE-AC-6 ($\{c\}$)
3 PROPAGATE-AC-6 (Q_{REVISE})

procedure RETRACT-CONSTRAINT-DNAC-6 (c)
1 $\{u, v\} \leftarrow$ proměnné svázané podmínkou c
2 $Q^{uv}_{RESTORE} \leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-6 ((u, v))
3 $Q^{vu}_{RESTORE} \leftarrow$ INITIALIZE-ARC-RETRACTION-DNAC-6 ((v, u))
4 $Q_{RESTORE} \leftarrow Q^{uv}_{RESTORE} \cup Q^{vu}_{RESTORE}$
5 $L_{CHECK} \leftarrow \emptyset$
6 $C \leftarrow C - \{c\}$ a zruš záznamy v S a *counter* příslušející podmínce c
$\vdots$

```
⋮
7     while Q_RESTORE ≠ Ø do
8         (v, d_v) ∈ Q_RESTORE , libovolná dvojice proměnné a hodnoty
9         Q_RESTORE ← Q_RESTORE − {(v, d_v)}
10        for each c ∈ C svazující proměnnou v do
11            {u, v} ← proměnné svázané podmínkou c
12            d_u^↑ ← NEXT-SUPPORT-AC-6 ((v, d_v), (u, NIL))
13            if d_u^↑ ≠ NIL then
14                S[u, d_u^↑] ← S[u, d_u^↑] ∪ {(v, d_v)}
15            for each d_u ∈ (D_0[u] − D[u]) do
16                if justification[u, d_u] = v then
17                    if c je splněna pro (d_u, d_v) then
18                        D[u] ← APPEND (d_u, D[u])
19                        S[v, d_v] ← S[v, d_v] ∪ {(u, d_u)}
20                        justification[u, d_u] ← NIL
21                        Q_RESTORE ← Q_RESTORE ∪ {(u, d_u)}
22                        L_CHECK ← L_CHECK ∪ {(u, d_u)}
23    RESTORE-ARC-CONSISTENCY-DNAC-6 (L_CHECK)

procedure RESTORE-ARC-CONSISTENCY-DNAC-6 (L_CHECK)
1     Q_REVISE ← Ø
2     while L_CHECK ≠ Ø do
3         (v, d_v) ∈ L_CHECK , libovolná dvojice proměnné a hodnoty
4         L_CHECK ← L_CHECK − {(v, d_v)}
5         if existuje u takové, že counter[(v, d_v), u] = 0 then
6             D[v] ← D[v] − {d_v}
7             Q_REVISE ← Q_REVISE ∪ {(v, d_v)}
8     PROPAGATE-AC-6 (Q_REVISE)

function INITIALIZE-ARC-RETRACTION-DNAC-6 ((u, v))
1     Q_RESTORE ← Ø
⋮
```

```
⋮
2     for each d_u ∈ (D_0[u] − D[u]) do
3         if justification[u, d_u] = v then
4             D[u] ← APPEND (d_u, D[u])
5             justification[u, d_u] ← NIL
6             Q_RESTORE ← Q_RESTORE ∪ {(u, d_u)}
7     return Q_RESTORE
```

Algoritmus DNAC-6 se skládá z procedur ADD-CONSTRAINT-DNAC-6, RETRACT-CONSTRAINT-DNAC-6 a RESTORE-ARC-CONSISTENCY-DNAC-6 a funkce INITIALIZE-ARC-RETRACTION-DNAC-6.

Přidání podmínky provádí procedura ADD-CONSTRAINT-DNAC-6, tato procedura dostává jako parametr přidávanou podmínku c. Navíc je předpokládána existence korektně inicializovaných globálních struktur - polí V a C uchovávajících problém, dále pole D a D_0 s doménami proměnných a struktur *counter*, S a *justification* s informacemi specifickými pro DNAC-6.

Po zařazení nové podmínky c do struktury problému a aktualizaci datových struktur tak, aby reflektovaly modifikovaný problém (řádky 1 a 2 procedury ADD-CONSTRAINT-DNAC-6), je spuštěna konzistenční procedura AC-6 pro novou podmínku c (řádek 3). Těmito kroky je původně hranově konzistentní problém transformován na hranově konzistentní problém obsahující navíc novou podmínku c.

Odebrání podmínky je prováděno procedurou RETRACT-CONSTRAINT-DNAC-6. Procedura dostává na místě svého jediného parametru odebíranou podmínku c. Opět se předpokládá existence a korektní obsah příslušných globálních datových struktur.

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 parametr dvojici proměnných u a v, jež jsou svázané odebíranou podmínkou. Hodnoty, které je potřeba navrátit do aktuálních domén proměnných u a v (cyklus na řádcích 2 až 6 funkce INITIALIZE-ARC-RETRACTION-DNAC-6), jsou zjištěny pomocí pole *justification*, jež obsahuje příčiny odstranění (řádek 3). Funkce INITIALIZE-ARC-RETRACTION-DNAC-6

je volána dvakrát (řádky 2 a 3 procedury RETRACT-CONSTRAINT-DNAC-6) - 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 (řádek 6), 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 5 procedury RETRACT-CONSTRAINT-DNAC-6, je podmínka c odstraněna z problému a z pomocných datových struktur a přichází ke slovu hlavní smyčka (řádky 7 až 22), v níž je zabudováno jádro celé myšlenky algoritmu.

Jak už bylo řečeno v jednom z předchozích odstavců, 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.

Obdobně jako v algoritmu DNAC-4, je i v DNAC-6 využívána fronta $Q_{RESTORE}$, která obsahuje dvojice proměnných a jejich hodnot, jež byly v minulosti navráceny do aktuálních domén, a je tedy nutné u nich provést revizi sousedů. Cyklus řízený frontou řízená frontou $Q_{RESTORE}$ se nachází na řádcích 7 až 22 procedury RETRACT-CONSTRAINT-DNAC-6. V každém kroku je z fronty $Q_{RESTORE}$ odebrána jedna libovolná dvojice proměnné a hodnoty (řádky 8 a 9), označme ji (v,d_v). Poté je pro každou proměnnou u sousedící prostřednictvím podmínky c s proměnnou v (cyklus na řádcích 10 až 22) uskutečněn pokus vrátit do její aktuální domény všechny hodnoty, které mohly být odstraněny kvůli podmínce c. Přesněji hodnoty, které mohly být odstraněny, kvůli tomu, že v aktuální doméně proměnné v chyběla hodnota d_v. Kandidáti na navrácení do aktuální domény jsou ty hodnoty z množiny

$D_0[u] - D[u]$ (hodnoty chybějící v aktuální doméně proměnné u), u nichž příslušná buňka pole *justification* obsahuje hodnotu v (řádky 15 a 16). Skutečně navrácená hodnota d_u pak musí navíc splňovat, že podmínka c je pro ohodnocení (d_u, d_v) splněna (řádek 17). Po připojení hodnoty d_u na konec seznamu aktuálních hodnot v doméně proměnné u (řádek 18) jsou naplánovány ke kontrole domény sousedních proměnných, zda i v nich nelze obnovit další hodnoty (řádek 21).

Zvláštní komentář si zaslouží aktualizace pomocných datových struktur, konkrétně struktury S na řádcích 13 a 14 a na řádku 19 procedury RETRACT-CONSTRAINT-DNAC-6. 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 hodnotě d_v z domény proměnné v vzhledem k danému hodnot uspořádání v aktuální doméně proměnné u. Po navrácení hodnoty d_v do aktuální domény proměnné v jsou tedy nutné následující aktualizace pole S.

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 jsou ale zajímavé pouze podpory nejmenší vzhledem k danému uspořádání hodnot. 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 $S[u, d_u]$. Aktualizace buňky pole S odpovídající této situaci se odehrává na řádcích 12 a 14 procedury RETRACT-CONS-TRAINT--DNAC-6.

Aby však byly změny v poli S úplné, je třeba také 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 okol-

ností 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 množiny $S[v,d_v]$ přidat dvojici (u,d_u). Právě popsaná aktualizace struktury S, probíhá na řádku 19 procedury 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 struktury S.

Množina podmínek L_{CHECK} (vyplňovaná na řádku 22 procedury RETRACT--CONSTRAINT-DNAC-6) má stejný význam jako v algoritmu DNAC-4, obsahuje dvojice proměnných a hodnot, které byly v průběhu odebírání podmínky obnoveny a které je třeba přezkoumat, zda jim někde nechybí podpora. Toto přezkoumání provádí procedura RESTORE-ARC-CONSISTENCY-DNAC-6 volaná na závěr procesu odebrání podmínky (řádek 23). Když je u nějaké hodnoty detekována chybějící podpora (řádek 5 procedury RESTORE-ARC-CONSISTENCY-DNAC-6), je tato hodnota odebrána a sousedící proměnné jsou naplánovány k dalšímu prověření (řádky 6 a 7). Ke kontrole a případnému vyloučení dalších nekonzistentních hodnot je pak použit konzistenční algoritmus AC-6 (řádek 8).

Výsledkem činnosti procedury RETRACT-CONSTRAINT-DNAC-6 je maximálně hranově konzistentní problém s odebranou podmínkou a korektně aktualizované pomocné datové struktury.

Algoritmus DNAC-6 ve verzi uvedené v [Deb96] používá při závěrečné revizi obnovených hodnot ještě o něco sofistikovanější mechanizmus. U prověřovaných hodnot - v programovém zápise uvedeném zde by to odpovídalo dvojicím proměnná a hodnota v seznamu L_{CHECK} - je 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]. 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 3.2 zobrazující odebírání podmínky algoritmem DNAC-4, neboť průběh samotných změn ve

struktuře řešeného problému je u obou algoritmů stejný (ovšem DNAC-6 dosáhne stejného výsledku výrazně menším počtem kroků).

Korektnost algoritmu DNAC-6 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		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|V\|\|D_0\| + \|C\|\|D\|)$	$O(\|V\|\|D_0\| + \|C\|\|D\|)$
	Různé domény	$O(\sum_{(u,v)\in C}(\|D[u]\| + \|D[v]\|) + \sum_{v\in V}\|D_0[v]\|)$	$O(\sum_{(u,v)\in C}(\|D_0[u]\| + \|D_0[v]\|) + \sum_{v\in V}\|D_0[v]\|)$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|\|D[v]\|)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$

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 značně tíživá. Na každou podmínku a hodnotu v aktuálních doménách proměnných, které svazuje, připadá jedna nejmenší podpora v datových strukturách algoritmu AC-6, což dohromady dává $O(\sum_{(u,v)\in C}(|D[u]|+|D[v]|))$ u přidání podmínky, resp. $O(\sum_{(u,v)\in C}(|D_0[u]|+|D_0[v]|))$ u odebrání podmínky. U obou operací je nutné ještě připočíst prostor $O(\sum_{v\in V}|D_0[v]|)$ potřebný pro uložení pole *justification*.

Časová složitost v nejhorším případě operace přidání podmínky přesně odpovídá časové složitosti konzistenční procedury AC-6. Podívejme se na časovou složitost odebrání podmínky. Inicializace odebrání podmínky svazující proměnné u a v na řádcích 2 a 3 procedury RETRACT-CONSTRAINT-DNAC-6 proběhne v čase $O(|D_0[u]|+|D_0[v]|)$. 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 $O(|D_0[u]||D_0[v]|)$, celkem má tedy hlavní smyčka časovou složitost $O(\sum_{(u,v)\in C}|D_0[u]||D_0[v]|)$. Konzistenční procedura volaná na závěr má složitost stejnou jako zmiňovaná 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 (testů podmínek na splnění pro dvojici hodnot) tak v celkovém čase. DNAC-6 je nejrychlejším z existujících algoritmů pro dynamickou hranovou konzistenci.

Nevýhoda algoritmu DNAC-4 ohledně omezení arity podmínek na hodnotu 2 přetrvává i zde. U algoritmu DNAC-6 však navíc neexistuje žádná jeho zobecněná verze pro podmínky vyšší arity, resp. algoritmus založený na stejné myšlence ale aplikovatelný i na podmínky vyšší arity (pro DNAC-4 zobecněná verze existuje - je to DNGAC-4). Je-li třeba pracovat s

podmínkami vyšší arity, je nutné u algoritmu DNAC-6 aplikovat některou z metod binarizace.

3.6.3 Shrnutí a srovnání vlastností algoritmů DNAC-4 a DNAC-6

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 jsou navrženy pro binární podmínky. To je podmíněno hlavně jejich závislostí na poměrně složitých a speciálních datových strukturách, které ve svém důsledku znesnadňují další modifikace a adaptace obou algoritmů (kam patří i rozšíření pro nebinární podmínky). Rozšíření myšlenky původního algoritmu pro podmínky vyšší arity bylo učiněno jen pro DNAC-4. Výsledkem je algoritmus DN-GAC-4, jehož charakteristikou je ale neúnosně velká paměťová složitost.

3.7 Dynamická hranová konzistence bez seznamů podpor

Jako podstatná nevýhoda se u předchozích algoritmů projevovaly jejich pomocné datové struktury sdružující informace o souvislostech mezi hodnotami v doménách proměnných, tj. počítadla podpor a hlavně seznamy podporovaných hodnot (značné paměťové nároky, náročná aktualizace). Úlohu udržování hranové konzistence v dynamických problémech lze ale zvládnout i bez těchto dodatečných datových struktur, resp. se strukturami méně nákladnými na paměť a snadněji zpracovatelnými, když dojde v problému ke změně.

V tomto oddíle ukážeme jeden existující algoritmus, který pracuje naznačeným způsobem, nepoužívá tedy struktury s počítadly a seznamy podpor. Současně budeme navrhovat nový algoritmus pracující podobným způsobem, který poté budeme analyzovat z hlediska prostorové a časové složitosti a zkoumat další jeho vlastnosti.

3.7.1 AC-3 pozpátku: Algoritmus AC|DC

Cestou snižování paměťových nároků a nároků na zpracování zaznamenaných informací se vydali Bertrand Neveu a Pierre Berlandier, když v článku [NeBe94] navrhli algoritmus AC|DC. 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 ani počítadla 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 proti předchozím algoritmům však spočívá v tom, že k této obnově používá minimum pomocných 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 pozpátku, jinými slovy, propagována nejsou zúžení aktuálních domén, nýbrž jejich rozšíření.

V rámci tohoto oddílu budeme také rozvíjet úplně nový algoritmus, který se ideově algoritmu AC|DC podobá. Pracovně jsme proto nový algoritmus nazvali *AC|DC-2*. V porovnání s algoritmem AC|DC přináší nový algoritmus mnohé časové úspory, přičemž mezi nejvýznačnější patří podstatně příznivější doba 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 zjednodušenou a méně efektivní verzi algoritmu AC|DC publikovaného v práci [NeBe94]. Algoritmus víceméně odpovídají-

cí původnímu AC|DC budeme prezentovat jako *AC|DC-1*. Narozdíl od původního AC|DC budou v AC|DC-1 použity mírně odlišné řídící datové struktury a jiný postup v závěrečné fázi při odstraňování chybně obnovených hodnot. AC|DC-1 bude získán vylepšením algoritmu AC|DC-0, přesněji odstraněním určité neefektivity. Analogicky bude postupováno i u nového AC|DC-2, ovšem zde se bude jednat o přechod od výchozího AC|DC-1.

3.7.2 Zjednodušení algoritmu AC|DC: Algoritmus AC|DC-0

Zjednodušený AC|DC - algoritmus AC|DC-0 - je v zavedeném programovém zápisu uveden jako 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 zadaných aktuálních domény (připomínáme, že přesný popis funkce EXTEND je uveden v předchozí kapitole).

Program je, jako obvykle, kvůli srozumitelnosti formulován pouze pro binární podmínky. Jelikož ale nezávisí na žádných speciálních datových strukturách, které by fungovaly pouze s binárními podmínkami, 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 originální algoritmus AC|DC (AC|DC-1).

Program algoritmu AC|DC-0 předpokládá existenci korektně vyplněných globálních datových struktur definujících problém - tedy polí V, C, a polí s doménami proměnných D, D_0.

Algoritmus 3.4. AC|DC-0

procedure ADD-CONSTRAINT-AC|DC-0 (c)
1 $C \leftarrow C \cup \{c\}$
2 PROPAGATE-AC-3 ($\{c\}$)

procedure RETRACT-CONSTRAINT-AC|DC-0 (c)
1 $\{u,v\} \leftarrow$ proměnné svázané podmínkou c
2 $Q_{RESTORE} \leftarrow \emptyset$
3 $D_u^+ \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-0 ((u,v))
4 $D_v^+ \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-0 ((v,u))
5 **if** $D_u^+ \neq \emptyset$ **then**
6 $\quad D[u] \leftarrow D[u] \cup D_u^+$
7 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{u\}$
8 **if** $D_v^+ \neq \emptyset$ **then**
9 $\quad D[v] \leftarrow D[v] \cup D_v^+$
10 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{v\}$
11 $C \leftarrow C - \{c\}$
12 $C_{REVISE} \leftarrow$ PROPAGATE-ARC-RETRACTION-AC|DC-0 ($Q_{RESTORE}$)
13 PROPAGATE-AC-3 (C_{REVISE})

function INITIALIZE-ARC-RETRACTION-AC|DC-0 ((u,v))
1 $c \leftarrow$ podmínka svazující proměnné u a v
2 $D_u^+ \leftarrow \emptyset$
3 **for each** $d_u \in (D_0[u] - D[u])$ **do**
4 $\quad$ **if not** HAS-SUPPORT (c , (u,d_u) , $(v,D[v])$) **then**
5 $\quad\quad D_u^+ \leftarrow D_u^+ \cup \{d_u\}$
6 **return** D_u^+

function PROPAGATE-ARC-RETRACTION-AC|DC-0 ($Q_{RESTORE}$)
1 $C_{REVISE} \leftarrow \emptyset$
2 **while** $Q_{RESTORE} \neq \emptyset$ **do**
3 $\quad u \in Q_{RESTORE}$, libovolná proměnná
4 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} - \{u\}$
5 $\quad$ **for each** $c \in C$ svazující proměnnou u **do**
6 $\quad\quad \{u,v\} \leftarrow$ proměnné svázané podmínkou c
7 $\quad\quad (D_u^+, D_v^+) \leftarrow$ EXTEND (c , $(D[u], D[v])$)
⋮

```
⋮
8        if D_v^+ ≠ Ø then
9            Q_RESTORE ← Q_RESTORE ∪ {v}
10           D[v] ← D[v] ∪ D_v^+
11       C_REVISE ← C_REVISE ∪ {c ∈ C | c svazuje proměnnou u}
12   return C_REVISE
```

Algoritmus je strukturován do dvou procedur ADD-CONSTRAINT-AC|DC-0, RETRACT-CONSTRAINT-AC|DC-0 a dvou funkcí INITIALIZE-ARC-RETRACTION--AC|DC-0 a PROPAGATE-ARC-RETRACTION-AC|DC-0.

Procedura ADD-CONSTRAINT-AC|DC-0 provádí přidání podmínky, přičemž jako parametr dostává přidávanou podmínku c. Příchozí podmínka je nejprve zapojena do struktury problému (řádek 1) a poté je spuštěn konzistenční algoritmus AC-3 (řádek 2). Původně hranově konzistentní problém procedura transformuje opět na hranově konzistentní s přidanou podmínkou.

Odebrání podmínky má v programu na starosti procedura RETRACT-CONSTRAINT-AC|DC-0 a dvě pomocné funkce. Jejím jediným parametrem je odebíraná podmínka c. Samotné odebrání podmínky zde probíhá ve dvou fázích, podobně 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 týká i algoritmu AC|DC a také nového 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 když dojde 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, která konzistentně rozšiřuje domény proměnných vzhledem k zadané podmínce. V tomto lze spatřovat určitou analogii ke konzistenčnímu

algoritmu AC-3, kde byla podobným způsobem (ovšem v opačné situaci, tj. po zúžení aktuálních domén) opakovaně volána funkce FILTER, která provádí přesný opak toho, co funkce EXTEND.

Inicializace procesu odebírání podmínky je z větší části zajišťována funkcí INITIALIZE-ARC-RETRACTION-AC|DC-0, která je volána na rádcích 3 a 4 procedury RETRACT-CONSTRAINT-AC|DC-0. Odebíráme-li podmínku c, která svazuje proměnné u a v, pak tato funkce vrátí hodnoty, které mají být obnoveny v aktuální doméně proměnné u, resp. v, a jejichž odebrání mohlo být přímo způsobeno podmínkou c. 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 v počáteční fázi běhu algoritmu 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 (řádek 4 funkce INITIALIZE-ARC-RETRACTION-AC|DC-0). 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 také uvedeno v rámci důkazu tvrzení 3.1. Uvědomme si ale, 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.

Po obnově domén přímo zasažených proměnných (řádky 5 až 10 procedury RETRACT-CONSTRAINT-AC|DC-0) a odebrání podmínky z problému (řádek 11) přichází na řadu rozšíření (propagace) změn do zbytku problému (volání na řádku 12).

Propagace počáteční obnovy probíhá ve funkci PROPAGATE-ARC-RETRACTION-AC|DC-0. Funkce dostává parametr frontu $Q_{RESTORE}$, která už byla připravena v rámci inicializace (řádek 7 a řádek 10 procedury RETRACT-CONSTRAINT-AC|DC-0) a která obsahuje proměnné, jejichž aktuální domény byly změněny a je třeba tuto změnu propagovat do sousedů.

Úkolem funkce PROPAGATE-ARC-RETRACTION-AC|DC-0 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 $Q_{RESTORE}$ (řádky 2 až 11). Předpokládejme, že byla z fronty $Q_{RESTORE}$ právě vybrána proměnná u (řádek 4), 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 v rámci cyklu na řádcích 5 až 10. Pro každou podmínku, která svazuje proměnnou u a nějakou další proměnnou, označme ji třeba v, je zavolána operace EXTEND (řádek 7). 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 (řádek 10). Jestliže je tato množina neprázdná (řádek 8), je i proměnná v naplánována do fronty $Q_{RESTORE}$ (řádek 9).

Během celého průběhu obnov aktuálních domén proměnných jsou s ohledem na možnost chybného obnovení hodnot plánovány k opětovné revizi podmínky svazující modifiko-

vané proměnné, k tomuto účelu slouží seznam podmínek C_{REVISE} (vyplňovaného na řádku 11). Tento seznam je nakonec vrácen jako návratová hodnota funkce PROPAGATE-ARC-RETRACTION-AC|DC-0.

Funkce RETRACT-CONSTRAINT-AC|DC-0 je na řádku 13 zakončena spuštěním konzistenční procedury AC-3 pro podmínky obsažené v seznamu C_{REVISE}. Výsledkem je maximálně hranově konzistentní problém s odstraněnou podmínkou.

Ilustraci k průběhu odebírání podmínky algoritmem AC|DC-0 ukazuje následující příklad společně s obrázkem 3.3. Obrázek znázorňuje obě fáze odebírání, tj. obnovu aktuálních domén a následnou opravu chyb pomocí konzistenční procedury.

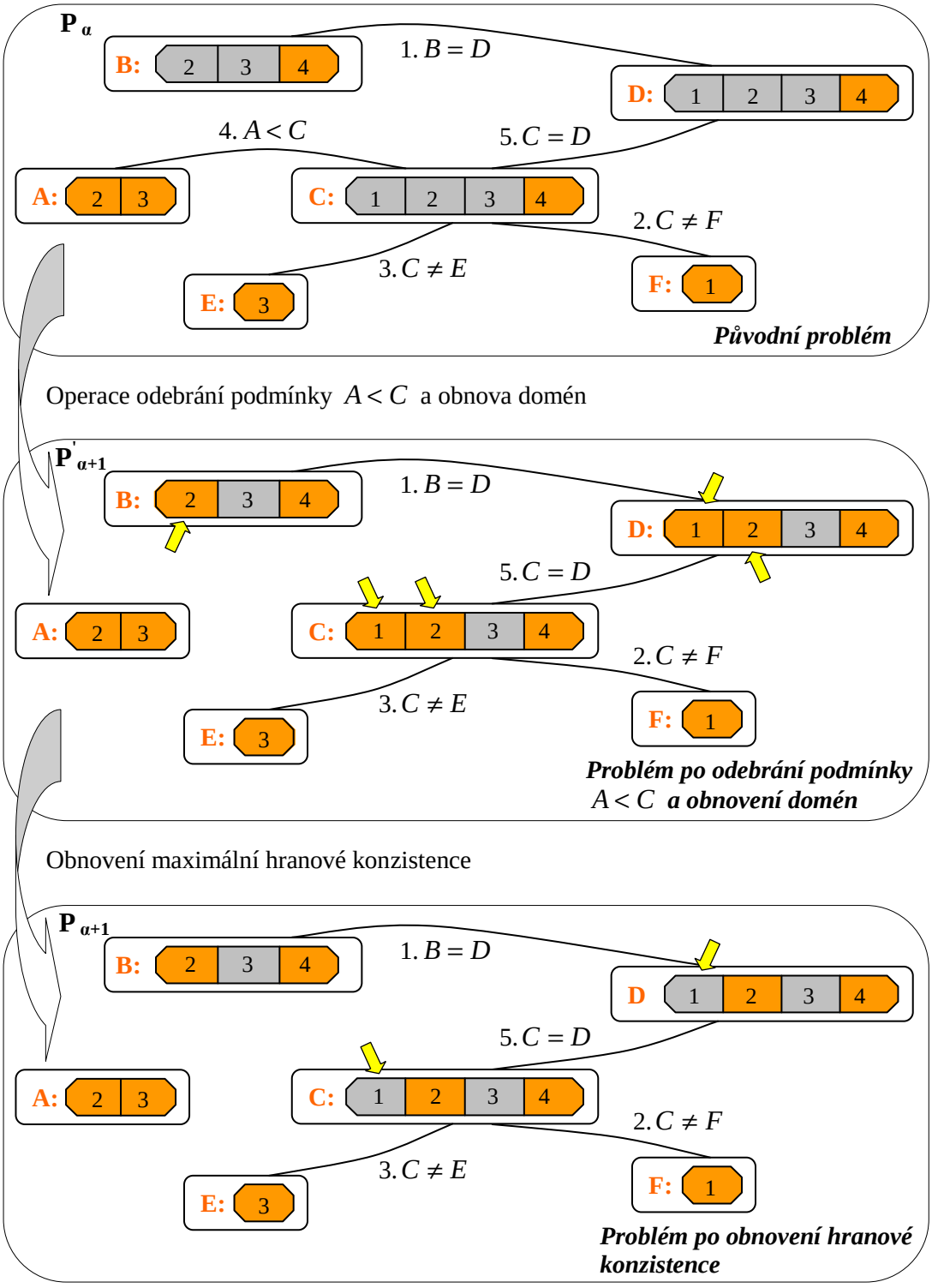
Příklad 3.3.

Původní problém: $P_\alpha = (V_\alpha, C_\alpha)$, kde $V_\alpha = \{A, B, C, D, E, F\}$;

$C_\alpha = \{(B = D); (C \neq F); (C \neq E); (A < C); (C = D)\}$,

Přechod od P_α k $P_{\alpha+1}$: $V_{ADD}^{\alpha+1} = \emptyset$; $V_{DEL}^{\alpha+1} = \emptyset$;

$C_{ADD}^{\alpha+1} = \emptyset; C_{DEL}^{\alpha+1} = \{(A < C)\}$, kde $\alpha \in \mathrm{N}$. ■



Obrázek 3.3: 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 [NeBe94]. 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 maximální hranovou konzistenci problému.* ■

Následující tabulka shrnuje složitost algoritmu AC|DC-0.

