Úlohy s ohraničeniami

Od teórie k programovaniu

Marián Mach
Ján Paralič

Košice

2000

Ing. Marián Mach, CSc.
Ing. Ján Paralič, PhD.
Katedra kybernetiky a umelej inteligencie
Fakulta elektrotechniky a informatiky
Technická univerzita v Košiciach
machm@tuke.sk
paralic@tuke.sk

Kniha bola vydaná s finančnou podporou Fakulty elektrotechniky a informatiky TU Košice.
Táto práca vznikla aj za podpory VEGA grantu MŠ SR č. 1/5032/98 "Integrácia prostriedkov inteligentných technológií".

Edícia vedeckých spisov Fakulty elektrotechniky a informatiky TU Košice.

Lektorovali: *doc. Ing. Július Csontó, CSc., FEI TU Košice*
Mgr. Roman Barták, Dr., MFF UK Praha
doc. Ing. Mária Bieliková, CSc., FEI STU Bratislava

ISBN 80-88964-48-2

Ani sto neúspechov
neznačí nič,
ak jediný úspešný pokus
bude môcť zmeniť osud sveta.

Arthur C. Clarke: Vesmírna odysea

Predslov

Pomerne značné množstvo úloh je možné formálne zapísať ako úlohy s ohraničeniami. Pre riešenie takýchto úloh existuje značné množstvo metód, zaoberajúcich sa procedurálnou stránkou riešenia týchto úloh. Toto umožní riešiteľovi sústrediť sa na deklaratívnu stránku riešenia problému – explicitnú definíciu objektov vystupujúcich v úlohe a ohraničení definovaných nad týmito objektami.

V rozpore s dôležitosťou, ktorá je metódam pre riešenie úloh s ohraničeniami prisudzovaná v cudzojazyčnej odbornej literatúre, však záujemca iba sporadicky natrafí na publikáciu v slovenskom jazyku, venovanú tejto problematike. Cieľom predkladanej knihy je vyplniť túto medzeru a podať slovenským čitateľom informácie o tom, ako je možné efektívnym spôsobom riešiť úlohy s ohraničeniami.

Kniha je v zásade rozdelená na štyri časti. Prvá časť zavádza základné pojmy z oblasti ohraničení. Tieto pojmy sú ilustrované na dvoch jednoduchých príkladoch, ktoré slúžia ako ilustračný nástroj aj v ďalších častiach knihy. Druhá časť je venovaná metódam pre riešenie úloh s ohraničeniami, známym pod spoločným označením *metódy splňania ohraničení*. Tretia časť sa zaoberá *logickým programovaním ohraničení*, pričom okrem výkladu princípov tejto oblasti prezentuje programovací jazyk ECL^iPS^e. Štvrtá časť ilustruje riešenie konkrétnych úloh pomocou jazyka ECL^iPS^e, pričom sa venuje ako úlohám s konečnými doménami, tak aj úlohám s nekonečnými doménami. Prílohy doplňujú hlavný text a prezentujú informácie, užitočné pre pochopenie hlavného textu a orientáciu v ňom. Ich súčasťou je aj zoznam zabudovaných ohraničení jazyka ECL^iPS^e.

Kniha sa nesnaží byť vyčerpávajúcim textom z predkladanej oblasti a v žiadnom prípade si nerobí nárok na úplnosť. Jej cieľom je priblížiť jednu časť metód použiteľných pre riešenie úloh s ohraničeniami, a tým podnietiť záujem čitateľov o túto problematiku. Je vhodná ako pre tých, ktorí by chceli získať základný prehľad a hľadajú informácie všeobecnejšieho charakteru, tak aj pre tých, ktorí majú hlbší záujem o prezentovanú problematiku a radi sa dozvedia detaily.

Vďaka organizácii knihy nie je nutné jej sekvenčné čítanie. Záujemci o problematiku metód spĺňania ohraničení, logické programovanie ohraničení, alebo jazyk ECL^iPS^e sa môžu zamerať priamo na príslušnú časť. Doporučujeme však predtým prečítať prvú časť, definujúcu problémy s ohraničeniami. Táto časť vytvára základ, na ktorom sú postavené všetky ostatné časti.

Kniha vznikla na Katedre kybernetiky a umelej inteligencie Fakulty elektrotechniky a informatiky Technickej univerzity v Košiciach. Pri jej písaní autori vychádzali zo svojich skúseností, získaných počas riešenia výskumných projektov *ENCODE – ENvironment for COnfiguration DEsign* (COPERNICUS projekt CIPA-CT94-0149 financovaný EK), *Metódy reprezentácie a riešenia úloh definovaných systémom ohraničení* (VEGA projekt 1/1686/94 financovaný MŠ SR) a *Integrácia prostriedkov inteligentných technológií* (VEGA projekt 1/5032/98 financovaný MŠ SR).

Výberom ilustračných úloh kniha nadväzuje na predmet *Rozvrhovanie a logistika* ponúkaný v rámci graduálneho štúdia na Technickej univerzite v Košiciach.

Autorom kapitol 1 až 4 je M. Mach, kapitoly 5 až 8 sú dielom J. Paraliča. Ostatné časti vznikli spoločným úsilím oboch autorov.

Na tomto mieste by sme chceli vyjadriť poďakovanie recenzentom za starostlivé prečítanie rukopisu, opravu mnohých formálnych a aj vecných chýb a za cenné pripomienky a námety, ktoré obohatili obsah predkladaného textu.

Košice, september 2000 autori

Obsah

Úvod

Pri svojej činnosti je človek obmedzovaný svojím okolím. Toto okolie neustále produkuje ohraničenia, ktoré mu určujú jeho oblasť pôsobnosti. Týkajú sa každej oblasti ľudskej činnosti, sú všadeprítomné, a nie je možné sa im vyhnúť. Nie je možné napr. postaviť strechu skôr ako základy, naplánovať schôdzku na dobu keď subjekty budú na rôznych miestach, používať jeden hrniec na varenie dvoch polievok naraz, ap.

Tak je to aj v oblasti technickej praxe. Či sa už jedná o plánovací problém, rozvrhovanie, prideľovanie zdrojov, návrhové úlohy, neustále je potrebné zohľadňovať rôzne požiadavky ako napr. dostupnosť zdrojov, časové závislosti, charakteristiky použitých elementov, ap. Na druhej strane, reprezentácia problémov pomocou explicitného popisu ohraničení je veľmi prirodzenou a názornou (a pritom pomerne jednoducho a rýchlo zostaviteľnou). Ak si uvedomíme, koľko úloh technickej praxe možno formulovať ako úlohy s ohraničeniami, tak je zrejmý dôvod, prečo metódy riešenia úloh s ohraničeniami majú takú dôležitosť.

Problémy s ohraničeniami majú kombinatorickú povahu. Veľa typov týchto problémov patrí do kategórie NP-úplných problémov, čo znamená, že zložitosť algoritmu ktorý by garantoval nájdenie riešenia spĺňajúceho všetky ohraničenia je exponenciálna (a teda časové požiadavky daného algoritmu sú vysoké – pre najhorší možný prípad rastú exponenciálne s veľkosťou problému). To však neznamená, že by praktické úlohy boli neriešiteľné.

Je ich možné riešiť pomocou pomerne značného počtu prístupov, pochádzajúcich z rôznych oblastí. Väčšina z týchto prístupov patrí do oblastí operačného výskumu a umelej inteligencie. Toto ilustruje napr. [55], kde pre riešenie toho istého problému je využitý predikátový počet, výroková logika, celočíselné programovanie, teória automatov, teória grafov, horolezecký algoritmus, neurónová sieť, genetický algoritmus, relačná algebra, syntéza ohraničení, disjunktívna a konjunktívna dekompozícia. Pritom toto vymenovanie nie je kompletné, existujú ďalšie metódy (napr. lokálne heuristiky a celulárne automaty [112]).

Jednou z možností ako riešiť takéto problémy je prístup označovaný v minulosti rôznymi názvami, napr. konzistentné značkovanie či spĺňajúce priradenia. V súčasnosti sa preň vžilo označenie *spĺňanie ohraničení*. Prvé práce spojené s oblasťou algoritmov momentálne zahŕňaných pod týmto spoločným názvom siahajú do polovice sedemdesiatych rokov.

Popudom k rozvoju tejto oblasti sa stali algoritmy používané pre interpretáciu trojrozmerných scén na základe dvojrozmerných obrazov (bolo potrebné zohľadňovať ohraničenia pri interpretácii čiarových obrazov – napr. trojrozmerná interpretácia nejakej čiary ako hrany musí byť rovnaká na oboch jej koncoch). Vtedajší študent MIT AI Lab (Laboratória pre umelú inteligenciu na Massachussetskom technologickom inštitúte) David Waltz si uvedomoval kombinatorickú explóziu, znižujúcu použiteľnosť vtedy známych interpretačných metód, a preto pre jej zmenšenie vytvoril filtračný algoritmus na odstránenie nekonzistentných interpretácií [130].

Jednou z prvých aplikačných domén sa stala interaktívna grafika, kde prvým systémom bol Sutherlandov Sketchpad [124] začiatkom šesťdesiatych rokov. Jeho pokračovateľom je Borningov ThingLab [20]. Tieto systémy dovoľovali používateľovi kresliť a manipulovať s geometrickými obrázkami, podliehajúcimi ohraničeniam, po displeji počítača.

Ďalší ranný vývoj tejto oblasti je spojený s takými menami ako Eugene C. Freuder, John Gaschnig, Robert M. Haralick, Alan K. Mackworth a Ugo Montanari [86], [51], [88]. Došlo k abstrahovaniu vlastností algoritmov, k ich oddeleniu od aplikácií a k formovaniu oblasti do takého tvaru, v akom je známa dnes.

O význame tejto oblasti pre riešenie praktických problémov svedčí aj úsilie zahrnúť algoritmy pre riešenie úloh s ohraničeniami priamo do výpočtových systémov tak, aby používateľ musel definovať iba problém, a nie aj to ako tento problém riešiť. Toto je označované ako programovanie ohraničení [10] a chápe sa pod ním počítačová implementácia algoritmov pre riešenie úloh s ohraničeniami.

Jedným z medzníkov sa stalo logické programovanie implementujúce prostredníctvom jazyka prolog jeden zo základných prehľadávacích algoritmov – prehľadávanie do hĺbky. Keďže však v prístupe "generuj a testuj" ohraničenia hrali iba pasívnu úlohu pri testovaní vygenerovaných riešení, tak ďalší rozvoj viedol cez aktívne využitie ohraničení v procese prehľadávania k logickému programovaniu ohraničení. Táto snaha podnietila vznik novej triedy programovacích jazykov. Vznikli také systémy ako CHIP (Constraint Handling in Prolog) a CLP($\mathcal{R}$). Prakticky všetky takéto komerčné systémy pochádzajú z krajín západnej Európy, najmä z Francúzska, kolísky prologu. Okrem komerčnej verzie CHIP-u, ktorý pôvodne vznikol v Mníchovskom ECRC (European Computer-Industry Research Centre), sú to najmä produkty, ktoré vznikli transferom tejto technológie do troch podielnických firiem ECRC. Ide o systémy DECISIONPOWER (anglický ICL), CHARME (francúzsky BULL, resp. jeho centrum umelej inteligencie CEDIAG) a SNI Prolog (nemecký Siemens-Nixdorf Information Systems). Z ďalších systémov je na trhu komerčná verzia Prolog III.

Existuje samozrejme aj celý rad ďalších CLP systémov, napr. ECL^iPS^e (používaný v tomto predkladanom texte), ktorý je priamym pokračovateľom CHIPu na báze logického programovania a je v súčasnosti ďalej vyvíjaný v Imperial College Parc vo Veľkej Británii.

Programovanie ohraničení našlo svoje uplatnenie aj v oblasti iných programovacích jazykov. Príkladom je ILOG Solver reprezentujúci knižnicu procedúr v jazyku C++, systém SCREAMER tvoriaci nedeterministickú nadstavbu jazyka Common Lisp alebo rozšírenie systému CLIPS o algoritmy spĺňania ohraničení [102]. Algoritmy pre prácu s ohraničeniami je možné pomerne jednoducho implementovať v štandardných imperatívnych programovacích jazykoch, čo činí túto technológiu ľahko prístupnou.

Programovanie ohraničení si získalo veľkú pozornosť medzi odborníkmi z mnohých oblastí najmä kvôli svojmu potenciálu pre riešenie náročných reálnych úloh. A že dnes sa už zďaleka nejedná len o teoretický výskum ale skôr naopak, dokázalo už veľa komerčných systémov demonštrujúcich svoju silu na mnohých reálnych aplikáciách.

Príkladmi niekoľkých takýchto funkčných aplikácií sú nasledujúce systémy založené na programovaní ohraničení:

1. Riadenie plánovania pracovných síl (Banque Bruxelles Lambert) pre generovanie ročných rozvrhov pracovných zmien.

2. PLANE (Dassault Aviation) pre podporu rozhodovania pri plánovaní výroby lietadiel s udržiavaním optimálnej skladby zásob na skladoch [12].

3. Krátkodobé plánovanie výroby (Renault) zabezpečujúce distribúcie zákazok na jednotlivé typy automobilov na príslušné podniky [33].

4. Rozvrhovanie lokomotív (železničná spoločnosť SNCF) pre zabezpečenie prevádzky skupiny železničných tratí s optimálnym využitím zdrojov [29].

5. PLANETS (elektrárenská spoločnosť ENHER) pre rozvrhovanie plánovanej údržby na elektrickej rozvodnej sieti a operatívne prerozvrhovanie v prípadoch súrnych opráv [30].

6. Systém pre plánovanie terminálových operácií (Hong-Kong International Terminals) pomáhajúci prideľovať zdroje pre zabezpečenie optimálneho spojenia [29].

7. Systém pre kontrolu znečistenia benátskej lagúny umožňujúci simulovať a vyhodnocovať možné zásahy za účelom redukcie aktuálneho znečistenia [93].

8. SEVE (Caisse des Dpots et Consignation) pre dlhodobé plánovanie finančných a burzových investícií určujúci optimálne stratégie pre nákup a predaj akcií [29].

9. CVE (Siemens) pre verifikáciu návrhov obvodov VLSI namiesto ich simulácie [29].

Časť I

Problémy s ohraničeniami

Kapitola 1

Čo sú problémy s ohraničeniami?

Pretože existuje viacero typov problémov s ohraničeniami, je možné nájsť viacej navzájom sa líšiacich definícií tohto typu problémov. Cieľom tejto časti je podať takú definíciu, ktorá by pokrývala všetky typy úloh s ohraničeniami, ktorými sa zaoberá táto publikácia v nasledujúcich kapitolách.

Problém s ohraničeniami je vo všeobecnosti definovaný pomocou troch množín:

- množina *premenných* $V = \{V_1, V_2, \ldots, V_n\}$
- množina *domén* $D = \{D_1, D_2, \ldots, D_n\}$
- množina *ohraničení* $C = \{C^1, \ldots, C^n, C^{1,2}, \ldots, C^{n-1,n}, \ldots, C^{1,2,\ldots,n}\}$.

D v uvedenom zápise reprezentuje vlastne systém množín. Každá doména $D_i \in D$ zodpovedá premennej $V_i \in V$ a reprezentuje jej definičný obor – definuje možné hodnoty, ktoré premenná V_i môže nadobúdať. Tieto domény môžu byť rôznych typov. Najčastejšie sa používa delenie domén na

- *konečné domény* (reprezentantmi môžu byť napríklad booleovské premenné, celočíselné ohraničené premenné alebo premenné s enumeračnými doménami)
- *nekonečné domény* (napríklad celočíselné neohraničené, racionálne a reálne premenné).

Veľa prístupov, používaných pre riešenie problémov s ohraničeniami, je schopných pracovať iba s jedným typom domén. Príkladom môže byť prístup využívajúci usporiadanie prvkov v doméne, ktorý nie je použiteľný pre tie domény, nad prvkami ktorých nie je definované usporiadanie.

V literatúre je väčšina prác venovaná problémom s konečnými množinami. Naproti tomu pri riešení praktických problémov sa často vyskytujú aj premenné, ktorých domény sú nekonečnými množinami.

Nad premennými z množiny V je definovaná množina ohraničení. Počty indexov jednotlivých ohraničení určujú árnosť týchto ohraničení a samotné indexy ohraničení označujú premenné, nad ktorými sú tieto ohraničenia definované. Tak napr. ohraničenie $C^{i,j}$ je binárnym ohraničením nad premennými V_i a V_j.

Definované ohraničenia stanovujú prípustnosť hodnôt definovaných premenných v rámci nejakého kontextu. Týmto kontextom sú hodnoty iných premenných (počet týchto premenných závisí od árnosti ohraničení). Napríklad, ak nejaké ohraničenie $C^{i,\dots,j}$ je definované nad podmnožinou premenných $V^{i,\dots,j} = \{V_i, \dots, V_j\} \subset V$, potom toto ohraničenie stanovuje prípustnosť priradenia $V_k = h$ (pre premennú $V_k \in V^{i,\dots,j}$ a jej hodnotu $h \in D_k$) v kontexte tvorenom hodnotami ostatných premenných z $V^{i,\dots,j}$. Jedinou výnimkou sú unárne ohraničenia, pri ktorých nie je potrebné brať do úvahy hodnoty iných premenných.

Z hľadiska ohraničení nezáleží na poradí tých premenných, nad ktorými sú definované, a teda platí $C^{i,j} = C^{j,i}$. Inak povedané, to isté ohraničenie je možné označiť toľkými spôsobmi, koľko existuje rôznych permutácií poradia jeho indexov.

Pre skupinu n premenných je teda možné definovať unárne, binárne, ..., n-árne ohraničenia. Celkový počet ohraničení, ktoré je možné definovať pre n premenných je

$$m_{MAX} = \sum_{i=1}^{n} \binom{n}{i} \tag{1.1}$$

pričom i označuje árnosť ohraničení. Je teda zrejmé, že je možné definovať až n unárnych ohraničení a jedno n-árne ohraničenie. Maximálny počet binárnych ohraničení je $(n^2 - n)/2$. Okrem n-árneho ohraničenia všetky ostatné sú definované iba nad nejakou podmnožinou premenných – tieto premenné sa označujú ako relevantné voči danému ohraničeniu. Všetky také ohraničenia, ktoré sú definované nad nejakou premennou V_i, vytvárajú množinu ohraničení $C_i^{all} = \{C^i, C^{i,1}, \dots, C^{1,\dots,n}\}$. Celkový počet ohraničení, ktoré je možné definovať nad nejakou premennou, je

$$\sum_{i=1}^{n} \binom{n-1}{i-1}. \tag{1.2}$$

Vo všeobecnosti, všetky ohraničenia sú definované nad podmnožinami premenných a obmedzujú kombinácie hodnôt, ktoré premenné daných podmnožín môžu nadobudnúť. Extrémnymi prípadmi sú ohraničenia, ktoré nepripúšťajú žiadne alebo naopak pripúšťajú všetky možné kombinácie hodnôt premenných, nad ktorými sú tieto ohraničenia definované. V prvom prípade nie je možné nájsť také riešenie, pre ktoré by takéto extrémne ohraničenie bolo splnené. V druhom prípade je také ohraničenie v množine ohraničení C zaradené iba formálne a možno ho z tejto množiny vypustiť.

Ohraničenia môžu mať rozličný tvar[1]. V zásade je možné použiť vymenovanie jednotlivých kombinácií (či už povolených alebo zakázaných) alebo ohraničenie možno zadať nejakým predpisom, ktorý pre každú kombináciu hodnôt premenných umožňuje rozhodnúť, či daná kombinácia porušuje dané ohraničenie alebo nie. To, ktorý tvar je využitý, závisí od typu domén premenných a od konkrétnej úlohy. Vo všeobecnosti možno povedať, že napríklad pre enumeračné premenné sa zvyčajne využíva vyjadrenie vymenovaním kombinácií hodnôt a pre reálne premenné sa zase využíva častejšie forma predpisu (napr. v tvare nejakej rovnosti alebo nerovnosti).

Niekoľko príkladov binárnych ohraničení definovaných nad dvoma premennými V_1 a V_2 je uvedených na obr. 1.1, kde plné krúžky znázorňujú prípustné kombinácie hodnôt a prázdne krúžky zase neprípustné kombinácie. Zobrazené ohraničenie pre enumeračné premenné je dané vymenovaním

$$\{(sym0, sym2), (sym0, sym3), \ldots\}$$

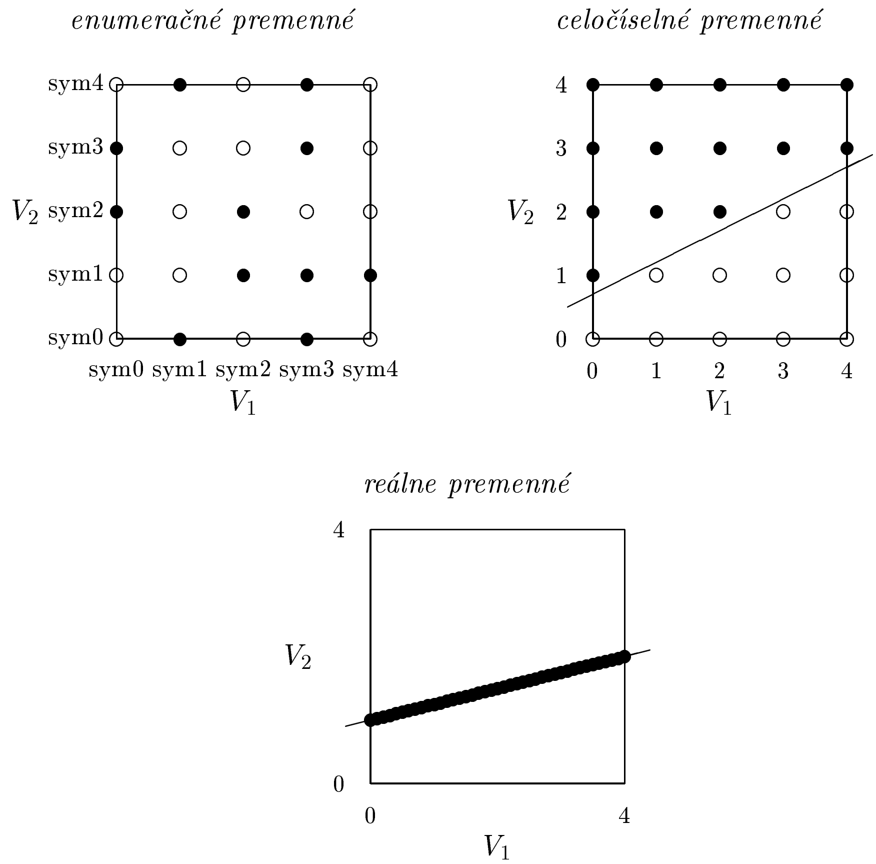
ohraničenie pre celočíselné premenné zase ako nerovnosť

$$V_2 > 0.5V_1 + 0.75$$

a ohraničenie pre reálne premenné ako rovnosť

$$V_2 = 0.25V_1 + 1.$$

[1] Obidva uvedené prípady (vymenovanie a predpis) ilustrujú explicitný tvar ohraničenia. Ohraničenie však môže byť vyjadrené aj implicitným spôsobom – samotné ohraničenie nie je priamo uvedené, ale je reprezentované iba prostredníctvom iných ohraničení, z ktorých môže byť indukované.