AC|DC-0		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|C\|)$	$O(\|V\|+\|C\|)$
	Různé domény	$O(\|C\|)$	$O(\|V\|+\|C\|)$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^3)$	$O(\|C\|\|D\|^3)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|^2\|D[v]\| + \sum_{(u,v)\in C}\|D[u]\|\|D[v]^2\|)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|^2\|D_0[v]\| + \sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]^2\|)$

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 $O(|D_0[u]\|D[v]|+|D[u]\|D_0[v]|)$ 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 $|D_0[u]|+|D_0[v]|$-krát, jelikož každé její spuštění musí být vyvoláno 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) $O(|D_0[u]\|D_0[v]|)$ pro podmínku (u,v), dostaneme pro funkci PROPAGATE-ARC-RET-RACTION--AC|DC-0 čas v nejhorším případě $O(\sum_{(u,v)\in C}(|D_0[u]|+|D_0[v]|)(|D_0[u]\|D_0[v]|))$, což je pro identické domény $O(|C\|D|^3)$.

Včetně následného spuštění konzistenční procedury AC-3 obdržíme pro časovou složitost operace odebírání podmínky výraz $O(|D_0[u]\|D[v]|+|D[u]\|D_0[v]|+$ $+\sum_{(u,v)\in C}(|D_0[u]|+|D_0[v]|)(|D_0[u]\|D_0[v]|))$, to je $O(\sum_{(u,v)\in C}(|D_0[u]|+|D_0[v]|)(|D_0[u]\|D_0[v]|))$, neboli $O(|C\|D|^3)$ pro identické domény.

Nyní se pokusíme algoritmus AC|DC-0 vylepšovat. První vylepšující úprava, kterou provedeme vyústí ve skoro totožný algoritmus s algoritmem AC|DC, proto o něm budeme hovořit jako o téměř původním AC|DC a označovat jej budeme AC|DC-1.

3.7.3 (Téměř) původní algoritmus AC|DC: Algoritmus AC|DC-1

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 vůči nim už také obnoveny.

Právě popsaný způsob odpovídá tomu, jak postupuje v obnovovací fázi publikovaný algoritmus AC|DC. Program algoritmu AC|DC budeme uvádět v upravené formě, kde budou používány jiné řídící datové struktury než v původním AC|DC, z tohoto důvodu budeme algoritmus, aby byl odlišen od originální verze, označovat jako AC|DC-1. Formálně AC|DC-1 popisuje v zavedeném programovém zápise algoritmus 3.5. Krátce se zastavme u rozdílů mezi oběma verzemi algoritmu. Zde uvedená symbolická implementace algoritmu se liší od programu uvedeného v článku [NeBe94] tím, že a autoři algoritmu AC|DC používají zdvojené aktuální domény proměnných, zatímco v AC|DC-1 je použita jen jedna aktuální doména. Při použití zdvojených domén slouží jedna z těchto domén slouží k uchovávání hodnot, které už byly navráceny, ale dosud vůči nim nebyly obnoveny hodnoty v aktuálních doménách sousedních proměnných. Po uskutečnění obnov sousedních proměnných se pak tyto hodnoty také přesunou do klasické aktuální domény proměnné. Originální AC|DC navíc aplikuje závěrečné volání konzistenční procedury je na tuto pomocnou doménu.

V algoritmu AC|DC-1 je místo zdvojených domén proměnných používána strukturovaná řídící fronta (detaily viz. samotný algoritmus). Hodnoty, vůči nimž má proběhnout obnova sousedů, jsou místo do extra aktuální domény vkládány přímo do řídící fronty algoritmu.

Algoritmus 3.5. AC|DC-1

procedure ADD-CONSTRAINT-AC|DC-1 (c)
1 $C \leftarrow C \cup \{c\}$
2 PROPAGATE-AC-3 ($\{c\}$)

procedure RETRACT-CONSTRAINT-AC|DC-1 (c)
1 $\{u,v\} \leftarrow$ proměnné svázané podmínkou c
2 $D_u^+ \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-1 ((u,v))
3 $D_v^+ \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-1 ((v,u))
4 $Q_{RESTORE} \leftarrow \emptyset$
5 **if** $D_u^+ \neq \emptyset$ **then**
6 $\quad D[u] \leftarrow D[u] \cup D_u^+$
7 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, D_u^+)\}$
8 **if** $D_v^+ \neq \emptyset$ **then**
9 $\quad D[v] \leftarrow D[v] \cup D_v^+$
10 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v, D_v^+)\}$
11 $C \leftarrow C - \{c\}$
12 $C_{REVISE} \leftarrow$ PROPAGATE-ARC-RETRACTION-AC|DC-1 ($Q_{RESTORE}$)
13 PROPAGATE-AC-3 (C_{REVISE})

function INITIALIZE-ARC-RETRACTION-AC|DC-1 ((u,v))
1 $c \leftarrow$ podmínka svazující proměnné u a v
2 $D_u^+ \leftarrow \emptyset$
3 **for each** $d_u \in (D_0[u] - D[u])$ **do**
4 $\quad$ **if not** HAS-SUPPORT (c , (u, d_u) , $(v, D[v])$) **then**
⋮

```
⋮
5                 D_u^+ ← D_u^+ ∪ {d_u}
6       return D_u^+
```

function PROPAGATE-ARC-RETRACTION-AC|DC-1 ($Q_{RESTORE}$)

```
1       C_REVISE ← Ø
2       while Q_RESTORE ≠ Ø do
3           (u, D_u^+) ∈ Q_RESTORE, libovolná dvojice
4           Q_RESTORE ← Q_RESTORE − {(u, D_u^+)}
5           for each c ∈ C svazující proměnnou u do
6               {u, v} ← proměnné svázané podmínkou c
7               D_v^restore ← Ø
8               for each d_v ∈ (D_0[v] − D[v]) do
9                   if HAS-SUPPORT (c, (v, d_v), (u, D_u^+)) then
10                      D_v^+ ← D_v^+ ∪ {d_v}
11              if D_v^+ ≠ Ø then
12                  D[v] ← D[v] ∪ D_v^+
13                  Q_RESTORE ← Q_RESTORE ∪ {(v, D_v^+)}
14          C_REVISE ← C_REVISE ∪ {c ∈ C | c svazuje proměnnou u}
15      return C_REVISE
```

Program algoritmu AC|DC-1 sestává z procedur ADD-CONSTRAINT-AC|DC-1 a RETRACT-CONSTRAINT-AC|DC-1 a dvou funkcí INITIALIZE-ARC-RETRACTION--AC|DC-1 a PROPAGATE-ARC-RETRACTION-AC|DC-1. Jejich význam přesně odpovídá obdobným funkcím z programu algoritmu AC|DC-0, 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 v její doméně obnovených hodnot naplánována do strukturované fronty $Q_{RESTORE}$. Ve frontě $Q_{RESTORE}$ se proměnná v

plus navíc dodatečný seznam D_v^+ hodnot obnovených v její aktuální doméně nachází (jako dvojice) právě tehdy, když je potřeba zkontrolovat, zda díky této změně nezískaly podpory další chybějící hodnoty v proměnných sousedících s v.

V hlavní smyčce algoritmu (řádky 2 až 14 funkce PROPAGATE-ARC-RETRACTION-AC|DC-1) je pak pokaždé z fronty $Q_{RESTORE}$ odebrán jeden libovolný záznam (řádky 3 a 4), pro tuto chvíli jej označme (u, D_u^+), což je následováno uskutečněním pokusu o navrácení dosud chybějících hodnot do všech aktuálních domén proměnných sousedících s u (cyklus na řádcích 5 až 14). Navrácení hodnoty je testováno pouze vzhledem k hodnotám v množině D_u^+, jinými slovy, jestliže je v tomto kroku obnovena nějaká hodnota, pak je to díky tomu, že získala podporu mezi hodnotami v D_u^+. Testování, zda má daná hodnota podporu v množině hodnot vzhledem k určité podmínce, je přenecháno na specializovanou operaci HAS-SUPPORT (řádek 9).

Tím, že jsou podpory hledány pouze vzhledem k obnoveným hodnotám, je ušetřeno významné množství kroků a také celkový čas běhu (jak uvidíme v další kapitole v experimentech). Konkrétní množina obnovených hodnot je ale stejná jako u AC|DC-0.

Korektnost algoritmu AC|DC-1 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 maximálně hranově konzistentní problém na modifikovaný problém, který je opět maximálně hranově konzistentní.* ■

Pro důkaz věty odkazujeme na článek [NeBe94] a na srovnání symbolických zápisů obou algoritmů - původního AC|DC a AC|DC-1.

Už z pouhého pohledu na programový zápis algoritmu AC|DC-1 je ale jasné, 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		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|C\|)$	$O(\|V\|\|D\|+\|C\|)$
	Různé domény	$O(\|C\|)$	$O(\sum_{v\in V}\|D_0[v]\|+\|C\|)$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^3)$	$O(\|C\|\|D\|^3)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|^2\|D[v]\| + \sum_{(u,v)\in C}\|D[u]\|\|D[v]\|^2)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|^2\|D_0[v]\| + \sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|^2)$

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 $\sum_{v\in V}|D_0[v]|$. 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.

Časová složitost v nejhorším případě operace odebrání podmínky zůstává stejná. Avšak obnovovací fáze operace odebrání podmínky nyní nepotřebuje čas $O(\sum_{(u,v)\in C}(|D_0[u]|+|D_0[v]|)(|D_0[u]||D_0[v]|))$, ale vystačí pouze s $O(\sum_{(u,v)\in C}|D_0[u]||D_0[v]|)$. K odůvodnění tohoto výsledku je nezbytné učinit pozorování, že každá dvojice hodnot v doménách sousedících proměnných je testována nejvýše jedenkrát. Tento výsledek platí pro výchozí implementaci operace HAS-SUPPORT, která pracuje přímo s dvojicemi kompatibilních hodnot (kdybychom použili specializovanou implementaci pro určitý speciální druh podmínek, byly by výsledky ještě pozitivnější). Celkovou časovou složitost operace odebrání podmínky nakonec ale pokazí konzistenční procedura AC-3 volaná na závěr.

Jak sami autoři algoritmu AC|DC podotýkají, mezi kladné 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 složitějších datových struktur i možnost implementace algoritmu pro podmínky vyšší arity.

3.7.4 Shrnutí vlastností algoritmů AC|DC a možnosti zlepšení

Nedostatek algoritmu AC|DC-1 představuje jeho poměrně značná teoretická časová složitost v nejhorším případě způsobená konzistenční procedurou AC-3. Ovšem zde je třeba zdůraznit, 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 (AC-4 či AC-6). Nicméně taková náhrada jde zásadně proti myšlence algoritmu AC|DC, která spočívá v jednoduché implementaci a nezávislosti na seznamech podpor.

Proto je mnohem přirozenější poohlédnout se například po algoritmu AC-2001 od Bessièra a Régina popsaném v [BeRe01] či po AC-3.1, o kterém pojednávají Zhang a Yap v práci [ZhYa01]. Oba algoritmy ideově vycházejí z AC-3, ale zlepšují jeho časovou složitost v nejhorším případě na optimální hodnotu $O(|C||D|^2)$ pro problémy s identickými doménami.

Druhou z jmenovaných možností využil Malek Mouhoub v [Mo93]. Jednoduše nahradil v AC|DC konzistenční proceduru AC-3 optimálním algoritmem AC-3.1. Výsledný algoritmus pojmenovaný AC3.1|DC se vyznačuje kratší dobou běhu závěrečné fáze vyřazující nesprávně obnovené hodnoty (výstižněji je to vidět v kapitole věnované experimentálnímu srovnání algoritmů). Je toho však docíleno za cenu vyšší paměťové složitosti. To je zapříčiněno konzistenční procedurou AC-3.1, která má větší paměťové nároky než AC-3 a jejíž paměťová složitost je $O(|C||D|)$.

Autor v článku [Mo93] uvádí paměťovou složitost algoritmu AC3.1|DC v nejhorším případě pro odebrání podmínky $O(|V||D|+|C|)$. My se však domníváme, že jde o chybu a že skutečná paměťová složitost je $O(|V||D|+|C||D|)$, za předpokladu, že domény proměnných stejně velké.

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 díky těmto testům (úspora spočívá v tom, že se obnovuje menší počet hodnot).

3.7.5 Nový algoritmus typu AC|DC: Algoritmus AC|DC-2

Neefektivitu způsobenou nepřesností ve fázi obnovování aktuálních domén považujeme za zásadní. Pokud by se nám ji podařilo odstranit, výsledkem by mohl být relativně výkonný a úsporný (máme na mysli srovnání s existujícími algoritmy DnAC-4 a DnAC-6), ovšem na druhou stranu také jednoduchý a rozšiřitelný algoritmus.

Na řešení uvedené neefektivity obnovovací fáze ale nahlížíme z docela jiného pohledu než autoři algoritmu AC|DC. Hlavní myšlenkou našeho nového přístupu je nějakým způsobem využít při obnově aktuálních domén informací, které by během své činnosti mohla vyprodukovat konzistenční procedura, v našem případě procedura AC-3.

Přesněji řečeno, při propagaci konzistenční procedurou AC-3 se budeme snažit zaznamenávat určité užitečné informace o jejím běhu, které později pomohou k přesnějšímu navracení hodnot do aktuálních domén proměnných a tím také k citelnému zefektivnění obnovovací fáze operace odebrání podmínky.

Během činnosti konzistenční procedury AC-3 budeme pro každou odebranou hodnotu zaznamenávat příčinu jejího odstranění, tj. sousední proměnnou, ve které odstraněná hodnota poprvé ztratila všechny podpory (stejnou informaci zaznamenávají i algoritmy DNAC-4 a DNAC-6).

Další informací, kterou budeme uchovávat, je čas odstranění každé chybějící hodnoty. K tomu pochopitelně není potřeba zaznamenávat fyzikální čas. Požadujeme jen, aby každé odebrání hodnoty proběhlo v jiném okamžiku (čase) a aby byly tyto události lineárně uspořádané. Čas tedy můžeme pro naše potřeby modelovat například pomocí globálního počítadla kroků programu.

Informaci o proměnné, kde hodnota poprvé ztratila všechny podpory a čase, kdy k tomu došlo, budeme, podobně jako u algoritmů DNAC-4 a DNAC-6, uchovávat v poli *justification*, jež bude indexováno dvojicí proměnná a její hodnota. Každá buňka tohoto pole bude mít nyní ale dvě složky - variable reprezentující proměnnou, která zapříčinila odebrání hodnoty a time uchovávající čas odebrání této hodnoty (tedy hodnoty figurující v indexu pole).

Dříve než přejdeme k vlastnímu algoritmu pro dynamickou hranovou konzistenci, je však nutné upravit konzistenční proceduru AC-3 tak, aby odrážela uvedené požadavky, tedy, aby odpovídajícím způsobem vyplňovala položky pole *justification*.

Konzistenční proceduru AC-3 doplníme o globální datovou strukturu *justification* 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 nutnou podmínkou, později v příloze ukážeme odpovídající rozšíření zahrnující též podmínky arity vyšší než 2.

Algoritmus 3.6. AC-3'

```
procedure PROPAGATE-AC-3' (C_REVISE)
1       Q_REVISE ← C_REVISE
2       while Q_REVISE ≠ Ø do
3       |       c ∈ Q_REVISE , libovolná podmínka
⋮       ⋮
```

```
⋮
4        Q_REVISE ← Q_REVISE − {c}
5        {u,v} ← proměnné svázané podmínkou c
6        (D_u^-, D_v^-) ← FILTER (c, (D[u], D[v]))
7        C_REVISE^uv ← FILTER-ARC-AC-3' ((u,v), D_u^-)
8        C_REVISE^vu ← FILTER-ARC-AC-3' ((v,u), D_u^-)
9        Q_REVISE ← Q_REVISE ∪ C_REVISE^uv ∪ C_REVISE^vu
```

```
function FILTER-ARC-AC-3' ((u,v), D_u^-)
1    C_REVISE ← Ø
2    c ← podmínka svazující proměnné u a v
3    if D_u^- ≠ Ø then
4        for each d_u ∈ D_u^- do
5            D[u] ← D[u] − {d_u}
6            justification[u, d_u].variable ← v
7            justification[u, d_u].time ← time
8            time ← time + 1
9        C_REVISE ← C_REVISE ∪ {e ∈ C | e svazuje proměnnou u & e ≠ c}
10   return C_REVISE
```

Z důvodů lepší čitelnosti byl algoritmus více strukturován, skládá se z procedury PROPAGATE-AC-3' a pomocné funkce FILTER-ARC-AC-3'. Význam procedury PROPAGATE-AC-3' přesně odpovídá proceduře PROPAGATE-AC-3 z algoritmu AC-3. Pomocná funkce FILTER-ARC-AC-3' zpracovává odebrání hodnot z aktuální domény proměnné a aktualizuje dodatečné informace v datové struktuře *justification* vzhledem k podmínce v jednom směru (tedy vzhledem ke hraně). Funkce je pro každou zasaženou podmínku volána zvlášť pro každý směr (řádky 7 a 8 procedury PROPAGATE-AC-3').

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.

Tyto informace jsou zaznamenávány na řádcích 6 a 7 funkce FILTER-ARC-AC-3'. Všimněme si ještě aktualizace globálního času na řádku 8.

Co se týče časové složitosti algoritmu nedošlo k žádné změně. Prostorová složitost ale narostla o paměť potřebnou k uložení pole *justification*, konkrétně se jedná o navýšení úměrné $O(\sum_{v \in V} |D_0[v]|)$, neboli $O(|V||D|)$ pro identické domény, protože ukládáme konstantně velká data pro každou hodnotu (původní domény).

Nový algoritmus pro dynamickou hranovou konzistenci jsme pojmenovali AC|DC-2. Algoritmus ideově vychází z AC|DC-1, avšak při obnovách aktuálních domén proměnných postupuje oproti AC|DC-1 opatrněji. Dodatečná režie opatrnějšího postupu je v tomto případě málo významná a navíc je bohatě vyvážena menším počtem navracených hodnot, tedy větším množstvím celkově 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 dodatečné informace do pole *justification*.

Základní schéma postupu při odebírání podmínky je prakticky totožné s ostatními popsanými algoritmy. 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 reverzní propagace změn (propagace zúžení domén). Operace odebrání podmínky je nakonec uzavřena voláním konzistenční procedury AC-3', která vyloučí chybně přidané hodnoty z předchozí fáze.

Jádrem nového algoritmu je způsob, jakým jsou obnovovány hodnoty v aktuálních doménách proměnných. Podívejme se tedy nyní na tento proces podrobněji.

V první řadě jsou obnoveny hodnoty v aktuálních doménách proměnných, které svazovala odebíraná podmínka. Jestliže odebíraná podmínka svazovala proměnné u a v, pak jsou do aktuální domény proměnné u navráceny všechny hodnoty, jež nemají v aktuální doméně proměnné v žádnou podporu a jejichž odebrání zapříčinila proměnná v, tj. v poli *justification* měla buňka náležející hodnotě testované na navrácení složku variable rovnou v. Analogicky je celý postup aplikován i na proměnnou v (v opačném směru podmínky). Jestliže dojde ke změně aktuální domény některé z proměnných u nebo v, je napláno-

ván pokus o obnovu jejích sousedů, neboť hodnoty dosud chybějící v aktuálních doménách těchto proměnných mohly díky provedené změně získat novou podporu.

Induktivně nyní předpokládejme, že algoritmus pokračoval a že právě zjistil, že do aktuální domény proměnné u (u je teď nová proměnná, ne ta z předchozího odstavce) byla v některém z minulých kroků navrácena množina hodnot D_u^+. V tomto okamžiku je třeba uskutečnit pokus o navrácení dalších hodnot do aktuálních domén všech sousedních proměnných. Kandidáti pro navrácení do aktuální domény sousední proměnné, nazvěme ji v (opět nová proměnná), sousedící s u jsou ty dosud chybějící hodnoty, jejichž odebrání zapříčinila proměnná u, tj. složka variable příslušné buňky pole *justification* (příslušející proměnné v) obsahuje proměnnou u, a jež byly odstraněny až po odstranění v minulosti nejdříve odstraněné hodnoty z množiny D_u^+. Formálně řečeno, čas ve složce time buňky pole *justification* náležející hodnotě kandidující na navrácení musí být větší než nejmenší čas uložený v buňkách pole *justification* patřící hodnotám z množiny D_u^+. Nakonec skutečně navrácená hodnota do aktuální domény proměnné v musí mít ještě navíc alespoň jednu podporu mezi hodnotami z množiny D_u^+.

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

procedure ADD-CONSTRAINT-AC|DC-2 (c)
1 $C \leftarrow C \cup \{c\}$
2 PROPAGATE-AC-3' ($\{c\}$)

procedure RETRACT-CONSTRAINT-AC|DC-2 (c)
1 $\{u,v\} \leftarrow$ proměnné svázané podmínkou c
2 $(time_u, D_u^+) \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2 (c, (u,v))
3 $(time_v, D_v^+) \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2 (c, (v,u))
4 $Q_{RESTORE} \leftarrow \emptyset$
⋮

```
⋮
5     if D_u^+ ≠ Ø then
6         D[u] ← D[u] ∪ D_u^+
7         Q_RESTORE ← Q_RESTORE ∪ {(u, time_u, D_u^+)}
8     if D_v^+ ≠ Ø then
9         D[v] ← D[v] ∪ D_v^+
10        Q_RESTORE ← Q_RESTORE ∪ {(v, time_v, D_v^+)}
11    C ← C − {c}
12    C_REVISE ← PROPAGATE-ARC-RETRACTION-AC|DC-2 (Q_RESTORE)
13    PROPAGATE-AC-3' (C_REVISE)

function INITIALIZE-ARC-RETRACTION-AC|DC-2 ((u, v))
1     c ← podmínka svazující proměnné u a v
2     time_u ← ∞
3     D_u^+ ← Ø
4     for each d_u ∈ (D_0[u] − D[u]) do
5         if justification[u, d_u].variable = v then
6             if not HAS-SUPPORT (c, (u, d_u), (v, D[v])) then
7                 D_u^+ ← D_u^+ ∪ {d_u}
8                 time_u ← MIN (time_u, justification[u, d_u].time)
9                 justification[u, d_u] ← NIL
10    return (time_u, D_u^+)

function PROPAGATE-ARC-RETRACTION-AC|DC-2 (Q_RESTORE)
1     C_REVISE ← Ø
2     while Q_RESTORE ≠ Ø do
3         (u, time_u, D_u^+) ∈ Q_RESTORE, libovolná trojice
4         Q_RESTORE ← Q_RESTORE − {(u, time_u, D_u^+)}
5         for each c ∈ C svazující proměnnou u do
6             {u, v} ← proměnné svázané podmínkou c
7             time_v ← ∞
8             D_v^+ ← Ø
⋮
```

```
⋮
9     for each d_v ∈ (D_0[v] − D[v]) do
10        if justification[v, d_v].variable = u and
11           justification[v, d_v].time > time_u and
12           HAS-SUPPORT (c, (v, d_v), (u, D_u^+))
13        then
14           D_v^+ ← D_v^+ ∪ {d_v}
15           time_v ← MIN ( justification[v, d_v].time,
16                          time_v )
17           justification[v, d_v] ← NIL
18    if D_v^+ ≠ ∅ then
19        D[v] ← D[v] ∪ D_v^+
20        Q_RESTORE ← Q_RESTORE ∪ {(v, time_v, D_v^+)}
21    C_REVISE ← C_REVISE ∪ {c ∈ C | c svazuje proměnnou u}
22 return C_REVISE
```

Přidání podmínky obstarává procedura ADD-CONSTRAINT-AC|DC-2, jejímž parametrem je přidávaná podmínka c. Po přidání podmínky do problému (řádek 1) je k obnovení hranové konzistence aplikována konzistenční proceduru AC-3' zavoláním funkce PROPAGATE-AC-3' (řádek 2). Výsledkem činnosti funkce ADD-CONSTRAINT--AC|DC-2, je hranově konzistentní problém s přidanou podmínkou c a korektně aktualizované pole *justification*.

Procedura RETRACT-CONSTRAINT-AC|DC-2 společně s funkcemi INITIALIZE-ARC-RETRACTION-AC|DC-2 a PROPAGATE-ARC-RETRACTION-AC|DC-2 uskutečňují odebrání podmínky.

Procedura RETRACT-CONSTRAINT-AC|DC-2 dostává jediný parametr, a to odebíranou podmínku c. Počáteční navrácení chybějících hodnot do aktuálních domén proměnných svázaných odebíranou podmínkou je, jak už je pro algoritmy typu AC|DC v této kapitole obvyklé, prováděno funkcí INITIALIZE-ARC-RETRACTION-AC|DC-2 (volána na řádcích 2 a 3 procedury RETRACT-CONSTRAINT-AC|DC-2). Funkce je pro danou pod-

mínku 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 v rámci funkce INITIALIZE-ARC-RETRACTION-AC|DC-2 prováděna ještě s ohledem na příčinu odebrání chybějící hodnoty. Jestliže některá z hodnot, které v aktuální doméně zainteresované proměnné chybí (řádek 4 funkce INITIALIZE-ARC-RETRACTION-AC|DC-2), splňuje podmínky pro navrácení (řádek 5 a řádek 6), je tato hodnota zařazena do množiny hodnot, které budou navráceny (řádek 7). Zároveň je průběžně počítán čas v minulosti nejdříve odstraněné hodnoty z těchto navracených (řádek 8). Přitom je také zrušen záznam v poli *justification* pro hodnotu, která bude obnovena (řádek 9). Množina hodnot, které mají být vloženy do aktuální domény dané proměnné a čas nejdřívějšího odebrání některé z nich, jsou funkcí INITIALIZE-ARC-RETRACTION-AC|DC-2 vráceny v návratové hodnotě.

Počáteční inicializace odebrání podmínky pak ještě pokračuje na řádcích 4 až 10 procedury RETRACT-CONSTRAINT-AC|DC-2, kde jsou hodnoty připravené ve funkci INITIALIZE-ARC-RETRACTION-AC|DC-2 vloženy do aktuálních domén proměnných (řádky 6 a 9) a kde také probíhá příprava řídící fronty $Q_{RESTORE}$ (řádky 7 a 10 - vysvětlení obsahu fronty bude následovat). Poté je podmínka odstraněna z problému (řádek 11) a algoritmus pokračuje voláním funkce PROPAGATE-ARC-RETRACTION-AC|DC-2 (řádek 12), která rozšíří dopad provedených změn do okolních proměnných.

Funkce PROPAGATE-ARC-RETRACTION-AC|DC-2 zajišťuje postupnou obnovu hodnot v aktuálních doménách proměnných v minulosti zasažených odebíranou podmínkou. Funkce je v podstatě představována cyklem řízeným frontou $Q_{RESTORE}$. Fronta $Q_{RESTORE}$ obsahuje trojice proměnná, čas a množina hodnot. Pokaždé, když je rozšířen obsah aktuální domény nějaké proměnné o určitou množinu hodnot, je zároveň vyhledána hodnota z této množiny, která měla nejmenší čas odebrání. Byla-li do aktuální domény proměnné u navrácena množina hodnot D_u^+ a $time_u$ je čas odebrání hodnoty, jež byla z hodnot v množině D_u^+ v minulosti odstraněna nejdříve, je do fronty $Q_{RESTORE}$ vložena trojice $(u, time_u, D_u^+)$.

Dokud je fronta $Q_{RESTORE}$ neprázdná, je z ní v hlavní smyčce funkce PROPAGATE-ARC-RETRACTION-AC|DC-2 (řádky 2 až 21) pokaždé vyjmuta jedna libovolná trojice (řádek 3), označme ji $(u, time_u, D_u^+)$. Po odstranění trojice z fronty $Q_{RESTORE}$ je uskutečněn

pokus o navrácení dalších dosud chybějících hodnot do aktuálních domén proměnných sousedících s proměnnou u (cyklus probíhající sousedy se nachází na řádcích 5 až 20). Kandidáti na navrácení jsou pouze ty hodnoty, za jejichž odstranění je odpovědná proměnná u (příslušná buňka pole *justification* obsahuje ve složce variable proměnnou u - řádek 10) a jejichž čas odstranění je větší než $time_u$ (odpovídající buňka v poli *justification* má složku time větší než $time_u$ - řádek 11). Skutečně navrácená hodnota pak navíc musí mít ještě aspoň jednu podporu v množině D_u^+ (řádek 12). K hledání podpor je použita specializovaná funkce HAS-SUPPORT.

Při realizaci podmíněného příkazu na řádcích 10 až 12 je výhodné použít tzv. líné vyhodnocování, kdy na náročnější test s funkcí HAS-SUPPORT nemusí vůbec dojít za předpokladu že předchozí predikáty nejsou splněny.

Během celého obnovovacího procesu jsou shromažďovány podmínky svazující obnovené proměnné do seznamu C_{REVISE} (řádek 12), který je funkcí PROPAGATE-ARC--RETRACTION-AC|DC-2 vrácen jako návratová hodnota.

Seznam C_{REVISE} je nakonec předán jako parametr konzistenční proceduře AC-3', jež v závěru funkce RETRACT-CONSTRAINT-AC|DC-2 (na řádku 13) vyloučí případné chybně navrácené hodnoty.

Výsledkem činnosti algoritmu je problém s odebranou podmínkou a znovu spočtenou hranovou konzistencí a aktualizované pomocné datové struktury.

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 maximální 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 nutné obnovit. Pokud navíc kromě těchto hodnot obnoví algoritmus ještě nějaké další, nevadí to, neboť na závěr operace odebrání pod-

mínky je spuštěna konzistenční procedura AC-3', která zkontroluje všechny přidané hodnoty a ty z nich, které byly přidány navíc, 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, kde $i\in\{1,2,\ldots,k\}$. 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, který by vznikl 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 byla podmínka c_i 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, než které svazuje podmínka c_i, jsou navráceny při počáteční obnově hodnot funkcí INITIALIZE-ARC-RETRACTION-AC|DC-2. To je třeba dokázat nejdříve.

Čas odebrání poslední z těchto hodnot označme $t_0+\Delta t$. Pro všechny hodnoty odstraněné v čase od t_0 do $t_0+\Delta t$ 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. 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 (řádky 5 a 6 funkce INITIALIZE-ARC-RETRACTION--AC|DC-2), jsou jistě navráceny i hodnoty, jež byly v minulosti odebrány v čase od t_0 do $t_0 + \Delta t$. Tím je ověřen počáteční krok důkazu matematickou indukcí.

Nyní budeme v induktivní úvaze pokračovat. 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 byla hodnota d_u odstraněna v čase t_1. K provedení indukčního kroku předpokládejme, že $t_1 > t_0 + \Delta t$. Z indukčního předpokladu dále platí, že všechny hodnoty, jež měly být obnoveny a jejichž čas odebrání byl menší než t_1, 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_1. Jelikož je ale hodnota d_u v aktuální doméně proměnné u u problému Q přítomna, znamená to, že u problému Q musí být 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_1, který je větší než u obnovených podpor, a získala podporu v proměnné v - řádky 10 až 12 funkce PROPAGATE-ARC-RETRACTION-AC|DC-2), 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 na dynamickém problému z příkladu 3.4. je znázorněn na obrázku 3.4. Příčiny a čas odebrání jednotlivých hodnot jsou vyznačeny v doménách ve

tvaru hodnota/*proměnná-čas*, kde údaje za lomítkem označují sousední proměnnou, kde daná hodnota poprvé ztratila všechny podpory a čas, kdy k tomu došlo.

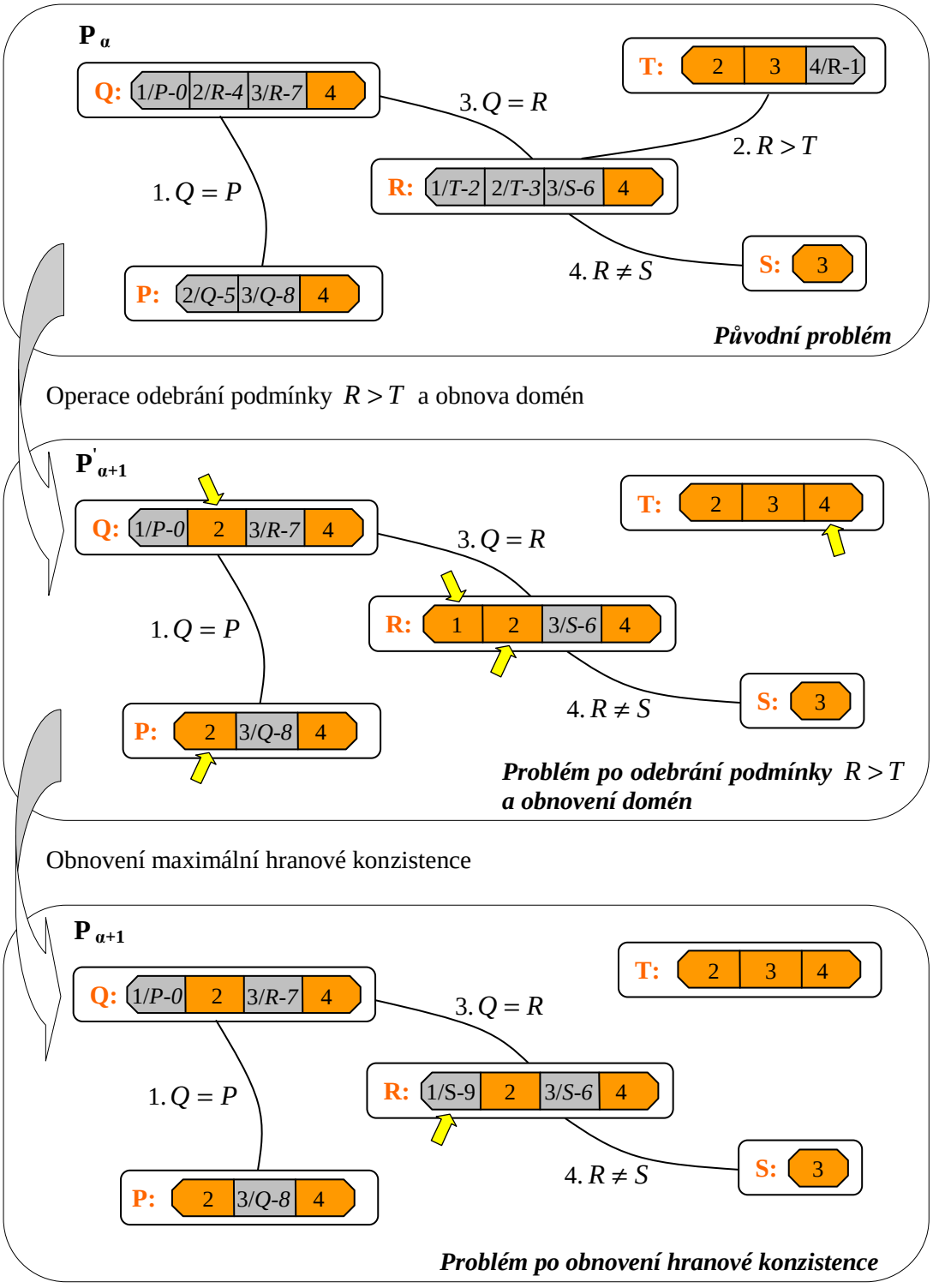
Příklad 3.4.

Původní problém: $P_\alpha = (V_\alpha, C_\alpha)$, kde $V_\alpha = \{P, Q, R, S, T\}$;

$C_\alpha = \{(Q = P); (R > T); (Q = R); (R \neq S)\}$,

Přechod od P_α k $P_{\alpha+1}$: $V_{ADD}^{\alpha+1} = \emptyset$; $V_{DEL}^{\alpha+1} = \emptyset$;

$C_{ADD}^{\alpha+1} = \emptyset; C_{DEL}^{\alpha+1} = \{(R > T)\}$, kde $\alpha \in \mathrm{N}$. ■



Obrázek 3.4: 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 tolik kroků (aditivní a multiplikativní konstanty v souladu s konvencemi zanedbáváme), 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 dvě tabulky shrnují časovou a prostorovou složitost algoritmu AC|DC-2 v nejhorším případě.

AC|DC-2		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|V\|\|D\|+\|C\|)$	$O(\|V\|\|D\|+\|C\|)$
	Různé domény	$O(\sum_{v\in V}\|D_0[v]\|+\|C\|)$	$O(\sum_{v\in V}\|D_0[v]\|+\|C\|)$

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

AC|DC-2		**Přidání podmínky**	**Odebrání podmínky**
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^3)$	$O(\|C\|\|D\|^3)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|^2\|D[v]\| + \sum_{(u,v)\in C}\|D[u]\|\|D[v]\|^2)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|^2\|D_0[v]\| + \sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|^2)$

Tabulka 3.6: Shrnutí časové složitosti algoritmu AC|DC-2

Oproti algoritmu AC|DC-1 zaznamenala nárůst prostorová složitost přidání podmínky o výraz $\sum_{v\in V}|D_0[v]|$, to má na svědomí pole *justification* udržující pomocné informace, 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 $O(\sum_{v\in V}|D_0[v]|+|C|)$. Dále je potřeba uvážit prostor potřebný k reprezentaci řídící fronty, ten ale již výraz $O(\sum_{v\in V}|D_0[v]|+|C|)$ 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 $O(\sum_{(u,v)\in C}|D_0[u]||D_0[v]|)$, 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', čímž bychom dostali optimální

teoretický čas pro závěrečnou filtrační fázi. A jednak chování algoritmu AC-3' není v průměrném případě zdaleka tak nepříznivé, jak to teoreticky vychází pro nejhorší případ.

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 poměrně snadno rozšířit též pro podmínky vyšší arity než 2. Jelikož se jedná víceméně o technickou záležitost ponecháme programový zápis příslušného rozšíření do přílohy A. Poznamenejme, že rozbor teoretické složitosti, který jsme zde uvedli, platí pouze pro verzi algoritmu s binárními podmínkami. Pro nebinární verzi je třeba provést příslušné zobecnění.

3.8 Na cestě k lepšímu algoritmu: Algoritmus AC|DC-2i

Poté, co jsme algoritmus AC|DC-2 a další algoritmy pro dynamickou hranovou konzistenci implementovali v rámci vyvíjeného testovacího prostředí (knihovna pro prácí s omezujícími podmínkami v jazyce C++, podrobněji viz. příloha B), bylo možné realizovat některé experimenty a provádět srovnání algoritmu AC|DC-2 s existujícími algoritmy. Testy byly prováděny na náhodně generovaných problémech (viz. další kapitola věnovaná experimentům a srovnání), nicméně fakt, že byly použity zrovna náhodné problémy a ne jiné, není pro výsledek až tak podstatný. Důkladná analýza běhu algoritmu AC|DC-2 ukázala určité skutečnosti, které si vyžádaly revize a motivovaly následné vylepšení algoritmu.

Co v našem případě znamenala ona důkladná analýza algoritmu AC|DC-2 ? Analýza se týkala operace odebrání podmínky. Cílem bylo zjistit jakým dílem přispívají jednotlivé úseky algoritmu do celkového času, který spotřebuje posloupnost volání operací odebrání podmínky. Úsekem algoritmu zde myslíme například tolikrát zmiňovanou inicializační fázi odebírání podmínky. Místo měření skutečného času, jelikož to by bylo technicky obtížně

proveditelné a navíc pro dané potřeby zbytečné, byly pouze počítány provedené operace, konkrétně testy podmínek (tedy například test, zda je podmínka splněná pro dvojici hodnot).

Algoritmus AC|DC-2 jsme rozdělili celkem do tří fází - *inicializační fáze*, *propagační fáze* a *filtrační fáze*. Není těžké uhodnout, kterým úsekům algoritmu uvedené fáze odpovídají. Inicializační fáze pokrývá počáteční obnovu podmínkou přímo zasažených hodnot, v symbolickém programovém zápisu algoritmu AC|DC-2 (algoritmus 3.7) tomu odpovídají řádky 1 až 11 procedury RETRACT-CONSTRAINT-AC|DC-2. Propagační fáze přesně odpovídá části algoritmu, kdy jsou přímo provedené změny z inicializační fáze šířeny do okolí, v programovém zápisu je tato fáze představována řádkem 12 v proceduře RETRACT--CONSTRAINT-AC|DC-2. Nakonec filtrační fáze pokrývá opravy chybných obnov z inicializační a propagační fáze, v algoritmu této fázi odpovídá řádek 13 ve stejné proceduře.

Testy překvapivě ukázaly, že největší část z celkového počtu operací algoritmu připadá na filtrační fázi. Naproti tomu propagační fáze spotřebuje řádově několikrát méně operací než fáze filtrační a z hlediska celkového času běhu operace odebrání podmínky je tedy čas strávený v propagační fázi málo významný. Tento výsledek motivoval k úpravě filtrační fáze algoritmu AC|DC-2.

Bezprostředně se nabízí úprava omezit závěrečnou filtrační fázi pouze na hodnoty, které obnovila předchozí inicializační a propagační fáze. Tato úprava nakonec přinesla dramatické zkrácení filtrační fáze. Celkově pak při prováděných testech upravený algoritmus AC|DC-2 předčil i dosud nejrychlejší algoritmus pro dynamickou hranovou konzistenci DNAC-6.

Prakticky byla restrikce filtrační fáze pouze na obnovené hodnoty implementována pomocí záznamu časů obnovy. Konkrétněji tedy, u každé hodnoty, která byla navrácena do aktuální domény proměnné si algoritmus zaznamená globální čas, kdy k této události došlo. Závěrečné vyřazování hodnot ve filtrační fázi se pak vztahuje jen na hodnoty, které byly obnoveny v předcházející inicializační a propagační fázi, resp. na ty hodnoty, jejichž čas obnovy je vyšší než čas zahájení operace odebrání podmínky.

Zároveň se u algoritmu AC|DC-2 ukázalo, že by mohlo být výhodné nepočítat minimum z časů odebrání u hodnot navrácených do aktuální domény proměnné ale používat rovnou původní časy odebrání. Takto by bylo zachováno více informace (místo jedné číselné hodnoty bychom používali vektor hodnot) a kritérium, které musí hodnota splnit předtím než bude obnovena, by bylo silnější. V důsledku by pak bylo obnoveno méně hodnot a celý proces odebrání podmínky by spotřeboval méně času.

Toto přísnější kritérium jsme rovněž zabudovali do upravené verze algoritmu. Ačkoli se tato úprava zdála být poměrně slibná, praktické testy provedené na náhodných problémech zatím nepotvrdily její přínos. Nicméně, jelikož se formálně jedná o silnější podmínku, která navíc nevyžaduje žádné další náklady, byla zařazena do nové verze algoritmu také.

Poslední úprava, kterou jsme zrealizovali, byl jiný způsob práce s podmínkami. Nově nepracujeme s celými podmínkami ale pouze s hranami, resp. s uspořádanými dvojicemi proměnných. Pro binární podmínky se tento přístup ukazuje jako efektivnější, což prokázaly i prováděné testy.

Novou verzi algoritmu jsme nazvali *AC|DC-2i* (z anglického improved AC|DC-2). Pro potřeby této verze algoritmu bylo nutné adaptovat i konzistenční proceduru AC-3 tak, aby umožňovala odstraňovat hodnoty, které byly do aktuálních domén vloženy až po určitém časovém okamžiku. K tomuto účelu jsme zavedli novou datovou strukturu - pole $restoration$, které je indexováno proměnnými a jejich hodnotami. Každá buňka pole $restoration$ obsahuje čas, kdy byla příslušná hodnota vložena do aktuální domény proměnné. Tato informace pak dovoluje provádět filtraci hodnot pouze nad vybranými hodnotami.

Odlišný způsob práce s podmínkami, kdy je pokaždé manipulováno s podmínkou jen v jednom směru (pracuje se s hranami), zohledníme jednoduše tak, že množina podmínek C nyní nebude obsahovat celé podmínky, ale pouze uspořádané dvojice proměnných (tedy hrany). Celá podmínka bude v množině podmínek zastupována dvěma hranami. Svazuje-li tedy podmínka c proměnné u a v, bude v množině C zastoupena dvojicemi proměnných (u,v) a (v,u).

Upravenou konzistenční proceduru AC-3 jsme označili jako *AC-3**, její symbolický zápis ukazuje algoritmus 3.8.

Algoritmus 3.8. AC-3*

procedure PROPAGATE-AC-3* (A_{REVISE} , $start$)
1 $Q_{REVISE} \leftarrow A_{REVISE}$
2 **while** $Q_{REVISE} \neq \emptyset$ **do**
3 $(u,v) \in Q_{REVISE}$, libovolná hrana
⋮

```
⋮
4       Q_REVISE ← Q_REVISE − {(u,v)}
5       A^uv_REVISE ← FILTER-ARC-AC-3* ((u,v), start)
6       Q_REVISE ← Q_REVISE ∪ A^uv_REVISE
```

```
function FILTER-ARC-AC-3* ((u,v), start)
1   changed ← false
2   {u,v} ← proměnné svázané podmínkou c
3   for each d_u ∈ D[u] takovou, že restoration[u,d_u].time > start do
4       if d_u nemá žádnou podporu v D[v] vzhledem k c then
5           D[u] ← D[u] − {d_u}
6           justifications[u,d_u].variable ← v
7           justifications[u,d_u].time ← time
8           time ← time + 1
9           changed ← true
10  if not changed then
11      return Ø
12  return {(w,u) | (w,u) ∈ C & w ≠ u & w ≠ v}
```

Program konzistenční procedury AC-3* se velice podobá programu algoritmu AC-3', skládá se z procedury PROPAGATE-AC-3* a pomocné funkce FILTER-ARC--AC-3*. Činnost obou programových konstrukcí se shoduje s odpovídajícími konstrukcemi u AC-3'.

Všimněme si aspoň řádku 3 funkce FILTER-ARC-AC-3*, kde jsou k revizi dávány pouze hodnoty, které byly do aktuální domény vloženy až po daném časovém okamžiku. Když je třeba, aby algoritmus AC-3* pracoval stejně jako AC-3, tj. bez rozlišování, kdy byly hodnoty vloženy do aktuální domény, stačí zadat nulový čas jako okamžik určující počátek vkládání hodnot, které mají být zrevidovány.

Asymptotická časová i prostorová složitost v nejhorším případě zůstává stejná jako u AC-3', tedy $O(\sum_{(u,v)\in C}|D[u]|^2|D[v]| + \sum_{(u,v)\in C}|D[u]||D[v]|^2)$ pro čas a $O(\sum_{v\in V}|D_0[v]|)$ pro paměť.

Nově zavedené pole *restoration* zabírá prostor úměrný $O(\sum_{v \in V} |D[v]|)$, resp. $O(|V||D|)$ pro identické domény, což paměťové nároky $O(\sum_{v \in V} |D_0[v]|)$ nezvětší.

Formální zápis algoritmu AC|DC-2i je v zavedeném symbolickém kódu uveden jako algoritmus 3.9.

Algoritmus 3.9. AC|DC-2i

procedure ADD-CONSTRAINT-AC|DC-2i (c)
1 $\{u,v\} \leftarrow$ proměnné svázané podmínkou c
2 $C \leftarrow C \cup \{(u,v);(v,u)\}$
3 PROPAGATE-AC-3* ($\{(u,v);(v,u)\}$, 0)

procedure RETRACT-CONSTRAINT-AC|DC-2i (c)
1 $\{u,v\} \leftarrow$ proměnné svázané podmínkou c
2 $start \leftarrow time$
3 $D_u^+ \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2i ((u,v))
4 $D_v^+ \leftarrow$ INITIALIZE-ARC-RETRACTION-AC|DC-2i ((v,u))
5 $Q_{RESTORE} \leftarrow \emptyset$
6 **if** $D_u^+ \neq \emptyset$ **then**
7 $\quad D[u] \leftarrow D[u] \cup D_u^+$
8 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(u, D_u^+)\}$
9 **if** $D_v^+ \neq \emptyset$ **then**
10 $\quad D[v] \leftarrow D[v] \cup D_v^+$
11 $\quad Q_{RESTORE} \leftarrow Q_{RESTORE} \cup \{(v, D_v^+)\}$
12 $C \leftarrow C - \{c\}$
13 $A_{REVISE} \leftarrow$ PROPAGATE-ARC-RETRACTION-AC|DC-2i ($Q_{RESTORE}$)
14 PROPAGATE-AC-3* (A_{REVISE} , $start$)

function INITIALIZE-ARC-RETRACTION-AC|DC-2i ((u,v))
1 $c \leftarrow$ podmínka svazující proměnné u a v
⋮

```
⋮
2       D_u^+ ← Ø
3       for each d_u ∈ (D_0[u] − D[u]) do
4           if justification[u, d_u].variable = v then
5               if not HAS-SUPPORT (c, (u, d_u), (v, D[v])) then
6                   D_u^+ ← D_u^+ ∪ {d_u}
7                   restoration[u, d_u].time ← time
8                   justification[u, d_u] ← NIL
9                   time ← time + 1
10      return D_u^+
```

function PROPAGATE-ARC-RETRACTION-AC|DC-2i ($Q_{RESTORE}$)

```
1       A_REVISE ← Ø
2       while Q_RESTORE ≠ Ø do
3           (u, D_u^+) ∈ Q_RESTORE, libovolná dvojice
4           Q_RESTORE ← Q_RESTORE − {(u, D_u^+)}
5           for each (u, v) ∈ C do
6               D_v^+ ← Ø
7               for each d_v ∈ (D_0[v] − D[v]) do
8                   if justification[v, d_v].variable = u and
9                   existuje d_u ∈ D_u^+ taková, že podporuje d_v
10                  vhledem k podmínce svazující u a v a
11                  zároveň justification[v, d_v].time >
12                          > justification[u, d_u].time
13                  then
14                      D_v^+ ← D_v^+ ∪ {d_v}
15                      restoration[v, d_v].time ← time
16                      justification[v, d_v] ← NIL
17                      time ← time + 1
18              if D_v^+ ≠ Ø then
19                  D[v] ← D[v] ∪ D_v^+
20                  Q_RESTORE ← Q_RESTORE ∪ {(v, D_v^+)}
⋮
```

```
⋮
21      └   A_REVISE ← A_REVISE ∪ {(u,v) | (u,v) ∈ C}
22      return A_REVISE
```

Struktura algoritmu AC|DC-2i se shoduje s AC|DC-2. Přidání podmínky zajišťuje procedura ADD-CONSTRAINT-AC|DC-2i, jejím parametrem je přidávaná podmínka c. Ta je rozdělena na dvě hrany, které jsou zařazeny do problému (řádek 2) a následuje volání konzistenční procedury AC-3* pro dvě nové hrany (řádek 3) s časovým parametrem rovným 0 (konzistence bude aplikována na všechny hodnoty v aktuálních doménách).

Odebrání podmínky realizuje procedura RETRACT-CONSTRAINT-AC|DC-2i, jíž sekundují funkce INITIALIZE-ARC-RETRACTION-AC|DC-2i a PROPAGATE-ARC--RETRACTION-AC|DC-2i. Při odebírání podmínky je v proceduře RETRACT-CONSTRAINT-AC|DC-2i nejprve zaznamenán čas zahájení této operace (řádek 2). Poté algoritmus pokračuje standardním způsobem do inicializační fáze, kde jsou obnovovány hodnoty do aktuálních domén proměnných přímo svázaných odebíranou podmínkou. Pokaždé dojdeli k obnově hodnoty, je zaznamenán čas této události do pole *restoration* (řádek 7 funkce INITIALIZE-ARC-RETRACTION-AC|DC-2i). Přitom je třeba aktualizovat globální počítadlo kroků modelující čas (řádek 9 funkce INITIALIZE-ARC-RETRACTION-AC|DC-2i).

Po provedení inicializace následuje šíření změn do okolí, to probíhá v rámci funkce PROPAGATE-ARC-RETRACTION-AC|DC-2i. Zde jsou nejzajímavějším místem řádky 8 až 12, kde se nachází podmínka, kterou musí hodnota splnit předtím, než je vložena do aktuální domény. Místo minima časů zde narozdíl od algoritmu AC|DC-2 používáme originální časy jednotlivě pro každou hodnotu zvlášť. AC|DC-2i se liší v propagační fázi ještě záznamem časů navrácení hodnot (řádek 15), zvyšováním počítadla kroků (řádek 17) a plánováním hran k revizi (nikoli celých podmínek - řádek 21).

Nakonec operace odebrání podmínky algoritmem AC|DC-2i je zavolána konzistenční procedura AC-3*, nyní ale pouze pro hodnoty, jež byly navráceny v aktuálním běhu operace.

Shrňme nyní ve formě tvrzení, co lze o novém algoritmu prohlásit.

Tvrzení 3.3. (KOREKTNOST ALGORITMU AC|DC-2i)**.** *Algoritmus AC|DC-2i je korektní, tj. operace přidání a odebrání podmínky algoritmem AC|DC-2i zachovávají maximální hranovou konzistenci problému.* ■

K důkazu tohoto tvrzení můžeme zopakovat argumentaci z důkazu tvrzení 3.1 (korektnost algoritmu AC|DC-2), předpoklady všech obratů použitých v důkazu tvrzení 3.1 zůstávají i zde v platnosti. ■

Tvrzení 3.4. (POČET OBOVENÝCH HODNOT ALGORITMEM AC|DC-2i)**.** *Operace odebrání podmínky algoritmem AC|DC-2i obnoví nejvýše tolik chybějících hodnot, kolik jich obnoví operace odebrání podmínky algoritmem AC|DC-2. Ve spojení s tvrzením 3.2. tedy platí, že AC|DC-1* $\prec$ *AC|DC-2* $\prec$ *AC|DC-2i, kde zápis A* $\prec$ *B značí, že algoritmus B obnoví nejvýše tolik hodnot, co algoritmus A.* ■

Vlastnost *AC|DC-1* $\prec$ *AC|DC-2* je přímo obsahem tvrzení 3.2. Stačí se tedy podívat na relaci *AC|DC-2* $\prec$ *AC|DC-2i*. Spojíme-li korektnost algoritmu AC|DC-2i a fakt, že hodnoty obnovované v algoritmu AC|DC-2i musejí splnit silnější kritérium než v AC|DC-2 předtím, než jsou obnoveny, dostáváme uvedené tvrzení. ■

Tvrzení 3.5. (POČET KROKŮ ALGORITMEMU AC|DC-2i)**.** *Operace odebrání podmínky algoritmu AC|DC-2i provede nejvýše tolik kroků (testů podmínek), kolik jich provede operace odebrání podmínky algoritmem AC|DC-2 (počítají se testy ze všech fází dohromady). Společně s tvrzením 3.2. tedy platí, že AC|DC-1* $\lhd$ *AC|DC-2* $\lhd$ *AC|DC-2i, kde zápis A* $\lhd$ *B značí, že algoritmus B provede nejvýše tolik kroků, co algoritmus A.* ■

Algoritmy AC|DC-1 a AC|DC-2 používají oba konzistenční algoritmus AC-3. Z předchozího tvrzení platí, že algoritmus AC|DC-2 obnoví nejvýše tolik hodnot, co algoritmus AC|DC-1. Z korektnosti obou algoritmů (tvrzení 3.1) vyplývá, že hodnoty obnovné algoritmem AC|DC-2 jsou podmnožinou hodnot obnovených algoritmem AC|DC-1. Dále platí, že na konci běhu oba algoritmy skončí s totožným obsahem aktuálních domén, to znamená, že algoritmus AC|DC-1 musel v závěrečné fázi svého běhu vyloučit z aktuálních domén proměnných stejně nebo více chybně obnovených hodnot. U algoritmu AC|DC-1 ale musí konzistenční procedura pracovat s většími aktuálními doménami (vzhledem k inkluzi), propagace je tedy slabší (větší domény znamenají méně vyloučených hodnot a méně vyřazených hodnot v jednom kroku automaticky implikuje méně vyřazených hodnot v následujících krocích atd.). Celkem tedy dostáváme, že algoritmus AC|DC-1 musí nutně provést alespoň tolik kroků, co algoritmus AC|DC-2.