Obr. 1.1: Príklady binárnych ohraničení.

Všetky tri príklady reprezentujú "pozitívnu" formu – reprezentujú prípustné kombinácie. Vo všeobecnosti ohraničenia môžu byť v pozitívnom alebo negatívnom tvare, reprezentujúc prípustné alebo neprípustné kombinácie hodnôt. Tie isté ohraničenia v "negatívnom" tvare by boli reprezentované ako $\{(sym0, sym0), \ldots\}$ pre enumeračné domény, $V_2 \leq 0.5V_1 + 0.75$ pre celočíselné premenné, resp. $V_2 \neq 0.25V_1 + 1$ pre reálne premenné. To, ktorá forma sa použije, zvyčajne závisí od jednoduchosti vyjadrenia.

V praktických úlohách sa vyskytujú ohraničenia rôznych árností. Pretože však unárne ohraničenia je možné zohľadniť príslušnou redukciou domén jednotlivých premenných (resp. naopak je ich možné rozšíriť na binárne ohraničenia) a ohraničenia vyšších árností môžu byť redukované na binárne ohraničenia (aj keď za cenu zavedenia nových pomocných premenných) [8] [131], väčšina prác sa zaoberá práve binárnymi problémami s ohraničeniami – problémami, v ktorých všetky ohraničenia sú definované najviac nad dvoma premennými.

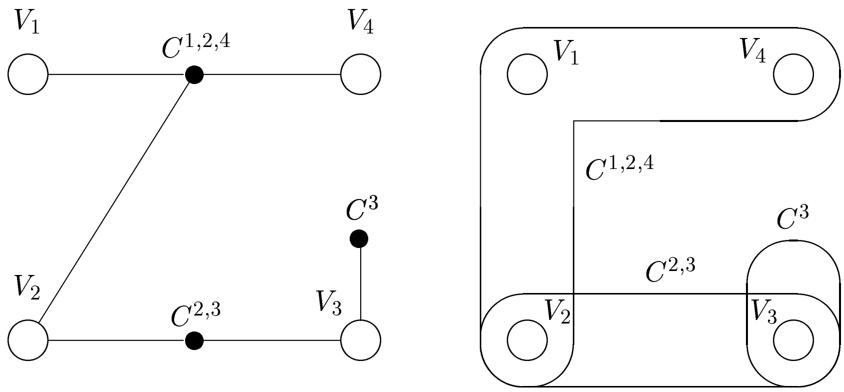
Ak je potrebné splniť všetky ohraničenia (teda aj unárne), je možné znížiť komplexnosť problému redukciou domén jednotlivých premenných. Po tejto redukcii premennej V_i bude zodpovedať redukovaná doména $D_i^r \subset D_i$, ktorá vznikne z pôvodnej domény vypustením všetkých prvkov nespĺňajúcich unárne ohraničenie viažúce sa na premennú V_i.

Táto redukcia sa môže rozšíriť aj na definície ohraničení – ak viacárne ohraničenie povoľuje nejakú takú kombináciu hodnôt, ktorá obsahuje hodnotu porušujúcu unárne ohraničenie, viažúce sa k príslušnej premennej (a teda táto hodnota sa nevyskytuje v redukovanej doméne danej premennej), potom je možné toto ohraničenie "sprísniť" a danú kombináciu hodnôt nepovoliť. Takýmto spôsobom dochádza aj k redukcii ohraničení – zvyšovaniu ich prísnosti.

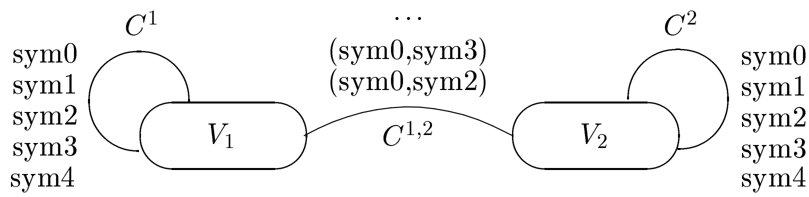
Premenné a ohraničenia je možné graficky znázorniť. Jednou z možností je znázornenie pomocou neorientovaného grafu – *siete ohraničení*. Vo všeobecnosti v tejto sieti existujú dva typy uzlov – pre premenné a pre ohraničenia. Hrany sú možné iba medzi uzlami rôzneho typu a vyjadrujú väzbu medzi ohraničeniami a premennými. Inou možnosťou je namiesto grafu použiť hypergraf. V takomto prípade bude existovať iba jeden typ uzlov reprezentujúcich premenné. Ohraničenia budú znázornené ako hyperhrany, spájajúce viac premenných súčasne. Na obr. 1.2 sú ilustrované oba spôsoby znázornenia siete ohraničení pre problém so štyrmi premennými $V_1, \ldots, V_4$, nad ktorými sú definované tri ohraničenia $C^{1,2,4}$, $C^{2,3}$ a C^3.

Pre binárne problémy je možné použiť jednoduchšie znázornenie. V tomto prípade premenné sú reprezentované uzlami a hrany zase reprezentujú ohraničenia. Unárne ohraničenie možno reprezentovať hranou, oboma koncami viazanou k tomu istému uzlu. Binárne ohraničenia sú znázornené ako hrany spájajúce rôzne uzly. Príklad takéhoto zjednodušeného spôsobu znázornenia binárneho problému s ohraničeniami je sieť ohraničení pre enumeračné premenné V_1 a V_2 z obr. 1.1, ktorá je zobrazená na obr. 1.3.

Domény premenných vlastne definujú *priestor prehľadávania P*. Je to vlastne priestor všetkých potenciálnych riešení – všetkých možných kombinácií hodnôt z domén premenných. Ak každá z domén bude reprezentovaná ako jedna súradnica tohto priestoru, potom každý bod tohto priestoru bude reprezentovať jednu kombináciu hodnôt premenných. Priestor prehľadávania takto vznikne ako kartézsky súčin domén premenných – teda ako množina



Obr. 1.2: Príklady zobrazenia premenných a ohraničení v sieti ohraničení.



Obr. 1.3: Príklad zjednodušeného zobrazenia siete ohraničení pre binárny problém.

všetkých kombinácií hodnôt jednotlivých premenných. V závislosti od použitých domén tento priestor môže obsahovať konečný alebo nekonečný počet bodov.

Každé ohraničenie rozdeľuje priestor prehľadávania na dve oblasti (nemusia byť súvislé). Pre každý bod tohto priestoru je potom možné povedať, či kombinácia hodnôt, reprezentovaná daným bodom priestoru, spĺňa dané ohraničenie (a patrí do jednej oblasti) alebo ho porušuje (a teda patrí do oblasti druhej).

Nad priestorom prehľadávania P je definovaná hodnotiaca funkcia[2] f_h,

[2] Je možné definovať aj iné funkcie nad bodmi priestoru prehľadávania P (alebo nejakej jeho časti). Napr. pri úlohách optimalizačného spĺňania ohraničení je definovaná okrem hodnotiacej funkcie ešte aj optimalizačná funkcia a riešením je taká kombinácia hodnôt z

ktorá pre každý bod priestoru prehľadávania P vyjadruje počet ohraničení, splnených tou kombináciou hodnôt, ktorá je reprezentovaná daným bodom priestoru. Aby potenciálne riešenie bolo považované za prijateľné, musí spĺňať všetky definované ohraničenia[3].

Riešiť problém s ohraničeniami potom zvyčajne znamená jednu z nasledujúcich úloh:

- dokázať existenciu alebo neexistenciu riešenia
- zistiť počet riešení
- nájsť jedno riešenie
- nájsť všetky riešenia.

Najčastejšie je v literatúre venovaná pozornosť úlohe nájdenia jedného riešenia. Je to preto, lebo jednak je to v praxi prevažujúca úloha, a jednak ostatné úlohy môžu byť riešené (viac či menej efektívne) využitím metód pre hľadanie jedného riešenia (ak ho je možné nájsť, tak riešenie existuje; opakovaným použitím je možné nájsť všetky riešenia ktoré stačí spočítať).

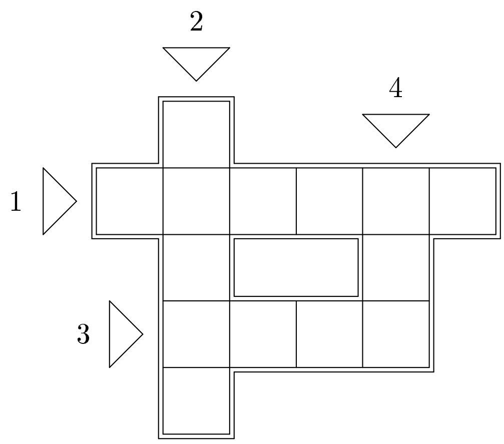
domén jednotlivých premenných, ktorá okrem splnenia všetkých ohraničení naviac súčasne dosahuje maximálnu hodnotu (alebo minimálnu – v závislosti od typu optimalizácie) tejto optimalizačnej funkcie.

[3] Existujú aj iné formulácie požadovaného tvaru riešenia. Napr. pri preurčených úlohách nie je možné nájsť takú kombináciu hodnôt z domén jednotlivých premenných, ktorá by splnila všetky definované ohraničenia. V tomto prípade za riešenie je považovaná taká kombinácia hodnôt, ktorá napr. spĺňa čo najväčší počet ohraničení resp. spĺňa nejakú skupinu najdôležitejších ohraničení.

Kapitola 2

Dva príklady

Ilustráciou uvedeného typu problémov s ohraničeniami je doplňovačka[1], ktorej tvar je znázornený na obr. 2.1.



Obr. 2.1: Ilustračný problém s konečnými doménami.

Do tejto doplňovačky je potrebné doplniť štyri slová tak, aby všetky riadky a stĺpce boli úplne zaplnené. K dispozícii je nasledovný výber slov:

[1]Táto doplňovačka bude využívaná ako ilustračný príklad v značnej časti textu (buď v hore uvedenej podobe alebo s modifikovaným výberom slov).

6 písmen	:	žirafa, európa
5 písmen	:	puška, kurín
4 písmená	:	auto, íver
3 písmená	:	pór, eso

Táto úloha môže byť transformovaná na problém s ohraničeniami pomerne priamočiaro. Z formálneho hľadiska, jednotlivé riadky a stĺpce môžu byť reprezentované premennými (pre daný problém je teda potrebné použiť štyri premenné – ich indexy zodpovedajú označeniu riadkov a stĺpcov podľa obr. 2.1). Domény týchto premenných sú rovnaké – uvedených osem slov. Jedná sa teda o problém s konečnými doménami.

Keďže riadok č. 1 má dĺžku 6 znakov, je možné zmysluplne do neho umiestniť iba niektoré zo šesťpísmenových slov. Toto je možné vyjadriť ako unárne ohraničenie, definované nad premennou V_1, zodpovedajúcou prvému riadku. Toto ohraničenie bude povoľovať pre premennú V_1 iba dve rôzne hodnoty a bude mať teda tvar $C^1 = \{($"žirafa"$)$, $($"európa"$)\}$. Podobným spôsobom je možné definovať unárne ohraničenia pre všetky ostatné premenné.

Jednotlivé slová, umiestnené do doplňovačky, zdieľajú niektoré písmená. Tieto závislosti sa dajú vyjadriť ako viacárne ohraničenia. Napríklad fakt, že druhé písmeno slova v prvom riadku musí byť rovnaké ako druhé písmeno slova umiestneného v stĺpci č. 2, je možné reprezentovať binárnym ohraničením $C^{1,2}$ nad premennými V_1 a V_2. Toto ohraničenie bude povoľovať značný počet rôznych kombinácií a bude mať tvar {("žirafa","žirafa"), …, ("eso","eso"), ("európa","puška"), …}.

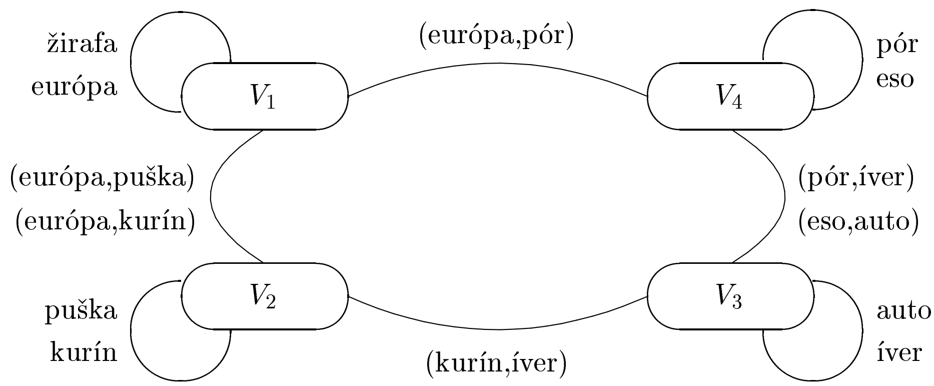
Celkovo pre daný problém je teda potrebné definovať štyri unárne a štyri binárne ohraničenia[2]. Priestor, ktorý je potrebné prehľadať, je konečný. Keďže každá doména obsahuje osem slov, tak tento priestor prehľadávania pozostáva z $8^4 = 4096$ štvoríc slov.

Použitím unárnych ohraničení je možné všetky domény redukovať na dvojprvkové množiny, čím sa získa priestor prehľadávania pozostávajúci zo šestnástich bodov. S ohľadom na to je možné sprísniť aj binárne ohraničenia, napr. keďže teraz v doméne D_1 sa nebude vyskytovať slovo "kurín", tak $C^{1,2}$ nebude povoľovať napr. dvojicu ("kurín","auto"). Podobne aj keď ohraničenie $C^{1,2}$ v pôvodnej neredukovanej podobe pripúšťalo aj kombináciu

[2]Ostatné ohraničenia sú reprezentované implicitne – môžu byť indukované z explicitne vyjadrených ohraničení. Takým je napr. ohraničenie $C^{1,3}$ – z explicitne vyjadrených binárnych ohraničení vyplýva, že toto ohraničenie nepovoľuje napr. kombináciu hodnôt ("žirafa", "íver") pre premenné V_1 a V_3.

("žirafa","žirafa"), táto v prípade redukovaných domén nie je prípustná.

Uvedené ohraničenia po redukcii vytvárajú sieť ohraničení podľa obr. 2.2 (jednotlivé ohraničenia sú zadané explicitným vymenovaním povolených hodnôt resp. kombinácií hodnôt).



Obr. 2.2: Sieť ohraničení pre redukovaný ilustračný problém s konečnými doménami.

Ilustráciou problému s ohraničeniami, definovanými nad premennými s nekonečnými doménami, môže byť doplňovačka znázornená na obr. 2.3.

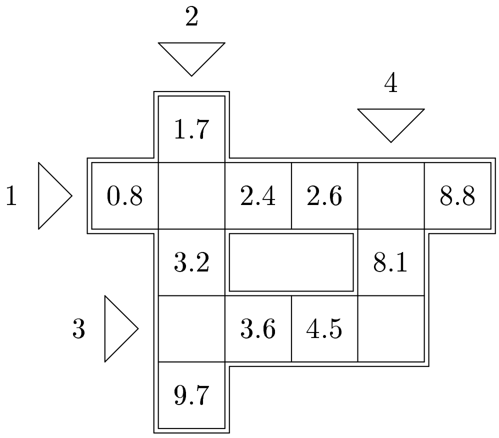
Do tejto doplňovačky je potrebné doplniť štyri kladné čísla tak, aby súčet čísiel v každom riadku a v každom stĺpci bol rovný číslu 20. Pritom musí byť zachované pravidlo, že jednotlivé čísla v políčkach sa zvyšujú v smere zľava doprava a taktiež v smere zhora nadol.

Opäť transformácia úlohy na problém s ohraničeniami (jeho formálne vyjadrenie) môže byť priamočiara. Jednotlivé voľné políčka môžu reprezentovať štyri premenné V_{12}, V_{23}, V_{34} a V_{14} (index premennej zodpovedá kompozícii indexov toho stĺpca a riadku, súčasťou ktorých je prázdne políčko reprezentované danou premennou). Domény týchto premenných budú reálne čísla.

Fakt, že doplniť je možné iba kladné čísla, môže byť reprezentovaný unárnym ohraničením rovnakým pre každú premennú. Naviac z podmienky "rastu" vyplývajú ďalšie unárne ohraničenia. Napr. unárne ohraničenie C^{12} prislúchajúce premennej V_{12} bude mať tvar:

$$C^{12} = \{0 < V_{12}, 0.8 \leq V_{12} \leq 2.4, 1.7 \leq V_{12} \leq 3.2\}$$

Napriek tejto redukcii jednotlivých domén priestor prehľadávania naďalej



Obr. 2.3: Ilustračný problém s nekonečnými doménami.

ostáva nekonečne veľkým.

Existujúce závislosti medzi premennými je možné reprezentovať binárnymi ohraničeniami[3]. Keďže pre každý stĺpec a každý riadok musí platiť podmienka "súčtu" čísel v riadku resp. stĺpci a súčasne aj podmienka "rastu" v zadanom smere, tak potom napr. ohraničenie $C^{12,14}$ bude mať tvar

$$\{V_{12} < V_{14}, 0.8 + V_{12} + 2.4 + 2.6 + V_{14} + 8.8 = 20\}.$$

Analogický tvar budú mať aj ďalšie ohraničenia $C^{12,23}$, $C^{14,34}$ a $C^{23,34}$. Celkovo je potrebné opäť definovať štyri unárne a štyri binárne ohraničenia. Výsledná sieť ohraničení má rovnakú štruktúru ako sieť na obr. 2.2, líši sa iba v označení premenných a v obsahu ohraničení. V tomto prípade jednotlivé ohraničenia nie sú dané explicitným vymenovaním povolených kombinácií hodnôt ale implicitne – predpisom, ktorý pre každú kombináciu hodnôt dovoľuje určiť, či táto kombinácia spĺňa dané ohraničenie alebo nie.

[3]Ak by úloha bola definovaná tak, že súčty čísel v jednotlivých riadkoch a stĺpcoch síce majú byť rovnaké, ale je jedno aká to bude hodnota, tak potom by bolo nutné definovať aj štyri ternárne ohraničenia (napr. $C^{14,23,34} = \{V_{23} + 3.6 + 4.5 + V_{34} = V_{14} + 8.1 + V_{34}\}$) a jedno 4-árne ohraničenie (obsahujúce napr. $1.7 + V_{12} + 3.2 + V_{23} + 9.7 = V_{14} + 8.1 + V_{34}$).

Časť II

Metódy riešenia problémov s ohraničeniami

Kapitola 3

Riešiace kroky a riešiace stratégie

Riešiť problém s ohraničeniami znamená hľadať vhodné priradenie hodnôt všetkým definovaným premenným. Toto hľadanie je procesom, ktorého stav v nejakom okamihu môže byť charakterizovaný počtom *neviazaných* premenných (to sú premenné, ktoré ešte nemajú priradenú hodnotu zo svojej domény – niekedy sú označované ako voľné) a počtom porušení ohraničení, definovaných nad premennými. Pri určovaní aktuálneho stavu riešenia môže dôjsť k situácii, keď nejaké ohraničenie je definované nad neviazanou premennou, a teda nie je možné overiť splnenie alebo porušenie tohto ohraničenia. Takéto ohraničenie v tomto prípade nie je považované za porušené.

Tento proces hľadania riešenia môže byť reprezentovaný v dvojrozmernom *stavovom priestore* S [132], kde

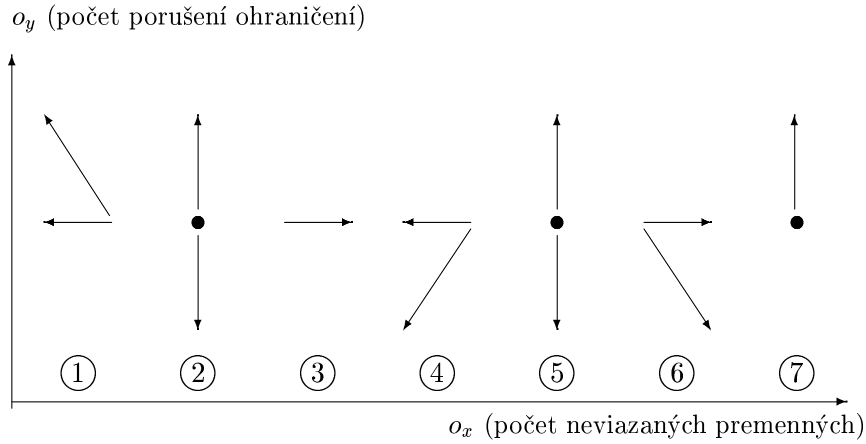
os o_x reprezentuje počet doposiaľ neviazaných premenných a

os o_y reprezentuje počet porušení ohraničení.

Z podstaty problémov s ohraničeniami vyplýva, že využívané sú iba nezáporné časti oboch súradných osí.

Proces hľadania riešenia môže byť znázornený v tomto stavovom priestore ako trajektória, spájajúca počiatočný bod hľadania riešenia s výsledným stavom. Na začiatku procesu hľadania riešenia žiadna premenná nemá priradenú hodnotu. Dôsledkom toho je, že žiadne ohraničenie nie je považované za porušené. Teda počiatočný bod trajektórie (reprezentujúci počiatočné riešenie ktoré je "prázdne") sa nachádza na osi o_x a jeho vzdialenosť od počiatku je rovná počtu premenných, definovaných na začiatku riešenia úlohy. Pretože konečné riešenie by malo byť kompletné (všetky premenné sú viazané – majú priradenú nejakú hodnotu zo svojej domény), trajektória končí na osi o_y. Ak konečné riešenie spĺňa všetky ohraničenia, potom koncový bod trajektórie je umiestnený v počiatku systému súradníc.

Trajektória v stavovom priestore, reprezentujúca hľadanie riešenia, pozostáva z rôznych krokov. Tieto kroky je možné rozdeliť do niekoľkých základných typov (ich možné tvary sú zobrazené na obr. 3.1):



Obr. 3.1: Základné typy riešiacich krokov.

1. priradenie

   Jednej neviazanej premennej sa priradí hodnota z jej domény. Toto naviazanie môže spôsobiť aj porušenie jedného alebo viacerých ohraničení. Tento krok je v stavovom priestore S reprezentovaný šípkou smerujúcou vľavo (je paralelná s osou o_x, ak žiadne ohraničenie nebolo porušené týmto krokom).

2. korekcia

   Hodnoty jednej alebo viacerých premenných sú modifikované s cieľom obnovenia platnosti nejakého ohraničenia. V S to možno reprezentovať šípkou paralelnou s osou o_y (ak počet porušení ohraničení klesol, tak šípka je orientovaná nadol) alebo bodom (ak počet porušených ohraničení sa nezmenil).

3. aktivácia

   Jedna alebo viac premenných je aktivovaných a zahrnutých do aktuálneho riešenia. Môže to byť sprevádzané rozšírením množiny ohraničení C o nové ohraničenia, viažúce sa na novo uvažované premenné. Pretože novo aktivované premenné sú neviazané, tento krok je v stavovom priestore reprezentovaný šípkou vpravo rovnobežnou s osou o_x.