Stejnou úvahou dospějeme k témuž výsledku pro dvojici algoritmů AC|DC-2 a AC|DC-2i. Tady ale navíc ve prospěch algoritmu AC|DC-2i hraje fakt, že použitá konzistenční procedura AC-3* pracuje jen s hodnotami obnovenými v propagační fázi a o to méně kroků provede ve filtrační fázi. ■

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

AC|DC-2i		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|V\|\|D\|+\|C\|)$	$O(\|V\|\|D\|+\|C\|)$
	Různé domény	$O(\sum_{v\in V}\|D_0[v]\|+\|C\|)$	$O(\sum_{v\in V}\|D_0[v]\|+\|C\|)$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^3)$	$O(\|C\|\|D\|^3)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|^2\|D[v]\|+\sum_{(u,v)\in C}\|D[u]\|\|D[v]\|^2)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|^2\|D_0[v]\|+\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|^2)$

Tabulka 3.7: Shrnutí časové a prostorové složitosti algoritmu AC|DC-2i

V porovnání s algoritmem AC|DC-2 nedošlo u složitosti v nejhorším případě k žádnému zlepšení ani zhoršení. Nový algoritmus tedy zatím teoreticky obhajují uvedená tvrzení. Praktický přínos musí být teprve ukázán, tím se bude zabývat kapitola věnovaná experimentům.

Podívejme se nyní na další možné zlepšení algoritmu AC|DC-2. Když v provedené analýze jednotlivých fází algoritmu AC|DC-2 na chvíli odhlédneme od operace odebrání podmínky a zaměříme se na přidávání podmínek, shledáme, že v tomto ohledu algoritmy typu AC|DC-2 (používající AC-3) stále poněkud zaostávají za DNAC-6 (který používá AC-6). Eliminaci tohoto nedostatku je věnována následující sekce.

3.9 Zapojení algoritmu AC-3.1: Algoritmus AC3.1|DC-2i

K řešení nastíněné neefektivity se nabízí poměrně přímočaré řešení - nahradit v AC|DC-2, resp. v AC|DC-2i teoreticky málo efektivní konzistenční algoritmus AC-3 optimálním AC-3.1. Podobně, jako to udělal Malek Mouhoub v [Mo93] u AC|DC. To jsme nakonec také zrealizovali. Název výsledného algoritmu jsme sestavili podle stejného schématu, jaký byl použit v [Mo93]. AC|DC-2i s vyměněnou konzistenční procedurou jsme pojmenovali *AC3.1|DC-2i*.

Praktické experimenty provedené opět na náhodných problémech (podrobnému srovnání se věnuje následující kapitola) potvrdily přínos této změny. Operace přidávání podmínek se u algoritmu AC3.1|DC-2i v počtu testů podmínek i v celkovém čase velmi přiblížila výkonu algoritmu DNAC-6. Nutno však říci, že zde vlastně prakticky soupeří pouze konzistenční metody AC-3.1 a AC-6, činnosti související s podporou dynamického odebírání podmínek se do celkových výsledků promítají málo.

Náhrada AC-3 optimálním algoritmem AC-3.1 má v AC3.1|DC-2i ještě další pozitivní efekt, dále zlepšuje výkon algoritmu v závěrečné filtrační fázi při odebírání podmínek a tím zefektivňuje celou operaci odebrání podmínky.

Programový zápis nového algoritmu nebudeme uvádět, protože samotná změna je oproti AC|DC-2i minimální. Co se týče konzistenční procedury AC-3.1, tak tu je třeba upravit kvůli zabudování do algoritmu AC3.1|DC-2i podobným způsobem, jako to bylo učiněno v případě AC-3 pro algoritmus AC|DC-2i. Konkrétně je do AC-3.1 třeba zařadit údržbu pomocných datových struktur (pole *justification* a *restoration*) a možnost spustit uvádění do konzistentního stavu pouze vybrané hodnoty, resp. hodnoty vložené do aktuálních domén proměnných až po určitém okamžiku (po zahájení procesu odebírání podmínky). Opět bychom tedy mohli hovořit o jakési AC-3.1* verzi. V rámci operace odebírání podmínky probíhá vylučování nesprávně navrácených hodnot jen pro hodnoty obnovené v aktuálním běhu operace.

Jelikož je AC-3.1 korektní je tedy tím pádem korektní i nový algoritmus. Platí také, že algoritmus AC3.1|DC-2i provede nejvýše tolik kroků, co algoritmus AC|DC-2. To je přímo dáno vlivem konzistenční procedury AC-3.1*, která si vystačí ve stejné situaci s menším počtem kroků než AC-3*.

Časovou a prostorovou složitost algoritmu AC3.1|DC-2i v nejhorším případě ukazuje následující tabulka.

AC3.1|DC-2i		**Přidání podmínky**	**Odebrání podmínky**
Prostorová složitost (nejhorší případ)	Identické domény	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|V\|\|D\|+\|C\|\|D\|)$
	Různé domény	$O(\sum_{v\in V}\|D[v]\| + \sum_{(u,v)\in C}(\|D[u]\|+\|D[v]\|))$	$O(\sum_{v\in V}\|D_0[v]\| + \sum_{(u,v)\in C}(\|D_0[u]\|+\|D_0[v]\|))$
Časová složitost (nejhorší případ)	Identické domény	$O(\|C\|\|D\|^2)$	$O(\|C\|\|D\|^2)$
	Různé domény	$O(\sum_{(u,v)\in C}\|D[u]\|\|D[v]\|)$	$O(\sum_{(u,v)\in C}\|D_0[u]\|\|D_0[v]\|)$

Tabulka 3.8: Shrnutí časové a prostorové složitosti algoritmu AC3.1|DC-2i

Důležité je, že náhradou konzistenčního algoritmu bylo dosaženo snížení teoretické časové složitosti filtrační fáze procesu odebírání podmínky na hodnotu $O(\sum_{(u,v)\in C}|D_0[u]\|D_0[v]|)$, resp. $O(|C\|D|^2)$ pro problém s identickými doménami proměnných. Vzhledem k tomu, že se asymptotická složitost ostatních fází operace odebrání podmínky rovněž vejde do této hodnoty, týká se snížení složitosti celé operace.

Prostorová složitost se zvětšila o hodnotu $O(\sum_{(u,v)\in C}(|D[u]|+|D[v]|))$ u přidávání podmínky, resp. o $O(\sum_{(u,v)\in C}(|D_0[u]|+|D_0[v]|))$ u odebrání podmínky (neboli $O(|C\|D|)$ pro identické domény). To je dáno potřebou algoritmu AC-3.1*, který si musí pamatovat pro každou hodnotu a vzhledem ke všem sousedním proměnným, kde v aktuální doméně sousední proměnné začíná hledání další podpory (viz. druhá kapitola).

Z hlediska AC|DC algoritmů se konzistenční procedura AC-3.1 vyznačuje příznivou vlastností a tou je její relativní jednoduchost. V důsledku toho pak není zapojení algoritmu AC-3.1 do AC3.1|DC-2i překážkou v rozšíření pro podmínky vyšší arity než dvě.

3.10 Závěrečné teoretické zhodnocení

Existující algoritmy typu DNAC se vyznačují nízkou teoretickou časovou složitostí, paměťové nároky jsou u nich poněkud vyšší. Další charakteristikou těchto algoritmů je obtížnější rozšiřitelnost, která je znesnadněna hlavně použitými datovými strukturami.

Algoritmy typu AC|DC a mezi nimi i nově navržené se naopak vyznačují horší teoretickou časovou složitostí. U operace odebrání podmínky se tento fakt v praxi odráží relativně málo. U operace přidání podmínky je situace horší. Výjimkou je algoritmus AC3.1|DC-2i, kde jsme operaci přidání podmínky zefektivnili.

AC|DC algoritmy mají také velmi příznivou paměťovou složitost. Výjimkou je zase AC3.1|DC-2i, kde se za efektivní přidávání podmínek platí větší paměťovou náročností. Všechny algoritmy typu AC|DC se vyznačují jednoduchostí a snadnou rozšiřitelností, což jsme mimochodem částečně prokázali zdokonalováním algoritmu AC|DC-2, do kterého byly integrovány nové vlastnosti.

Nově navržené algoritmy jsme v této kapitole prozatím obhájili z teoretického hlediska. Teoretické výsledky nám ale prozatím neposkytují obraz o chování nových algoritmů v průměrném případě na skutečných problémech. Dokázaná tvrzení hovoří pouze o tom, že nové algoritmy jsou aspoň tak dobré, jako srovnatelné existující. Je tedy nutné ještě pokračovat v nějakém druhu empirické analýzy a porovnání nových algoritmů s existujícími v situacích, které se blíží průměrnému případu.

Kapitolu věnovanou udržování hranové konzistence v dynamických problémech s podmínkami uzavřeme shrnující přehledovou tabulkou ukazující teoretické složitosti existujících algoritmů (DNAC-4, DNAC-6, AC|DC-1 a AC3.1|DC) a nově navržených (AC|DC-0, AC|DC-2, AC|DC-2i a AC3.1|DC-2i). Výsledky uvedené v tabulce jsou pouze pro problémy s identickými doménami.

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	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
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	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
AC3.1|DC	*Přidání* podmínky	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
AC|DC-2	*Přidání* podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
AC|DC-2i	*Přidání* podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|)$	$O(\|C\|\|D\|^3)$
***AC3.1|DC-2*i**	*Přidání* podmínky	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$
	Odebrání podmínky	$O(\|V\|\|D\|+\|C\|\|D\|)$	$O(\|C\|\|D\|^2)$

Tabulka 3.9: Shrnutí časové a prostorové složitosti v nejhorším případě algoritmů pro dynamickou hranovou konzistenci

Kapitola 4

Empirická analýza algoritmů pro dynamickou hranovou konzistenci

4.1 Účel empirických testů

Dosud jsme v několika tvrzeních teoreticky ukázali, že nové algoritmy AC|DC-2, AC|DC-2i a AC3.1|DC-2i použijí při řešení stejného úkolu nejvýše tolik kroků, co existující AC|DC-1 (AC|DC v obnovovací fázi). Tím jsme ale dokázali pouze to, že si nové algoritmy nepovedou hůře. Což ovšem ještě neznamená, že by si algoritmy typu AC|DC-2 měly vést výrazně lépe, resp. o tolik lépe, aby by to znamenalo nějaký praktický přínos. Při návrhu nových algoritmů bylo pochopitelně cílem, aby dosahovaly při řešení hranové konzistence na reálných dynamických problémech co nejlepších časů, pokud možno lepších než srovnatelné existující algoritmy. Tento cíl ale provedená teoretická analýza podpořila jen částečně. K tomu, abychom důkladněji prozkoumali chování nových algoritmů, přesněji, abychom potvrdili či vyvrátili jejich kvality, je tedy nezbytné provést ještě nějaký druh empirické analýzy.

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 instancích určitých oblíbených testovacích problémů. My jsme k testování a srovnávání nových algoritmů zvolili tzv. *náhodné problémy*. O vlastnostech náhodných problémů splňování podmínek píší například MacIntyre, Prosser, Smith a Walsh v článku [MPSW98].

Z našeho hlediska mají náhodné problémy velmi výhodnou vlastnost a to, že je lze snadno parametrizovat a generovat. Potom není obtížné vytvořit velké množství dostatečně různorodých pokusných instancí problému.

Cílem testů, které budou představeny v této kapitole, je ověřit chování algoritmů AC|DC-2, AC|DC-2i a AC3.1|DC-2i v průměrném případě a porovnat je se srovnatelnými existujícími algoritmy. Pro srovnání byly jako soupeři nových algoritmů vybráni AC|DC (nikoli AC|DC-1), AC3.1|DC a DNAC-6. Algoritmus DNAC-4 jsme do srovnávacích empirických testů nezařadili, neboť se dá předpokládat, že by si v nich stejně nevedl dobře. Toto domněnka je podložena srovnáním algoritmů DNAC-4 s AC|DC, jak je uvedeno například v [NeBe94], a srovnáním DNAC-4 a DNAC-6, což je uvedeno v práci [Deb96]. Další srovnání těchto algoritmů lze nalézt také v [Mo03].

Na základě testů se budeme rovněž snažit zjistit, za jakých okolností a v jakých situacích se případně projeví výhody či nevýhody nových algoritmů.

Ale dříve než přejdeme k samotné empirické analýze algoritmů, popíšeme přesně pojem náhodného problému.

4.2 Náhodný problém splňování podmínek

Asi nejběžněji je náhodný binární problém splňování podmínek zadán čtveřicí parametrů (n,d,p_1,p_2), kde n určuje počet proměnných, d udává velikost původních domén proměnných, p_1 určuje hustotu grafu podmínek (*density*) a nakonec p_2 je tzv. *tightness* vyjadřující sílu, jakou podmínky omezují domény svých proměnných. Parametry p_1 a p_2 jsou interpretovány jako pravděpodobnost a předpokládá se tedy, že platí $p_1 \in [0,1]$ a $p_2 \in [0,1]$. Je-li třeba, lze pochopitelně používat rozšířené náhodné problémy zadávané dalšími dodatečnými parametry.

Podle čtveřice parametrů (n,d,p_1,p_2) je generována struktura problému splňování podmínek. Způsobů, jak to provádět existuje hned několik. Jak je uvedeno v [MPSW98], v praxi se nejčastěji používají čtyři způsoby generování náhodných problémů označované jako *model A*, *model B*, *model C* a *model D*. Přičemž úplně nejčastěji je z těchto čtyřech možností využíváno modelu B. Pro dobré vlastnosti modelu B, což je opět podrobně argumentováno v [MPSW98], jej budeme využívat při testech i my.

Podle modelu B je náhodný problém zadaný parametry (n,d,p_1,p_2) generován následujícím způsobem. Ze všech možných $n(n-1)/2$ podmínek (hran v grafu problému) je rovnoměrně vybráno přesně $p_1 n(n-1)/2$ a ty jsou zařazeny do problému. Pro každou zařazenou podmínku je pak ze všech možných d^2 kompatibilních dvojic vybráno přesně $p_2 d^2$ a ty jsou dále brány jako nekompatibilní, tj. daná podmínka pro ně není splněna (zbylé dvojice hodnot jsou brány jako kompatibilní, tj. podmínka pro tyto dvojice hodnot zůstává splněna).

Parametry n, d, p_1 a p_2 dohromady určují jakousi složitost problému, tedy jak obtížný bude vygenerovaný problém pro určitý algoritmus. 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 parametr udávající hustotu podmínek a parametr tightness (zkoumá se tedy závislost výkonu algoritmů na těchto parametrech, přičemž další parametry zůstávají pevné). To dokládá řada prací zabývající se touto tématikou, jako například [Deb96] a [Mo03], kde se autoři zaměřují právě na jmenované parametry. S ohledem na tato fakta jsme navrhli i svoji empirickou analýzu pro otestování nových algoritmů AC|DC-2, AC|DC-2i a AC3.1|DC-2i.

Samotné empirické testování algoritmů pomocí náhodných problémů probíhá tak, že testovaný algoritmus je postupně vyzkoušen na množině náhodných problémů vygenerovaných podle různých parametrů, přičemž při každém běhu algoritmu jsou sledovány a zaznamenávány určité charakterizující veličiny (prakticky hlavně celkový čas běhu či počet vykonání určité význačné operace).

4.3 Empirická analýza rychlosti běhu nových algoritmů

Nové algoritmy - AC|DC-2, AC|DC-2i a AC3.1|DC-2i a existující algoritmy - AC|DC, AC3.1|DC a DnAC-6 - jsme implementovali v jazyce C++ jako součást experimentální knihovny pro práci s podmínkami. Korektnost implementace všech algoritmů byla ještě ověřena porovnáním s jednoduchým dynamickým algoritmem založeným čistě na konzistenční proceduře AC-3, který počítá hranovou konzistenci po změně struktury problému pokaždé úplně od začátku, tedy natolik jednoduše, že je prakticky možné vyloučit jakoukoli chybu v implementaci. Dynamické algoritmy byly zkontrolovány, zda dávají stejný výstup jako popsaný jednoduchý referenční algoritmus.

Detaily implementace testovaných algoritmů jsou podrobněji probrány v příloze B, proto ji zde nebudeme zvlášť rozebírat. Pouze poznamenejme, že testovací implementace algoritmů popisovaných v kapitole 3 přesně odpovídá, co se funkčnosti týče, uvedenému programovému zápisu. U ostatních algoritmů, tj. u AC|DC a AC3.1|DC, vychází implementace z programového popisu v příslušných článcích.

Algoritmy pro dynamickou hranovou konzistenci byly testovány z hlediska rychlosti běhu a z hlediska spotřeby paměti. Nejprve se budeme věnovat prvnímu hledisku.

Pro otestování rychlosti běhu byl každý z implementovaných algoritmů spuštěn na sadě náhodných problémů s parametry $(100, 50, 0.50, p_2)$, kde se parametr p_2 pohyboval v intervalu $[0.70, 1.00]$ po krocích velikosti 0.20, a na sadě problémů se stejnými parametry, kde se ale parametr p_2 pohyboval v intervalu $[0.87, 0.89]$ tentokrát s krokem o velikosti 0.025. V druhém případě se jedná o oblast parametrů, kde se nachází tzv. *fázový přechod* (vysvětlení ještě bude následovat).

Velikost domén proměnných a hustota podmínek pro testování algoritmů byly zvoleny podle výsledků předběžných experimentů. S doménami proměnných obsahujícími 50 prvků a hustotou podmínek 50% dávaly experimenty dostatečně různorodé výsledky, z čehož se dalo usuzovat, že pro potřebu srovnání algoritmů budou mít testy s těmito parametry nejlepší vypovídací hodnotu. Na počet proměnných byla kladena jediná podmínka a to, aby šly všechny experimenty dokončit na dostupném hardwarovém vybavení v rozumném čase. Tento požadavek omezil počet proměnných zhruba na hodnotu 100, která nakonec byla v testech použita.

Každý algoritmus dynamické hranové konzistence při testech pracoval tak, že byly nejprve pomocí operace přidání podmínky do problému (který na začátku neobsahoval žádnou podmínku) postupně zařazeny všechny podmínky vygenerované podle zadaných parametrů popisujících náhodný problém. Po dokončení přidání všech vygenerovaných podmínek přišla na řadu fáze odebírání podmínek. Ve fázi odebírání podmínek bylo z vytvořeného problému náhodně odebráno pomocí operace odebrání podmínky 10% všech přítomných podmínek.

Data charakterizující běh algoritmů byla sbírána z obou fází odděleně. Uvedený poměr 10% odebíraných podmínek byl opět zvolen podle předběžných experimentů.

Abychom při empirické analýze vyloučili náhodné fluktuace, bylo od každé čtveřice parametrů vygenerováno 10 různých náhodných instancí problému. Výsledky ke každé čtveřici parametrů pak byly získány jako aritmetický průměr z výsledků celé sady 10-ti instancí náhodných problémů.

V okamžiku, kdy dojde při přidávání podmínek k vyprázdnění aktuální domény nějaké proměnné, nepokračují implementované algoritmy pro dynamickou hranovou konzistenci v další propagaci a přidávání podmínek je ukončeno. 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 před-

chozího hranově konzistentního stavu. Nastane-li taková situace, tedy, když dojde k vyprázdnění aktuální domény proměnné, nachází se problém na konci fáze přidávání podmínek v hranově nekonzistentním stavu. Implementované algoritmy se však s takovou situací dokáží vypořádat, všechny umožňují odebírat podmínky i z hranově nekonzistentních stavů. Pokud přidávání podmínek uvízne v hranově nekonzistentním stavu, je fáze odebírání podmínek zahájena odebráním právě té podmínky, která nekonzistenci způsobila. Další náhodně odebírané podmínky jsou pak již odebírány z hranově konzistentních stavů.

Průběh operace odebrání podmínky byl při empirických testech zkoumán zvlášť pro konzistentní a zvlášť pro nekonzistentní stavy. Interval hodnot parametru p_2, kde dochází k přelomu mezi konzistentními a nekonzistentními stavy, nazýváme oním už zmiňovaným fázovým přechodem. Tato oblast hodnot parametru tightness byla v experimentech také zohledněna zvlášť.

Implementované algoritmy při svém běhu shromažďují řadu statistických údajů. Konkrétně se jedná o údaje shrnuté v tabulce 4.1.

Statistické údaje shromažďované během testů
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***
Počet volání operace ***EXTEND***
Celkový ***čas běhu*** v sekundách

Tabulka 4.1: Údaje shromažďované během empirických testů

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 (kvalitu konkrétní implementace posuzujeme například

podle vyspělosti použitých datových struktur). Další operace, konkrétně tedy HAS--SUPPORT, FILTER a EXTEND, jsou ve své výchozí implementaci, která byla použita v této empirické analýze, realizovány právě pomocí testů podmínky pro dvojice hodnot.

Jak bylo uvedeno v předchozích odstavcích, fáze přidávání podmínek a fáze odebírání podmínek byly měřeny zvlášť. Údaje z běhů algoritmů ve fázi přidávání podmínek nepředstavují nic jiného než data o běhu příslušných konzistenčních procedur v nedynamické verzi.

Následující tabulka přehledně ukazuje situace, které byly při empirickém testování algoritmů rozlišovány a měřeny nezávisle (jeden celý řádek tabulky vždy postihuje jednu rozlišovanou situaci).

Fáze **přidávání** podmínek		**Řídké pokrytí** širšího rozsahu parametru *tightness*
		Husté pokrytí parametru *tightness* v oblasti fázového přechodu
Fáze **odebírání** podmínek	**Konzistentní** stavy	**Řídké pokrytí** širšího rozsahu parametru *tightness*
		Husté pokrytí parametru *tightness* v oblasti fázového přechodu
	Nekonzistentní stavy	**Řídké pokrytí** širšího rozsahu parametru *tightness*
		Husté pokrytí parametru *tightness* v oblasti fázového přechodu

Tabulka 4.2: Situace rozlišované při testování algoritmů

Výsledkem provedených testů jsou statistické údaje uvedené v tabulce 4.1 přepočítané na jednu podmínku (přidávanou nebo odebíranou) pro všechny výše specifikované hodnoty parametrů definujících náhodný problém a pro všechny výše specifikované situace (tabulka 4.2).

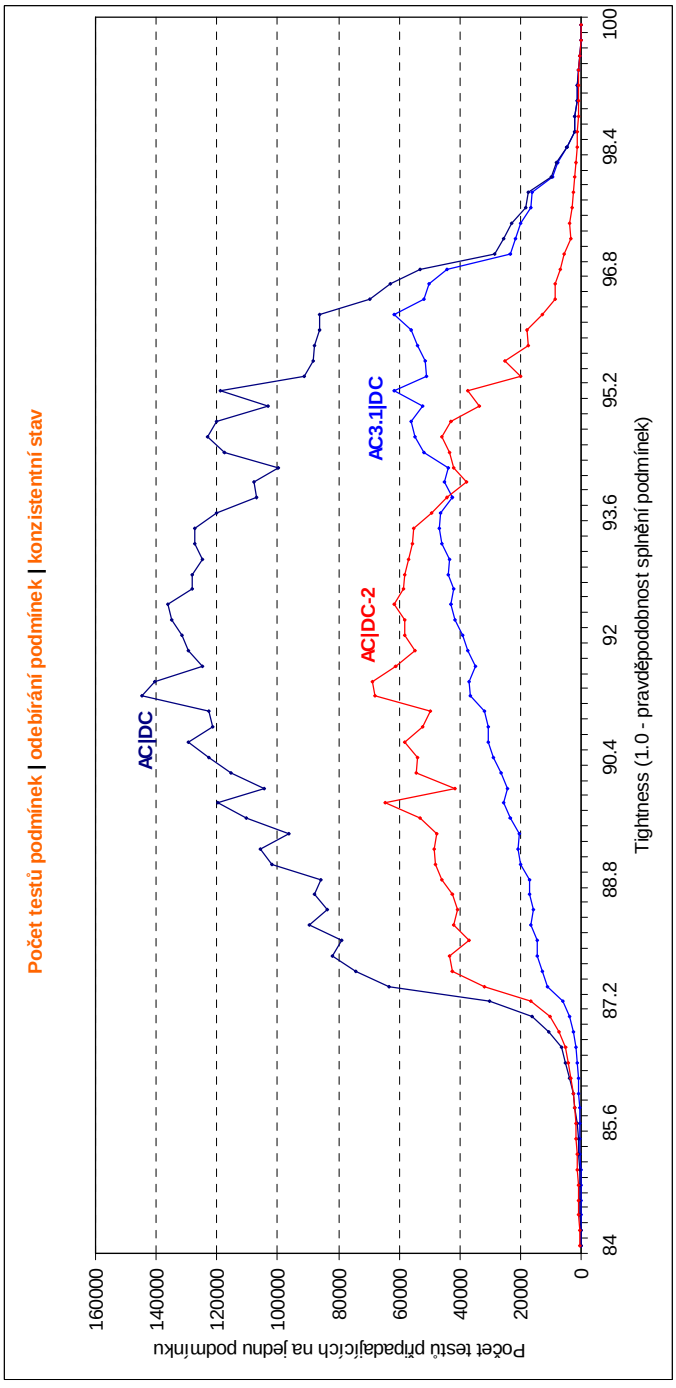
V této kapitole uvedeme některé srovnávací grafy závislosti počtu testů podmínek pro dvojici hodnot a celkového času běhu na hodnotě parametru tightness. Další srovnávací grafy přenecháme do přílohy C.

Kompletní soubory dat získaných během měření, tj. dat, z nichž byly konstruovány srovnávací grafy, jsou uloženy na přiloženém CD. Původní údaje získané z měření jsou na CD uloženy v množství textových souborů (výpisy statistik běhu programu), kde každý soubor odpovídá měření jednoho z algoritmů a jedné čtveřici parametrů udávajících náhodný problém. Data použitá pro vytvoření srovnávacích grafů jsou z originálních textových souborů extrahována a na přiloženém CD jsou uložena ještě ve formě tabulek v programu Microsoft Excel, to se týká to údaje o počtu testů podmínek a celkového času běhu algoritmů.

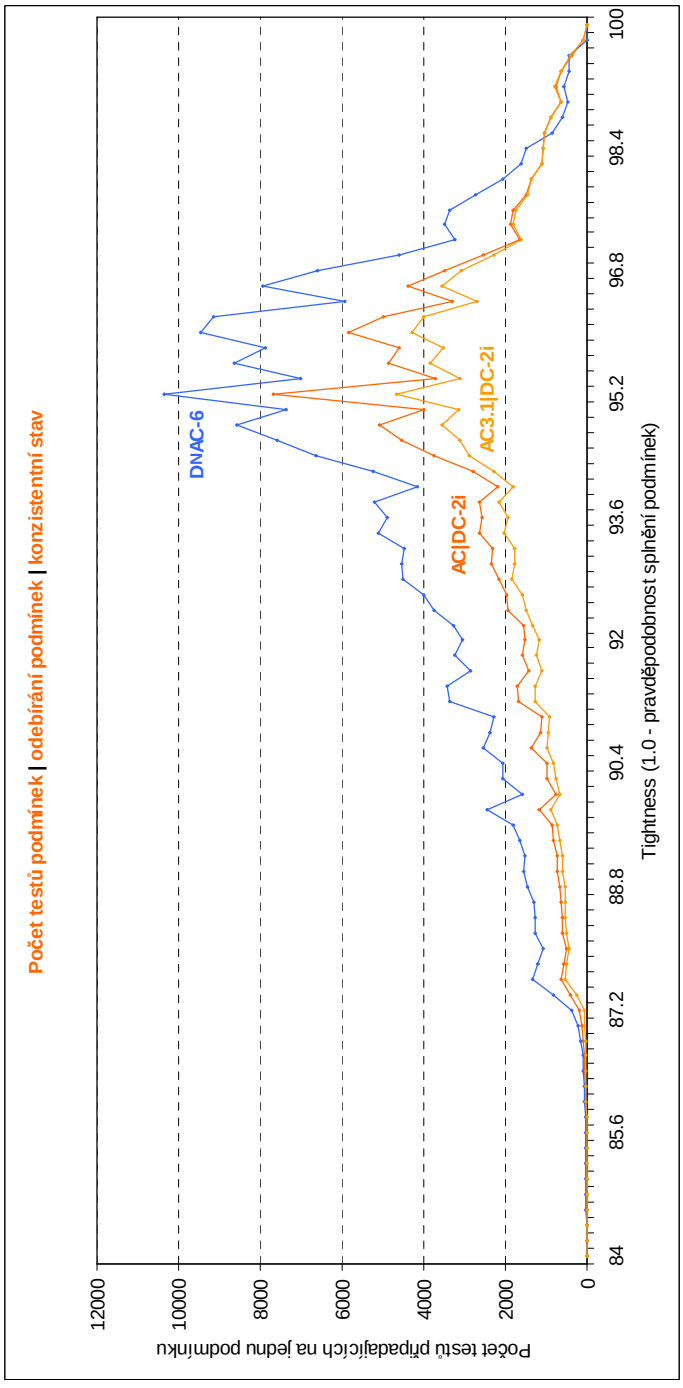
4.3.1 Výsledky experimentů zaměřených na rychlost běhu

Grafy 4.1 a 4.2 ukazují empirické srovnání rychlosti běhu algoritmů AC|DC, AC3.1|DC, AC|DC-2, AC|DC-2i, AC3.1|DC-2i a DNAC-6 při odebírání podmínek z konzistentních stavů. Grafy zobrazují průměrný počet testů připadajících na jednu podmínku. Stejným způsobem jsou zpracovány i grafy 4.3 a 4.4., zde se ovšem jedná o operaci přidání podmínky.

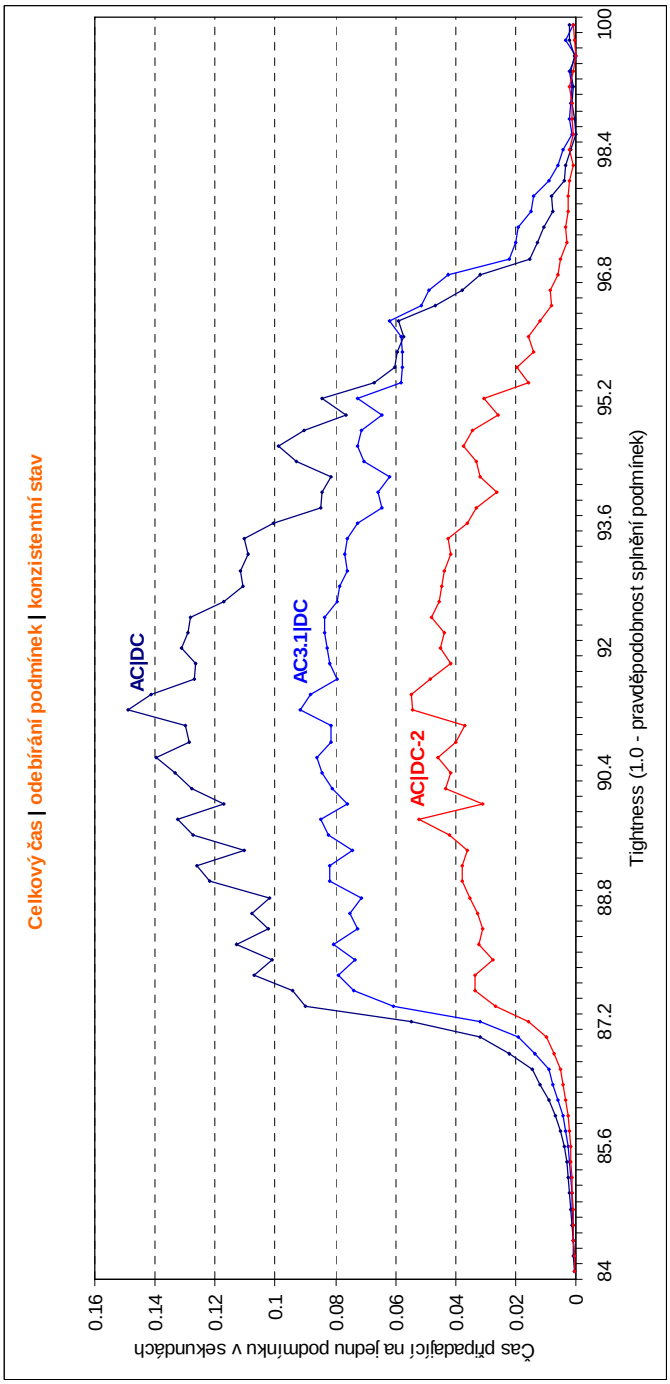
Grafy 4.5 a 4.6 ukazují stejně jako grafy 4.1 a 4.2 srovnání implementovaných algoritmů při odebírání podmínek z konzistentních stavů. V tomto případě však není srovnáván počet testů podmínek ale celkový čas běhu algoritmů.



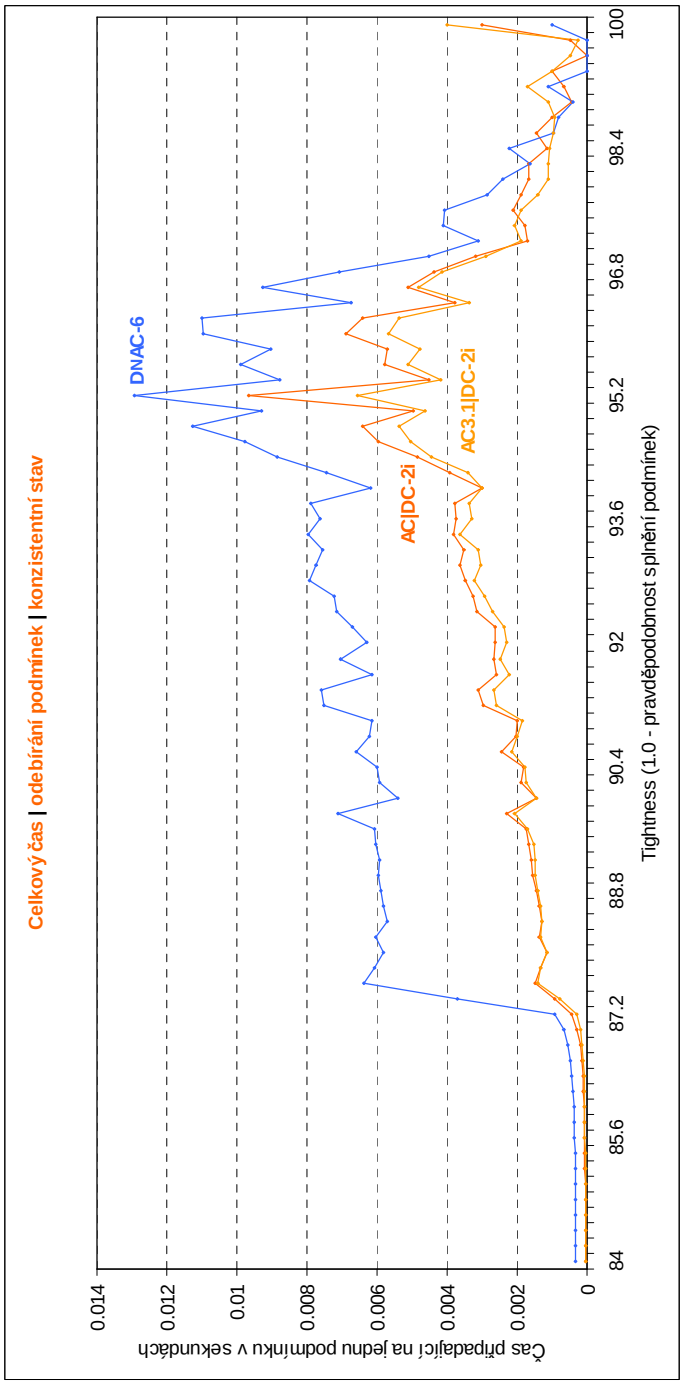
Graf 4.1: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle počtu testů při odebírání podmínek z konzistentního stavu



Graf 4.2: Srovnání algoritmů DnAC-6, AC|DC-2i a AC3.1|DC-2i podle počtu testů při odebírání podmínek z konzistentního stavu



Graf 4.3: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle celkového času běhu při odebírání podmínek z konzistentního stavu



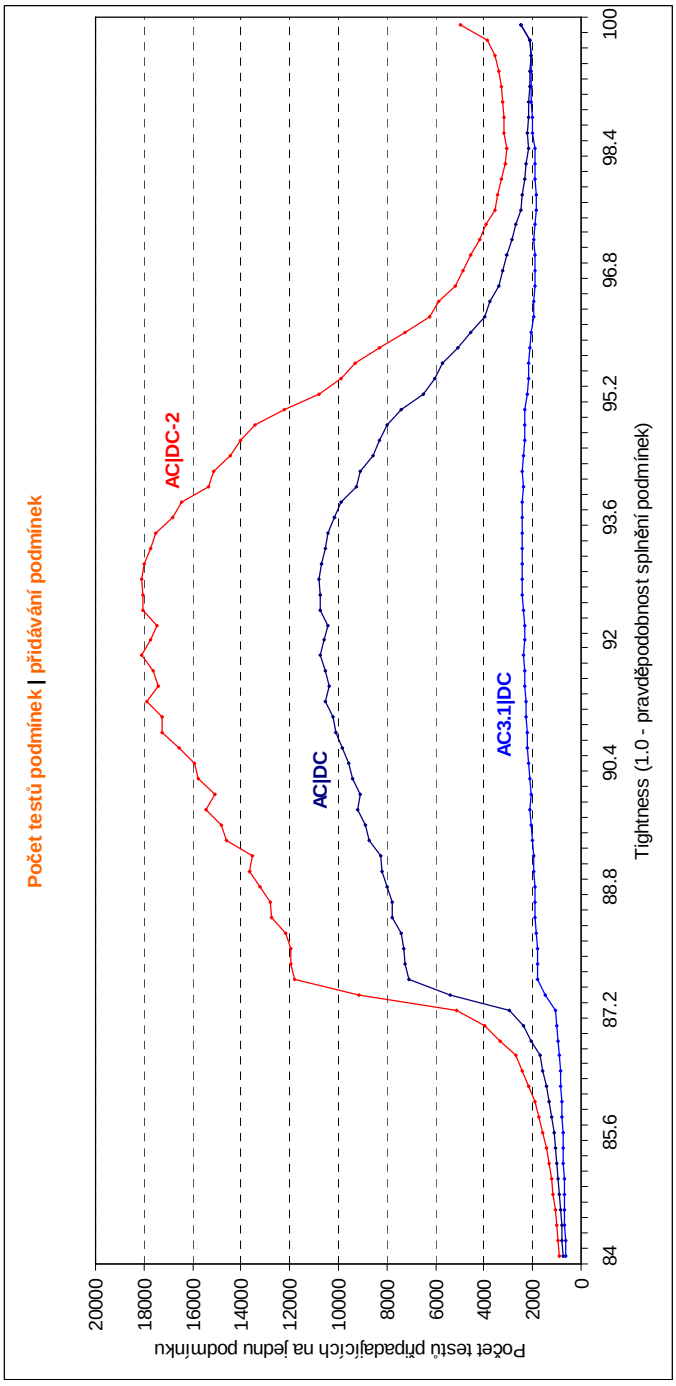
Graf 4.4: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle celkového času běhu při odebírání podmínek z konzistentního stavu

Graf 4.1 ukazuje, že nový algoritmus AC|DC-2 jednoznačně poráží existující algoritmus AC|DC při odebírání podmínek. Špatně si nevede ani v porovnání se silnějším existujícím algoritmem AC3.1|DC. Při vyšších hodnotách parametru tightness nový algoritmus AC3.1|DC dokonce také předčí.

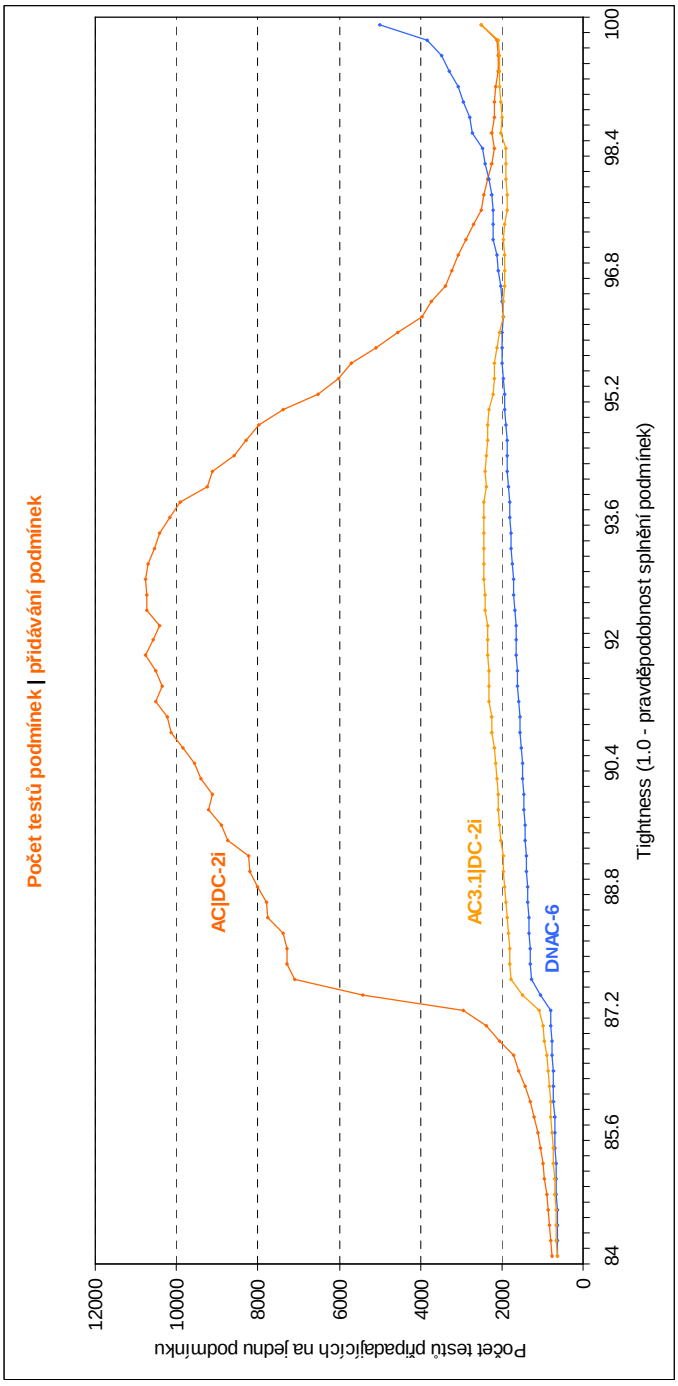
Zdokonalené verze algoritmu AC|DC-2 - tedy algoritmy AC|DC-2i a AC3.1|DC-2i - jsou v grafu 4.2 srovnávány s nejlepším dosud existujícím algoritmem DNAC-6. Opět se jedná o srovnání operace odebírání podmínek. Všimněme si, že v porovnání s grafem 4.1 se naměřené počty testů podmínek pohybují zhruba o jeden řád níže. Z grafu je vidět, že oba nové algoritmy potřebují ke své činnosti podstatně méně testů podmínek než dosud nejrychlejší algoritmus DNAC-6 a stávají se tak nejrychlejšími algoritmy pro udržování hranové konzistence při odebírání podmínek (alespoň co se počtu testů podmínek týče, z dalších grafů je ale vidět, že pro celkový čas běhu jsou výsledky skoro stejné).

Celkové časy běhu algoritmů při odebírání podmínek jsou shrnuty v grafech 4.3 a 4.4[1]. Společně s grafy 4.1 a 4.2 ukazují souvislost celkového času běhu a počtu testů podmínek na splnění pro dvojici hodnot. Z těchto čtyřech grafů je vidět, že srovnání algoritmů pro počet testů podmínek a pro celkový čas je prakticky stejné. Počet testů podmínek je tedy z pohledu rychlosti konzistenčních algoritmů dobře charakterizující veličina. Celkově dostáváme, že AC|DC-2i a AC3.1|DC-2i mají nejlepší výsledky ze všech existujících algoritmů jak v počtu testů podmínek, tak v celkovém čase běhu.

[1] Měření celkového času běhu probíhala na počítači vybaveném procesorem AMD AthlonXP-M 3000+ (1600 MHz), s 512 MB operační paměti, pod operačním systémem Mandrake Linux 10.0. Testovací programy byly přeloženy kompilátorem gcc-3.3.2 se stupněm optimalizace O9.



Graf 4.5: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle počtu testů při přidávání podmínek



Graf 4.6: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle počtu testů při přidávání podmínek

Grafy 4.5 a 4.6 srovnávají rychlost algoritmů při operaci přidávání podmínek. Grafy ukazují v podstatě známý výsledek, tedy to, jak si vzájemně výkonnostně stojí konzistenční procedury AC-3, AC-3.1 a AC-6. Pozitivní výsledek pro nový algoritmus AC3.1|DC-2i je ne o příliš vyšší počet testů při přidávání podmínek ve srovnání s DnAC-6. Jinak řečeno, AC-3.1 dosahuje skoro stejně nízkého počtu testů podmínek jako AC-6.

4.4 Analýza paměťových nároků

Provedli jsme rovněž analýzu všech implementovaných algoritmů pro dynamickou hranovou konzistenci z hlediska prostorové složitosti, resp. prostorové složitosti v průměrném případě. Testy byly opět prováděny na náhodných problémech, ovšem konstrukce samotných experimentů byla zcela odlišná od měření rychlosti běhu algoritmů.

Pro zvolenou čtveřici parametrů definujících náhodný problém byla vždy generována instance problému pomocí příslušné operace přidání podmínky. Jakmile bylo vytváření náhodného problému dokončeno, byl proveden odečet spotřebované paměti. Vzhledem k tomu, že množství spotřebované paměti je přímo úměrné počtu chybějících hodnot v aktuálních doménách proměnných neboli přímo úměrné počtu přítomných podmínek, nebylo v rámci testování paměťových nároků prováděno žádné odebírání podmínek (pomocné datové struktury všech testovaných dynamických algoritmů uchovávají záznamy právě pro chybějící hodnoty). Množství paměti spotřebované algoritmem bylo měřeno pomocí UNIXového příkazu `top`.

Měnícím se parametrem určujícím náhodný problém byla v případě paměťových testů velikost domén proměnných. Zkoumána tedy byla spotřeba paměti v závislosti na velikosti domén proměnných. Hodnotu parametru tightness jsme pro každou čtveřici parametrů náhodného problému volili v celých procentech co největší tak, aby problém zůstal po přidání všech podmínek ještě konzistentní (zvětšení o 1% už by vyústilo v nekonzistentní stav na konci procesu přidávání podmínek). Uvedený postup zajišťuje, že v okamžiku odečtu množství spotřebované paměti jsou datové struktury všech algoritmů maximálně naplněny a paměťové nároky jsou největší.

Počet proměnných v náhodných problémech byl pro tyto experimenty roven 200 a hustota podmínek 30%. Tyto hodnoty byly zvoleny tak, aby nebyla překročena kapacita dostupného hardwarového vybavení.

4.4.1 Výsledky experimentů zaměřených na spotřebu paměti

Tabulky 4.3 a 4.4 ukazují výsledky provedených paměťových experimentů. První tabulka pokrývá výsledky spotřeby paměti pro velikosti domén proměnných od 10 do 55 prvků, druhá pak pro velikosti domén 60 až 100. Množství spotřebované paměti je udáváno v MB.

Výsledky lze stručně shrnout asi takto. Dynamické algoritmy postavené na bázi konzistenční procedury AC-3 vykazují prakticky neměřitelnou spotřebu paměti a z tohoto hlediska jsou velmi úsporné. Dynamické algoritmy postavené na sofistikovanějších konzistenčních procedurách AC-6, resp. AC-3.1 mají téměř shodně zvýšené paměťové nároky, přičemž algoritmy používající AC-6 jsou na tom o něco hůře.

Velikost domén d	10	15	20	25	30	35	40	45	50	55
Tightness p_2	49%	62%	70%	75%	79%	81%	84%	85%	87%	88%
DNAC-6	4	6	7	9	11	13	15	17	18	19
AC|DC	1	1	<1	<1	<1	<1	1	1	1	<1
AC3.1|DC	4	5	6	7	8	10	12	13	14	15
AC|DC-2	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
AC|DC-2i	1	1	<1	<1	<1	<1	1	1	1	<1
***AC3.1|DC-2*i**	4	5	6	7	8	10	12	13	14	15

Tabulka 4.3: Paměťové nároky algoritmů pro dynamickou hranovou konzistenci v megabytech (velikosti domén proměnných v rozmezí 10 až 55 prvků)

Velikost domén d	60	65	70	75	80	85	90	95	100
Tightness p_2	89%	89%	90%	91%	91%	92%	92%	92%	93%
DNAC-6	21	24	26	26	29	30	33	34	36
AC|DC	<1	1	1	<1	<1	1	1	<1	1
AC3.1|DC	16	19	20	20	22	23	25	26	27
AC|DC-2	<1	<1	<1	<1	<1	<1	<1	<1	<1
AC|DC-2i	<1	1	1	<1	<1	1	1	<1	1
***AC3.1|DC-2*i**	16	19	20	20	22	23	25	26	27

Tabulka 4.4: Paměťové nároky algoritmů pro dynamickou hranovou konzistenci v megabytech (velikosti domén proměnných v rozmezí 60 až 100 prvků)

4.5 Zhodnocení výsledků

Měření celkového času běhu algoritmů dává prakticky stejné výsledky jako měření počtu testů podmínek (grafy 4.5 a 4.6 a dále v příloze C). To znamená, že nové algoritmy se stávají nejrychlejšími algoritmy pro dynamickou hranovou konzistenci i z hlediska celkového času běhu. Příznivější počet testů podmínek na splnění se tedy odpovídajícím způsobem odráží i na celkovém čase běhu algoritmu, a to i přesto, že experimentální implementace nebyla nijak speciálně optimalizována. Na základě těchto výsledků můžeme vyslovit tvrzení, že počet provedených testů podmínek velmi dobře odpovídá celkovému času běhu, resp. dobře charakterizuje časovou složitost v průměrné případě. To v důsledku také znamená, že menší počet testů podmínek není u nových algoritmů typu AC|DC-2 zaplacen navýšením jiné práce.

Z testů je dále vidět, že nové algoritmy vykazují výrazně lepší výsledky než odpovídající srovnatelné existující algoritmy 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 nových algoritmů - AC|DC-2 oproti AC|DC, resp. AC|DC-2i a AC3.1|DC-2i oproti DNAC-6. Na druhou stranu v situaci, kdy je

obnova jednoduchá, tedy, když se obnovuje malé množství hodnot, jsou výkony testovaných algoritmů srovnatelné.

Podívejme se podrobněji nejprve na srovnání nových algoritmů s existujícím AC|DC. Zde je jasně vidět, že nové algoritmy přinášejí výraznou úsporu počtu testů podmínek a stejně tak celkového času.

Převaha nových algoritmů nad AC|DC se projevuje zejména, pokud je třeba do aktuálních domén zasažených proměnných navrátit větší množství chybějících hodnot (jsou-li podmínky přísnější - vyšší hodnoty parametru tightness). Za těchto okolností algoritmus AC|DC 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 algoritmů typu AC|DC-2 skutečně vyplácí (neboli naopak, že algoritmus AC|DC na tento jev výrazně doplácí).

Jak už bylo naznačeno, je-li naopak potřeba navracet do aktuálních domén proměnných menší množství chybějících hodnot, jsou výkony všech testovaných algoritmů srovnatelné. Tento fakt lze interpretovat tak, že při menším počtu obnovovaných hodnot je menší pravděpodobnost, že algoritmus AC|DC provede řetěz chybných obnov a obnova se tak více blíží ideálnímu průběhu, resp. je také menší pravděpodobnost, že do celkového průběhu obnov výrazněji promluví techniky užívané v algoritmech AC|DC-2, AC|DC-2i a AC3.1|DC-2i.

Podíváme-li se na výsledky algoritmu AC3.1|DC, můžeme říci, že významně do celkového výkonu algoritmů při odebírání podmínek promlouvá i fáze odstraňování chybně obnovených hodnot. Náhrada konzistenční procedury AC-3 kvalitnějším algoritmem AC-3.1 se velmi pozitivně odrazila na celkovém výkonu algoritmu. Stejně tak se pozitivně projevilo vylepšení filtrační fáze, které bylo diskutováno v předchozí kapitole a které bylo aplikováno u algoritmů AC|DC-2i a AC3.1|DC-2i.

Když se pokusíme interpretovat výsledky srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i, je třeba nejprve říci, že jsou zde srovnávány zcela odlišné techniky obnov aktuálních domén proměnných. Obě techniky usilují o navracení co nejmenšího počtu zbytečných hodnot. Algoritmus DNAC-6 za tímto účelem používá seznamů podporovaných hodnot, zatímco algoritmy AC|DC-2, AC|DC-2i a AC3.1|DC-2i jsou založeny na využití

informací o čase, kdy došlo k odstranění hodnot z aktuálních domén proměnných. Z provedených testů vyplývá, že druhý způsob je efektivnější.

Z hlediska paměťových nároků jsou algoritmy AC|DC-2 a AC|DC-2i velmi úsporné. A to zejména v porovnání s DNAC-6. Pokud však vyšší paměťové nároky nejsou překážkou, pak se jako nejlepší alternativa pro uživatele jeví algoritmus AC3.1|DC-2i.

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

Veškerý software pro testování algoritmů AC|DC, AC3.1|DC, AC|DC-2, AC|DC-2i, AC3.1|DC-2i a DNAC-6 na náhodných problémech, který byl použit k získání uvedených výsledků, se nachází na přiloženém CD včetně kompletních zdrojových textů a dalších podpůrných dat.

Závěr a možnosti dalšího vývoje

V této práci jsem podal 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 algoritmy, jako jsou AC|DC nebo AC3.1|DC, které pracují bez užití seznamů podpor, ale například v porovnání s DnAC-6 značně zaostávají. 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 jsem proto navrhl několik zcela nových algoritmů pro řešení dynamické hranové konzistence - AC|DC-2, AC|DC-2i a AC3.1|DC-2i. Všechny se obejdou bez udržování nákladných seznamů podpor a řeší tak nedostatek algoritmů DnAC-4 a DnAC-6 a zároveň nabízejí oproti AC|DC a AC3.1|DC výrazně vyšší úsporu počtu operací i celkového času, což dokládají dokázaná tvrzení a provedená empirická analýza. Hlavním výsledkem této práce jsou algoritmy AC|DC-2i a AC3.1|DC-2i. Oba algoritmy jsou podle provedených experimentů rychlejší než dosud nejrychlejší algoritmus pro udržování hranové konzistence v dynamických problémech DnAC-6. Ještě jednou bych zde ale chtěl zdůraznit, že výsledky byly získávány postupně. Původní algoritmus AC|DC-2 jsem navrhnul u6 ve své diplomové práci. Algoritmy AC|DC-2i a AC3.1|DC-2i vznikly až později, při přípravě příspěvků na mezinárodní konference, jak jsem psal v úvodu.

Nové algoritmy jsou v práci popsány pomocí symbolického programového zápisu. Následují nezbytná tvrzení o korektnosti navržených přístupů a příslušná analýza prostorové a časové složitosti. V práci je rovněž teoreticky dokázáno několik tvrzení o nově navržených algoritmech. Přínos nových algoritmů je dále demonstrován v možnostech jejich dalších

rozšíření, z nichž konkrétně rozšíření algoritmu AC|DC-2 pro podmínky vyšší arity než 2 je formulováno jako program v uvedený příloze A.