4. deaktivácia

   Jedna alebo viac premenných je vyradených z aktuálneho riešenia. Ohraničenia, ktoré boli definované nad týmito deaktivovanými premennými, sa stávajú neaktívnymi. V stavovom priestore to je reprezentované ako šípka vľavo (paralelná s osou o_x, ak žiadne porušené ohraničenie nebolo deaktivované).

5. relaxácia[1]

   Jedno alebo viacero ohraničení sa stáva neaktívnymi (môže to byť sprevádzané aktiváciou nejakých iných neaktívnych ohraničení). Tento krok je možné reprezentovať šípkou paralelnou s osou o_y (ak počet porušení ohraničení klesol, tak šípka je orientovaná nadol) alebo bodom (ak počet porušených ohraničení sa nezmenil).

6. uvoľnenie

   Jedna alebo niekoľko premenných (s priradenou hodnotou) sa stáva neviazanými. Toto môže mať za následok pokles celkového počtu porušení ohraničení, ak nejaké ohraničenie porušené aktuálnym riešením je definované nad premennou, ktorá je ovplyvnená týmto krokom. V stavovom priestore tomuto kroku zodpovedá šípka vpravo (môže byť rovnobežná s osou o_x, ak počet porušení neklesol).

7. redukcia

   Jedna alebo niekoľko hodnôt je vypustených z domén premenných alebo z ohraničení. Ak sa jedná o neviazané premenné resp. o ohraničenia definované nad dosiaľ neviazanými premennými, potom tento krok nemá žiadny vplyv na počet porušení ohraničení ani na počet neviazaných premenných, a vtedy je v stavovom priestore znázorňovaný ako bod. Inak môže znamenať zvýšenie počtu porušených ohraničení.

[1]Krok je typicky používaný v preurčených úlohách, keď nie je možné splniť súčasne všetky ohraničenia. Vtedy je potrebné sa niektorého ohraničenia vzdať alebo ho nahradiť inými, menej prísnymi ohraničeniami.

Tieto riešiace kroky slúžia na transformáciu aktuálneho riešenia na riešenie nové. Sekvencia týchto krokov potom umožňuje počiatočné riešenie transformovať na riešenie konečné.

Z obrázku je zrejmé, že niektoré kroky (alebo niektoré ich varianty) majú v stavovom priestore rovnaké znázornenie. Teda z tvaru trajektórie nie je možné vždy presne určiť, z ktorých riešiacich krokov je zložená. Navyše, medzi jednotlivými riešiacimi krokmi existujú aj závislosti vo forme vzájomnej zameniteľnosti. Napr. prípad keď nejaká premenná je uvoľnená (a teda sa stáva neviazanou) s následným priradením inej hodnoty z jej domény môže byť chápaný ako korekcia jej hodnoty.

Základným riešiacim krokom (prítomným vždy) je priradenie. Zvyčajne však nestačí pre nájdenie vyhovujúceho riešenia a preto je tento krok doprevádzaný korekčným krokom alebo uvoľnením premenných. V niektorých typoch problémov s ohraničeniami však ani tieto tri kroky nepostačujú pre nájdenie riešenia a musia byť využité aj ostatné kroky. Napr. pri rôznych transformáciách problému môže dôjsť k zmenám premenných a teda k uplatneniu aktivačného a/alebo deaktivačného kroku. Vo všeobecnosti, čím viac typov riešiacich krokov nejaká riešiaca metóda vie využiť, tým širší je rozsah aplikovateľnosti tejto metódy.

Existuje viacero stratégií pre riešenie problémov s ohraničeniami. Každá stratégia vytvára v stavovom priestore trajektóriu charakteristického tvaru. V podstate existujú dva základné tvary trajektórie, ktoré zodpovedajú dvom stratégiám prehľadávania priestoru P:

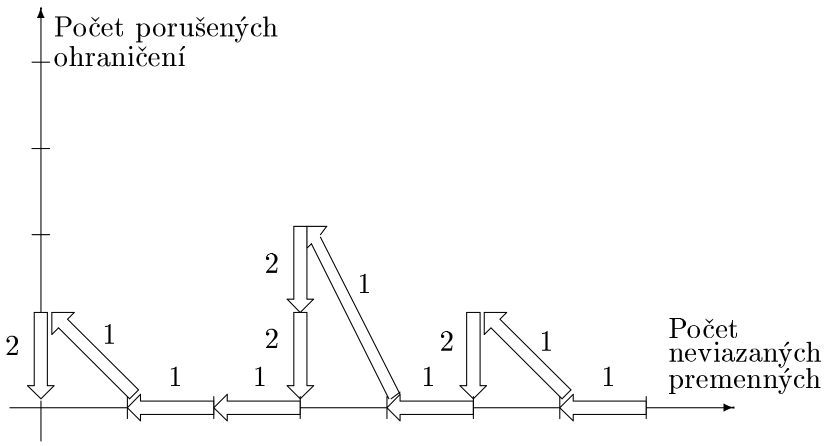
- rozširuj a reviduj

  Aktívnym premenným sú priraďované hodnoty z ich domén až pokiaľ sa nevyskytne prvé porušenie nejakého ohraničenia. Najprv sa koriguje riešenie tak, aby sa toto porušenie ohraničenia odstránilo, a až po jeho odstránení sa pokračuje v ďalšom priraďovaní hodnôt premenným.

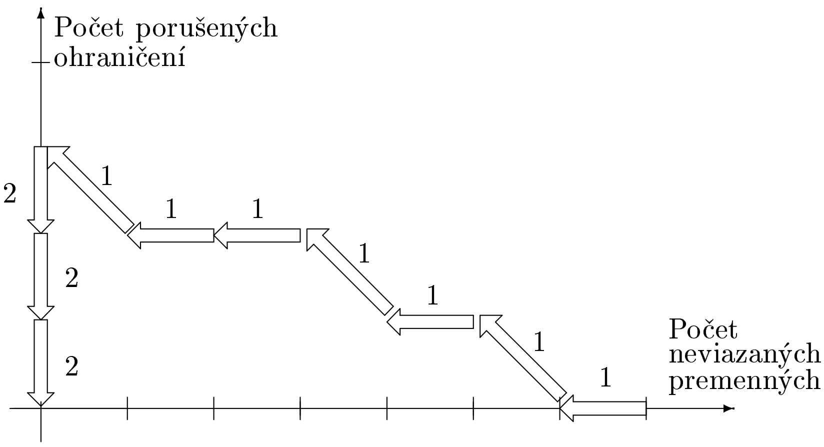
- vytvor a reviduj

  Najprv sa priradia hodnoty všetkým premenným bez ohľadu na ohraničenia. Až po naviazaní všetkých premenných nasleduje modifikácia riešenia vedúca k odstráneniu porušení ohraničení.

Príklady možných trajektórií vytvorených týmito dvoma základnými stratégiami sú znázornené na obr. 3.2 a obr. 3.3. Číselné označenie jednotlivých riešiacich krokov zodpovedá vyššie uvedenému deleniu týchto krokov do uvedených tried.



Obr. 3.2: Stratégia "rozširuj a reviduj".



Obr. 3.3: Stratégia "vytvor a reviduj".

Z pohľadu spôsobu prehľadávania priestoru prehľadávania P (priestoru potenciálnych riešení) uvedené dve stratégie zodpovedajú dvom rôznym spôsobom prehľadávania:

- hierarchickému prehľadávaniu
- bodovému prehľadávaniu.

Postupným sekvenčným priraďovaním hodnôt jednotlivým premenným s ich okamžitým testovaním je vlastne priestor prehľadávania P (definovaný doménami jednotlivých premenných) prehľadávaný hierarchickým spôsobom – priradením hodnoty h nejakej premennej V_i z jej domény D_i sa zúži oblasť momentálneho záujmu z celého priestoru prehľadávania na menší podpriestor, charakterizovaný hodnotou h pre i-tu dimenziu. Ak v tomto podpriestore nebola objavená žiadna nekonzistencia, tak sa tento podpriestor ďalej zužuje na ešte menší podpriestor (priradením hodnoty ďalšej premennej). Tento proces cyklicky pokračuje až kým nie je zistená nekonzistencia skúmaného podpriestoru alebo až kým celý priestor prehľadávania nebol redukovaný na jeden bod, reprezentujúci hľadané riešenie. Ak je v niektorom podpriestore zistená nekonzistencia, tak tento podpriestor sa celý vylúči z ďalšieho prehľadávania (jeho body sa už detailnejšie skúmať nebudú) a prechádza sa na iný podpriestor.

Pri bodovom prehľadávaní sa konzistencia (platnosť ohraničení) skúma pre jednotlivé body priestoru prehľadávania. Ak skúmaný bod je konzistentný, tak reprezentuje hľadané riešenie. Ak nie je konzistentný, prechádza sa na skúmanie iného bodu, ktorý môže byť z okolia predchádzajúceho bodu alebo aj z celkom inej (vzdialenej) časti priestoru prehľadávania. Príkladom bodového prehľadávania je algoritmus *generuj a testuj*. Tento algoritmus pri oddelení fáz generovania a testovania (testovanie riešenia nasleduje až po jeho úplnom vygenerovaní) umožňuje systematickým spôsobom generovať a testovať jednotlivé body priestoru prehľadávania.

Pre hierarchické prehľadávanie je typickým krokom uvoľňovací krok (návrat pri nájdení nekonzistencie v nejakom podpriestore), zatiaľ čo bodové prehľadávanie je charakterizované korekčným krokom (umožňujúcim zmenu hodnoty jednej alebo viacerých premenných).

Kapitola 4

Úlohy s konečnými doménami

4.1 Redukčné algoritmy

Spoločným znakom redukčných algoritmov je, že sa snažia redukovať domény jednotlivých premenných a/alebo jednotlivé ohraničenia tak, aby nájdenie riešenia bolo čo najjednoduchšie. Pritom dochádza k redukcii priestoru prehľadávania. Existujú dve základné skupiny redukčných algoritmov, odlišujúce sa spôsobom redukcie:

- konzistenčné algoritmy
- substitučné algoritmy.

V oboch prípadoch sa pôvodný problém transformuje na nový (z hľadiska hľadania riešenia jednoduchší ako bol pôvodný). Rozdiel medzi oboma typmi algoritmov je ten, že pri použití konzistenčných algoritmov nový problém má tú istú množinu riešení ako pôvodný, kým pri použití substitučných algoritmov to zaručené nie je – množina riešení nového problému môže byť redukovaná iba na podmnožinu množiny riešení problému pôvodného (ak však pôvodný problém mal aspoň jedno riešenie, tak aj nový problém bude mať aspoň jedno riešenie). V prípade, že pôvodný problém nemal riešenie, pri oboch typoch redukčných algoritmov ho nebude mať ani nový problém.

Pri hľadaní jedného riešenia táto uvedená vlastnosť substitučných algoritmov nie je prekážkou, ale pri hľadaní všetkých riešení je potom potrebné použiť nejaký rekonštrukčný algoritmus, ktorý všetky riešenia pôvodného problému rekonštruuje spätne z množiny riešení nového problému a množiny realizovaných substitúcií [27].

Výhodou konzistenčných algoritmov je, že ak robia "fair" šírenie ohraničení (t.j. každé ohraničenie, ktorého premenné sa zmenili, je znovu aktivované), tak takéto šírenie vedie k jedinečnému stavu siete ohraničení – a teda nezáleží na poradí, v ktorom sú jednotlivé ohraničenia uvažované [65]. Naproti tomu pri substitučných algoritmoch v závislosti od použitého poradia premenných a hodnôt v doménach premenných je možné vygenerovať rôzne sekvencie substitučných operácií. Tieto rôzne sekvencie aplikovania substitúcií môžu viesť k rôznym výsledným redukciám.

Konzistenčné a substitučné algoritmy môžu poskytovať rôzny stupeň redukcie priestoru prehľadávania – a teda sú možné situácie, keď jeden z nich už nevie viac redukovať žiadnu z domén premenných ani žiadne ohraničenie, zatiaľ čo použitie druhého prístupu môže viesť k ďalšej redukcii. Je preto možné oba prístupy kombinovať pre dosiahnutie čo najväčšej redukcie. Podľa [27] je najvhodnejšou stratégiou:

1. zabezpečiť prísnu k-konzistenciu (viď kapitolu 4.1.1)

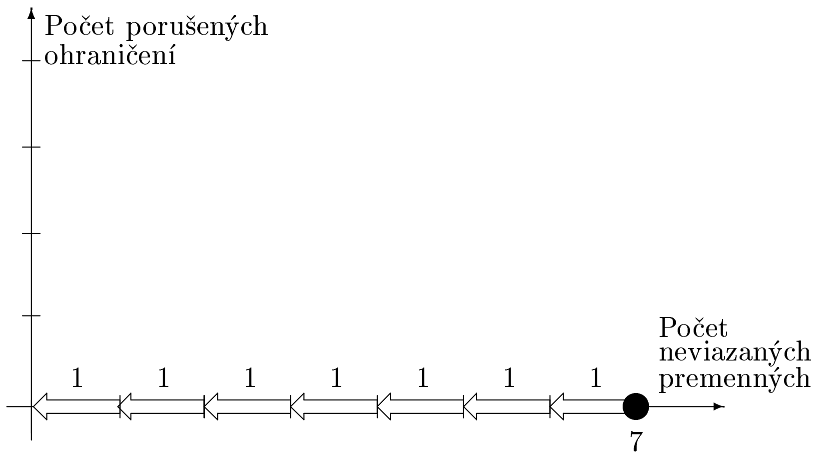
2. realizovať substitúciu.

Redukčné algoritmy sa najčastejšie používajú v kombinácii s prehľadávacími algoritmami. Je ich však možné použiť aj samostatne (často je to však vďaka zložitosti možnosť iba teoretická).

Z hľadiska znázornenia procesu riešenia úlohy v stavovom priestore je pre samostatné použitie redukčných algoritmov charakteristický tvar trajektórie podľa obr. 4.1. Táto trajektória je zložená iba z dvoch riešiacich krokov: redukcie (krok 7) a priradenia (krok 1).

Prvý typ kroku zodpovedá procesu šírenia ohraničení pri konzistenčných algoritmoch alebo substitúcii hodnôt pri substitučných algoritmoch, keď dochádza k úpravám jednotlivých ohraničení a domén. Počet krokov takéhoto typu je premenlivý a závisí od konkrétneho problému.

Druhý typ krokov slúži pre vygenerovanie riešenia. Počet priraďovacích krokov je n – je potrebné priradiť hodnotu každej premennej. Pritom priraďovanie hodnoty nejakej premennej V_i sa nedeje ako výber ľubovoľnej hodnoty z jej domény D_i, ale pre tento výber sa použijú iba tie hodnoty z domény D_i, ktoré sú konzistentné s hodnotami už priradenými ostatným premenným. Tým sa napríklad pri existencii viacerých disjunktných riešení zabráni tomu, aby sa nejakej premennej priradila hodnota patriaca k jednému rieše-



Obr. 4.1: Typická trajektória pre redukčné algoritmy.

niu a nejakej inej premennej zase hodnota z druhého riešenia. Formálnejšie možno tento postup zapísať napr. v nasledovnom tvare: [1]

```
let no_solution_flag = #f;
let i = 0;
while ( and ( not ( no_solution_flag ), i < n ) )
   i := i + 1;
   D[i] := remove ( D[i], non_consistent( D[i] ) );
   if ( empty ( D[i] ) )
       no_solution_flag := #t;
   else
       V[i] := select ( D[i] );
   end if;
end while;
```

4.1.1 Konzistenčné algoritmy

Konzistenčné algoritmy pracujú s pojmom *k-konzistencie* [2]. Pritom v úlohe hodnoty k môžu vystupovať prirodzené čísla z intervalu $< 1, n >$. Voľne by

[1]Pre zápis algoritmov je použitý programovací jazyk Dylan. Pre zachovanie syntaktickej správnosti bolo nutné označenie D_i, reprezentujúce doménu, nahradiť tvarom `D[i]`. Znaky `#t` a `#f` reprezentujú logické hodnoty *true* a *false*.

[2]Existuje aj všeobecnejší pojem (i,j)-konzistencie. Tento typ konzistencie môže byť redukovaný na k-konzistenciu v špeciálnom prípade $i = k - 1$ a $j = 1$ [53].

sa dalo povedať, že sa jedná vlastne o konzistenciu podmnožín premenných z množiny všetkých premenných $\{V_1, \ldots, V_n\}$ a hodnota k vlastne udáva kardinalitu týchto podmnožín. Úlohou konzistenčných algoritmov je túto k-konzistenciu zabezpečiť. Konzistenčné algoritmy vlastne zabezpečia, že nejaká subsieť siete ohraničení je konzistentná voči svojmu okoliu. Pritom pod týmto okolím sa chápe nejaká premenná, ktorá sa nevyskytuje v danej subsieti – teda takýchto okolí pre nejakú subsieť môže existovať vo všeobecnosti viac.
Sieť ohraničení je k-konzistentná, ak platí:

> Nech ľubovoľným $k-1$ premenným sú priradené také hodnoty z ich domén, aby všetky ohraničenia definované nad touto (k-1)-ticou premenných boli splnené. Pre ľubovoľnú ďalšiu k-tu premennú je možné vybrať takú hodnotu z jej domény, že všetky ohraničenia definované nad vzniknutou k-ticou premenných sú splnené.

Takto definovaná k-konzistencia teda vlastne znamená, že v sieti ohraničení neexistuje žiadne také ohraničenie definované nad k-1 premennými, ktorého explicitné vyjadrenie by pripúšťalo nejakú takú kombináciu hodnôt, ktorá nie je prípustná indukovanou podobou daného ohraničenia (v príklade na obr. 2.2 napr. ohraničenia $C^{3,4}$ a $C^{1,4}$ indukujú neprípustnosť kombinácie ("európa","auto") pre premenné V_1 a V_3). Použitie konzistenčných algoritmov je potom možné chápať ako transformáciu siete ohraničení na ekvivalentnú ale explicitnejšiu sieť, pričom existujúce ohraničenia sú použité pre odvodenie ďalších a ich pridanie do siete. Tento proces sa nazýva šírenie (propagácia) ohraničení [36].

Sieť ohraničení je *prísne* k-konzistentná, ak je j-konzistentná pre všetky také hodnoty j, pre ktoré je splnená nerovnosť $1 \leq j \leq k$.

Pojem 1-konzistencie znamená, že doména ľubovoľnej premennej V_i obsahuje iba také hodnoty, ktoré neporušujú unárne ohraničenie C^i viažúce sa na túto premennú. Pretože sa to týka vždy iba jedného uzla v sieti ohraničení, tento typ konzistencie sa nazýva *uzlová* konzistencia (node consistency).

Naproti tomu 2-konzistencia znamená, že v doméne ľubovoľnej premennej V_i sa nachádzajú iba také hodnoty, ktoré ak spĺňajú ohraničenie C^i, tak sú zároveň konzistentné s doménou ľubovoľnej inej premennej V_j, $j \neq i$. To znamená, že pre príslušné $h_i \in D_i$ existuje taká hodnota $h_j \in D_j$, že dvojica (h_i, h_j) neporušuje ohraničenia definované nad premennými V_i a V_j (napr. $C^{i,j}$ a C^j). Takýto typ konzistencie v sieti ohraničení zahŕňa vždy dva uzly a preto sa nazýva *hranovou* konzistenciou (arc consistency). Ak dva uzly nie sú spojené hranou, reprezentujúcou binárne ohraničenie, tak je možné túto hranu dodefinovať – bude jednoducho reprezentovať prázdne ohraničenie, pripúšťajúce všetky kombinácie hodnôt z domén príslušných premenných.

Podobne 3-konzistencia znamená, že ak z domén ľubovoľných dvoch rôznych premenných V_i a V_j je možné vybrať takú dvojicu hodnôt, ktorá neporušuje žiadne ohraničenie, tak táto dvojica je konzistentná s doménou ľubovoľnej inej premennej V_k, pre $k \neq i$ a $k \neq j$. To znamená, že pre každú dvojicu (h_i, h_j) ($h_i \in D_i$ a $h_j \in D_j$) neporušujúcu žiadne z trojice ohraničení $C^{i,j}$, C^i a C^j existuje nejaká hodnota $h_k \in D_k$ taká, že trojica (h_i, h_j, h_k) neporušuje žiadne z ohraničení $C^{i,j,k}$, $C^{i,k}$, $C^{j,k}$ a C^k.

3-konzistencia vlastne zaručuje, že ľubovoľná kombinácia dvoch hodnôt z domén dvoch rôznych premenných, povolená priamym ohraničením definovaným nad týmito dvoma premennými, je súčasne povolená všetkými možnými cestami dĺžky 2 medzi týmito dvoma premennými (teda je povolená ohraničeniami, indukovanými skupinami ohraničení vytvárajúcimi dané cesty v sieti ohraničení). Z tohto dôvodu je možné v literatúre nájsť označovanie 3-konzistencie ako konzistencie *po ceste* (path consistency).

Analogickým spôsobom je možné definovať konzistencie vyšších stupňov. Dôležité je uvedomiť si, že definícia konzistencie vyššieho stupňa nevyžaduje konzistenciu stupňa nižšieho.

Uvažujme ilustračný príklad podľa obr. 2.2. Pretože je prezentovaný v redukovanej podobe (domény jednotlivých premenných boli redukované tak, aby boli splnené všetky unárne ohraničenia), tak je vlastne uzlovo konzistentný. Nie je však aj hranovo konzistentný, pretože ak napríklad premennej V_1 je priradená hodnota "žirafa", tak nie je možné premennej V_2 priradiť žiadnu hodnotu z jej domény bez porušenia ohraničenia $C^{1,2}$. Podobne nie je 3-konzistentný, pretože ak napríklad premennej V_3 je priradená hodnota "auto" a premennej V_4 zase hodnota "eso", tak nie je možné priradiť premennej V_1 takú hodnotu z jej domény, aby nebolo porušené ohraničenie $C^{1,4}$. Analogicky možno zistiť, že nie je ani 4-konzistentný, lebo pri priradení hodnôt "európa" premennej V_1, "puška" premennej V_2 a "pór" premennej V_4 nie je možné vybrať vhodnú hodnotu pre premennú V_3.

Platnosť resp. neplatnosť k-konzistencie vo všeobecnosti nič nehovorí o platnosti alebo neplatnosti j-konzistencie pre $j \neq k$. V ilustračnom príklade 2-, 3- a 4-konzistencia splnené neboli. Toto je však možné zmeniť. Napríklad ak by doména premennej V_2 neobsahovala hodnotu "puška", tak sieť ohraničení by bola 4-konzistentná, hoci 3-konzistencia naďalej splnená nie je. Naopak, ak by sa hodnota "žirafa" nahradila hodnotou "medveď" (D_1)

a hodnota "puška" zase hodnotou "dekan" (D_2), tak by sa sieť stala hranovo konzistentná, pričom naďalej by nebola 3-konzistentná (napr. po voľbe "medveď" pre V_1 a "íver" pre V_3).