Všechny nové algoritmy byly za účelem ověření a získání výsledků o chování v průměrném případě implementovány jako součást experimentální knihovny `SPlan` pro práci s omezujícími podmínkami v jazyce C++. Knihovna `SPlan` je přiměřeným způsobem stručně popsána v příloze B. Stejně tak jsem implementoval v rámci knihovny `SPlan` i srovnatelné existující algoritmy pro dynamickou hranovou konzistenci, aby bylo možné nové algoritmy empiricky porovnávat s existujícími přístupy.

Dalším vývoj prací k tématu dynamických algoritmů 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 [JuBo97] 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. Ostatně jistý druh takové vizualizace, ale ne příliš dokonalé, jsem používal v rámci knihovny `SPlan` při vývoji nových algoritmů.

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. Ještě si všimněme, že na řádku 11 funkce PROPAGATE-ARC-RETRACTION--AC|DC-2 je složka *variables* pole *justifications* testována na obsahování prvku, nikoli na rovnost, jako je tomu u odpovídající konstrukce v binární verzi algoritmu.

Algoritmus A.1. OBECNÝ AC-3'

procedure PROPAGATE-AC-3' (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 $\{v_1, v_2, \ldots, v_n\} \leftarrow$ proměnné svázané podmínkou c
6 $(D^-_{v1}, D^-_{v2}, \ldots, D^-_{vn}) \leftarrow$ FILTER(c, $(D[v_1], D[v_2], \ldots, D[v_n])$)
⋮

```
⋮
7       for each i ∈ {1,2,…n} do
8           C^vi_REVISE ← FILTER-ARC-AC-3' ( v_i , D⁻_vi , {v_1, v_2,…
9                                                   …, v_(i−1), v_(i+1),…, v_n} )
10          Q_REVISE ← Q_REVISE ∪ C^vi_REVISE
```

function FILTER-ARC-AC-3' (v_i , D_{vi}^-, *variables*)

```
1   C_REVISE ← Ø
2   c ← podmínka svazující proměnné v_1, v_2,…, v_n
3   if D⁻_vi ≠ Ø then
4       for each d_vi ∈ D⁻_vi do
5           D[v_i] ← D[v_i] − {d_vi}
6           justification[v_i, d_vi].variables ← variables
7           justification[v_i, d_vi].time ← time
8           time ← time + 1
9       C_REVISE ← C_REVISE ∪ {e ∈ C | e svazuje nějakou proměnnou
10                                  z v_1, v_2,…, v_n & e ≠ c}
11  return C_REVISE
```

Algoritmus A.2. OBECNÝ AC|DC-2

procedure ADD-CONSTRAINT-AC|DC-2 (c)

```
1   C ← C ∪ {c}
2   PROPAGATE-AC-3' ({c})
```