Pojem k-konzistencie má pre $k > 1$ charakter orientovanosti. Pre hranovú konzistenciu, ak uzol V_i siete ohraničení je konzistentný s uzlom V_j, neznamená to ešte, že to platí aj naopak. Uzol V_j ešte nemusí byť konzistentný s uzlom V_i (príkladom sú V_1 a V_2 na obr. 2.2 kde D_2 je konzistentná voči D_1 ale neplatí to naopak). Podobne pri 3-konzistencii, ak premenné V_i a V_j sú konzistentné s premennou V_k, to ešte neznamená, že konzistentné budú napr. aj V_j a V_k voči premennej V_i (v ilustračnom príklade napr. trojica V_1, V_3 a V_4).

Pretože existujú rôzne typy konzistencie, musia existovať aj rôzne navzájom odlišné algoritmy, ktoré umožňujú tieto konzistencie dosahovať. Ich spoločnou vlastnosťou je *redukcia* – upravujú jednotlivé ohraničenia zužovaním povolených kombinácií hodnôt. Táto redukcia sa niekedy označuje ako šírenie ohraničení. Pokiaľ redukujú aj unárne ohraničenia, tak sa to priamo premietne do zužovania domén jednotlivých premenných. Vo všeobecnosti, od toho, aký typ konzistencie daný algoritmus zabezpečuje, závisí aj zložitosť (a tým aj praktická použiteľnosť) tohto algoritmu.

Algoritmus NC pre zabezpečenie uzlovej konzistencie je jednoduchý – stačí prehľadať doménu každej premennej a jej hodnoty porovnať voči príslušnému unárnemu ohraničeniu: [3]

```
for ( i::<integer> from 1 to n )
   for ( hi::<value> in D[i] )
      unless ( satisfy ( C[i], hi ) )
         D[i] := remove ( D[i], hi );
      end unless;
   end for;
end for;
```

Ak niektorá hodnota v doméne skúmanej premennej porušuje unárne ohraničenie, definované nad touto premennou, tak doména tejto premennej je redukovaná o túto hodnotu.

Algoritmus pre zabezpečenie hranovej konzistencie je o niečo zložitejší, pretože je potrebné porovnávať navzájom domény dvojíc premenných. Ak jedna doména obsahuje takú hodnotu, pre ktorú nie je možné vybrať z domény druhej premennej žiadnu hodnotu bez toho, aby binárne ohraničenie

[3] Pre zachovanie syntaktickej správnosti bolo nutné označenie C^i reprezentujúce ohraničenie nahradiť tvarom $C[i]$.

definované nad týmito dvoma premennými bolo porušené, tak je nutné danú hodnotu z domény prvej premennej vypustiť.

Naviac je potrebné zohľadniť fakt, že aj keď v určitom momente je nejaká premenná V_i hranovo konzistentná voči premennej V_j, toto sa môže neskôr zmeniť po redukcii domény D_j prislúchajúcej premennej V_j – a teda vlastne redukcia domény premennej V_j zapríčiňuje nutnosť návratu ku kontrole premennej V_i. Na druhej strane po zmene domény D_i premennej V_i vyvolanej redukciou D_j nie je potrebné opätovne kontrolovať premennú V_j, pretože prvky vypustené z D_i neboli konzistentné so žiadnym prvkom z D_j.

Principiálny algoritmus AC pre zabezpečenie hranovej konzistencie môže mať nasledovnú podobu:

```
let change_flag::<boolean> = #t;
while ( change_flag )
   change_flag := #f;
   for ( i::<integer> from 1 to n )
   for ( hi::<value> in D[i] )
      let delete_flag::<boolean> = #f;
      for ( j::<integer> from 1 to n )
         let consistent_flag::<boolean> = #f;
         for ( hj::<value> in D[j] )
            if ( satisfy ( C[i,j], list ( hi, hj ) ) )
               consistent_flag := #t;
            end if;
         end for;
         unless ( consistent_flag )
            delete_flag := #t;
         end unless;
      end for;
      if ( delete_flag )
         C[i] := remove ( C[i], hi );
         change_flag := #t;
      end if;
   end for; end for;
end while;
```

Namiesto úpravy ohraničenia C^i s nutnosťou následného použitia algoritmu pre zabezpečenie uzlovej konzistencie je možné upravovať priamo doménu D_i.

Tento principiálny algoritmus je značne neefektívny. Pre praktické použitie je možné ho zefektívniť niekoľkými spôsobmi:

- v prípade redukcie domény D_i premennej V_i je postačujúce opätovne testovať iba domény tých premenných, ktoré môžu byť ovplyvnené redukciou D_i (sú to všetky tie premenné, medzi ktorými a premennou V_i existuje binárne ohraničenie)
- pri kontrole premennej V_i nie je potrebná kontrola voči každej premennej, ale iba voči tým, ktoré sú viazané s V_i binárnym ohraničením
- v prípade zistenia nekonzistencie nejakej hodnoty z domény premennej V_i voči nejakej inej premennej, nie je nutné robiť ďalšie testy a je možné túto hodnotu vymazať z domény
- ak je zistená konzistencia nejakej hodnoty z domény premennej V_i voči nejakej inej premennej V_j, nie je nutné pokračovať v testoch voči ostatným hodnotám z domény premennej V_j.

Algoritmus pre zabezpečenie hranovej konzistencie môže mať viacero podôb s rôznou zložitosťou a priestorovými nárokmi, danými jednak spôsobom implementácie a jednak mierou redundantných resp. zbytočných testov. V literatúre je možné nájsť celý rad rôznych implementácií: AC1 v podstate zhodnú s uvedeným algoritmom, AC2 realizujúcu iba jednu úplnú iteráciu cez všetky premenné [86] (je ekvivalentný Waltzovmu filtračnému algoritmu [130]), AC3 používajúcu dynamickú frontu uzlov čakajúcich na kontrolu [86], AC4 založenú na pojme podpory hodnôt premenných hodnotami iných premenných [95], AC5 v generickom tvare umožňujúcu špecializáciu okrem iného aj na AC3 a AC4 [129], AC6 narábajúcu s nutnou podporou hodnôt premenných [14], AC6+ [16] a AC7 [56] založené na symetrii ohraničení $C^{xy} = C^{yx}$, a AC8 dekomponujúcu problém do série menších problémov [24].

Ak by sa algoritmus pre zabezpečenie hranovej konzistencie použil pre príklad na obr. 2.2, pričom by sa jednotlivé premenné skúmali v poradí V_1, V_2, V_4 a V_3, tak by sa dosiahlo nasledovné poradie redukcie domén premenných:

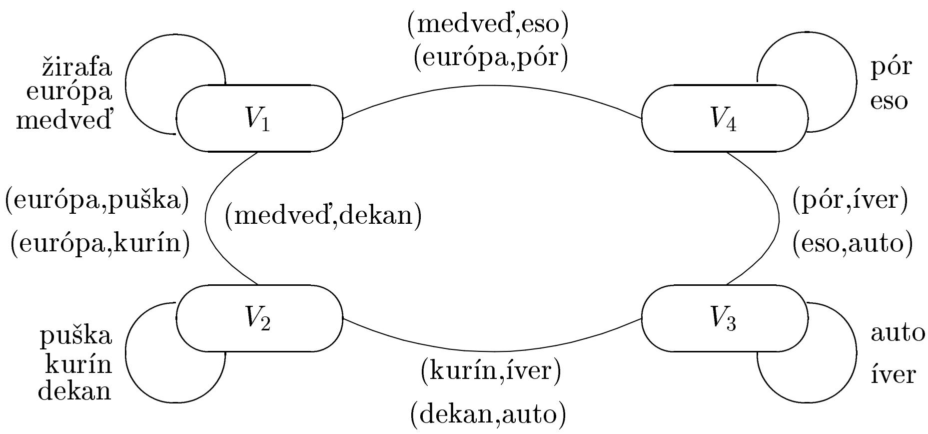
- pri skúmaní hrany (V_1,V_2) je z domény premennej V_1 vypustená hodnota "žirafa"
- kontrola hrany (V_2,V_3) zase spôsobí vypustenie hodnoty "puška" z domény premennej V_2
- hrana (V_4,V_1) zabezpečí odstránenie hodnoty "eso" z D_4
- skúmanie (V_3,V_2) má za následok vypustenie hodnoty "auto" z D_3.

Ak by sa zvolilo iné poradie kontroly domén premenných, poradie vypúšťaných hodnôt by mohlo byť iné, avšak jednotlivé domény budú mať po orezaní výsledný tvar:

$$\begin{aligned} D_1 &: \text{ európa} \\ D_2 &: \text{ kurín} \\ D_3 &: \text{ íver} \\ D_4 &: \text{ pór} \end{aligned}$$

Takýto tvar domén umožňuje priamo určiť riešenie pôvodného problému. Táto redukcia domén má vlastne za úlohu redukovať priestor prehľadávania. Postupné vypustenie štyroch hodnôt z domén premenných má za následok redukciu pôvodného priestoru prehľadávania až na jediný bod.

Ak by však v tomto príklade boli pred orezaním k dispozícii aj ďalšie dve slová "medveď" (6 písmen) a "dekan" (5 písmen), tak sieť ohraničení by mala podobu podľa obr. 4.2, kde voči sieti na obr. 2.2 boli ohraničenia $C^{1,2}$, $C^{1,4}$ a $C^{2,3}$ rozšírené o jednu novú prípustnú kombináciu hodnôt príslušných premenných.



Obr. 4.2: Sieť ohraničení pre ilustračný problém s dvoma riešeniami.

V tomto prípade pri aplikácii algoritmu pre zabezpečenie hranovej konzistencie by došlo k vypusteniu iba dvoch hodnôt – hodnoty "žirafa" z domény D_1 premennej V_1 a hodnoty "puška" z D_2. Domény by po zabezpečení hranovej konzistencie mali tvar

$$
\begin{array}{lcl}
D_1 & : & \text{európa, medveď} \\
D_2 & : & \text{kurín, dekan} \\
D_3 & : & \text{auto, íver} \\
D_4 & : & \text{pór, eso}
\end{array}
$$

Teraz je možné nájsť dve rôzne riešenia: ("európa", "kurín", "íver", "pór") alebo ("medveď", "dekan", "auto", "eso"). Ich hľadanie však nemusí byť priamočiare. Napríklad voľba hodnoty "európa" pre premennú V_1 a hodnoty "auto" pre premennú V_3 vedie do slepej uličky, keď nie je možné zvoliť hodnoty pre premenné V_2 a V_4 bez toho, aby nedošlo k porušeniu žiadneho ohraničenia. Dôvodom je, že nie je zabezpečená 3- a 4-konzistencia. Ich zabezpečenie by viedlo k tomu, že by nebolo možné vybrať nevhodnú východziu kombináciu hodnôt ("európa","auto") pre premenné V_1 a V_3.

Vo všeobecnosti, ak sieť ohraničení obsahuje n uzlov, tak iba prísna n-konzistencia zaručuje priame nájdenie riešenia bez hrozby uviaznutia v "slepej uličke", keď nie je možné nasledujúcej premennej priradiť žiadnu hodnotu z jej domény bez porušenia žiadneho ohraničenia.

Existujú algoritmy pre úpravu siete ohraničení na k-konzistentné pre $k > 2$ [26]. Bohužiaľ zložitosť takýchto algoritmov (napr. pre získanie n-konzistentnej siete s n uzlami) je exponenciálna. Príkladom je principiálny algoritmus PC pre zabezpečenie konzistencie po ceste:

```
let change_flag::<boolean> = #t;
while ( change_flag )
   change_flag := #f;
   for ( i::<integer> from 1 to n )
   for ( hi::<value> in D[i] )
      for ( j::<integer> from i + 1 to n )
      for ( hj::<value> in D[j] )
         let delete_flag::<boolean> = #f;
         for ( k::<integer> for 1 to n )
            let consistent_flag::<boolean> = #f;
            for ( hk::<value> in D[k] )
               if ( and ( satisfy ( C[i,k], list ( hi, hk )),
                        satisfy ( C[k,j], list ( hk, hj )),
                        satisfy ( C[i,j,k], list ( hi, hj, hk ))))
                  consistent_flag := #t;
               end if;
            end for;
            unless ( consistent_flag )
               delete_flag := #t;
            end unless;
```

```
        end for;
        if ( delete_flag )
           C[i,j] := remove ( C[i,j], list ( hi, hj ));
           change_flag := #t;
        end if;
   end for; end for; end for; end for;
end while;
```

V tomto prípade je potrebné porovnávať navzájom trojice domén. Ak dvojica domén umožňuje vybrať nejakú dvojicu hodnôt, ktorá nie je konzistentná so žiadnou hodnotou z tretej domény (jedná sa o konzistenciu dvojice hodnôt ako celku voči tretej hodnote ale zároveň aj o konzistenciu každej zložky tejto dvojice voči tretej hodnote), tak je nutné zabrániť výberu takejto dvojice – toto je možné dosiahnuť úpravou binárneho ohraničenia, prislúchajúceho tým premenným, z domén ktorých daná dvojica hodnôt bola vyberaná. K redukcii samotných domén premenných nedochádza.

Zložitosť algoritmu sa zvyšuje tým, že podobne ako pri zabezpečovaní hranovej konzistencie aj teraz je potrebné riešiť otázku opätovnej kontroly po zmene nejakého ohraničenia.

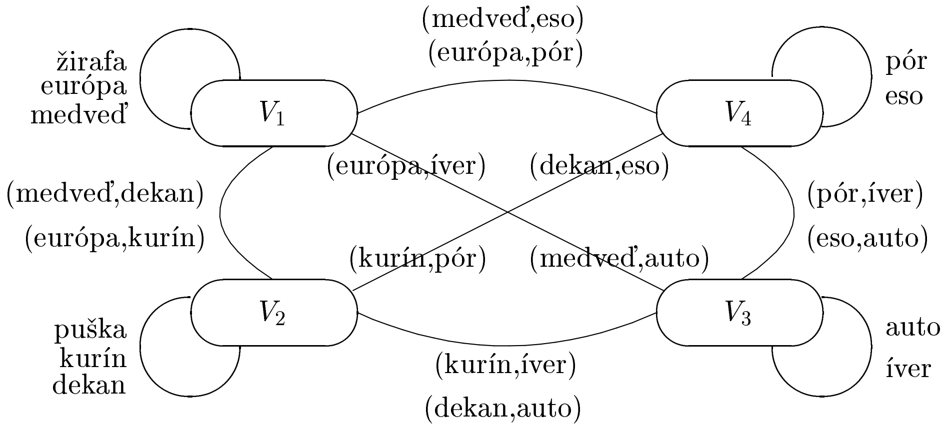
Podobne ako algoritmus pre zabezpečenie hranovej konzistencie aj PC algoritmus môže mať viacero podôb s rôznou zložitosťou a rôznym spôsobom implementácie. Vďaka väčšej zložitosti (a tým menej častému používaniu) je ich menej ako pri AC algoritme: PC1 podobný uvedenej variante, PC2 využívajúci dynamickú frontu uzlov čakajúcich na kontrolu [86], PC3 pracujúci s explicitným vyjadrením podpory hodnôt premenných [95] a PC4 opravujúci skrytú chybu v PC3 [67].

Ak by sa v predchádzajúcom ilustračnom príklade s dvoma možnými riešeniami na obr. 4.2 jednotlivé premenné skúmali v poradí V_1, V_2, V_4 a V_3, tak by sa dosiahlo nasledovné poradie úprav binárnych ohraničení:

- pri skúmaní dvojice premenných (V_1,V_2) voči premennej V_3 je z ohraničenia $C^{1,2}$ vypustená kombinácia hodnôt ("európa","puška")
- kontrola premenných V_1 a V_3 voči premennej V_2 zase spôsobí vypustenie štyroch kombinácií ("žirafa","auto"), ("žirafa","íver"), ("európa","auto") a ("medveď","íver") z ohraničenia $C^{1,3}$
- kontrola premenných V_2 a V_4 voči premennej V_1 spôsobí vypustenie štyroch kombinácií ("puška","pór"), ("puška","eso"), ("kurín","eso") a ("dekan","pór") z ohraničenia $C^{2,4}$.

Keďže pred aplikovaním algoritmu ohraničenie $C^{1,3}$ nebolo definované, predpokladalo sa, že všetkých šesť možných kombinácií hodnôt premenných V_1 a V_3 je prípustných – a teda po redukcii v druhej úprave bolo zavedené nové ohraničenie $C^{1,3}$ obsahujúce iba dve prípustné kombinácie hodnôt ("európa","íver") a ("medveď","auto"). Podobná situácia nastala pre ohraničenie $C^{2,4}$.

V sieti ohraničení pre zabezpečenie 3-konzistencie bolo potrebné pridať dve nové hrany, reprezentujúce dve nové binárne ohraničenia. Výsledná sieť ohraničení je potom ilustrovaná na obr. 4.3.



Obr. 4.3: 3-konzistentná sieť ohraničení pre ilustračný problém s dvoma riešeniami.

Z obrázku je zrejmé, že teraz boli modifikované iba binárne ohraničenia. Unárne ohraničenia (a tým aj domény jednotlivých premenných) naďalej obsahujú hodnoty, ktoré sa zjavne v riešení neobjavia ("žirafa" a "puška"), pretože ich nepripúšťa žiadne binárne ohraničenie. Mohli by byť odstránené zabezpečením prísnej 3-konzistencie (konkrétne použitím algoritmu pre hranovú konzistenciu).

Algoritmus pre dosiahnutie k-konzistencie bude veľmi podobný algoritmu pre 3-konzistenciu: bude potrebné porovnávať navzájom k domén. Ak $k-1$ domén umožní vybrať takú (k-1)-ticu hodnôt, ktorá nie je konzistentná so žiadnou hodnotou z k-tej domény, potom je potrebné zabrániť výberu takejto (k-1)-tice (je to možné úpravou (k-1)-árneho ohraničenia definovaného nad príslušnými premennými). Inak povedané, dosiahnutie k-konzistencie je možné prostredníctvom úpravy (sprísnenia) (k-1)-árnych ohraničení (s ich prípadným dodefinovaním, ak také ohraničenia ešte definované neboli).

Ak sa požaduje prísna k-konzistencia, tak ju možno dosiahnuť postupným zabezpečovaním j-konzistencie, pričom j musí nadobúdať hodnoty v poradí $j = k, k-1, k-2, \ldots, 2, 1$. Inou možnosťou je použitie algoritmov, ktoré priamo zabezpečia prísnu konzistenciu. Príkladom je algoritmus pre zabezpečenie prísnej k-konzistencie pre ľubovoľnú hodnotu k [26].

Zaujímavým prístupom je aj spájanie rôznych algoritmov do kooperačných schém. Napr. v [13] sú kombinované AC a PC algoritmy. Po zabezpečení hranovej konzistencie sa skúmajú všetky ostávajúce hodnoty v doménach premenných. Akonáhle sa pomocou PC (implementovaného iba v parciálnej podobe) zistí, že nejaká hodnota v doméne nejakej premennej stratila svoju podporu v doméne inej premennej, tak táto hodnota je vypustená z domény skúmanej premennej a algoritmus sa prepne na AC (implementovaný v plnom rozsahu). Výsledkom je zaručená hranová konzistencia, pričom PC algoritmus vlastne pomáha AC algoritmu k väčšiemu orezaniu domén.

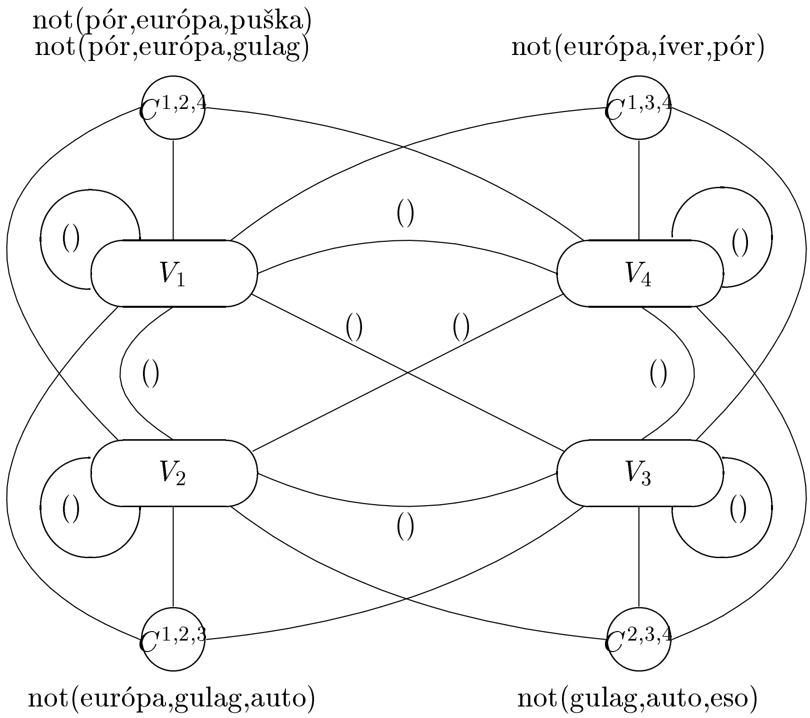
Uvažujme ilustračný príklad podľa obr. 2.2 s tým, že hodnota "kurín" v doméne D_2 premennej V_2 bude nahradená hodnotou "gulag". To má za následok zmenu ohraničení $C^{1,2}$ a $C^{2,3}$:

$$C^{1,2} = \{(\text{európa,puška}),(\text{európa,gulag})\}$$
$$C^{2,3} = \{(\text{gulag,auto})\}$$

Takáto konfigurácia hodnôt jednotlivých domén spôsobí, že problém nemá riešenie (na prvý pohľad to však viditeľné nie je). Pre zabezpečenie prísnej 4-konzistencie je potrebné najprv zabezpečiť 4-konzistenciu. Pri tom sa napr. zistí, že kombinácia hodnôt "európa","gulag" a "auto" z trojice domén D_1, D_2 a D_3 nie je konzistentná so žiadnou hodnotou z domény D_4. Následkom toho je definované ohraničenie $C^{1,2,3}$, ktoré nebude pripúšťať kombináciu ("európa","gulag","auto").

Pri následnom zabezpečovaní 3-konzistencie kombinácia hodnôt "gulag" a "auto" z dvojice domén D_2 a D_3 nebude konzistentná so žiadnou hodnotou z D_1 práve vďaka v predchádzajúcom kroku definovanému ternárnemu ohraničeniu $C^{1,2,3}$ – a to spôsobí, že ohraničenie $C^{2,3}$ všetky kombinácie hodnôt z domén D_2 a D_3 považuje za neprípustné. Prázdnosť $C^{2,3}$ pri zabezpečovaní hranovej konzistencie si vynúti vyprázdnenie unárnych ohraničení C^2 a C^3 a to spôsobí, že po zabezpečení uzlovej konzistencie doména D_2 premennej V_2 bude prázdna. Výsledná sieť ohraničení je na obr. 4.4.

Výsledkom prísnej n-konzistencie (pre n premenných) je minimálna sieť – taká, že každá dvojica hodnôt dvoch premenných je súčasťou aspoň jedného



Obr. 4.4: Prísne 4-konzistentná sieť ohraničení pre ilustračný problém bez riešenia.

riešenia. Ak teda problém s n premennými nemá riešenie, počas zabezpečovania prísnej n-konzistencie sa doména niektorej premennej stane prázdnou. Následné použitie algoritmu pre hranovú konzistenciu spôsobí, že sa vyprázdnia domény všetkých premenných – a teda detekcia neexistencie riešenia je jednoduchá.