procedure RETRACT-CONSTRAINT-AC|DC-2 (c)

```
2   {v_1, v_2,…, v_n} ← proměnné svázané podmínkou c
1   Q_RESTORE ← Ø
3   for each i ∈ {1,2,…n} do
4       (time_vi, D⁺_vi) ← INITIALIZE-ARC-RETRACTION-AC|DC-2( c ,
5                                   v_i , {v_1, v_2,…, v_(i−1), v_(i+1),…, v_n})
⋮
```

```
⋮
6        if D+vi ≠ Ø then
7            D[vi] ← D[vi] ∪ D+vi
8            QRESTORE ← QRESTORE ∪ {(vi, timevi, D+vi)}
9    C ← C − {c}
10   CREVISE ← PROPAGATE-ARC-RETRACTION-AC|DC-2 (QRESTORE)
11   PROPAGATE-AC-3' (CREVISE)

function INITIALIZE-ARC-RETRACTION-AC|DC-2 (c, vi, variables)
1    {v1, v2, …, v(i−1), v(i+1), …, vn} ← variables
2    timevi ← ∞
3    D+vi ← Ø
4    for each dvi ∈ (D0[vi] − D[vi]) do
5        if justification[vi, dvi].variables = variables then
6            if not HAS-SUPPORT (c, vi, dvi, (v1, v2, …, vi−1, vi+1, …
7                …, vn), (D[v1], D[v2], …, D[vi−1], D[vi+1], …
8                …, D[vn]))
9            then
10               D+vi ← D+vi ∪ {dvi}
11               timevi ← MIN (timevi, justification[vi, dvi].time)
12               justification[u, du] ← NIL
13   return (timevi, D+vi)

function PROPAGATE-ARC-RETRACTION-AC|DC-2 (QRESTORE)
1    CREVISE ← Ø
2    while QRESTORE ≠ Ø do
3        (vi, timevi, D+vi) ∈ QRESTORE, libovolná trojice
4        QRESTORE ← QRESTORE − {(vi, timevi, D+vi)}
5        for each c ∈ C svazující proměnnou vi do
6            {v1, v2, …, vn} ← proměnné svázané podmínkou c
7            for each j ∈ {1,2,…,i−1,i+1,…,n} do
⋮
```

```
⋮
8                    time_vj ← ∞
9                    D+_vj ← Ø
10                   for each d_vj ∈ (P.D0[v_j] − D[v_j]) do
11                     if v_i ∈ justification[v_j, d_vj].variables and
12                       justification[v_j, d_vj].time > time_vi and
13                       HAS-SUPPORT(c, v_i, d_vi, (v1, v2, ...
14                       ..., v_{i−1}, v_{i+1}, ..., v_n), (D[v1], D[v2], ...
15                       ..., D[v_{i−1}], D[v_{i+1}], ..., D[v_n]))
16                     then
17                       D+_vj ← D+_vj ∪ {d_vj}
18                       time_vj ← MIN(justification[v, d_vj].time,
19                                    time_vj)
20                       justification[v_j, d_vj] ← NIL
21                 for each j ∈ {1,2,...,i−1,i+1,...,n} do
22                   if D+_vj ≠ Ø then
23                     D[v_j] ← P.D[v_j] ∪ D+_vj
24                     Q_RESTORE ← Q_RESTORE ∪ {(v_j, time_vj, D^restore_vj)}
25               C_REVISE ← C_REVISE ∪ {c ∈ C | nějakou proměnnou
26                                             z v1, v2, ..., v_n}
27           return 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, někdy dokonce o něco jednodušeji, lze upravit za účelem zpracování podmínek vyšší arity též algoritmy AC|DC-0, AC|DC-1, AC|DC-2i a AC3.1|DC-2i.

Příloha B

Implementace experimentální knihovny

Pro ověřování teoretických hypotéz a k různým praktickým testům byla implementována vlastní knihovna pro práci s omezujícími podmínkami. Experimentální knihovnu jsme na první pohled nelogicky nazvali `SPlan`[1]. Knihovna se s veškerými svými zdrojovými texty a dalšími podpůrnými soubory nachází na přiloženém CD.

Vzhledem k tomu, že mezi dostupnými softwarovými prostředky 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 byla také skutečnost, 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, a která bude plně pod kontrolou autora.

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 použitelný softwarový nástroj a při implementaci byl brán zřetel i na možnou přenositelnost.

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

B.1 Jazyk a systémové prostředí

Knihovna `SPlan` byla implementována v jazyce *C++*, základní literaturou k tomuto jazyku je publikace [Str86] od Bjarna Stroustrupa, autora jazyka. 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 [Sgi05].

Jako systémové prostředí pro experimentální výzkum byl zvolen operační systém Linux. Knihovna je vyvíjena pomocí softwarových prostředků běžně dodávaných jako součást

[1] Autorovým cílem je, aby časem knihovna dospěla v plánovací systém, jehož výkonné jádro by tvořily algoritmy pro práci s podmínkami. Tento cíl je v současnosti realizován.

většiny distribucí operačního systému Linux, tedy hlavně pomocí kompilátoru `gcc` a souvisejících nástrojů. Překlad knihovny dále využívá standardních nástrojů pro automatickou kompilaci jako jsou `automake`, `autoconf`, atd..

B.2 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.

Do druhé kategorie by spadaly všechny aktivní objekty, které typicky 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.. Jinak řečeno, konstrukce, které jsou nějakým způsobem samy od sebe aktivní.

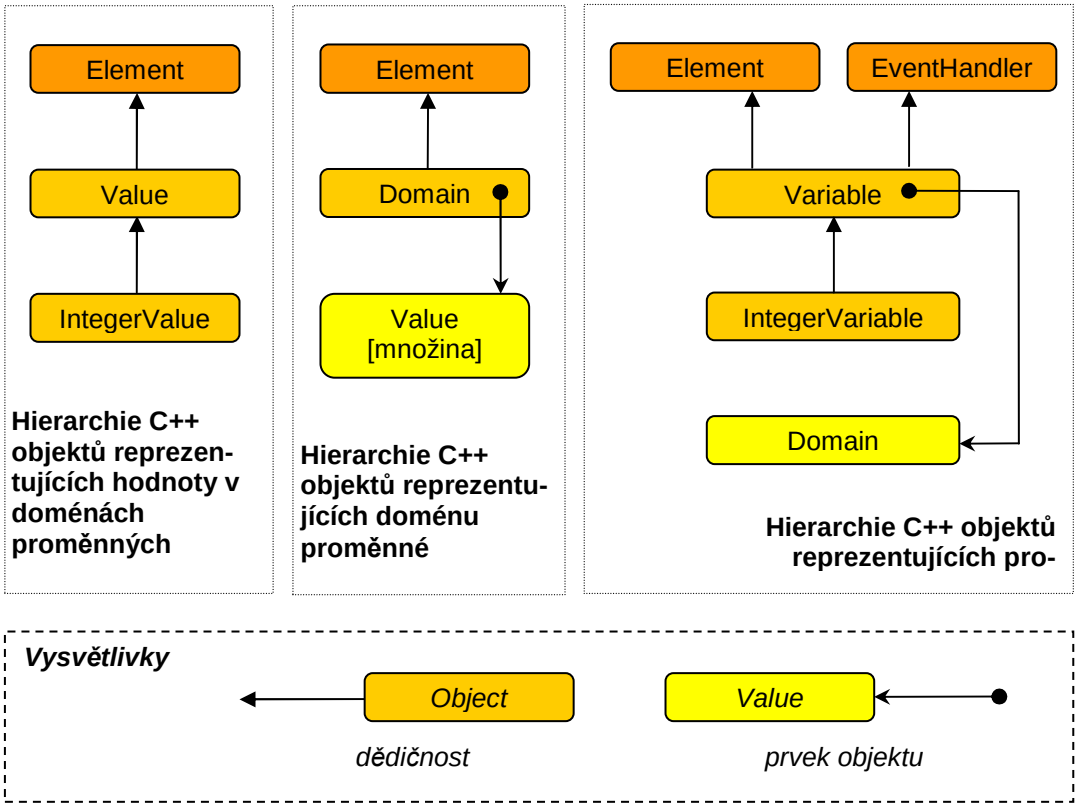
Do poslední kategorie patří všechny ostatní objekty, jež nespadají ani do první ani do druhé kategorie objektů. Příkladem takového objektu je třeba objekt zajišťující šíření událostí mezi různými částmi knihovny (přičemž událostí je například odstranění prvku).

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

B.2.1 Reprezentace problému splňování podmínek

V této sekci stručně charakterizujeme objekty realizující reprezentaci problémů splňování podmínek. Obrázky B.1 až B.3 ukazují hierarchii konkrétních objektů (tříd) jazyka C++, které zajišťují tuto reprezentaci. Obyčejné šipky znázorňují dědičnost mezi objekty, resp. specializace objektů, šipky začínající zvýrazněným počátkem znázorňují prvky objektů.

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.



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

Hodnoty v doménách proměnných (obrázek B.1) 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 lineární uspořádání, což je důležité pro efektivní reprezentaci domény proměnné.

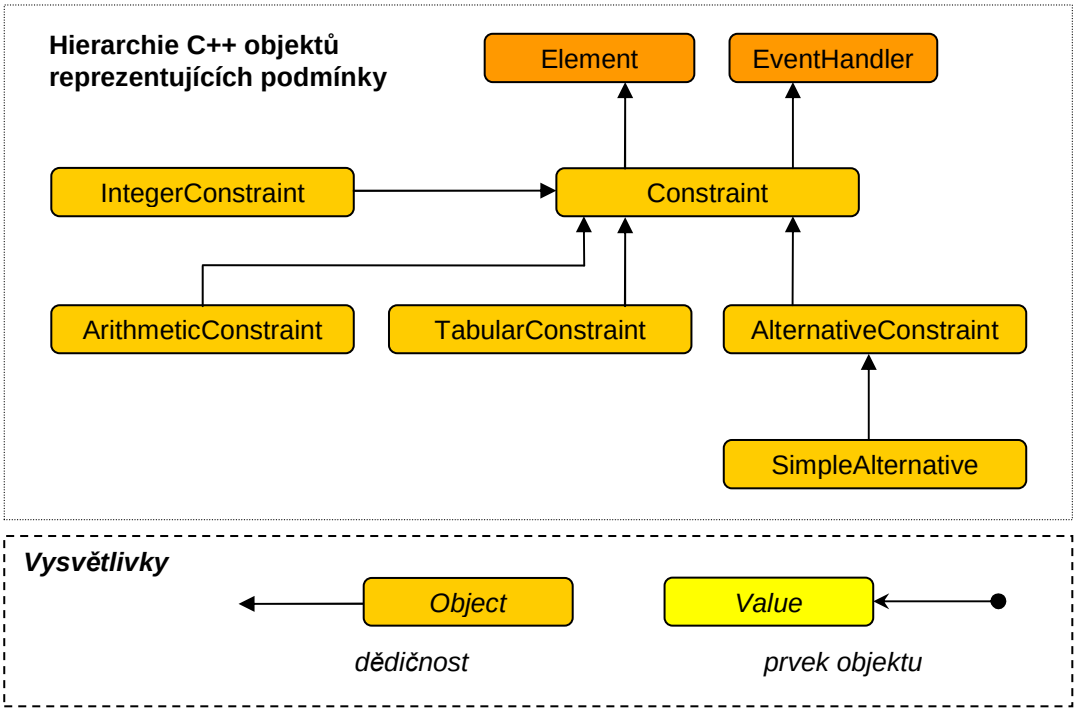
Dalším důležitým objektem je třída Domain reprezentující doménu proměnné (obrázek B.1). Prvky objektu Domain jsou tři uspořádané seznamy hodnot reprezentující původní a aktuální doménu a seznam hodnot momentálně nepřítomných v aktuální doméně. Objekt Domain poskytuje sadu operací nad doménou proměnné, 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 (obrázek B.1). Objekt Variable poskytuje operace přidání a odebrání hodnoty z aktuální domény proměnné, přičemž může eventuálně 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 (obrázek B.2). 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 pro danou 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 při implementaci nové odvozené podmínky je nutné povinně dodat 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, které pracují podle základních definicí uvedených operací. Objekt Constraint může rovněž, pokud to uživatel požaduje, zaznamenávat informace pro návrat do předchozích stavů (což se zde týká stavů svazovaných proměnných).

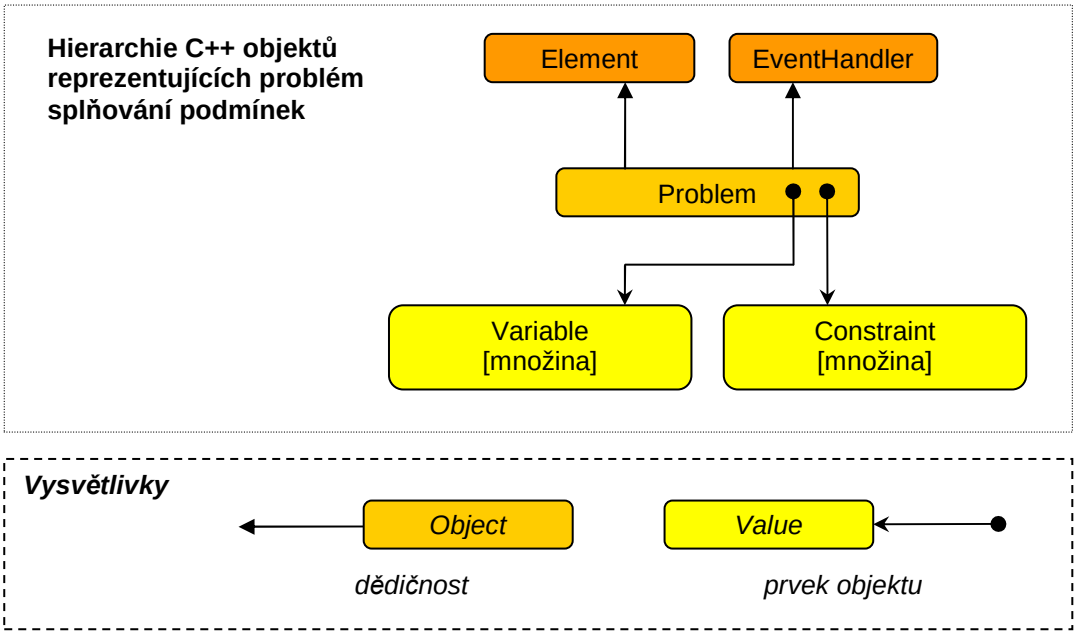
Objekt IntegerConstraint realizuje rovnosti a nerovnosti mezi dvojicemi proměnných. Tato třída poskytuje specializované efektivní verze operací HAS-SUPPORT, FILTER a EXTEND založené na intervalových výpočtech. Podmínka implementovaná objektem ArithmeticConstraint realizuje lineární aritmetické relace mezi dvojicemi proměnných,

nicméně narozdíl od IntegerConstraint využívá pouze výchozích implementací operací HAS-SUPPORT, FILTER a EXTEND. Podmínka realizovaná třídou TabularConstraint představuje podmínku reprezentovanou výčtem všech kompatibilních dvojic hodnot. Pro generování náhodných podmínek v náhodných problémech jsou využívány podmínky právě typu TabularConstraint.



Obrázek B.2: Hierarchie C++ objektů reprezentujících podmínku problému splňování podmínek

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 negování, což je dáno tím, že při prohledávání prostoru všech možných ohodnocení pomocí ohodnocujících podmínek je nutné mít k dispozici pokrytí všech možných alternativ pro daný bod rozhodování, tj. podmínku v pozitivním i negativním tvaru (pokud jsou alternativy dvě).

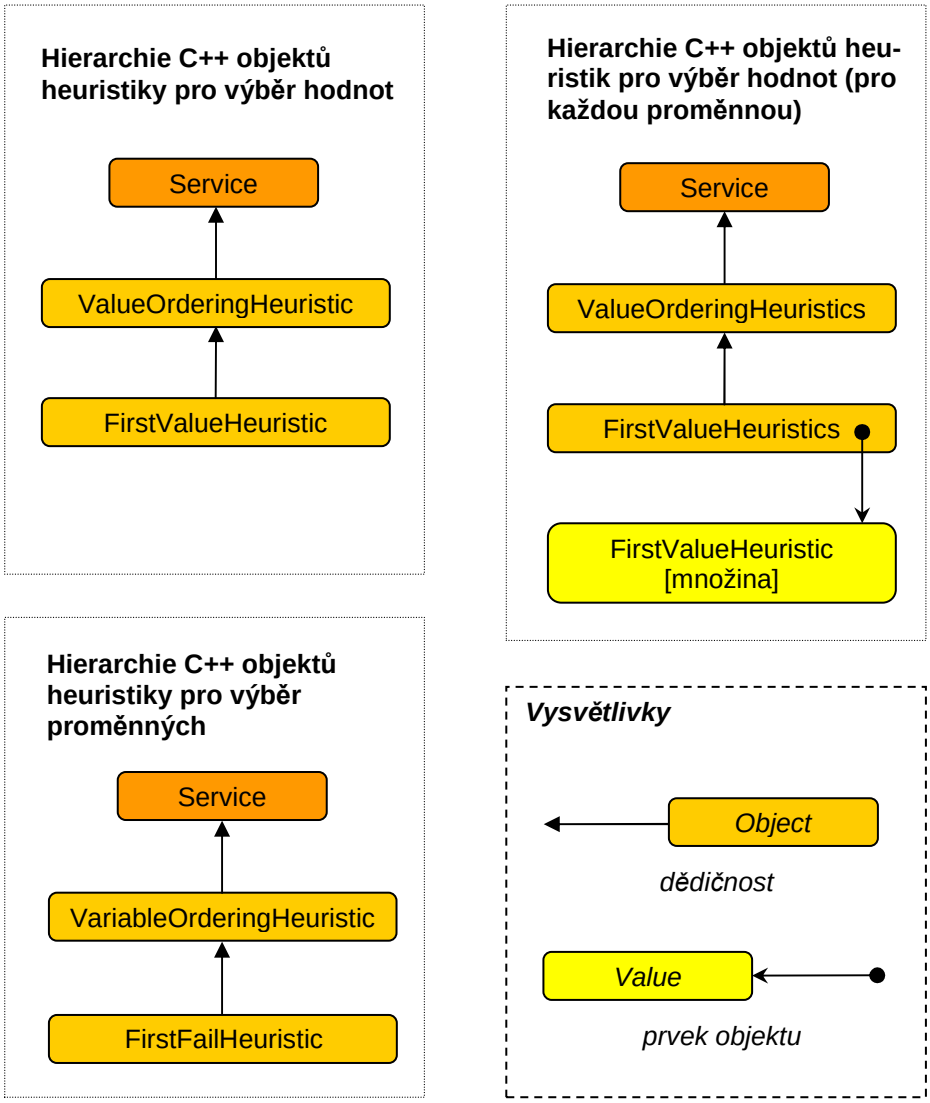


Obrázek B.3: Hierarchie C++ objektů reprezentujících problém splňování podmínek

Posledním objektem, který v této sekci zmíníme, je třída Problem (obrázek B.3). 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 žádný druh konzistence) 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.2.2 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.4 až B.6.

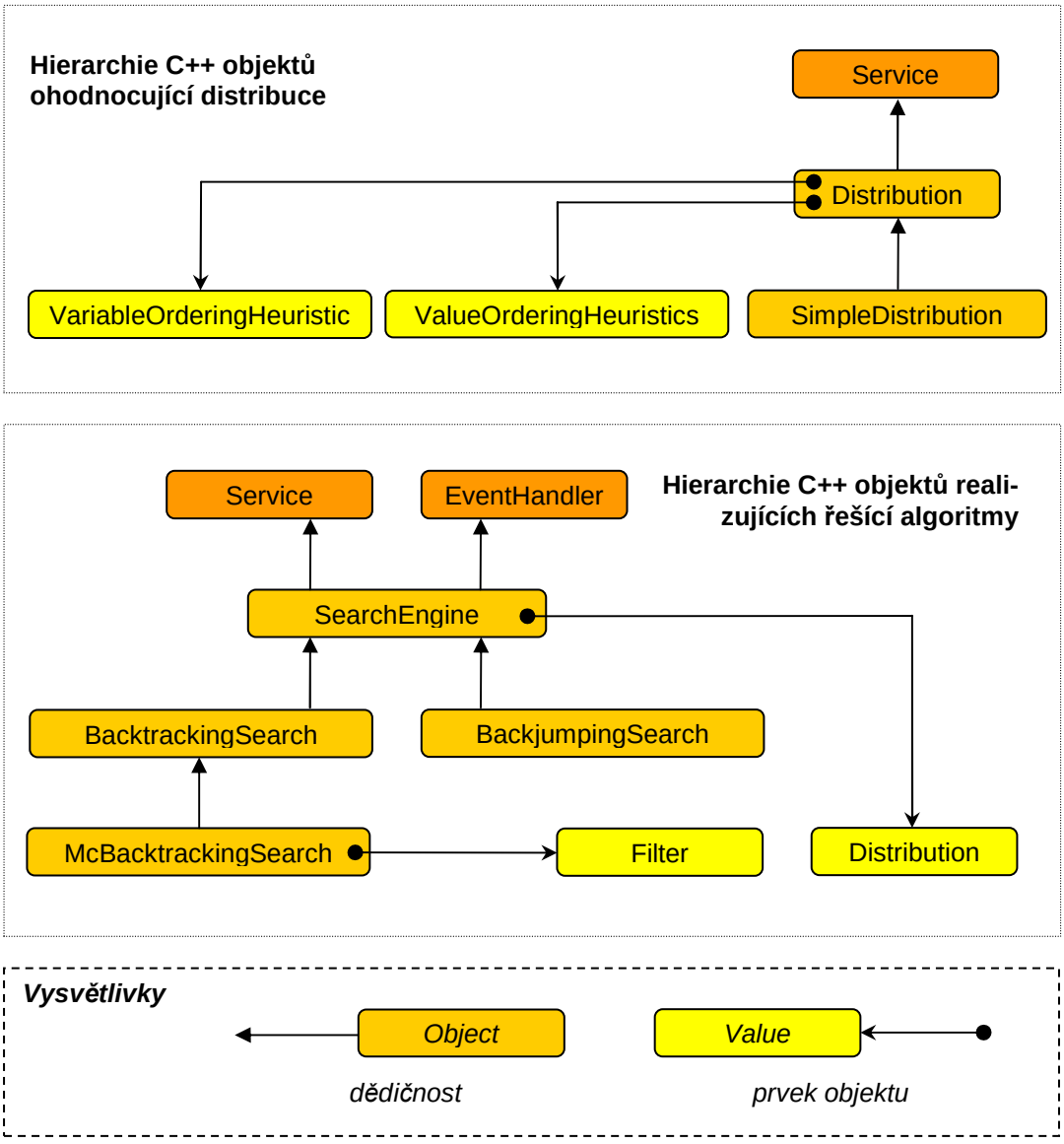


Obrázek B.4: Hierarchie C++ objektů reprezentujících ohodnocující heuristiky

Bázovým objektem, tj. objektem, jenž je výchozím bodem pro odvozování, všech aktivních tříd je třída Service. Nejprve popíšeme objekty, které slouží jako pomůcky při řešení problému splňování podmínek.

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, ukazují obrázky B.4 a B.5. 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].



Obrázek B.5: Hierarchie C++ objektů tvořících řešící algoritmus

Implementovány jsou heuristiky pro výběr hodnot z aktuálních domén proměnných, jež je výhodné v daném kroku řešícího algoritmu vybrat pro ohodnocení (obrázek B.4). 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. *alternativ*. O tomto způsobu ohodnocování pojednává například publikace [Sch02]. Třída Distribution slouží ke generování ohodnocujících podmínek (obrázek B.5), 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 (obrázek B.5), 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.

Třída Filter (obrázek B.6) představuje základ všech konzistenčních algoritmů, její specializací tohoto objektu je například třída AC3Filter, která implementuje konzistenční algoritmus AC-3, přesněji řečeno, jeho doplněnou verzi AC-3', jež byla popsána v kapitole 3.

Dynamické konzistenční algoritmy (obrázek B.6) 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, ACDC2DynamicFilter, ACDC1BDynamicFilter, AC3_1DCDynamicFilter, ACDC2BDynamicFilter, ACDC2CDynamicFilter, ACDC2D-

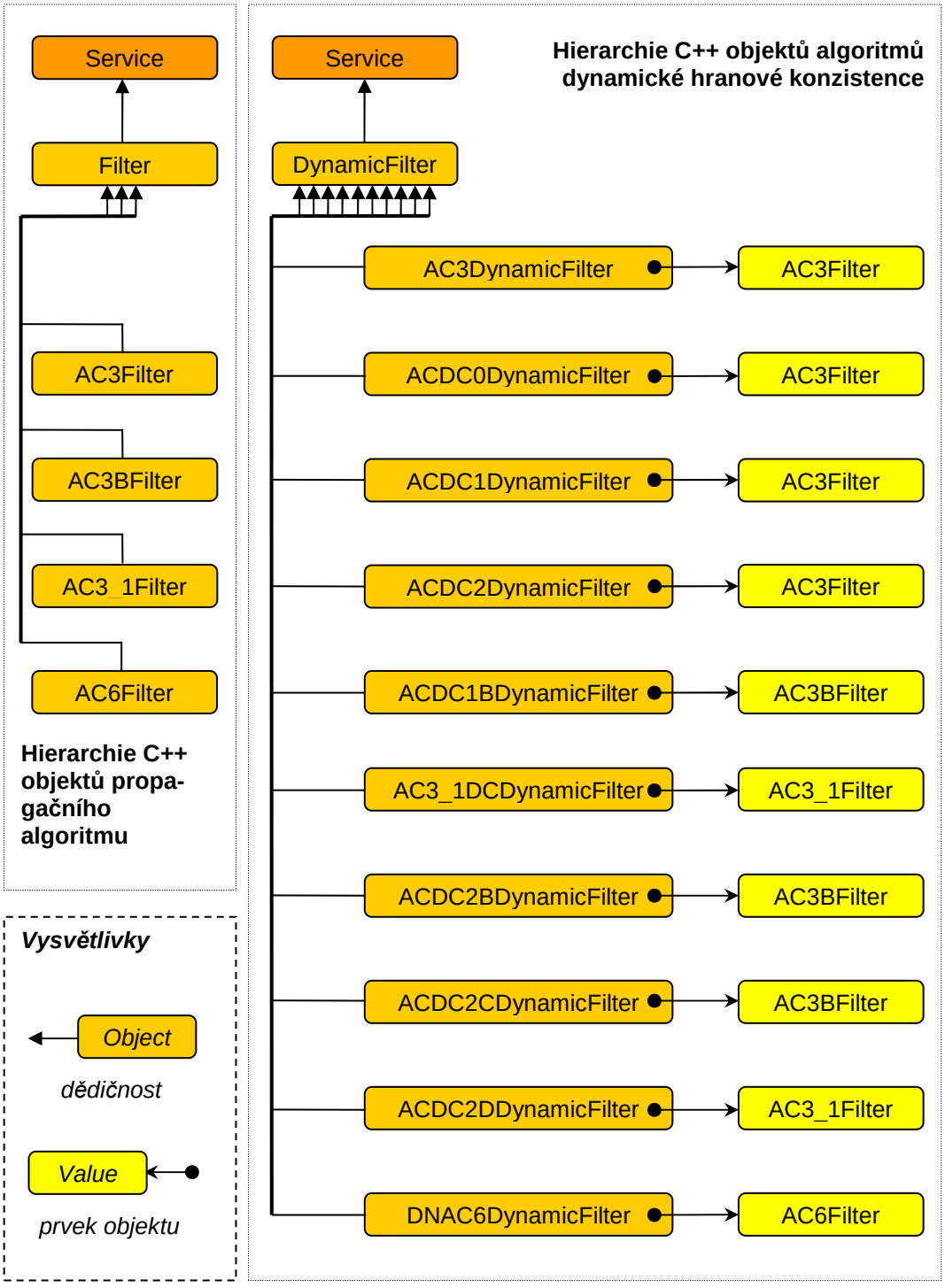
-DynamicFilter a DNAC6DynamicFilter, které reprezentují konkrétní algoritmy pro dynamickou hranovou konzistenci. V knihovně `SPlan` je užíváno trochu jiného označení algoritmů, než je běžné ve vědecké literatuře. Je tomu tak proto, že od některých algoritmů bylo implementováno více zkušebních verzí a bylo potřeba je pracovně také nějak pojmenovat. Korespondenci mezi jmény algoritmů, které se běžně užívají ve vědeckých článcích, a pojmenováním v rámci knihovny ukazuje tabulka B.1.

Jméno algoritmu užívané v rámci knihovny `SPlan`	**Odpovídající jméno algoritmu užívané ve vědeckých článcích**
`acdc0`	AC|DC-0 *(nepublikován)*
`acdc1`	AC|DC-1 *(nepublikován)*
`acdc1b`	AC|DC
`acdc2`	AC|DC-2
`acdc2b`	*(nepublikován)*
`acdc2c`	AC|DC-2i
`acdc2d`	AC3.1|DC-2i
`ac3_1dc`	AC3.1|DC
`dnac6`	DNAC-6

Tabulka B.1: Korespondence jmen algoritmů v rámci knihovny `SPlan` s obecně užívanými jmény

Zastavme se u několika implementovaných dynamických algoritmů. Dynamický algoritmus AC3 je tím ve kapitole 4 zmiňovaným referenčním dynamickým algoritmem, kdy je konzistenční procedura AC-3 pro modifikovaný problém spouštěna pokaždé od začátku. Další algoritmy - AC|DC-0, AC|DC-1 a AC|DC-2, také používají pro svou funkci užívají propagační algoritmus AC-3 (AC-3'), který uchovávají jako svůj vnitřní prvek. Směrová verze AC-3, která je zde označovaná jako `AC3B`, je základem pro algoritmy `ACDC1B`, `ACDC2B` a `ACDC2C`. Podobně AC-3.1 je základem pro algoritmy `AC3_1DC` a `ACDC2D`.

Implementace všech uvedených dynamických algoritmů prakticky přesně odpovídá programovému zápisu z kapitoly 3. Totéž platí i pro konzistenční procedury z kapitoly 2.



Obrázek B.6: Hierarchie objektů reprezentující algoritmy dynamické hranové konzistence

B.3 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

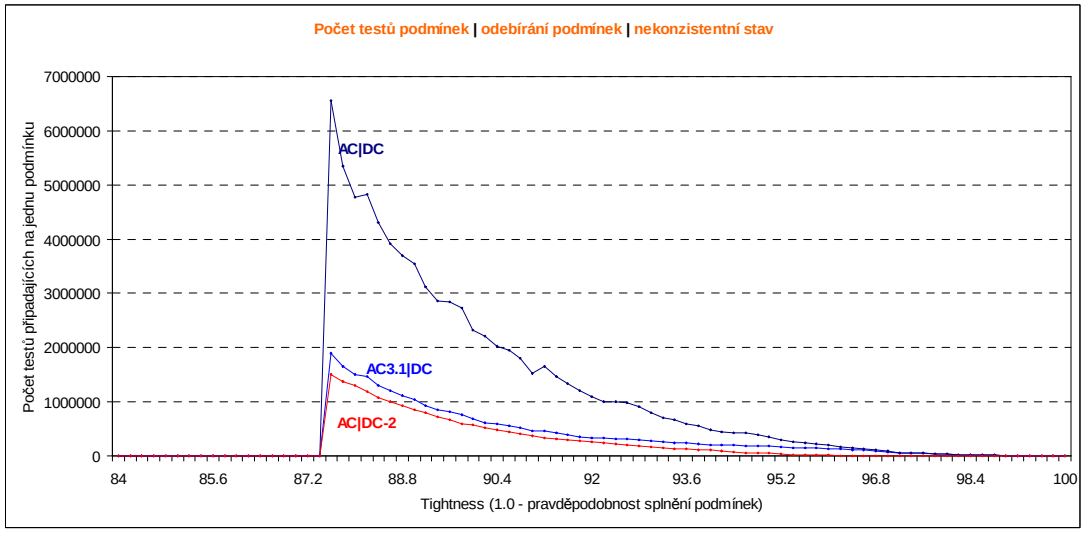
Další empirické testy

Do této přílohy byl přenechán kompletní přehled výsledků provedených testů. V několika následujících grafech jsou shrnuty další výsledky, které jsme neuváděli v rámci 4. kapitoly, ale na jejichž základě jsme také činili závěry uvedené v kapitole 4.

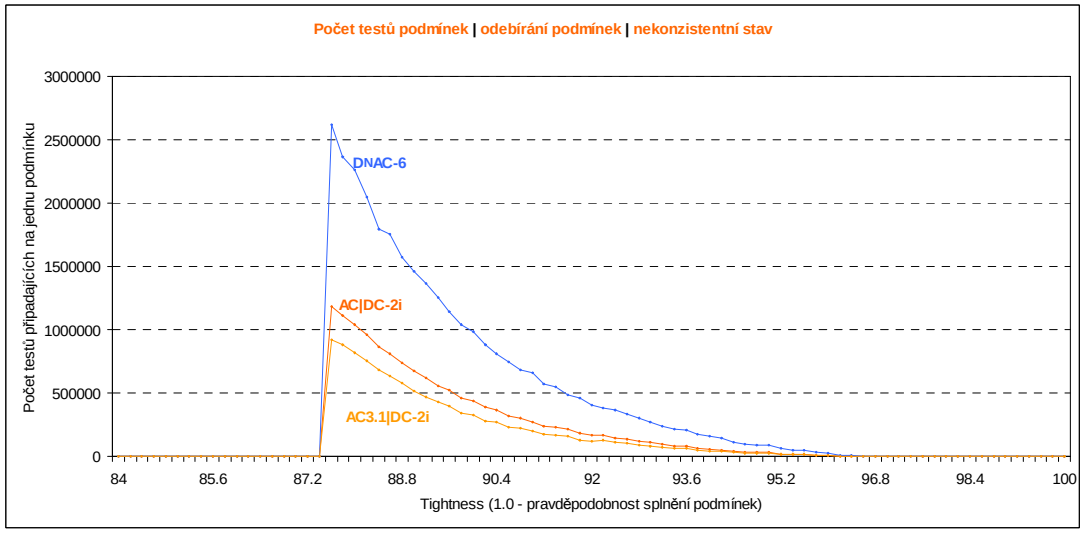