V tejto kapitole uvedené algoritmy pracovali so všeobecným typom ohraničení. Pri zohľadnení sémantiky jednotlivých ohraničení je možné navrhnúť špecifické algoritmy, ktoré sú efektívnymi špecializáciami všeobecných (generických) algoritmov. Vďaka tomu, že využívajú sémantiku jednotlivých ohraničení, pri ich testovaní dosahujú väčšiu výkonnosť.

Príkladom môže byť špecializácia algoritmu AC5 pre funkcionálne, antifunkcionálne a monotónne ohraničenia [129] resp. pre špeciálnu triedu funkcionálnych ohraničení [85]. Iným príkladom je algoritmus pre hranovú konzistenciu, navrhnutý špeciálne pre funkcionálne ohraničenia [1], alebo filtračný algoritmus pre n-árne diferenčné ohraničenia [109] (premenné v rámci takéhoto ohraničenia musia mať rôzne hodnoty). Inferenčná schéma AC-Inference umožňuje zmenšiť počet nutných kontrol pri zabezpečovaní hranovej konzistencie využitím metaznalostí o binárnych ohraničeniach (napr. bidirekcionalita, nereflexívnosť, komutatívnosť, ap.) [15]. PC-CRC algoritmus [45] umožní zabezpečiť 3-konzistenciu pre špeciálny typ riadkovo konvexných ohraničení.

4.1.1.1 Stupeň konzistencie

Pri zohľadnení konkrétneho tvaru siete ohraničení je možné niekedy nájsť priamo riešenie pri prísnej k-konzistencii pre $k < n$ (n je počet premenných – uzlov siete). Stupeň k konzistencie, ktorú je potrebné dosiahnuť, závisí od konkrétneho prípadu. Príkladom sú siete ohraničení stromového tvaru, pre ktoré je postačujúce zabezpečiť uzlovú a hranovú konzistenciu [41].

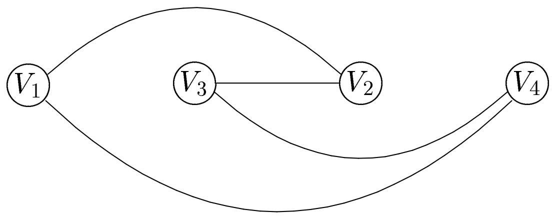
Hodnotu k (jej postačujúcu hodnotu) je možné určiť napríklad z veľkosti domén a árnosti ohraničení [37] podľa vzťahu

$$\max_i(\| D_i \|)\ (arita - 1) + 1 \tag{4.1}$$

Pretože v príklade podľa obr. 2.2 v každej doméne sú iba dve hodnoty a samotná sieť obsahuje iba unárne a binárne ohraničenia, tak postačujúcou je prísna 3-konzistencia. Ak by však doména premennej V_1 obsahovala aj hodnoty "kuchyňa" a "teleso", tak by postačujúcou bola prísna 5-konzistencia (hoci samotná sieť má iba štyri premenné). Z tohto je zrejmé, že takto určená hodnota môže byť pomerne značne nadhodnotená a teda že je skutočne iba postačujúcou a nie nutnou hodnotou.

Inou možnosťou je využiť pre tento účel pojem *šírky*. Pri tomto spôsobe je potrebné usporiadať jednotlivé premenné do takého poradia, v akom sa im pri zostavovaní riešenia budú priraďovať hodnoty. Ak by sme v príklade podľa obr. 2.2 uvažovali premenné v poradí V_1, V_3, V_2 a V_4, tak by bolo možné sieť ohraničení zobraziť v tvare usporiadanej siete ohraničení podľa obr. 4.5 (pre zjednodušenie bez uvažovania konkrétneho tvaru ohraničení a bez znázornenia unárnych ohraničení).

Podľa obrázku je možné všetky ohraničenia, súvisiace s nejakou premennou rozdeliť do dvoch skupín - spájajúce danú premennú s premennými, ktoré sú pred ňou v použitom poradí a s tými, ktoré sú až za ňou. Potom je možné definovať tri typy šírok [52]:



Obr. 4.5: Určovanie šírok uzlov a usporiadania pre poradie premenných V_1, V_3, V_2 a V_4.

šírka uzla je počet hrán, ktorými je uzol spojený s uzlami, ktoré podľa daného usporiadania sú umiestnené pred ním

šírka usporiadania je maximálna hodnota šírok všetkých uzlov pri danom usporiadaní

šírka grafu je minimálna hodnota šírok usporiadania pre všetky možné usporiadania premenných.

To, aká je šírka nejakej siete ohraničení, je možné zistiť pomocou nasledujúceho jednoduchého algoritmu (graph width – GW): [4]

```
let width::<integer> = -1;
while ( not empty( V ) )
   width := width + 1;
   for ( Vi::<value> in V )
      if ( card ( Vi ) <= width )
         V := remove ( V, Vi );
      end if;
   end for;
end while;
```

Pre stupeň k požadovanej prísnej konzistencie postačuje, ak bude väčší ako šírka grafu riešeného problému [52], [41]. Pre náš prípad podľa obr. 4.5

[4] Funkcia `card` zisťuje, s koľkými uzlami je uzol `Vi` spojený nejakou hranou, pričom sa uvažujú iba tie uzly siete, ktoré ešte stále sú zaradené v zozname uzlov `V`.

šírka uzla V_3 je 0 a šírka uzla V_4 zase 2. Keďže žiadny z ostatných uzlov nemá väčšiu šírku, tak šírka usporiadania je tiež 2 a šírka grafu tak isto (o čom sa možno presvedčiť určením šírok usporiadania pre všetkých 24 možných usporiadaní premenných). Teda pre tento problém je potrebné zabezpečiť aspoň prísnu 3-konzistenciu.

Stanovenie hodnoty k je možné nielen podľa topológie siete, ale aj podľa použitých ohraničení[5], napríklad z prísnosti ohraničení (pomer počtu povolených kombinácií hodnôt a počtu všetkých možných kombinácií) [128].

Iným prípadom sú obmedzenia na typ povolených ohraničení. Pre ne je možné uviesť prípad binárnych riadkovo konvexných ohraničení – ak je ohraničenie vyjadrené binárnou maticou, tak táto matica je riadkovo konvexná ak jednotky v žiadnom riadku nie sú navzájom oddelené žiadnou nulou. Platí, že ak binárna sieť ohraničení je prísne 3-konzistentná a existuje také usporiadanie domén premenných, že všetky ohraničenia sú riadkovo konvexné (každá doména má rovnaké usporiadanie pre všetky ohraničenia), tak taká sieť je globálne konzistentná [127].

Napríklad ak by doména premennej V_1 z obr. 2.2 obsahovala hodnoty "európa" a "medveď" a doména premennej V_2 zase hodnoty "puška", "kurín" a "dekan", tak potom ohraničenie $C^{1,2}$ vyjadrené v tvare binárnej matice má tvar

	D_2		
D_1	1	1	0
	0	0	1

a teda matica je konvexná. Ak by však doména D_2 bola usporiadaná "puška", "medveď" a "kurín" tak matica reprezentujúca toto ohraničenie by už konvexná nebola.

Obmedzením tohto prístupu založeného na dosiahnutí prísnej k-konzistencie pre nejakú odhadnutú hodnotu k menšiu než počet premenných je fakt, že pre $k > 2$ zabezpečenie k-konzistencie môže pridávať ďalšie hrany do siete ohraničení, čím sa zároveň mení štruktúra tejto siete ohraničení (a určená hodnota k sa môže stať nedostatočnou). Z tohto dôvodu existuje trieda algoritmov, ktoré sa snažia dosiahnuť väčšiu filtráciu domén premenných a ohraničení ako je to pri hranovej konzistencii (keď sa ešte štruktúra siete nemení) a zároveň nemenia ohraničenia. Reprezentantom je SAC (singletonová hranová konzistencia) [35]. Princíp je vcelku jednoduchý - sieť

[5] V [81] sú charakterizované všetky možné typy ohraničení pre ktoré zabezpečenie prísnej k-konzistencie ($k < n$) garantuje globálnu konzistenciu.

ohraničení je hranovo konzistentná a navyše platí pre každú hodnotu každej premennej, že ak by sa doména danej premennej zúžila iba na tú jednu hodnotu, vo výslednom grafe by pri zabezpečovaní hranovej konzistencie nedošlo k vyprázdneniu žiadnej domény.

4.1.1.2 Smerová konzistencia

Pretože konzistenčné algoritmy sú pomerne zložité, tak každé možné zjednodušenie týchto algoritmov je vítané. Jednou z možností je nezabezpečovať plnú konzistenciu, ale iba konzistenciu v určitom smere. Vychádza sa z toho, že pri priraďovaní hodnôt premenným sa hodnoty priraďujú týmto premenným v určitom poradí. Ak by bolo toto poradie premenných známe vopred, tak je postačujúce zabezpečiť prísnu k-konzistenciu iba pre toto usporiadanie premenných – pre ostatné možné usporiadania premenných ju nie je potrebné zabezpečovať.

Túto úlohu hrá smerová k-konzistencia, ktorej definícia je vlastne definícia plnej k-konzistencie upravená o pojem smerovosti. Podľa nej je sieť ohraničení smerovo k-konzistentná, ak platí:

> Nech je definované nejaké usporiadanie premenných. Nech ľubovoľným $k-1$ premenným sú priradené také hodnoty z ich domén, aby všetky ohraničenia definované nad touto (k-1)-ticou premenných boli splnené. Pre ľubovoľnú ďalšiu k-tu premennú, ktorá v danom usporiadaní nasleduje až za týmito $k-1$ premennými, je možné vybrať takú hodnotu z jej domény, že všetky ohraničenia definované nad vzniknutou k-ticou premenných sú splnené.

Využitie smerovej konzistencie prinesie zmenšenie počtu nutných testov a prípadných úprav ohraničení a domén premenných (napr. doménu tej premennej, ktorej sa hodnota priraďuje ako posledná, nie je potrebné redukovať o nekonzistentné hodnoty). Algoritmus pre zabezpečenie smerovej hranovej konzistencie (directional arc consistency – DAC) by mal potom nasledovný tvar: [6]

```
for ( i::<integer> from n-1 to 1 )
```

[6] Ak by prvý riadok mal tvar "`for ( i::<integer> from 1 to n-1 )`", tak by sa jednalo o usporiadanie kontroly premenných použité v algoritme čiastočného dopredného pohľadu (partial looking ahead – PLA) [125]. Navyše by algoritmus musel byť vložený do "while" cyklu rovnakým spôsobom ako algoritmus AC pre zabezpečenie hranovej konzistencie.

```
for ( hi::<value> in D[i] )
   let delete_flag::<boolean> = #f;
   for ( j::<integer> from i+1 to n )
      let consistent_flag::<boolean> = #f;
      for ( hj::<value> in D[j] )
         if ( satisfy ( C[i,j], list ( hi, hj ) ) )
            consistent_flag := #t;
         end if;
      end for;
      unless ( consistent_flag )
         delete_flag := #t;
      end unless;
   end for;
   if ( delete_flag )
      C[i] := remove ( C[i], hi );
   end if;
end for; end for;
```

Problém je smerovo hranovo konzistentný podľa určitého usporiadania premenných, ak pre každú hodnotu z domény premennej V_i existuje nejaká kompatibilná hodnota v doméne každej premennej, ktorá v danom usporiadaní nasleduje až za V_i.

Dôležité je poradie kontroly premenných (prvý riadok uvedeného algoritmu). Pre ilustráciu nech sa nejaká premenná V_i nachádza v použitom usporiadaní premenných pred inou premennou V_j a teda doména D_i bude kontrolovaná voči doméne D_j a nie naopak. Ak v nejakom okamihu pri kontrole domény D_j premennej V_j dôjde k jej redukcii, tak pri radení premenných od začiatku usporiadania už v tomto okamihu domény všetkých premenných, nachádzajúcich sa pred premennou V_j (vrátane premennej V_i), boli skontrolované voči pôvodnej doméne premennej V_j. A teda zmena domény premennej V_j môže mať vplyv na doménu premennej V_i až v nasledujúcom opakovaní algoritmu. Ak je však použité radenie premenných od konca usporiadania, tak v okamihu zmeny domény premennej V_j premenná V_i ešte kontrolovaná nebola – a teda jej doména bude testovaná už voči zmenenej doméne premennej V_j. Takto sa zmena domény nejakej premennej môže prejaviť ešte v rámci toho istého prechodu cyklom algoritmu. Inak povedané, pri použití radenia premenných od konca (DAC) dochádza minimálne k takému rýchlemu orezávaniu domén premenných ako pri použití radenia premenných od začiatku (PLA) [125].

Ak by sa algoritmus pre zabezpečenie smerovej hranovej konzistencie použil pre príklad na obr. 2.2, pričom by sa použilo usporiadanie premenných

V_1, V_3, V_2 a V_4 (zobrazené na obr. 4.5), tak by sa dosiahlo nasledovné poradie redukcie domén premenných:

- pri skúmaní hrany (V_2,V_3) je z domény premennej V_3 vypustená hodnota "auto"
- skúmanie premennej V_1 voči premennej V_2 má za následok vypustenie hodnoty "žirafa" z D_1.

Jednotlivé domény budú mať po orezaní výsledný tvar:

D_1	:	európa
D_2	:	puška, kurín
D_3	:	íver
D_4	:	pór, eso

a teda nedošlo k orezaniu domén premenných V_2 a V_4, pretože by to vzhľadom na použité usporiadanie premenných bolo zbytočné.

Dosahovanie vyšších stupňov smerovej konzistencie je rovnako ako pri plnej konzistencii spojené s definovaním nových ohraničení, s čím môže súvisieť zmena šírky grafu (a teda potrebný stupeň smerovej konzistencie).

Toto je riešené algoritmom ADAPT pre adaptívnu konzistenciu [39] [38]. Pre nejaké dané usporiadanie premenných algoritmus zabezpečuje smerovú i-konzistenciu, pričom hodnota i sa mení od uzla k uzlu tak, aby zodpovedala šírke daného uzla v okamihu, keď tento uzol je spracovávaný – teda šírka tohto uzla sa mohla meniť pri spracovávaní predchádzajúcich uzlov. Algoritmus ADAPT je vlastne smerovou zjednodušenou variantou prísnej n-konzistencie. Jeho špecializáciou môže byť napríklad odvodený algoritmus DAC, keď sa skúmajú iba podmnožiny uzlov kardinality 1.

4.1.2 Substitučné algoritmy

Substitučné algoritmy vychádzajú na rozdiel od konzistenčných algoritmov z myšlienky, že ak úloha viacerých hodnôt z domény nejakej premennej V_i vzhľadom na určité kritériá je rovnaká, tak potom tieto hodnoty sú navzájom zameniteľné. V tomto prípade je postačujúce, ak v doméne premennej ostáva iba jedna z týchto hodnôt.

Prirodzeným kritériom substitúcie sa zdá byť kritérium súvisiace s riešeniami problému. Z takéhoto kritéria vychádza definícia plnej zameniteľnosti hodnôt. Podľa nej dve hodnoty nejakej premennej sú navzájom plne zameniteľné, ak každé riešenie problému v ktorom vystupuje prvá hodnota ostane riešením toho istého problému aj po nahradení prvej hodnoty hodnotou druhou, a to isté platí aj o náhrade druhej hodnoty hodnotou prvou. Inak povedané, existujú podobné riešenia, ktoré sa navzájom líšia iba v hodnote jednej premennej.

Bohužiaľ identifikácia všetkých plne zameniteľných hodnôt vyžaduje vo všeobecnosti nájdenie všetkých riešení. Z praktického hľadiska sa preto používajú lokálne kritériá pre zameniteľnosť hodnôt.

Príkladom substitučného princípu založeného na lokálnom kritériu je náhrada podľa okolia (neighbourhood substitution - NS) [27]. Pri takomto type substitúcie úlohu kritéria hrajú všetky tie ohraničenia, ktoré sú definované nad skúmanou premennou. Pre ľubovoľnú dvojicu hodnôt $sym0$ a $sym1$ z domény D_i premennej V_i je možné použiť substitúciu $sym0 \rightarrow sym1$ (teda hodnotu $sym0$ nahradiť hodnotou $sym1$), ak pre každé ohraničenie $C^{\dots,i,\dots}$ definované nad V_i platí implikácia[7]

$$sym0 \text{ spĺňa } C^{\dots,i,\dots} \rightarrow sym1 \text{ spĺňa } C^{\dots,i,\dots}.$$

Inak povedané, platnosť hodnoty $sym1$ pokrýva platnosť hodnoty $sym0$ vzhľadom na lokálnu podmnožinu ohraničení.

Toto substitučné pravidlo je možné aplikovať opakovane na sieť ohraničení, až pokiaľ už nie je možná žiadna ďalšia substitúcia. Principiálny algoritmus NS pre tento typ substitúcie má nasledovný tvar:

```
let change_flag::<boolean> = #t;
while ( change_flag )
   change_flag := #f;
   for ( i::<integer> from 1 to n )
      for ( sym0::<value> in D[i] )
      for ( sym1::<value> in D[i] )
         if ( different ( sym0, sym1 ) )
            if ( subsume ( sym0, sym1,
                           relevant_constraints ( C, i ) ) )
               D[i] := remove ( D[i], sym0 );
               change_flag := #t;
            end if;
         end if;
      end for; end for;
```

[7] Ak by namiesto implikácie platila ekvivalencia, tak by sa jednalo o zameniteľnosť podľa okolia (premenné sú zameniteľné, iba ak rovnakým spôsobom spĺňajú všetky pre ne relevantné ohraničenia). Táto je postačujúcou (nie však nutnou) podmienkou pre plnú zameniteľnosť hodnôt.

```
    end for;
end while;
```

Ak by sa algoritmus pre zabezpečenie tohto typu substitúcie použil pre príklad na obr. 2.2, pričom by sa jednotlivé premenné skúmali v ich prirodzenom poradí, tak by sa dosiahlo nasledovné poradie redukcie domén premenných:

- pri skúmaní premennej V_1 je z jej domény vypustená hodnota "žirafa" a je nahradená hodnotou "európa"
- kontrola domény premennej V_2 zase spôsobí vypustenie hodnoty "puška" z domény tejto premennej kvôli jej náhrade hodnotou "kurín".

Jednotlivé domény budú mať po substitúcii výsledný tvar:

$$\begin{array}{lcl} D_1 & : & \text{európa} \\ D_2 & : & \text{kurín} \\ D_3 & : & \text{auto, íver} \\ D_4 & : & \text{pór, eso} \end{array}$$

Hodnota "auto" (ktorá bola eliminovaná pri konzistenčných algoritmoch zabezpečujúcich hranovú konzistenciu) teraz nemôže byť nahradená hodnotou "íver", pretože kombinácia ("íver","eso") porušuje ohraničenie $C^{3,4}$, zatiaľ čo kombinácia ("auto","eso") ho neporušuje.

Z príkladu vidno, že hodnota, ktorá nespĺňa ani jedno ohraničenie ("žirafa"), môže byť nahradená ľubovoľnou inou hodnotou z tej istej domény. Hodnota, ktorá v nejakej kombinácii ostatných hodnôt spĺňa nejaké ohraničenie ("puška" spolu s "európou" spĺňajú $C^{1,2}$), môže byť nahradená inou hodnotou iba vtedy, ak tá spĺňa to isté ohraničenie v tej istej kombinácii ostatných hodnôt ("kurín" a "európa" voči $C^{1,2}$).

Ak by však v tomto príklade bolo pred orezaním k dispozícii aj ďalšie slovo "pýr" (ktoré by bolo súčasťou jedného z dvoch možných riešení), tak pri spracovaní premennej V_4 by bolo eliminované jedno zo slov "pór" a "pýr", čím by sa znížil počet riešení zjednodušeného problému (ale nekonzistentná hodnota "eso" by v doméne D_4 ostala naďalej). To, ktoré slovo by bolo nahradené ktorým, by záviselo od poradia prehľadávania domén jednotlivých premenných.

Uvedená substitúcia (na rozdiel od zameniteľnosti) vlastne obetuje niektoré riešenia ako výmenu za zjednodušené hľadanie iného riešenia. Z nájdeného riešenia a realizovaných substitúcií je totiž pomerne obtiažne zostaviť alternatívne riešenia, pretože nahrádzané a nahrádzajúce hodnoty nemusia spĺňať relevantné ohraničenia rovnakým spôsobom.

Okrem jednoduchej substitúcie jednej hodnoty inou hodnotou tej istej premennej je možný aj všeobecnejší prípad, keď sa nahrádza kombinácia hodnôt rôznych premenných inou kombináciou hodnôt tých istých premenných [80]. Takáto úprava nie je realizovaná redukciou domén premenných, ale úpravami príslušných ohraničení. Príkladom môže byť napr. čiastočná zameniteľnosť [54], keď dve hodnoty nejakej premennej sú vtedy čiastočne zameniteľné, ak ľubovoľné riešenie obsahujúce prvú hodnotu je možné zámenou tejto hodnoty za hodnotu druhú transformovať na iné riešenie, pričom nejaká podmnožina premenných musí zmeniť svoje hodnoty tiež. Ak by napr. domény premenných z príkladu na obr. 2.2 mali tvar

$$\begin{array}{lcl} D_1 & : & \text{žirafa, európa} \\ D_2 & : & \text{puška, kurín} \\ D_3 & : & \text{kvér, íver} \\ D_4 & : & \text{pór, eso} \end{array}$$

tak riešenie

(európa, kurín, íver, pór)

môže byť transformované na iné náhradou hodnoty premennej V_3 "íver" za hodnotu "kvér", pričom sa súčasne musí vhodne zmeniť aj hodnota premennej V_2.

4.2 Prehľadávacie algoritmy

Základným algoritmom pre hierarchické prehľadávanie priestoru riešení je algoritmus *spätného navracania*, ktorý realizuje vlastne slepé prehľadávanie tohto priestoru do hĺbky. Samotný algoritmus spätného navracania bol v histórii niekoľkokrát znovuobjavený [19].

Premenným sa priraďujú hodnoty sekvenčne. Platnosť každého ohraničenia je testovaná hneď ako je to možné (teda hneď po priradení hodnôt všetkým tým premenným, nad ktorými je dané ohraničenie definované). Ak sa zistí porušenie nejakého ohraničenia, algoritmus sa vráti k premennej, ktorá získala svoju hodnotu ako posledná a pokúsi sa jej priradiť hodnotu novú. Ak to nie je možné, vracia sa k premennej, ktorá získala svoju hodnotu ako predposledná atď. Pri tomto navracaní sa vlastne používa obrátené chronologické poradie premenných než pri priraďovaní hodnôt.

Akonáhle sa zistí, že v nejakom podpriestore priestoru riešení (danom nejakým parciálnym priradením hodnôt niektorým premenným) nie je splnené nejaké ohraničenie, tak tento podpriestor už nebude ďalej prehľadávaný delením na menšie a menšie podpriestory, ale celý tento podpriestor sa vylúči z ďalšieho prehľadávania, pretože počet porušení ohraničení by priradením hodnôt aj ostatným (zatiaľ neviazaným) premenným už neklesol. Takýmto spôsobom dochádza k orezávaniu priestoru prehľadávania, čím sa znižuje počet vykonaných priradení.

V prípade, že riešenie existuje, sa algoritmu v určitom štádiu hľadania podarí zúžiť priestor prehľadávania iba na jeden bod a priradiť hodnoty všetkým premenným bez toho, aby nejaké ohraničenie bolo porušené. Ak však riešenie neexistuje, tak po určitej dobe sa algoritmus navracaním vráti až k premennej, ktorá získala hodnotu ako prvá, s tým, že pokus o priradenie novej hodnoty tejto premennej zlyhá.

Premenné sú usporiadané v určitom poradí a podľa tohto usporiadania sú postupne vyberané pre priradenie hodnoty. Podobne hodnoty v doménach premenných sú tiež určitým spôsobom usporiadané a vyberané z domén pre priradenie premenným podľa daného poradia. Tieto usporiadania premenných a hodnôt v doménach premenných môžu byť statické (platné počas celej doby prehľadávania priestoru riešení) alebo dynamické (počas hľadania riešenia sa môžu meniť).

Algoritmus spätného navracania (backtracking – BT) možno potom vyjadriť napr. v tvare: [8]

```
define method backtracking( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      V[i] := next( D[i], V[i] );
      if ( not ( V[i] ) )
         backtracking ( i-1 );
      end if;
      if ( satisfy_all_constraints ( ) )
         backtracking ( i+1 );
      else
         backtracking ( i );
      end if;
   end if;
end method backtracking;
```

[8] Funkcia `next` ako svoj vstup očakáva zoznam hodnôt a hodnotu. Ak zadaná hodnota sa nenachádza v zozname, tak vracia prvú hodnotu zoznamu. Ak sa zadaná hodnota nachádza v zozname, tak vracia nasledujúcu hodnotu zo zoznamu. Ak zadaná hodnota je poslednou hodnotou v zozname, tak funkcia vracia hodnotu `#f` (false).

Pomocou zaznamenávania (a vizualizácie) činnosti algoritmu, menovite toho, kedy ktorej premennej priradil ktorú hodnotu z jej domény a s akým úspechom, je možné získať *strom prehľadávania*. Jeho uzly reprezentujú stavy riešenia úlohy, charakterizované určitým (parciálnym) priradením hodnôt premenným. Hrany označujú prechody medzi stavmi. Tieto prechody môžu byť dvojakého typu:

- premennej sa priradí nejaká hodnota z jej domény (a teda v strome prehľadávania sa vygeneruje nový uzol)
- premenná s priradenou hodnotou sa uvoľní – nebude mať priradenú žiadnu hodnotu a teda sa stáva neviazanou.

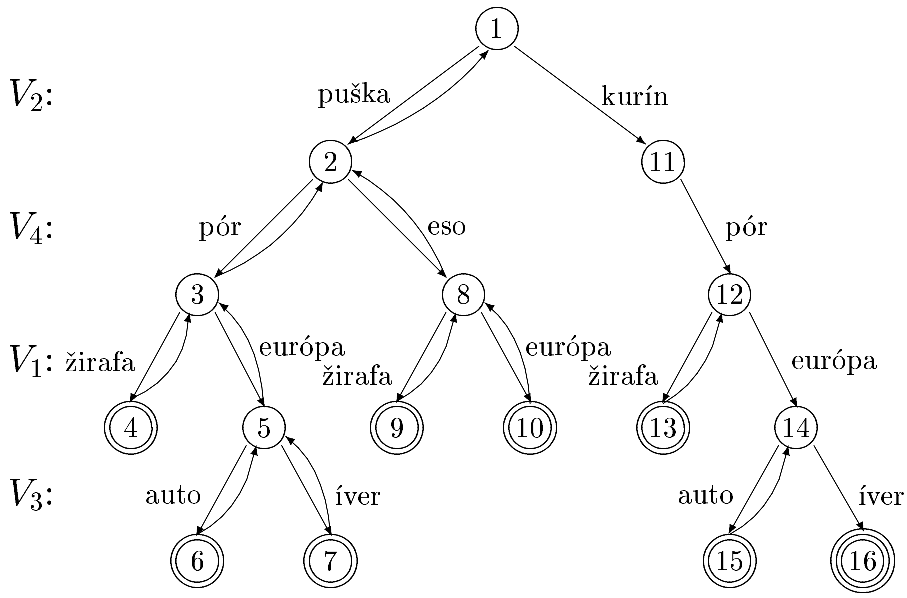
Obidva uvedené typy prechodov znamenajú pohyb v hierarchii podpriestorov priestoru prehľadávania. Priradenie zužuje podpriestor na jednu jeho časť (charakterizovanú priradenou hodnotou), uvoľnenie naopak znamená rozšírenie podpriestoru.

Konkrétny tvar stromu prehľadávania závisí od spôsobu výberu premenných a hodnôt. Ak by sa použili rôzne usporiadania premenných alebo hodnôt v doménach premenných, získali by sa stromy prehľadávania rôznych tvarov. Ak riešený problém má práve jedno riešenie, tak všetky možné stromy prehľadávania by smerovali priamejším alebo spletitejším spôsobom k tomuto riešeniu. Ak by však problém mal viac riešení, tie by mohli byť nájdené rôznymi stromami v rôznom poradí (a pri hľadaní iba jedného riešenia by nebolo nájdené vždy to isté riešenie).

Ak by bol algoritmus spätného navracania použitý pre ilustračný príklad na obr. 2.2, pričom by bolo použité zmenené usporiadanie premenných V_2, V_4, V_1 a V_3 a usporiadanie hodnôt v doménach by ostalo nezmenené, tak by bol vygenerovaný strom prehľadávania podľa obr. 4.6.

V tomto strome operácie priradenia hodnôt premenným sú znázornené ako orientované hrany smerujúce nadol a operácie uvoľnenia premenných zase ako hrany orientované nahor. Dvojitý krúžok označuje situáciu keď bolo detekované porušenie nejakého ohraničenia a následne bolo vyvolané navracanie k predošlým stavom hľadania riešenia. Trojitý krúžok označuje úspešné vygenerovanie riešenia – stav keď priestor prehľadávania bol redukovaný na jeden bod a pritom nebolo detekované žiadne porušenie definovaných ohraničení.

Z obrázku je zrejmé orezávanie priestoru prehľadávania. Pri prehľadávaní došlo štyrikrát k orezaniu (v uzloch 4, 9, 10 a 13), pričom celkovo sa ušetrilo



Obr. 4.6: Trasovanie riešenia ilustračného príkladu pomocou spätného navracania pre poradie premenných V_2, V_4, V_1 a V_3.

generovanie ôsmich uzlov v strome prehľadávania (v podstromoch týchto štyroch uzlov).

Algoritmus spätného navracania nemá však len prednosti ale aj nevýhody. Obzvlášť sú mu vyčítané tieto dve necnosti:

- opakované zlyhávanie hľadania riešenia v rôznych častiach priestoru prehľadávania z tých istých príčin (označované ako *trashing* [86])
- redundantné prehľadávanie.

Strom prehľadávania na obr. 4.6 ilustruje obe tieto nevýhody spätného navracania. Príkladom opakovaného zlyhania sú uzly 6 a 15. V oboch zlyhalo priradenie hodnoty "auto" premennej V_3 z toho istého dôvodu – premenná V_4 mala priradenú hodnotu "pór" (uzly 3 a 12) a následkom toho bolo porušené ohraničenie $C^{3,4}$. Dôvodom pre toto chovanie je to, že algoritmus spätného navracania nemá pamäť.

Absencia pamäti je tiež príčinou nasledujúceho redundantného chovania algoritmu. Na začiatku prehľadávania bolo zistené, že hodnota "pór" je kompatibilná iba s jednou hodnotou z domény D_1 (hodnota "európa") – uzly 3, 4 a 5. Keďže však hodnota "puška" (uzol 2) nebola kompatibilná so žiadnou hodnotou z domény D_3, navracanie sa vrátilo do uzla 2 a naučená znalosť o vzťahu hodnoty "pór" k doméne D_1 bola zabudnutá. V neskoršom štádiu prehľadávania táto znalosť musela byť znovuobjavená (uzly 12, 13 a 14). Teda algoritmus to isté musel redundantne objavovať druhý raz.

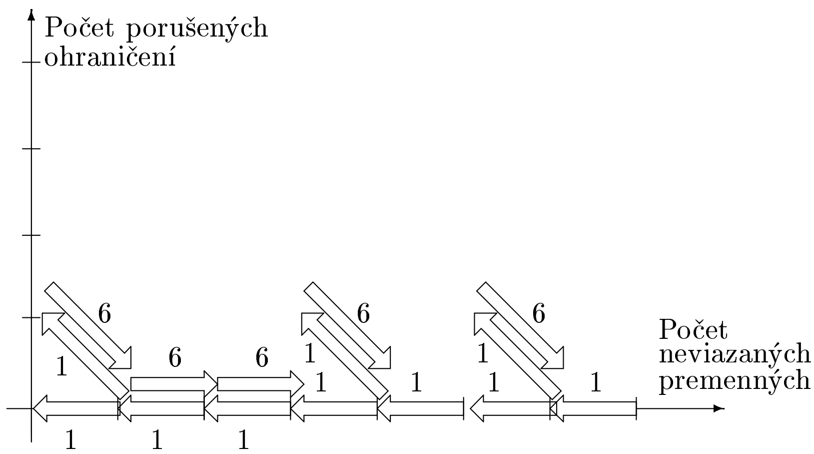
Toto bolo príčinou vzniku radu ďalších algoritmov snažiacich sa tieto nevýhody štandardného spätného navracania eliminovať. V zásade tieto algoritmy možno rozdeliť do dvoch hlavných skupín – skokové algoritmy (snažiace sa o znižovanie počtu nutných navracaní) a pamäťové algoritmy (založené na znižovaní počtu kontrol platnosti jednotlivých ohraničení). Samozrejme sú možné aj hybridy využívajúce princípy oboch skupín [82], [107].

Niektoré z týchto algoritmov sú uvedené v nasledujúcich podkapitolách. Tieto vylepšenia algoritmu spätného navracania generujú vo všeobecnosti menšie stromy prehľadávania. Cenou za to však je to, že majú väčšiu výpočtovú réžiu ako samotný algoritmus spätného navracania. Z tohto dôvodu je možné, že niekedy štandardný algoritmus spätného navracania nájde riešenie skôr než jeho zdokonalené varianty.

Zmenšovanie priestoru prehľadávania pri použití prehľadávacích algoritmov má iný charakter ako v prípade redukčných algoritmov (aj keď v prípade binárnych premenných to nie je viditeľné navonok). Zatiaľ čo pri redukčných algoritmoch bol prítomný princíp "vypustiť všetky kombinácie hodnôt premenných obsahujúce hodnotu, ktorá je vypúšťaná z domény nejakej premennej", tak teraz to je "ponechať všetky kombinácie hodnôt premenných obsahujúce hodnotu priraďovanú nejakej premennej".

Aplikáciou tohto princípu v kombinácii s postupným viazaním premenných vzniká tvar trajektórie v stavovom priestore, ktorý je typický pre prehľadávacie algoritmy. Táto trajektória je znázornená na obr. 4.7.

Z hľadiska stavového priestoru podobne ako algoritmy šírenia ohraničení aj uvedené prehľadávacie algoritmy využívajú iba dva typy riešiacich krokov. V tomto prípade to sú priradenie a uvoľnenie. Keďže je možné aj také priradenie, ktoré spôsobí porušenie nejakého ohraničenia, tak priraďovací krok môže byť znázorňovaný dvojako. Naproti tomu je uvoľňovaná vždy tá premenná ako prvá, ktorej viazanie na konkrétnu hodnotu z jej domény spôsobilo všetky porušenia ohraničení – a teda uvoľňovací krok vždy končí na vodorovnej osi. Pri algoritme spätného navracania dĺžka kroku (vo vodorovnom smere) je vždy jednotková. Pri skokových algoritmoch táto dĺžka



Obr. 4.7: Typická trajektória pre prehľadávacie algoritmy.

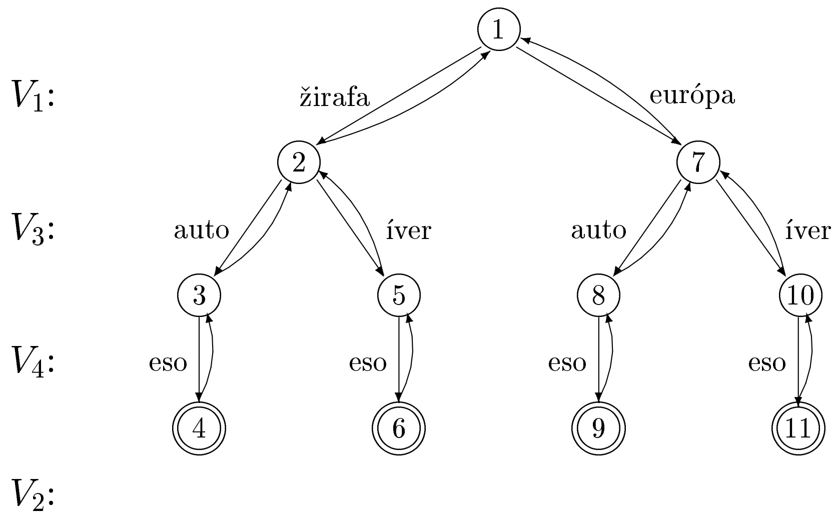
môže byť väčšia (a teda jeden uvoľňovací krok nie je inverziou jedného priraďovacieho).

Tieto dva typy riešiacich krokov sa striedajú podľa toho, či momentálnej etape hľadania riešenia zodpovedá v strome prehľadávania pohyb smerom nadol alebo nahor. Ich počet nemôže byť vopred presne daný. Počet priraďovacích krokov je minimálne n (je potrebné priradiť hodnoty všetkým premenným) a maximálne $\parallel D_1 \parallel \times \ldots \times \parallel D_n \parallel$ (v najhoršom prípade je potrebné vyskúšať všetky možné kombinácie hodnôt z domén premenných). V ideálnom prípade nie je nutné realizovať žiadny uvoľňovací krok. Celkový počet uvoľňovacích krokov nepresiahne počet priraďovacích krokov (v prípade, že úloha nemá riešenie, je potrebné uvoľniť všetky realizované priradenia, a teda počet priraďovacích krokov je rovnaký ako počet uvoľňovacích krokov).

Ak úloha má riešenie, tak trajektória končí v počiatku. Ak však úloha riešenie nemá, tak potom trajektória končí v tom istom bode, v ktorom začala (žiadna premenná nemá priradenú hodnotu a teda žiadne ohraničenie nemôže byť porušené).

4.2.1 Návrat k príčine neúspechu

Ak by v príklade na str. 51 v doméne D_4 nebola prítomná hodnota "pór" a ak by sa zmenilo usporiadanie premenných na V_1, V_3, V_4 a V_2, tak by strom prehľadávania vygenerovaný pri použití algoritmu spätného navracania mal tvar podľa obr. 4.8.



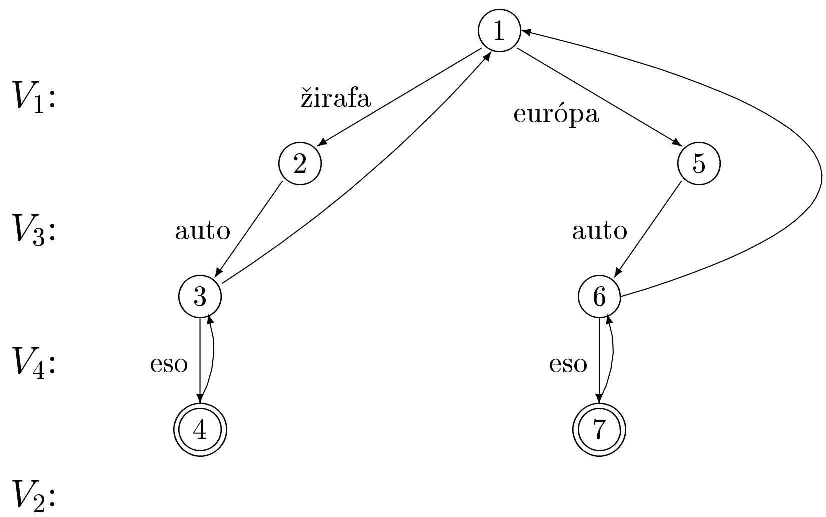
Obr. 4.8: Trasovanie riešenia ilustračného príkladu pomocou spätného navracania pre poradie premenných V_1, V_3, V_4 a V_2.

Pretože v tomto prípade úloha nemá riešenie, všetky listové uzly stromu signalizujú zahájenie navracania. V tomto strome možno nájsť inú podobu opakovaného zlyhávania. Prehľadávanie v uzle 4 zlyhalo preto, lebo hodnota "žirafa" nie je konzistentná so žiadnou hodnotou z domény premennej V_4. Algoritmus sa však vrátil k premennej V_3 a snažil sa jej priradiť inú hodnotu. Keďže neriešil príčinu neúspechu v uzle 4, tak následné prehľadávanie zlyhalo z tej istej príčiny (uzol 6). Bola vykonávaná zbytočná práca, pretože algoritmus nedetekoval príčinu neúspechu, a tým pádom nereagoval optimálnym spôsobom.

Takýto prípad opakovaného zlyhania z tej istej príčiny sa snaží riešiť napr. *spätné navracanie riadené závislosťami* [123], ktoré pri výskyte konfliktu v priradení hodnôt premenným sa pokúša meniť premennú, ktorá je príčinou tohto konfliktu. Podobný prístup reprezentuje Gaschnigov algoritmus *spätného skoku* [61].

Tento algoritmus zaručuje, že strom prehľadávania nebude väčší ako ten, ktorý je dosiahnutý pomocou spätného navracania. Jeho podstatou je to, že pri detekovaní *slepého konca* (deadend) prehľadávania (po vyčerpaní všetkých možností výberu hodnoty z nejakej domény) sa dokáže vrátiť späť naraz aj o niekoľko úrovní – vykoná spätný skok až priamo k tej premennej, ktorá je príčinou neúspechu. Ak neúspech bol zapríčinený viacerými premennými, tak sa vracia k tej z nich, ktorá je najbližšie (teda ktorá získala svoju momentálnu hodnotu najneskôr).

Ak by algoritmus spätného skoku bol použitý pre riešenie toho istého problému ako bol riešený na obr. 4.8, tak by bol vygenerovaný zjednodušený strom prehľadávania podľa obr. 4.9.



Obr. 4.9: Trasovanie riešenia ilustračného príkladu pomocou algoritmu spätného skoku pre poradie premenných V_1, V_3, V_4 a V_2.

Po neúspechu v uzle 4 a následnom návrate k uzlu 3 bola detekovaná nemožnosť konzistentného priradenia hodnoty premennej V_4. Preto je potrebné realizovať navracanie k jednej z už naviazaných premenných (premenné V_1

a V_3). Pretože premenná V_4 bola v konflikte s priradením V_1 a nie s V_3, tak sa vykoná spätný skok od uzla 3 k uzlu 1. Rovnaký dôvod mal za následok spätný skok od uzla 6 k uzlu 1.

Nevýhodou tohto variantu je to, že ak od premennej V_i sa spätne skočí k premennej V_j a tá už nemá žiadnu ďalšiu hodnotu vo svojej doméne na vyskúšanie, tak návrat k premenným pred premennou V_j už prebieha ako pri spätnom navracaní, pretože premenná V_j nie je momentálne v konflikte so žiadnou z premenných ktoré získali svoju hodnotu skôr ako ona.

Túto nevýhodu sa snaží odstrániť algoritmus spätného skoku riadený konfliktmi [82]. Pre každú premennú je udržiavaná *konfliktná množina* – zoznam tých premenných, s ktorými daná premenná bola v konflikte (pri ich momentálnom spôsobe naviazania) pri priraďovaní hodnôt z jej domény (vo všeobecnosti rôzne hodnoty z jej domény môžu zapríčiňovať konflikt s rôznymi premennými). Tento algoritmus spätného skoku riadený konfliktmi (dependency-directed backjumping – DDBJ) môže mať nasledujúci tvar: [9]

```
define method backjumping( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      V[i] := next( D[i], V[i] );
      if ( not ( V[i] ) )
         let j = max ( CS[i] );
         CS[j] := union ( CS[j], del( CS[i], j ) );
         for ( k::<integer> from j+1 to i )
            CS[k] := #();
         end for;
         backjumping ( j );
      end if;
      if ( satisfy_all_constraints ( ) )
         backjumping ( i+1 );
      else
         CS[i] := add ( CS[i], conflict_variables ( i ) );
         backjumping ( i );
      end if;
   end if;
end method backjumping;
```

Ak teda po priradení nejakej hodnoty nejakej premennej V_i nastáva porušenie jedného alebo viacerých ohraničení, tak do zoznamu CS_i prislúchajúceho danej premennej sa vložia všetky tie premenné, s ktorými je daná premenná momentálne v konflikte. Pri nutnosti návratu k predchádzajúcim

[9] Znak `#()` označuje prázdny zoznam.

premenným je zoznam usporiadaný – premenné sa v ňom zoradia podľa toho, v akom poradí im boli priradené hodnoty – a algoritmus pokračuje skokom k tej premennej V_j, ktorá získala hodnotu najneskoršie. Zároveň zvyšok konfliktnej množiny CS_i premennej V_i sa pridáva ku konfliktnej množine tej premennej, na ktorú sa skáče, takže informácia o konfliktoch sa nestráca.

Pretože premenná V_i po realizácii spätného skoku nebude mať priradenú žiadnu hodnotu, tak súčasne nebude v konflikte so žiadnou inou premennou a teda zoznam prislúchajúci tejto premennej sa vyprázdni. Po návrate k opätovnému pokusu naviazať premennú V_i na nejakú hodnotu sa zoznam prislúchajúci tejto premennej začne budovať z prípadných neúspechov nanovo.

Ak by bol použitý pre riešenie rovnakého problému, ako bol riešený na obr. 4.8, tak by bol opäť vygenerovaný strom prehľadávania zobrazený na obr. 4.9. Pretože v uzle 4 nastal konflikt (bolo porušené ohraničenie $C^{1,4}$), do zoznamu CS_4 zodpovedajúcemu premennej V_4 bola vložená premenná V_1. Po vyčerpaní všetkých možností ako priradiť hodnotu premennej V_4 (návrat do uzla 3) bolo rozhodnuté o návrate k nejakej predchádzajúcej premennej. Zo zoznamu CS_4 bola vybratá premenná V_1. Opäť došlo k väčšiemu orezaniu priestoru prehľadávania ako v prípade použitia spätného navracania.

Existuje aj iný variant algoritmu spätného skoku, ktorý sa snaží eliminovať nutnosť udržiavať konfliktnú množinu pre každú premennú tým, že používa iba jeden zoznam uzlov, ktoré majú byť skontrolované pri výskyte nasledujúceho slepého konca. Je to algoritmus spätného skoku založený na grafe [36].