Jak už bylo zmíněno ve 4. kapitole, testy algoritmů byly prováděny jednak s řídkým vzorkováním širšího rozsahu parametru *tightness* a jednak s hustým vzorkováním parametru *tightness* v oblasti fázového přechodu. V kapitole 4 jsme uvedli pouze grafy s řídkým pokrytím širokého rozsahu parametru. Zde grafy s širším rozsahem parametru doplníme ještě o několik dalších (grafy pro nekonzistentní stavy a grafy ukazující celkový čas běhu). Poté bude následovat sada grafů získaných při hustém vzorkování oblasti fázového přechodu parametru *tightness*.

Podklady pro grafy, které srovnávají algoritmy při odebírání podmínky z nekonzistentního stavu, byly získány během odebírání první (a jediné) podmínky, která způsobila nekonzistenci. U grafů, kde figuruje celkový čas běhu algoritmu, 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.

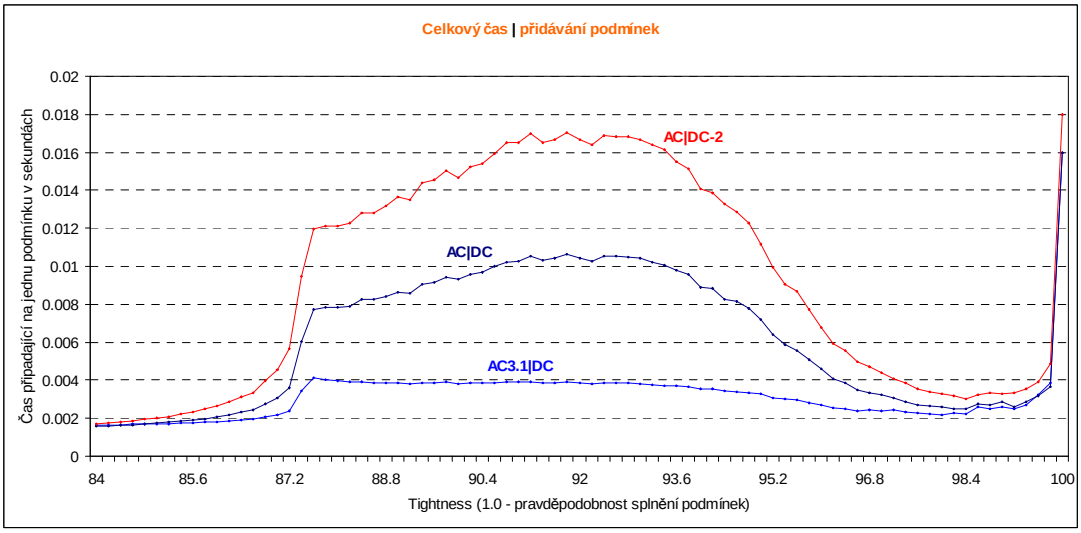
Zdrojová data, na jejichž základě byly vytvořeny grafy v této příloze, jsou uložena na přiloženém CD.



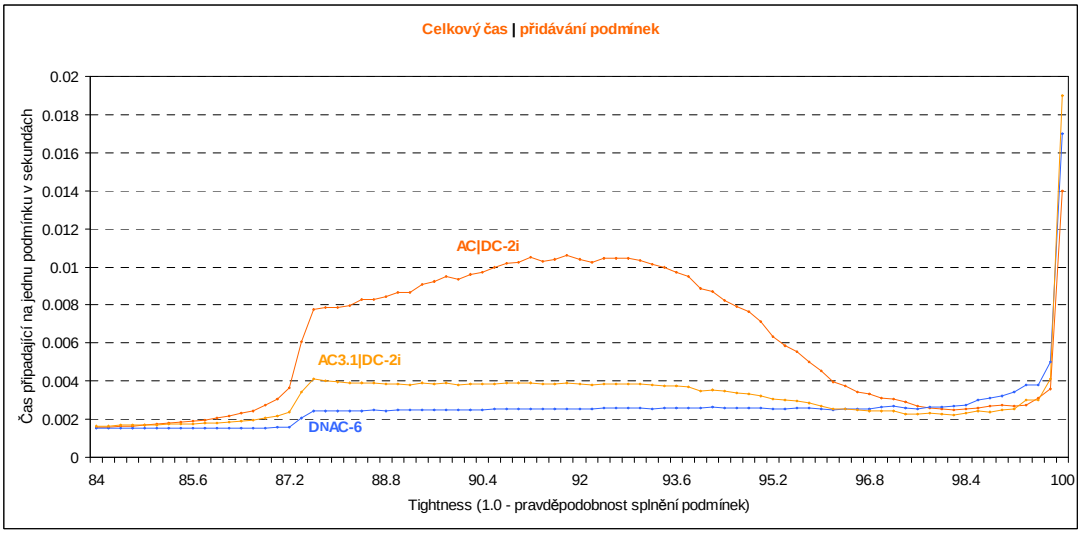
Graf C.1: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle počtu testů při odebírání podmínek z nekonzistentního stavu



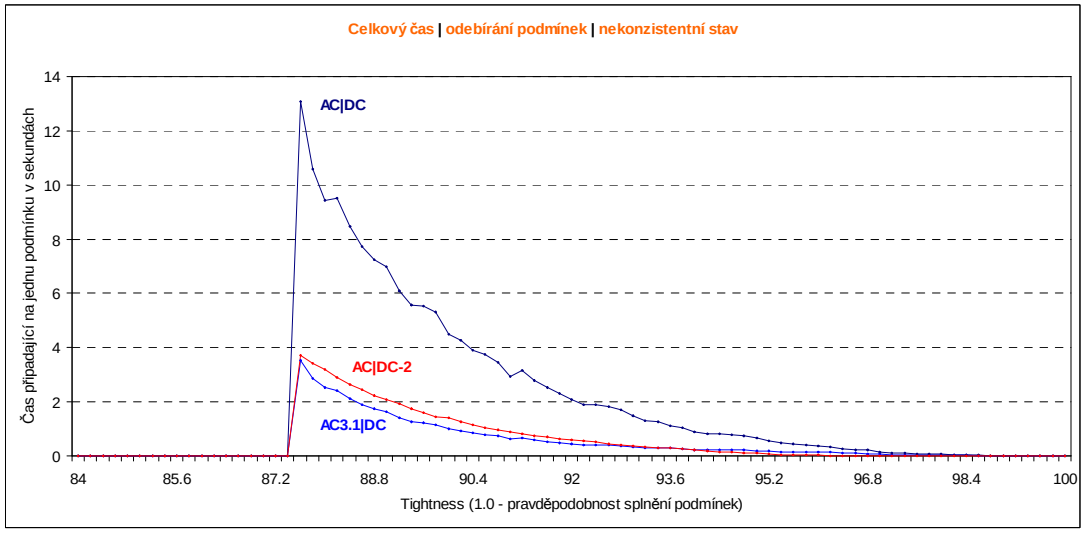
Graf C.2: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle počtu testů při odebírání podmínek z nekonzistentního stavu



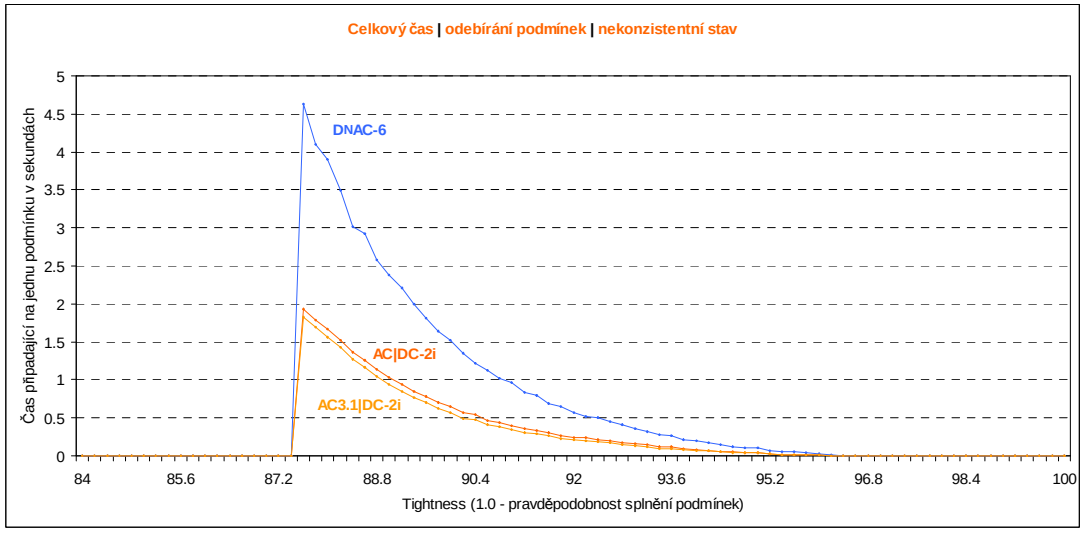
Graf C.3: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle celkového času běhu při přidávání podmínek



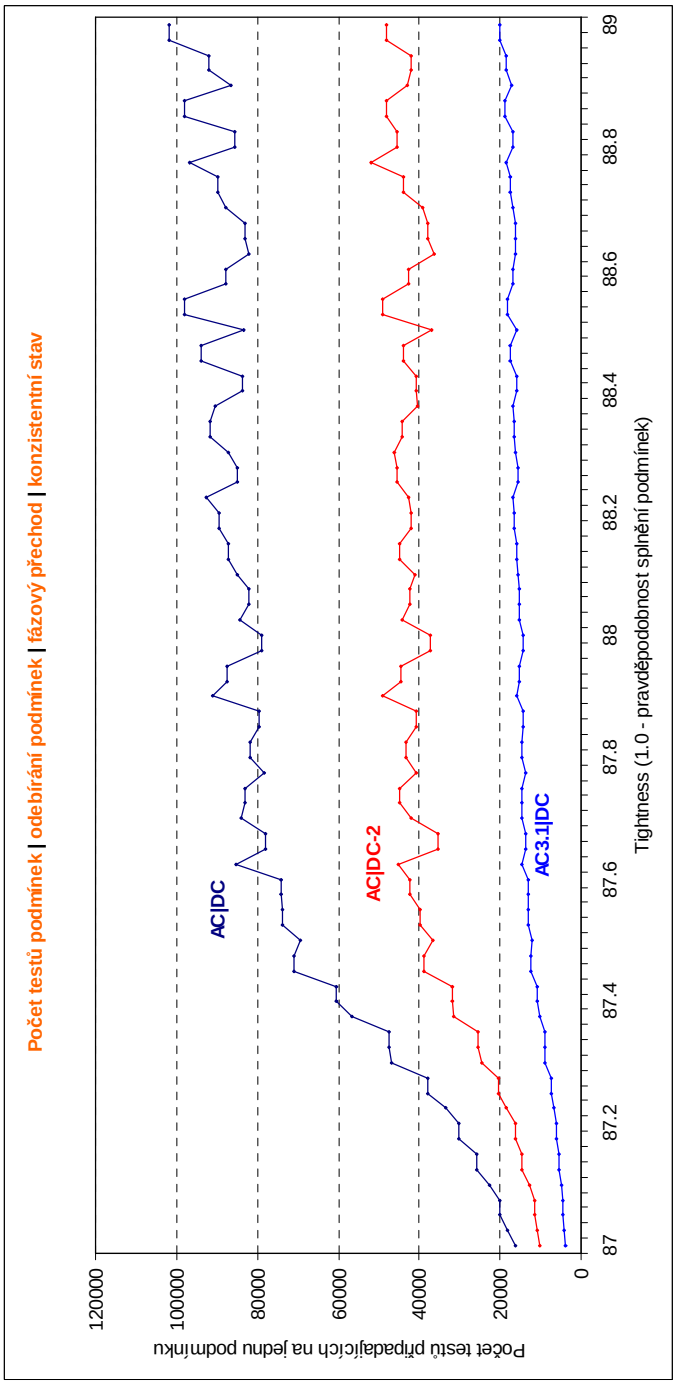
Graf C.4: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle celkového času běhu při přidávání podmínek



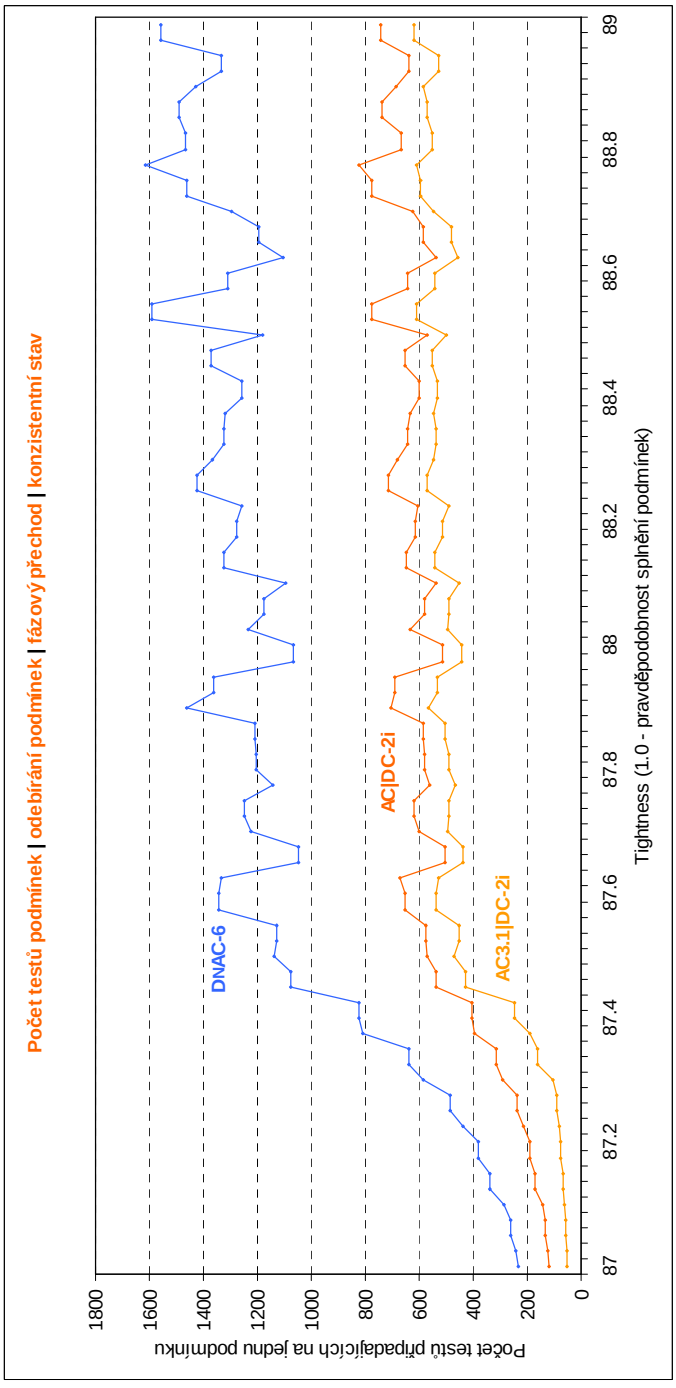
Graf C.5: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle celkového času běhu při odebírání podmínek z nekonzistentního stavu



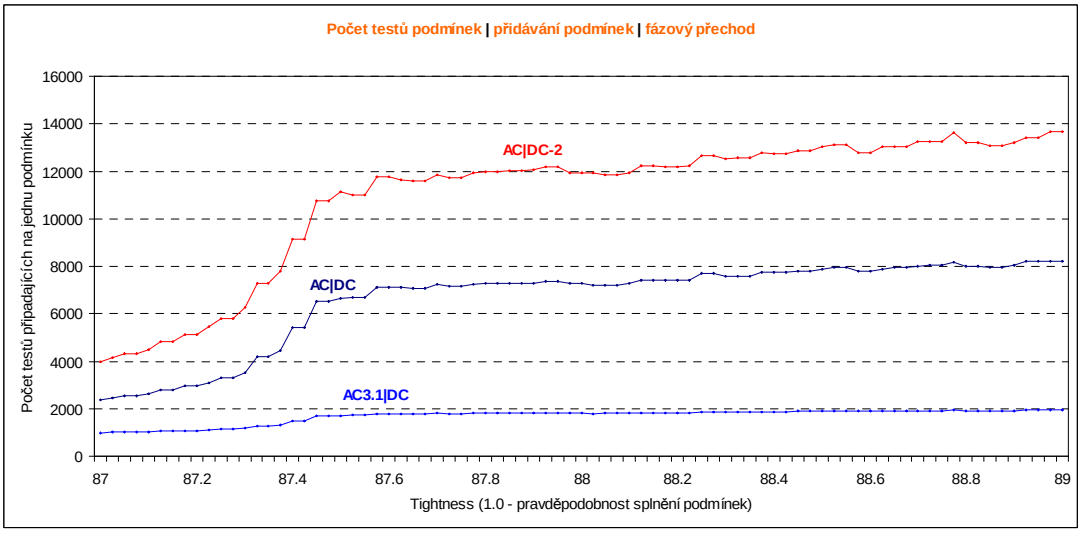
Graf C.6: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle celkového času běhu při odebírání podmínek z nekonzistentního stavu



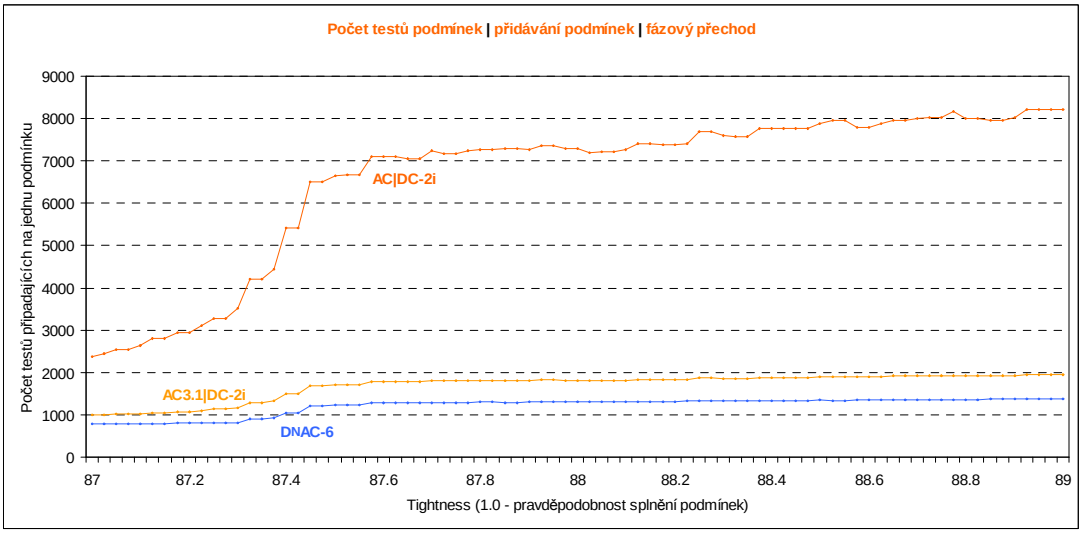
Graf C.7: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle počtu testů při odebírání podmínek z konzistentního stavu (fázový přechod)



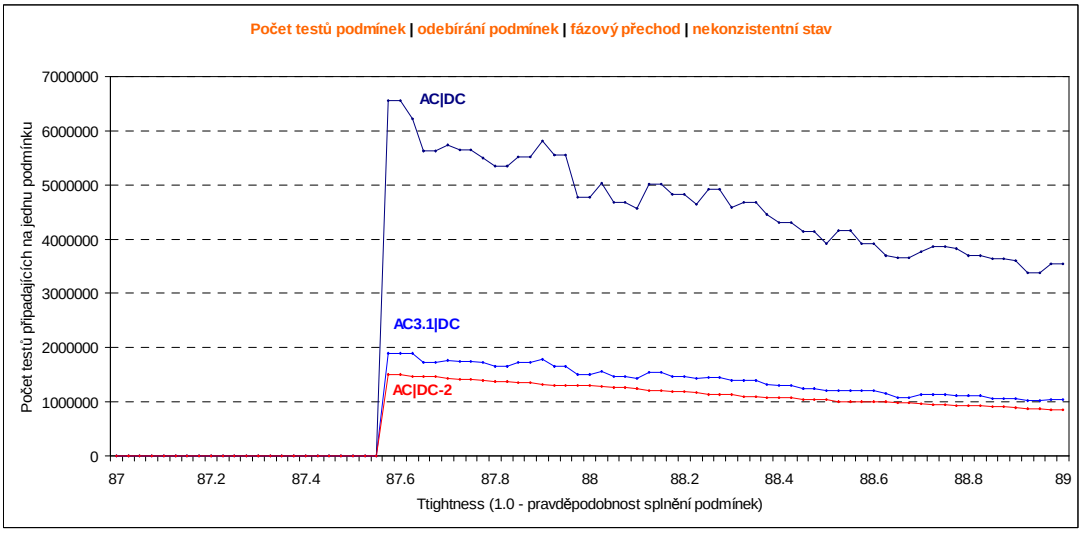
Graf C.8: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle počtu testů při odebírání podmínek z konzistentního stavu (fázový přechod)



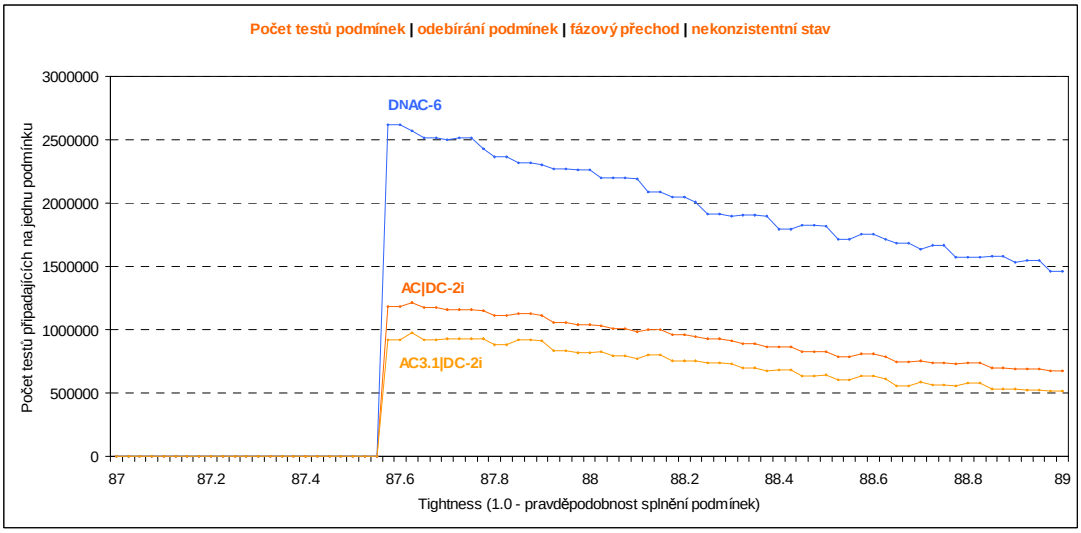
Graf C.9: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle počtu testů při přidávání podmínek (fázový přechod)



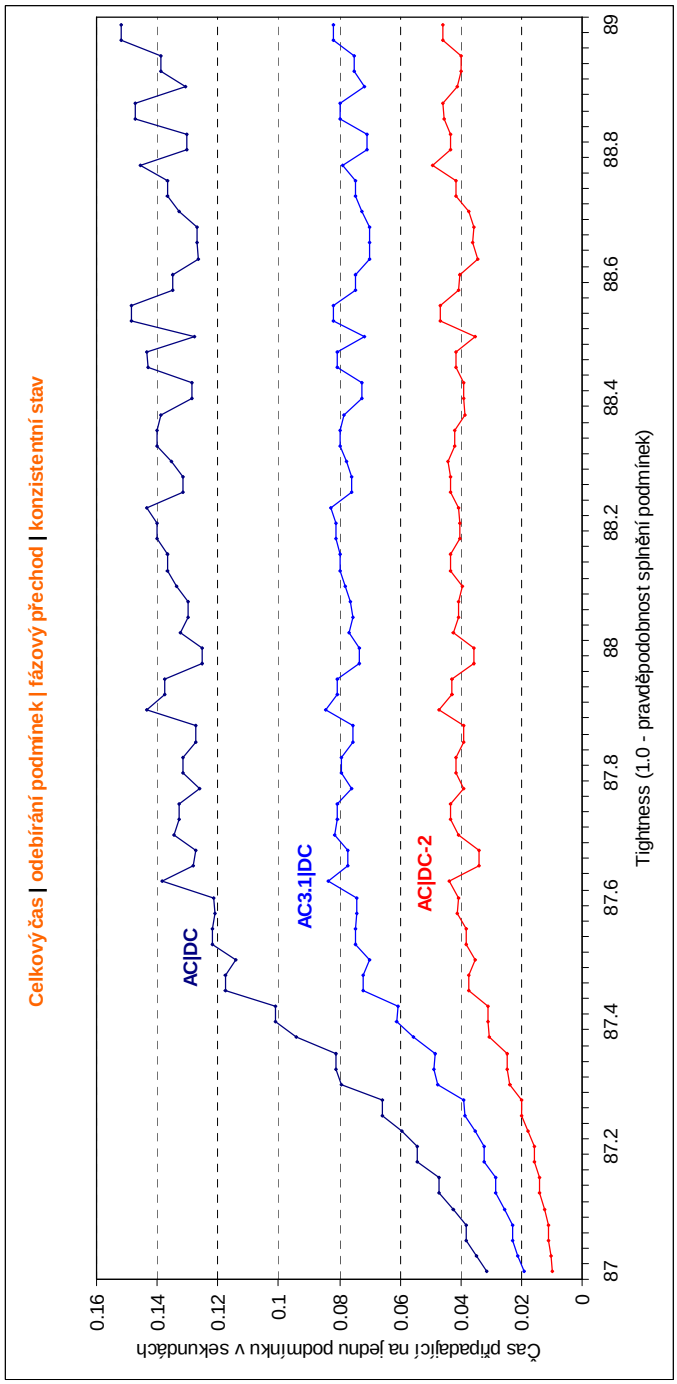
Graf C.10: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle počtu testů při přidávání podmínek (fázový přechod)



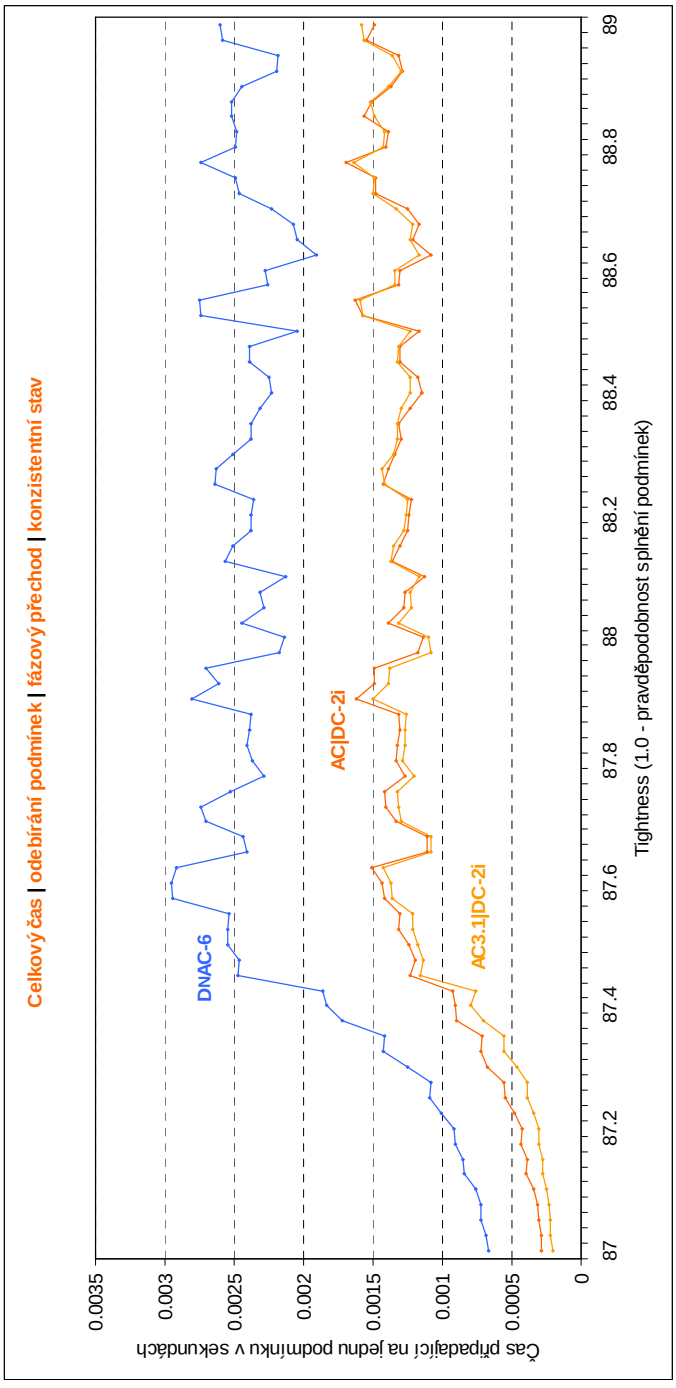
Graf C.11: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle počtu testů při odebírání podmínek z nekonzistentního stavu (fázový přechod)



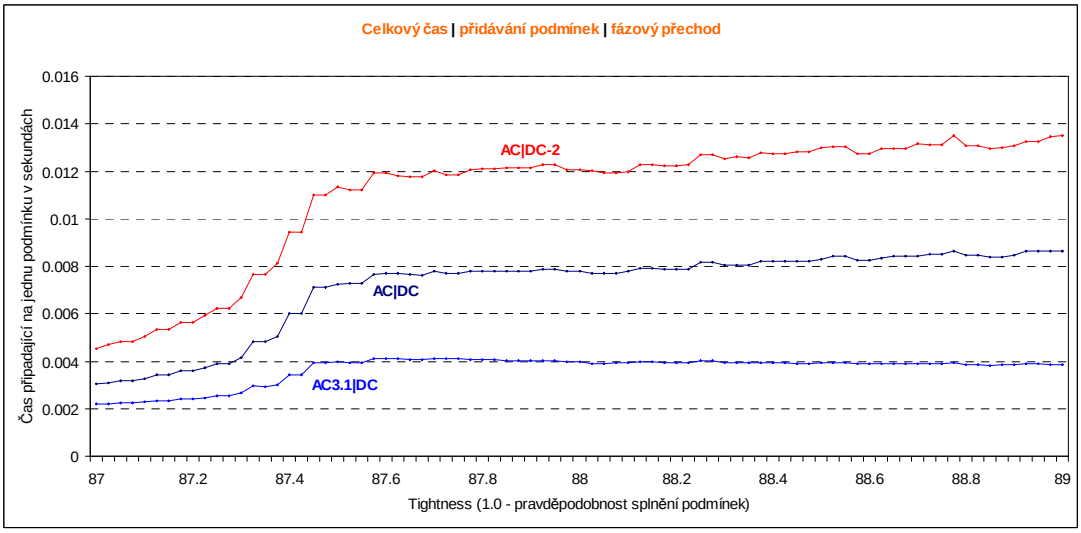
Graf C.12: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle počtu testů při odebírání podmínek z nekonzistentního stavu (fázový přechod)



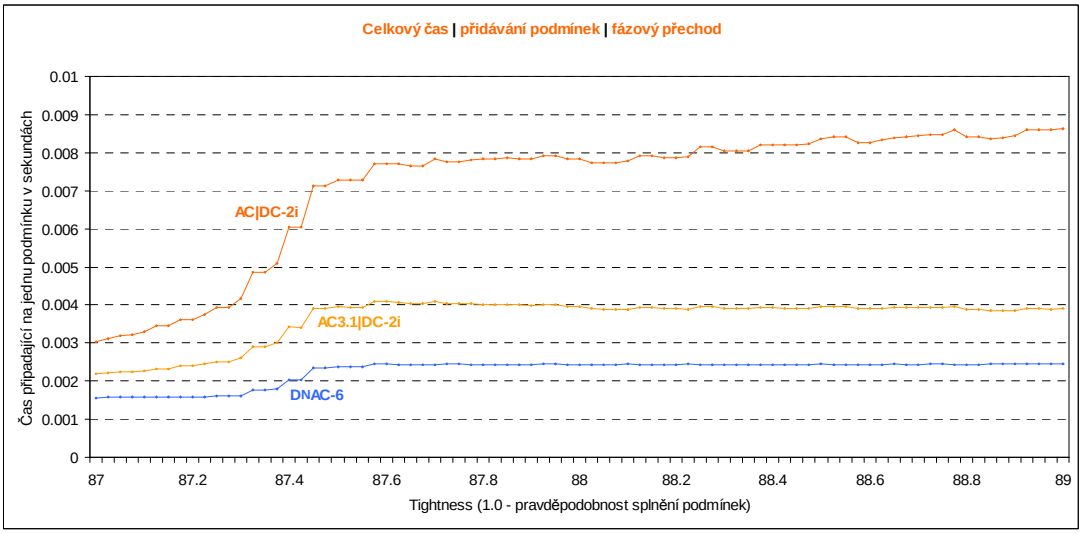
Graf C.13: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle celkového času běhu při odebírání podmínek z konzistentního stavu (fázový přechod)



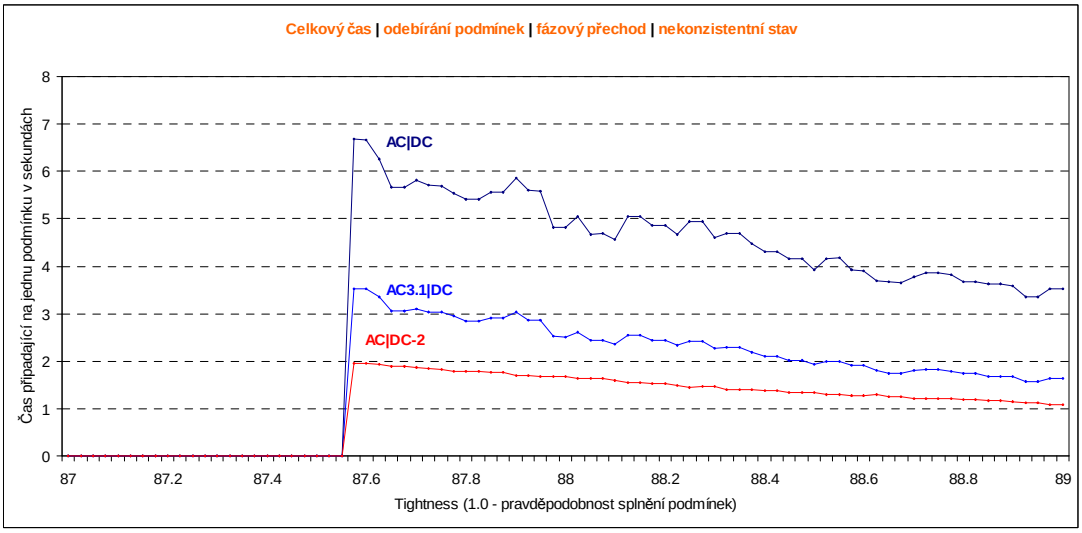
Graf C.14: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle celkového času běhu při odebírání podmínek z konzistentního stavu (fázový přechod)



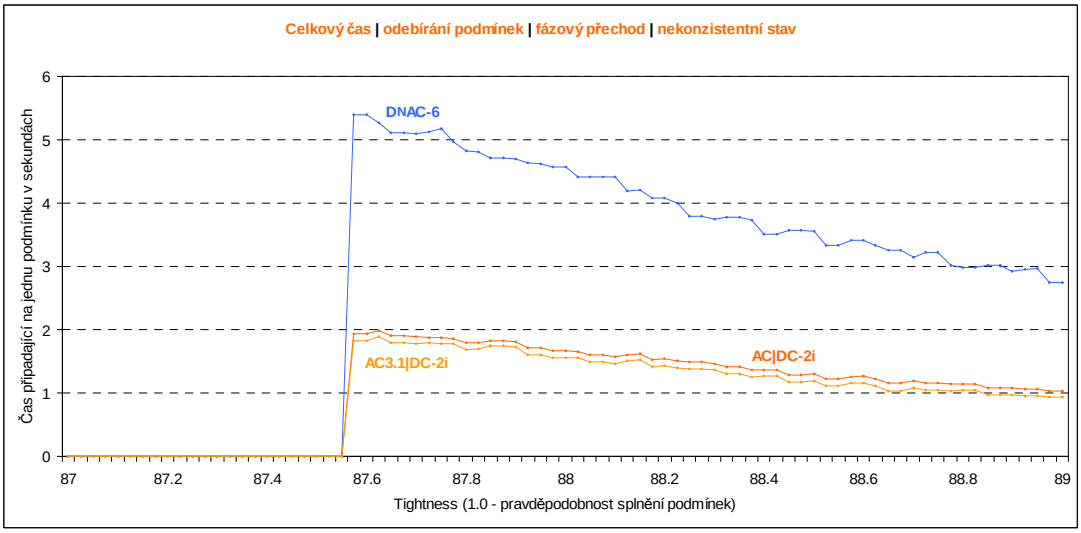
Graf C.15: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle celkového času běhu při přidávání podmínek (fázový přechod)



Graf C.16: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle celkového času běhu při přidávání podmínek (fázový přechod)



Graf C.17: Srovnání algoritmů AC|DC, AC3.1|DC a AC|DC-2 podle celkového času běhu při odebírání podmínek z nekonzistentního stavu (fázový přechod)



Graf C.18: Srovnání algoritmů DNAC-6, AC|DC-2i a AC3.1|DC-2i podle celkového času běhu při odebírání podmínek z nekonzistentního stavu (fázový přechod)

Literatura

[Bak95] Andrew B. Baker: Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.
Doctoral Thesis, University of Oregon, Eugene, OR, USA, 1995.

[Bar98] Roman Barták: On-line Guide to Constraint Programming.
http://kti.ms.mff.cuni.cz/constraints/, Matematicko-fyzikální fakulta Univerzity Karlovy, Praha, Česká republika, 1998.

[BaSu05] Roman Barták, Pavel Surynek: An Improved Algorithm for Maintaining Arc Consistency in Dynamic Constraint Satisfaction Problems.
In Ingrid Russell, Zdravko Markov, editors, Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005), Clearwater Beach, FL, USA, 161-166, AAAI Press, 2005.

[BeCo94] Christian Bessière, Marie-Odile Cordier: Arc-Consistency and Arc-Consistency Again.
Artificial Intelligence 65, 179-190, 1994.

[BeRe01] Christian Bessière, Jean Charles Régin: Refining the Basic Constraint Propagation Algorithm.
In Proceedings of the 17th International Joint Conference in Artificial Intelligence (IJCAI-01), Seattle, WA, USA, 309-315, Morgan Kaufmann, 2001.

[Bes91] Christian Bessière: Arc-Consistency in Dynamic Constraint Satisfaction Problems.
In Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91), Anaheim, CA, USA, 221-226, AAAI Press, 1991.

[Bes92] Christian Bessière: Arc-Consistency for Non-Binary Dynamic Constraint Satisfaction Problems.
In Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92), 23-27, Vienna, Austria, John Wiley and Sons, 1992.

[BMR04] Roman Barták, Tomáš Müller, Hana Rudová: A New Approach to Modeling and Solving Minimal Perturbation Problem.
In Proceedings of the CSCLP-2003 Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, Budapest, Hungary, LNCS 3010, Springer-Verlag, 233-249, 2004.

[CDR96] Philippe Codognet, Daniel Diaz, Francesca Rossi: Constraint Retraction in Finite Domains.
In Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS-96), Hyderabad, India, LNCS 1180, 168-179, Springer-Verlag, 1996.

[Deb96] Romuald Debruyne: Arc-Consistency in Dynamic CSPs Is No More Prohibitive.
In Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-96), Toulouse, France, 299-306, IEEE Computer Society, 1996.

[DeDe88] Rina Dechter, Avi Dechter: Belief Maintenance in Dynamic Constraint Networks.
In Proceedings the 7th National Conference on Artificial Intelligence (AAAI-88), St. Paul, MN, USA, 37-42, AAAI Press, 1988.

[Chip05] CHIP V5
http://www.ilog.com, Cosytec - Complex Systems Technologies SA, France, 2005.

[GaWe94] Christine Gaspin, Eric Westhof: The Determination of the Secondary Structures of RNA as a Constraint Satisfaction Problem.
In S. Schulze-Kremer, editor, Advances in Molecular Bioinformatics 22, Frontiers in Artificial Intelligence and Applications, IOS Press, 1994.

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

[Har95] William D. Harvey: Nonsystematic Backtracking Search.
Doctoral Thesis, University of Oregon, Eugene, OR, USA, 1995.

[Ilog05] ILOG Solver.
http://www.ilog.com, ILOG SA, France, 2005.

[JDB00] Narendra Jussien, Romuald Debruyne, Patrice Boizumault: Maintaining Arc-Consistency within Dynamic Backtracking.
In Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP-2000), Singapore, LNCS 1894, 249-261, Springer-Verlag, 2000.

[JJNVC90] P. Janssen, P. Jégou, B. Nouguier, M. Vilarem, B. Castro: SYNTHIA: Assisted Design of Peptide Synthesis Plans.
In New Journal of Chemistry, 14-12, 969-976, 1990.

[JuBo97] Narendra Jussien, Patrice Boizumault: Dynamic Backtracking with Constraint Propagation: Application to Static and Dynamic CSPs.
In Proceedings of the CP-97 Workshop on the Theory and Practice of Dynamic Constraint Satisfaction, Schloss Hagenberg, Austria, 1997.

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

[MiFa90] Sanjay Mittal, Brian Falkenhainer: Dynamic Constraint Satisfaction Problems.
In Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90), Boston, MA, USA, 25-32, AAAI Press / The MIT Press, 1990.

[Mo03] Malek Mouhoub: Arc Consistency for Dynamic CSPs.
In Vasile Palade, Robert J. Howlett, Lakhmi C. Jane, editors, Proceedings of the 7th International Conference on Knowledge Based Intelligent Information and Engineering Systems - Part I (KES-2003), Oxford, UK, LNCS 2773, 393-400, Springer-Verlag, 2003.

[MoHe86] Roger Mohr, Thomas C. Henderson: Arc and Path Consistency Revised.
Artificial Intelligence 28, 225-233, 1986.

[MoMa88] Roger Mohr, Gérald Masini: Good Old Discrete Relaxation.
In Proceedings of the 8th European Conference on Artificial Intelligence (ECAI-88), Munich, Germany, 651-656, Pitmann Publishing, London, 1988.

[MPSW98] Ewan MacIntyre, Patrick Prosser, Barbara Smith, Toby Walsh: Random Constraint Satisfaction: theory meets practice.
In Michael Maher, Jean-Francois Puget, editors, 4th International Conference on Principles and Practice of Constraint Programming (CP-98), Pisa, Italy, LNCS 1520, 325-339, Springer-Verlag, 1998.

[NeBe94] Bertrand Neveu, Pierre Berlandier: Maintaining Arc-Consistency Through Constraint Retraction: An RMS-free Approach.
In Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-94), New Orleans, LA, USA, 426-431, IEEE Computer Society, 1994.

[RDP90] Francesca Rossi, Vasant Dhar, Charles Petrie: On the Equivalence of Constraint Satisfactions Problems.
In Proceedings of the 9th European Conference on Artificial Intelligence (ECAI-90), Stockholm, Sweden, 550-556, 1990.

[Sgi05] STL - Standard Template Library.
http://www.sgi.com/tech/stl, Silicon Graphics Inc., USA, 2005.

[Sch02] Christian Schulte: Programming Constraint Services: High-Level Programming of Standard and New Constraint Services.
LNCS 2302, Springer-Verlag, 2002.

[SchVe93a] Thomas Schiex, Gérard Verfaillie: Two Approaches to the Solution Maintenance Problem in Dynamic Constraint Satisfaction Problems.
In Proceedings of the IJCAI-93 Workshop on Knowledge-Based Production Planning, Scheduling and Control, Chambery, France, 1993.

[SchVe93b] Thomas Schiex, Gérard Verfaillie: Nogood Recording for Static and Dynamic Constraint Satisfaction Problems.
In Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-93), Boston, MA, USA, 48-55, IEEE Computer Society, 1993.

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

[SuBa04a] Pavel Surynek, Roman Barták: A New Algorithm for Maintaining Arc Consistency After Constraint Retraction.
ITI Series 2004-205, technická zpráva, Institute for Theoretical Computer Science, Matematicko-fyzikální fakulta Univerzity Karlovy, Praha, Česká republika, 2004.

[SuBa04b] Pavel Surynek, Roman Barták: A New Algorithm for Maintaining Arc Consistency After Constraint Retraction.
In Mark Wallace, editor, 10th International Conference Principles and Practice of Constraint Programming (CP-2004), Toronto, Canada, LNCS 3258, 767-771, Springer-Verlag, 2004.

[Sur04] Pavel Surynek: Řešení dynamických problémů s podmínkami.
Diplomová práce, Matematicko-fyzikální fakulta Univerzity Karlovy, Praha, Česká republika, 2004.

[VBChe99] Peter Van Beek, Xinguang Chen: A Constraint Programming Approach to Planning.
In Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99), Orlando, FL, USA, 585-590, AAAI Press, 1999.

[VeSch94a] Gérard Verfaillie , Thomas Schiex: Dynamic Backtracking for Dynamic CSPs.
In Thomas Schiex and Christian Bessière, editors, Proceedings of the ECAI-94 Workshop on Constraint Satisfaction Issues raised by Practical Applications, Amsterdam, The Netherlands, 1994.

[VeSch94b] Gérard Verfaillie , Thomas Schiex: Solution Reuse in Dynamic Constraint Satisfaction Problems.
In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, WA, USA, 585-590, AAAI Press, 1994.

[WaFr98] Richard J. Wallace, Eugene C. Freuder: Stable Solutions for Dynamic Constraint Satisfaction Problems.
In Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP-98), Pisa, Italy, LNCS 1520, 447-461, Springer-Verlag, 1998.

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

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

[ZhYa01] Yuanlin Zhang, Roland H. C. Yap: Making AC-3 an Optimal Algorithm.
In Proceedings of the 17th International Joint Conference in Artificial Intelligence (IJCAI-01), Seattle, WA, USA, 316-321, Morgan Kaufmann, 2001.

Výtisk rigorosní práce číslo: elektronická verze