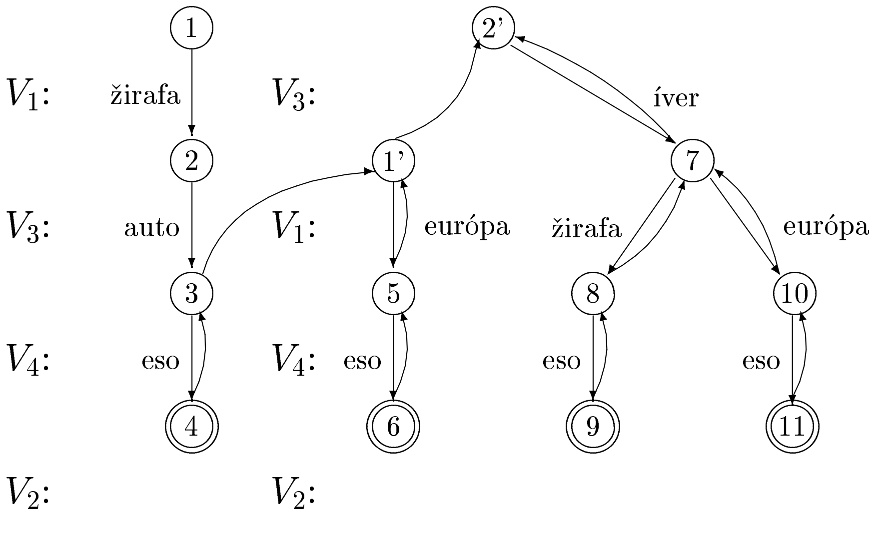
Znalosti o závislostiach vyberá zo samotnej siete ohraničení. Kedykoľvek sa počas procesu prehľadávania vyskytne slepý koniec pri pokuse naviazať nejakú premennú V_i, algoritmus sa vracia na najneskôr naviazanú premennú V_j spomedzi tých premenných, ktoré sú spojené v sieti ohraničení s premennou V_i nejakým ohraničením. Ak V_j už nemá viac hodnôt na vyskúšanie, tak sa algoritmus vracia k najposlednejšej premennej V_k spomedzi tých premenných, ktoré v sieti ohraničení súvisia s V_i alebo V_j atď.

Je to o niečo slabšia verzia ako variant riadený konfliktmi – nevracia sa priamo k príčine neúspechu, ale k možnej príčine neúspechu, čím jeho spätný skok môže byť kratší. Príkladom je opäť problém podľa obr. 4.8. Teraz nebude vygenerovaný strom prehľadávania podľa obr. 4.9, pretože konfliktná premenná V_4 v sieti ohraničení susedí s oboma premennými V_1 a V_3. Výsledkom by v tomto prípade bol strom prehľadávania rovnaký ako na obr. 4.8.

Všetky podoby algoritmu spätného skoku majú tú nevýhodu, že pri navracaní od premennej V_i až k premennej V_j sa spätne uvoľnia všetky premenné, ktorým bola priradená hodnota po naviazaní V_j a pred priradením hodnoty V_i. Tieto premenné budú v ďalších pokusoch o nájdenie riešenia naväzované znova.

Dynamické spätné navracanie [63] toto rieši zmenou poradia niektorých z tých premenných, ktoré už majú priradenú hodnotu. Vráti sa síce k premennej V_j ale tá sa zároveň stane poslednou z minulých premenných a všetky tie premenné ktoré boli naviazané po nej a pred V_i budú presunuté pred V_j. Potom je teda možné zmeniť hodnotu premennej V_j bez modifikácie priradení premenným, ktoré boli naviazané až po nej. To umožní uchovať výsledky predchádzajúcej práce.

Pri použití dynamického spätného navracania pre rovnaký problém ako v predošlom (riešený chronologickým spätným navracaním na obr. 4.8) by bol vygenerovaný strom prehľadávania podľa obr. 4.10 (zároveň to je príklad toho, že použitie zložitejšieho (inteligentnejšieho) variantu voči jednoduchšiemu nie je vždy výhodné).



Obr. 4.10: Trasovanie riešenia ilustračného príkladu pomocou dynamického spätného navracania pre prvotné poradie premenných V_1, V_3, V_4 a V_2.

V tomto prípade vykonanie spätného skoku z uzla 3 nemá za následok stratu hodnoty “auto” pre premennú V_3. Hodnotu stratí iba premenná V_1 (hodnotu “žirafa”) a premenné V_1 a V_3 si navzájom vymenia poradie.

4.2.2 Algoritmy s pamäťou

Doplnenie pamäti k prehľadávaciemu algoritmu umožní tomuto algoritmu v rámci riešenia nejakého problému poučiť sa z výsledkov svojej činnosti v skorších etapách prehľadávania, a tieto svoje skúsenosti využiť v neskorších etapách prehľadávania priestoru riešení. Typicky sa jedná o učenie z neúspechov – algoritmus si tieto svoje neúspechy pamätá a to mu umožní nezopakovať tieto chyby ešte raz.

Najjednoduchším z takýchto algoritmov je algoritmus *spätnej kontroly* [68]. Vždy po priradení hodnoty nejakej premennej je toto priradenie kontrolované na konzistenciu s predchádzajúcimi priradeniami. V prípade, že nejaká podmnožina priradených hodnôt porušuje nejaké ohraničenie, tak ešte pred začiatkom navracania sa táto nekonzistentná podmnožina hodnôt zapamätá.

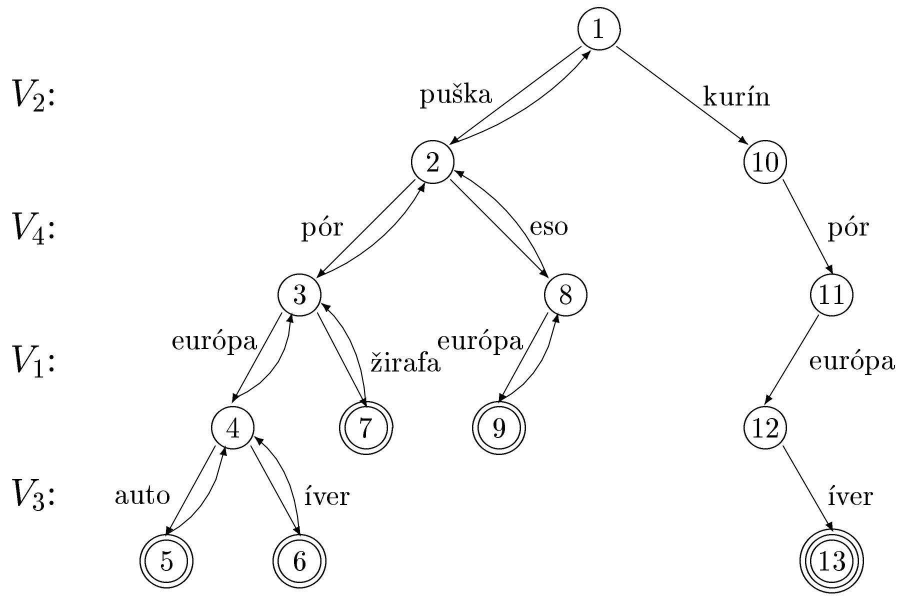
Ak by sa v budúcnosti priradením určitej hodnoty nejakej premennej táto pamätaná kombinácia hodnôt mala opäť vyskytnúť ako súčasť budovaného riešenia, tak k priradeniu nedôjde. Teda vždy pred priradením nejakej hodnoty nejakej premennej sa kontroluje, či rozšírenie aktuálneho riešenia o uvažovanú hodnotu nepovedie k výskytu nejakej kombinácie hodnôt, o ktorej už bolo zistené, že je nekonzistentná.

Ak by algoritmus spätnej kontroly bol využitý pre riešenie príkladu z obr. 2.2 pričom by bolo použité usporiadanie premenných V_2, V_4, V_1 a V_3 a poradie hodnôt v doméne D_1 by bolo opačné, tak by vygenerovaný strom prehľadávania mal tvar podľa obr. 4.11.

Tento strom prehľadávania je menší o dva negenerované uzly než strom, ktorý by bol získaný pomocou chronologického spätného navracania. V uzle 8 nie je pokus priradiť hodnotu "žirafa" premennej V_1. Je to preto, lebo tento pokus by bol neúspešný – táto hodnota spolu s hodnotou "puška" priradenou premennej V_2 by porušovala ohraničenie $C^{1,2}$. A pretože tento fakt bol zistený už v uzle 7, tak v uzle 8 sa tento neúspešný pokus už nezopakuje. Z rovnakého dôvodu v uzle 12 nie je pokus o použite hodnoty "auto" – nevhodnosť tohto pokusu bola zistená v uzle 5.

V princípe spätnú kontrolu je možné použiť na kontrolu ohraničení ľubovoľnej árnosti. Ak sa však uvažujú iba binárne ohraničenia, je potrebné si pamätať iba nekonzistentné dvojice.

Mechanizmus spätnej kontroly sa samostatne nepoužíva, ale zvyčajne sa dopĺňa o mechanizmus eliminácie niektorých redundantných kontrol konzistencie do tvaru algoritmu *spätného značkovania*. Ten vynecháva niektoré kontroly, ktoré boli vykonané v minulosti, boli vtedy úspešné a ak by boli



Obr. 4.11: Strom prehľadávania generovaný algoritmom spätnej kontroly pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

vykonané opäť, tak by boli opäť úspešné [68]. Výsledný strom prehľadávania má rovnaký tvar ako pri spätnej kontrole.

Pre vysvetlenie možno predpokladať, že pokus o priradenie hodnoty nejakej premennej V_i stroskotal, pretože táto premenná nemala vo svojej doméne žiadnu takú hodnotu, ktorá by bola konzistentná s aktuálnym parciálnym riešením. Z toho dôvodu bolo vyvolané navracanie cez premenné $V_{i-1}, \ldots,$ V_{j+1} až k premennej V_j. Tej bola priradená nejaká iná hodnota z jej domény. Premenným $V_{j+1}, \ldots, V_{i-1}$ boli taktiež znovu priradené nejaké hodnoty. Potom pri novom pokuse o priradenie nejakej hodnoty premennej V_i (takej, o ktorej ešte nie je známe podľa algoritmu spätnej kontroly, že by bola nekonzistentná s aktuálnym parciálnym priradením) túto hodnotu pri binárnych ohraničeniach nie je potrebné kontrolovať voči hodnotám tých premenných ktoré svoje hodnoty nadobudli skôr ako V_j – voči tým je hodnota premennej V_i určite konzistentná (keďže tie hodnoty sa nemenili, ale ostali rovnaké aké boli pred začatím navracania od V_i, tak potrebné kontroly už boli realizované pri prvom pokuse priradiť hodnotu premennej V_i).

Pri použití spätného značkovania na rovnaký príklad ako vyššie by bol opäť získaný strom prehľadávania podľa 4.11. Rozdiel oproti spätnej kontrole je ten, že pri spätnom značkovaní v uzle 9 nie je potrebné kontrolovať hodnotu priradenú premennej V_1 voči hodnote premennej V_2, pretože rovnaká kontrola v uzle 4 potvrdila konzistenciu tejto dvojice hodnôt. V uzle 9 sa skontroluje konzistencia iba dvojice hodnôt ("európa", "eso").

Nevýhodou predchádzajúcich algoritmov je to, že si iba zapamätávajú nekonzistentné kombinácie hodnôt, porušujúce existujúce ohraničenia. Nemajú schopnosť sa učiť – nevedia indukovať ohraničenia a pracovať s nimi rovnakým spôsobom ako s ostatnými explicitnými ohraničeniami (teda vytvárať nové alebo sprísňovať už existujúce explicitné ohraničenia). A práve o toto sa snaží prístup *plytkého* a *hlbokého učenia* [36] [58].

Príležitosť indukovať nejaké ohraničenie sa vyskytne vždy pri detekcii slepého konca. Ak nejaké parciálne riešenie, charakterizované priradením hodnôt premenným $V_1 \dots V_i$, nie je možné konzistentne rozšíriť o žiadnu hodnotu z domény premennej V_{i+1} tak principiálne je možné vytvoriť resp. upraviť ohraničenie $C^{1,\dots,i}$ tak, aby daná kombinácia hodnôt nebola dovolená (parciálne riešenie vlastne vytvára konfliktnú množinu). Problémom je to, že by to bolo zbytočné – tá istá kombinácia hodnôt sa vďaka stratégii spätného navracania už v ďalšej fáze prehľadávania nevyskytne. Ak by však nejaká podmnožina nájdenej konfliktnej množiny bola v konflikte s hodnotami v doméne premennej V_{i+1}, tak táto podmnožina sa ešte v ďalšom prehľadávaní môže znovu vyskytnúť a preto má cenu si ju zapamätať vo forme ohraničenia. Cieľom potom je vybrať z danej konfliktnej množiny všetky minimálne konfliktné podmnožiny a tie si pamätať. [10].

Ak nejaká konfliktná množina je v konflikte s nejakou premennou V_{i+1} (nedá sa konzistentne rozšíriť priradením žiadnej hodnoty tejto premennej), tak je z nej možné získať nejakú konfliktnú podmnožinu vypustením priradenia hodnoty nejakej premennej V_k, ak toto priradenie je irelevantné voči V_{i+1}. Toto je splnené napr.:

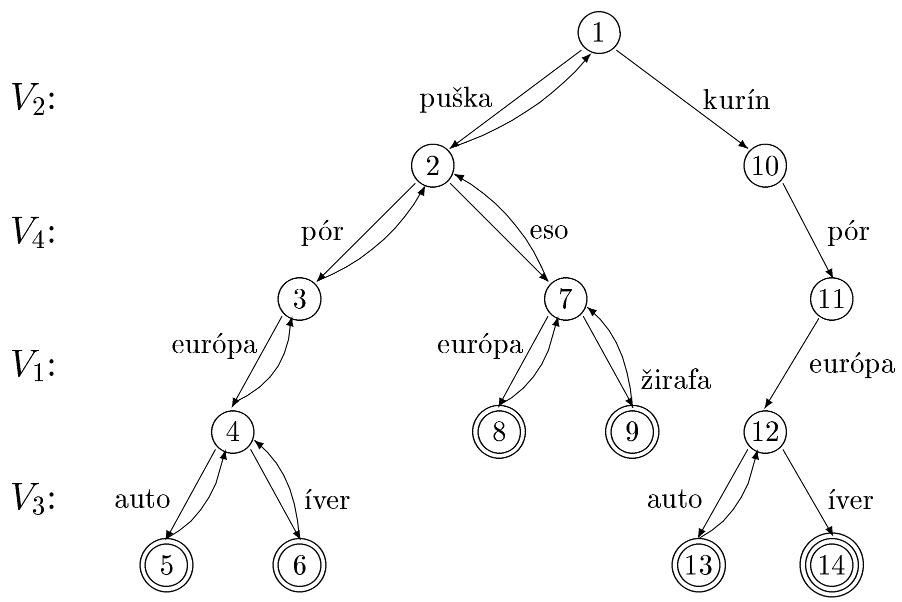
1. ak žiadne ohraničenie nie je definované súčasne nad V_k a V_{i+1}
2. ak priradenie premennej V_k je konzistentné so všetkými možnými hodnotami z domény premennej V_{i+1}.

[10] Je možný aj selektívny výber – pamätať sa budú napr. iba konfliktné podmnožiny určitej veľkosti (napr. binárne) alebo iba tie, ktoré povedú na sprísnenie už existujúceho ohraničenia a nevyžadujú vytvorenie nového ohraničenia, čím by došlo k zmene topológie siete ohraničení.

Takýmto spôsobom sa hľadajú konfliktné podmnožiny pri plytkom učení. Ak sa použije iba prvý bod, tak sa jedná o plytké učenie založené na grafe. Sieť ohraničení poskytuje jednoduchý návod ako identifikovať navzájom irelevantné premenné. Nevýhodou je to, že samotná topológia siete negarantuje identifikáciu všetkých irelevantných priradení v konfliktnej množine.

Na rozdiel od plytkého učenia hlboká varianta postupne analyzuje všetky možné podmnožiny konfliktnej množiny a vyhľadáva všetky minimálne konfliktné podmnožiny.

Pri použití princípu plytkého učenia spolu so spätným navracaním na rovnaký príklad ako je riešený na obr. 4.11 by bol vygenerovaný strom prehľadávania podľa obr. 4.12.



Obr. 4.12: Strom prehľadávania generovaný algoritmom plytkého učenia pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

Po návrate do uzla 4 z uzla 6 bol vlastne detekovaný slepý koniec s tým, že kombinácia hodnôt ("európa", "puška", "pór") je v konflikte s premennou V_3, pretože sa nedá konzistentne rozšíriť ani o hodnotu "auto" ani o hodnotu "íver". Jedinou irelevantnou hodnotou v tejto kombinácii je "európa", pretože neexistuje žiadne ohraničenie definované nad V_1 a V_3. Následkom učenia bude vytvorené nové ohraničenie $C^{2,4}$, ktoré nebude povoľovať kombináciu hodnôt ("puška", "pór"). Toto bude dôvodom toho, že po návrate do uzla 3 sa nevyskytne neúspešný pokus priradiť premennej V_1 hodnotu "žirafa". Podobne po detekcii slepého konca v uzle 7 sa ohraničenie $C^{2,4}$ sprísni tak, aby nepovoľovalo kombináciu ("puška", "eso").

Ak by však bolo použité hlboké učenie namiesto plytkého, tak po detekcii slepého konca v uzle 4 by z konfliktnej množiny ("európa", "puška", "pór") bola vybraná minimálna konfliktná podmnožina ("puška") a táto hodnota by bola zakázaná unárnym ohraničením C^2. Následkom toho by sa negenerovali ani uzly 7, 8 a 9 a navracanie by išlo z uzla 4 až späť k uzlu 1.

Ešte ďalej ide prístup nazývaný *zaznamenávanie nevhodností* [116] [117]. Nevhodnosť je vlastne objekt, zahŕňajúci nekonzistentnú kombináciu hodnôt a zároveň aj dôvod tejto nekonzistencie. Pritom ako dôvod nekonzistencie danej kombinácie hodnôt slúži nejaká kombinácia ohraničení. Tak napr. v uzle 5 podľa obr. 4.12 bolo zistené, že kombinácia hodnôt ("európa", "puška", "auto", "pór") nie je konzistentná, pretože porušuje ohraničenia $C^{2,3}$ a $C^{3,4}$. Na základe toho možno zostrojiť nevhodnosť

$$((\text{európa, puška, auto, pór}), (C^{2,3})).$$

To nové, čo prináša tento prístup, je to, že s nevhodnosťami (reprezentujúcimi nejaké ohraničenia) je možné určitým spôsobom narábať tak, že vznikajú nevhodnosti nové (teda je možné vlastne z existujúcich ohraničení odvádzať ohraničenia nové). Pritom sú možné dve operácie:

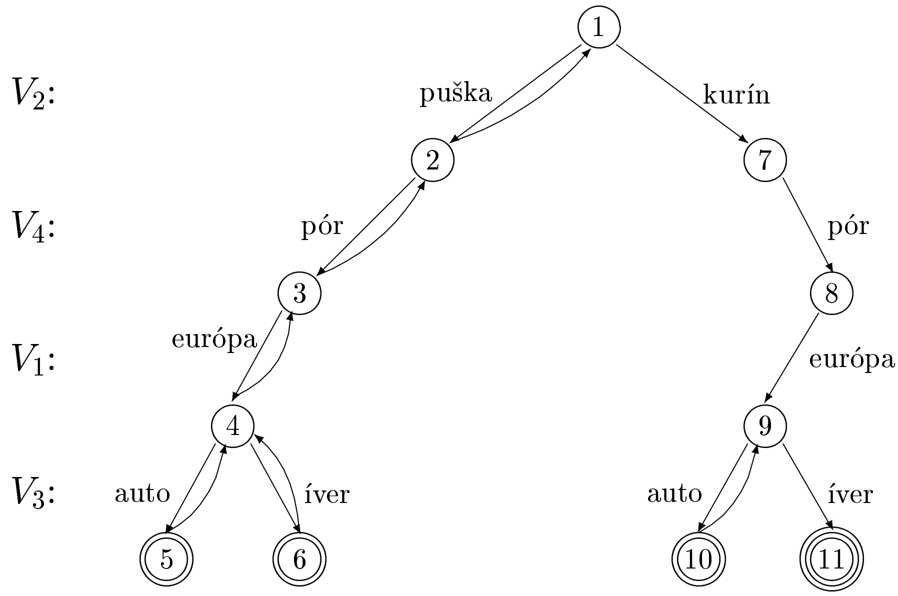
- spájanie

  Je možné spojiť viacero nevhodností do jednej, ak ich kombinácie hodnôt sú také, že obsahujú všetky možné hodnoty z domény nejakej premennej V_i a súčasne sa navzájom líšia iba v hodnote priradenej tejto premennej. Vo výslednej nevhodnosti kombinácia hodnôt nebude obsahovať hodnotu priradenú premennej V_i a kombinácia ohraničení bude vytvorená zjednotením kombinácií ohraničení príslušných spájaných nevhodností.

- projekcia

  Z nevhodnosti je možné vytvoriť novú vypustením tej hodnoty z kombinácie hodnôt, ktorá je priradená takej premennej, nad ktorou nie je definované žiadne ohraničenie z kombinácie ohraničení danej nevhodnosti.

Pri použití tohto princípu spolu so spätným navracaním na rovnaký príklad ako vyššie by bol vygenerovaný strom prehľadávania podľa obr. 4.13.



Obr. 4.13: Strom prehľadávania generovaný algoritmom zaznamenávania nevhodností pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

V uzloch 5 a 6 bola detekovaná nekonzistencia aktuálneho riešenia a preto bolo možné vygenerovať tri základné nevhodnosti:

uzol 5

((európa, puška, auto, pór), ($C^{2,3}$))
((európa, puška, auto, pór), ($C^{3,4}$))

uzol 6

((európa, puška, íver, pór), ($C^{2,3}$))

Dve nevhodnosti je možné spojiť do jednej, pričom to je možné urobiť dvoma spôsobmi (vždy pre jednu z tých ktoré vznikli v uzle 5). Výsledkom sú dve nové nevhodnosti, ktoré už povoľujú ľubovoľnú hodnotu priradenú premennej V_3:

((európa, puška, pór), ($C^{2,3}$))
((európa, puška, pór), ($C^{3,4}$, $C^{2,3}$)).

Na obidve je možné aplikovať operáciu projekcie. V prvom prípade v kombinácii hodnôt sa vyskytuje priradenie premenných V_1 a V_4, hoci nad týmito premennými nie je definované žiadne ohraničenie, ktoré by sa súčasne nachádzalo aj v kombinácii ohraničení danej nevhodnosti. Preto tieto dve priradenia je možné z danej kombinácie hodnôt vypustiť. V druhom prípade je možné vypustiť iba priradenie premennej V_1. Pomocou projekcie sa získajú

((puška), ($C^{2,3}$))
((puška, pór), ($C^{3,4}$, $C^{2,3}$)).

Prvé z nich je vlastne ekvivalentné úprave unárneho ohraničenia C^2, pri ktorom sa zakáže hodnota "puška". A práve vďaka tejto úprave v uzle 3 nie je vyskúšaná hodnota "žirafa" pre premennú V_1 a v uzle 2 zase nie je pokus priradiť hodnotu "eso" premennej V_4.

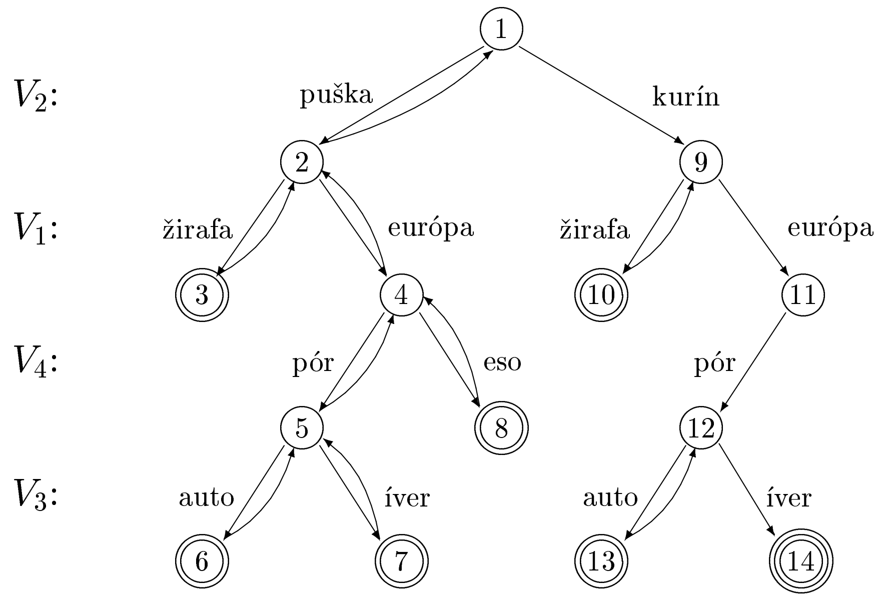
4.2.3 Usporiadanie premenných a ich domén

Proces hľadania riešenia prehľadávaním priestoru riešení závisí nielen od použitého algoritmu prehľadávania, ale je ovplyvnený aj tým, v akom poradí sa daný algoritmus pokúša priradiť hodnoty jednotlivým premenným a v akom poradí tieto hodnoty vyberá z domén týchto premenných. Vhodnou voľbou poradia je možné ovplyvniť tvar prehľadávacieho stromu a tým aj efektívnosť použitia daného algoritmu.

Napríklad použitie spätného navracania pre ilustračný príklad z obr. 2.2 vytvorilo strom prehľadávania so šestnástimi uzlami a dvoma slepými koncami (obr. 4.6). Ak by sa však napr. vymenilo navzájom poradie premenných V_1 a V_4 (a teda celkové poradie premenných by bolo V_2, V_1, V_4 a V_3) a opäť by sa použil algoritmus spätného navracania, výsledný strom prehľadávania by mal iba 14 uzlov a jeden slepý koniec. Jeho tvar je na obr. 4.14.

Z obrázku je zrejmé, že ak by napríklad súčasne v doménach D_1 a D_3 bolo použité opačné poradie hodnôt, tak výsledný strom prehľadávania by bol menší o dva uzly (uzly č. 10 a 13).

V predchádzajúcich kapitolách v ilustračných príkladoch aplikácie prehľadávacích algoritmov bolo vždy uvedené poradie, v akom sa jednotlivým premenným priraďujú hodnoty (poradie hodnôt v doménach sa väčšinou



Obr. 4.14: Strom prehľadávania generovaný algoritmom spätného navracania pre ilustračný príklad pre poradie premenných V_2, V_1, V_4 a V_3.

nemenilo)[11]. Pri riešení nejakého konkrétneho problému je však cieľom vyriešiť tento problém čo najskôr – a teda je vhodné voliť také poradie premenných a hodnôt, ktoré umožňuje čo najrýchlejšie nájdenie hľadaného riešenia (vo všeobecnosti tomu zodpovedá čo najmenší strom prehľadávania).

Veľmi často je možné v literatúre nájsť intuitívny postup, odporúčajúci inštalovať "obtiažnejšie" premenné najprv, čím sa vo všeobecnosti je možné vyhnúť budovaniu parciálnych riešení, ktoré nebude možné dobudovať. Pri takomto výbere ak sa systém dostane do slepej uličky, tak má šancu na to prísť skôr. To je "fail-first" princíp [68] zaručujúci, že každá vetva stromu prehľadávania, ktorá nevedie k riešeniu, bude orezaná čo najskôr a teda celkovo sa bude veľkosť stromu prehľadávania minimalizovať.

Podobne pre výber hodnôt sa odporúča vyberať najmenej ohraničujúce hodnoty – také, o ktorých sa predpokladá, že vystupujú v čo najväčšom počte

[11]Toto poradie bolo volené účelovo, kde účelom bolo získať taký tvar stromu prehľadávania, ktorý by čo najlepšie vystihoval tú-ktorú popisovanú vlastnosť.

riešení, kompatibilných s daným stavom prehľadávania. Takýmto spôsobom ostáva vlastne čo možno najväčší počet možností pre budúce priradenie ostatným voľným premenným. Tento postup zosobňuje "succeed-first" princíp, ktorý volí takú hodnotu, ktorá najpravdepodobnejšie vedie k riešeniu a tým minimalizuje riziko, že bude potrebné navracanie kvôli zmene tejto hodnoty. V praxi sa hodnota, ktorá najpravdepodobnejšie vedie k riešeniu, zamieňa za hodnotu, ktorá najmenej pravdepodobne vedie ku konfliktu.

Na základe mnohých publikovaných porovnaní sa zaužívala predstava, že voľba využiť "fail-first" princíp pre usporiadanie premenných a "succeed-first" princíp pre usporiadanie hodnôt nie je zlou voľbou. Zvyčajne to funguje, ale neplatí to všeobecne – existujú príklady, ktoré protirečia tomuto všeobecnému výberu [121] [122].

V nasledujúcich dvoch kapitolách budú prezentované niektoré všeobecné heuristiky pre výber premenných a hodnôt z domén týchto premenných. Tieto všeobecné heuristiky sú však niekedy málo účinné, lebo neberú do úvahy špeciálne charakteristiky riešených úloh, a preto v praktických aplikáciách môžu byť nahrádzané heuristikami, špeciálne orientovanými na konkrétny typ riešenej úlohy (napr. prideľovanie zdrojov [113]).

4.2.3.1 Usporiadanie premenných

Voľba poradia premenných môže byť dvojakého typu:

- statická
- dynamická.

Pri statickom zotriedení sa určí poradie premenných ešte pred začiatkom priraďovania hodnôt premenným. Toto zotriedenie sa potom používa počas celého hľadania riešenia. Naproti tomu dynamické zotriedenie sa určuje vždy nanovo v okamihu, keď je potrebné rozhodnúť, ktorej premennej sa bude priraďovať hodnota[12]. V tomto prípade sa jedná o zoradenie iba tých premenných, ktorým ešte nebola priradená žiadna hodnota a vlastne nie je potrebné určiť ich celkové usporiadanie ale iba to, ktorá premenná bude na začiatku tohto usporiadania.

[12] Aby dynamická voľba mala zmysel, použitá metóda usporiadania musí mať "dopredný charakter" – musí byť založená na takej charakteristike premenných, ktorá je ovplyvnená aktuálnym priradením hodnôt ostatným premenným. Rôzne priradenia hodnôt nejakej skupine premenných musia spôsobovať rôzne hodnoty tejto charakteristiky pre dosiaľ voľné premenné.

Vo všeobecnosti statické zotriedenia sú výpočtovo menej náročné ako dynamické, pretože tie je potrebné opakovať viacnásobne (čím je väčší strom prehľadávania, tým viackrát musia byť opakovane určované). Na druhej strane však dynamické triedenia umožňujú použiť rôzne poradia premenných v rôznych častiach stromu prehľadávania, čím ich účinok na znižovanie rozmernosti stromu môže byť väčší.

Vo všeobecnosti pre menšie a stredné problémy sú výhodnejšie statické metódy, nevyžadujúce veľa počítania. Pre veľké problémy sa oplatí investovať do dynamických metód [113].

Metódy statického zotrieďovania premenných sú viazané na sieť ohraničení. Najčastejšie v literatúre uvádzané heuristické metódy tohto typu sú [40]:

- minimálna šírka

  Vzostupné usporiadanie podľa počtu tých susediacich uzlov, ktoré zodpovedajú premenným bez priradenej hodnoty. Ako nasledujúci sa vyberá vždy uzol s minimálnym stupňom v podgrafe voľných premenných.

- maximálny stupeň

  Zostupné usporiadanie podľa počtu hrán, ktorými je daný uzol spojený s inými uzlami. Vyberá sa vždy tá premenná, ktorá zodpovedá uzlu s maximálnym stupňom v sieti ohraničení.

- maximálna kardinalita

  Prvá premenná sa volí ľubovoľne. Ostatné sa zoradia zostupne podľa počtu tých susediacich uzlov, ktoré už majú priradenú hodnotu. Teda na každom stupni (okrem prvého) sa vyberá tá premenná, ktorá je spojená s čo najväčším počtom už naviazaných premenných.

- prehľadávanie do hĺbky

  Prehľadávaním do hĺbky (pozdĺž hrán) sa vygeneruje cesta, obsahujúca všetky uzly.

Uvedené metódy zotriedenia nie sú úplné – existujú situácie, keď podľa nich nie je možné rozhodnúť jednoznačne o poradí premenných (napr. ktorým uzlom začať pri prehľadávaní do hĺbky alebo pri rovnakom počte susediacich uzlov daného typu či pri rovnakom počte hrán). Vtedy je možné rozhodnúť náhodne[13] alebo použiť ako pomocné kritérium nejakú inú z uvedených metód zotriedenia. Metóda usporiadania teda nezaručuje vygenerovanie iba jedného poradia premenných.

Z popisu týchto metód by sa mohlo zdať, že niektoré z nich (napr. minimálna šírka) majú dynamický charakter – zohľadňujú uzly siete ohraničení, ktoré už majú alebo ešte nemajú priradenú hodnotu. Jedná sa však o simulované priraďovanie, pri ktorom sa nezohľadňuje, aká konkrétna hodnota bola priradená tej-ktorej premennej.

Uvažujme sieť ohraničení podľa obr. 2.2. Pre metódu minimálnej šírky všetky premenné majú stupeň rovnaký (2). Ak by napríklad bola vybratá premenná V_2, tak stupeň premenných V_1 a V_3 klesol na 1. Pretože stupeň premennej V_4 sa nezmenil, nemôže byť vybratá. Ak teraz bude vyberaná premenná V_1, tak stupeň oboch ostávajúcich premenných je rovnaký a je možné vybrať ľubovoľnú z nich. Pomocou tejto metódy teda možno získať napr. usporiadanie V_2, V_1, V_4 a V_3 (strom prehľadávania zodpovedajúci tomuto usporiadaniu je na obr. 4.14), ale nie je možné získať napr. poradie V_2, V_4, V_1 a V_3.

Pre metódu maximálnej kardinality by bolo možné ako prvú premennú zvoliť V_2 (ľubovoľná voľba prvej premennej to umožňuje). Potom uzol reprezentujúci premennú V_4 nesusedí so žiadnym uzlom reprezentujúcim viazanú premennú a nemôže byť vybraný ako druhý, pretože kardinalita uzlov V_1 a V_3 je vyššia ako jeho kardinalita. Použitím tejto metódy by bolo tiež možné získať poradie premenných použité na obr. 4.14.

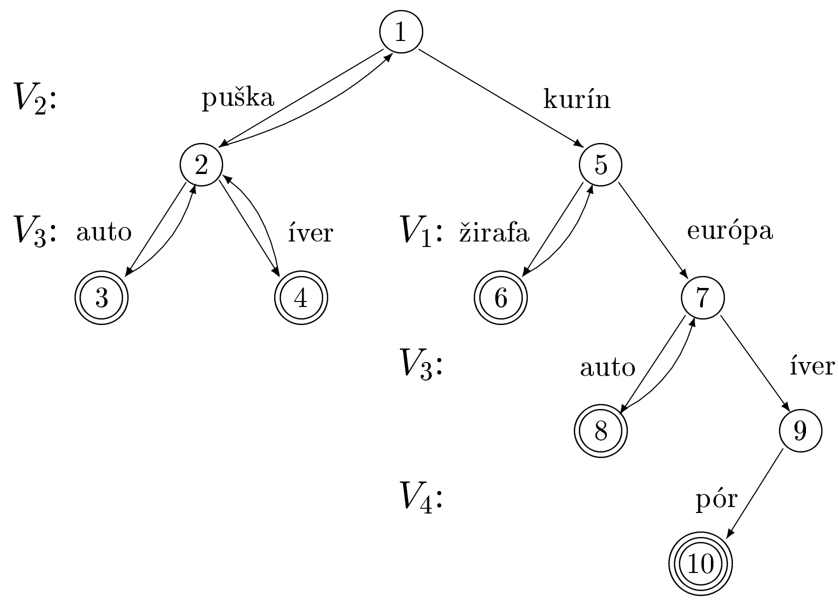
Jednotlivé metódy môžu viesť na rovnaké usporiadanie ako vo vyššie uvedenom prípade. To však neznamená, že by tieto metódy boli ekvivalentné – pre každú dvojicu metód existuje taký tvar siete ohraničení, pri ktorom produkujú rôzne usporiadania.

Ak by sme chceli použiť metódu prehľadávania do hĺbky, mohli by sme ako počiatočný uzol zvoliť V_2. Z tohto uzla sa dá prejsť iba do uzla V_1 alebo V_3. Nech sa zvolí ktorýkoľvek z nich, treťou v poradí bude premenná V_4. Touto metódou nie je možné získať usporiadanie, v ktorom by hneď za sebou nasledovali premenné V_2 a V_4 alebo premenné V_1 a V_3.

Na rozdiel od statických metód, dynamické metódy zotrieďovania zohľadňujú aj konkrétne hodnoty, ktoré už sú priradené premenným. Jednou z

[13] Náhodnosť v riešení nerozhodných situácií je použitá v [64], kde je stanovený prah na počet kontrol. Ak pri riešení úlohy sa dosiahne tento prah, tak riešenie je odštartované odznovu, pričom sa použije iná inicializácia generátora náhodných čísel ako v predchádzajúcom behu.

dynamických metód je *dynamická zmena prehľadávania* [40] (niekedy označovaná ako *minimum ostávajúcich hodnôt* [9]). Táto metóda vyberá vždy tú premennú, ktorá má vo svojej doméne najmenší počet hodnôt konzistentných s aktuálnym priradením. Pre použitý ilustračný príklad je jeden z možných stromov prehľadávania generovaných touto metódou na obr. 4.15.



Obr. 4.15: Strom prehľadávania generovaný algoritmom spätného navracania pre ilustračný príklad pre dynamicky určované poradie premenných.

Na začiatku všetky premenné majú rovnakú šancu na výber, pretože každá z nich má vo svojej doméne dve hodnoty. Po priradení napr. hodnoty “puška” premennej V_2 je vyberaná premenná V_3 – ako jediná z neviazaných premenných nemá v doméne žiadnu hodnotu konzistentnú s hodnotu premennej V_2. Po návrate k premennej V_2 a priradení hodnoty “kurín” tejto premennej je možné voliť spomedzi premenných V_1 a V_3 – obe majú jednu hodnotu zo svojej domény konzistentnú s aktuálnym parciálnym priradením. Po naviazaní premennej V_1 všetky zvyšné premenné majú po jednej konzistentnej hodnote vo svojej doméne a opäť je možno medzi nimi voliť náhodne.

V práci [62] je uvedených niekoľko ďalších heuristík pre dynamický výber premenných. Tieto heuristiky sú založené na snahe odhadovať ohraničenosť subproblému, tvoreného voľnými premennými. Snažia sa o to, aby sa vybrala taká premenná, po naviazaní ktorej subproblém, tvorený zvyšnými premennými, je ohraničený čo najmenej.

Jednou takou heuristikou je *E(N)*, ktorá sa snaží maximalizovať očakávaný počet riešení. Teda snaží sa zároveň o čo najväčší subproblém (s najväčším priestorom prehľadávania) a o čo najväčšiu hustotu riešení (teda čo najmenšiu ohraničenosť). Jej cieľom je vybrať takú premennú V_i, ktorej výber maximalizuje vzťah[14]

$$\left(\prod_{V_k \in V^{free} - V_i} \| D_k \|\right) \left(\prod_{c \in (C^{free} - C_i^{all})} (1 - p_c)\right) \tag{4.2}$$

teda vzťah zvažujúci veľkosti domén voľných premenných (bez ohodnocovanej premennej) a prísnosť ohraničení týkajúcich sa iba voľných premenných (s výnimkou práve ohodnocovanej premennej).

To je vlastne identické so snahou vybrať tú premennú, ktorá najmenej prispieva k ohraničenosti subproblému tvorenému voľnými premennými (po naviazaní vybranej premennej) a teda minimalizuje

$$\| D_i \| \prod_{c \in (C^{free} \cap C_i^{all})} (1 - p_c) \tag{4.3}$$

kde $\| D_i \|$ je veľkosť domény D_i premennej V_i a $(C^{free} \cap C_i^{all})$ sú všetky ohraničenia nad voľnými premennými zahŕňajúce aj V_i a p_c je prísnosť ohraničenia (udávajúca akú časť možných kombinácií hodnôt premenných vylučuje). Pri rovnakej prísnosti všetkých ohraničení sa táto heuristika redukuje na už spomínanú dynamickú zmenu prehľadávania.

Zaujímavou myšlienkou je použitie viacerých heuristík súčasne nielen v tom zmysle, že jedna je primárna a ak tá nevie rozhodnúť, tak sa použije sekundárna, ale ako kombinácia viacerých heuristík do jednej výslednej. Pri dom/deg usporiadaní [17] sa vyberá tá premenná, ktorá má najmenší pomer veľkosti domény a stupňa (jedna časť uvažuje domény hodnôt premenných a druhá zase štruktúru siete ohraničení).

Uvedené zotriedenia je možné používať v súčinnosti s ľubovoľným prehľadávacím algoritmom. Nie vždy to je však vhodné, pretože spôsob zotriedenia

[14] C^{free} označuje všetky ohraničenia definované nad voľnými (dosiaľ neviazanými) premennými a C_i^{all} zase všetky ohraničenia, ktoré sú definované nad premennou V_i.

premenných môže kolidovať s činnosťou algoritmu. Tak napr. použitie dynamickej zmeny prehľadávania v spojení s algoritmom spätného skoku je zbytočné, pretože pri navracaní sa bude tento vracať vždy na najbližšiu úroveň (bude sa chovať ako obyčajné spätné navracanie, ibaže bude mať väčšiu réžiu) [9]. Inou možnosťou je návrh zotriedenia určený pre použitie s nejakým konkrétnym prehľadávacím algoritmom. Takéto zotriedenie má možnosť vychádzať z vlastností tohto konkrétneho algoritmu a využívať ich. Príkladom je usporiadanie premenných pre algoritmus doprednej kontroly, založené na minimalizovaní očakávaného počtu uzlov a očakávaného počtu kontrol [100].

4.2.3.2 Usporiadanie domén

Z pravej vetvy stromu na obr. 4.15 je zrejmá úloha usporiadania hodnôt v doménach (podľa obrázku je zrejmé nevhodné usporiadanie domén D_1 a D_3). V tomto prípade hrajú dominantnú úlohu dynamické heuristické metódy. Niektoré z nich sú:

- lepkavé hodnoty

  preferovanie hodnôt, ktoré sa v minulosti ukázali úspešné

- dopredné heuristiky

  všetky hodnoty selektovanej premennej sú porovnávané voči doménam budúcich[15] premenných podľa nejakého kritéria

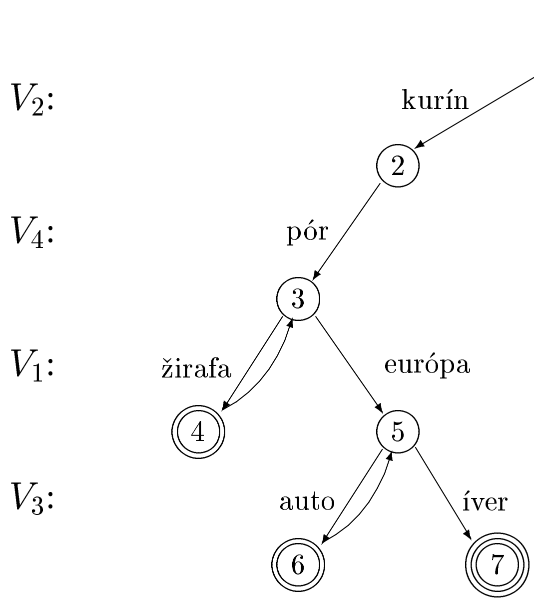
- minimalizácia zložitosti

  preferencia hodnoty, ktorá vedie na jednoduchší problém.

Použitie lepkavých hodnôt [57] možno ilustrovať na strome prehľadávania uvedeného na obr. 4.6. Pri prvom obsadzovaní premennej V_1 v uzle 3 bola hodnota "európa" úspešná (teda aspoň dočasne). Preto pri druhom pokuse o priradenie hodnoty premennej V_1 v uzle 8 bude najprv vyskúšaná hodnota "európa" a až potom hodnota "žirafa". Keďže teraz ani jedna hodnota nebola úspešná, tak toto druhé priraďovanie nebude mať vplyv na poradie hodnôt v doméne D_1. Pri treťom pokuse o priradenie hodnoty premennej V_1 sa opäť začne s hodnotou "európa" – a dôsledkom bude, že uzol 13 nebude generovaný.

[15]Sú to voľné premenné, ktorým v ďalšom postupe prehľadávania bude priradená nejaká hodnota z ich domény.

Príkladom doprednej heuristiky je maximalizácia počtu budúcich možností [83], ktorá maximalizuje počet možností pre budúce priraďovanie hodnôt. Pri tomto spôsobe výberu je preferovaná taká hodnota, po priradení ktorej počet hodnôt v doménach susedných uzlov, ktoré sa stanú nekonzistentnými s aktuálnym priradením, bude minimálny. Ak v príklade na obr. 2.2 sa ako prvej bude priraďovať hodnota premennej V_2, tak sa bude preferovať priradenie hodnoty "kurín", ktorá spôsobí nekonzistenciu dvoch hodnôt ("žirafa" v D_1 a "auto" v D_3), zatiaľ čo použitie hodnoty "puška" by negatívne ovplyvnilo tri hodnoty ("žirafa", "íver" a "auto"). Pri výbere hodnoty pre nasledujúcu premennú V_4 je voľba tiež jednoznačná. Pri obsadení tretej premennej V_1 obe hodnoty sú rovnako možné (neovplyvnia žiadnu hodnotu z D_3). Pre priradenie hodnoty poslednej premennej nie je možné použiť tento spôsob zotriedenia hodnôt a teda výber môže byť náhodný. Ukážka jedného možného stromu prehľadávania je na obr. 4.16.



Obr. 4.16: Strom prehľadávania generovaný algoritmom spätného navracania pre ilustračný problém pre dynamický výber hodnôt z domén.

Iným príkladom doprednej heuristiky je preferencia takej hodnoty, ktorá vytvára najväčšiu minimálnu doménu (z domén voľných premenných) [59] – vychádza z intuície, že je pravdepodobnejšie, že subproblém má riešenie, ak nemá premenné s malými doménami.

Pri metóde minimalizácie zložitosti musí byť daný nejaký spôsob ako určiť zložitosť problému. V [83] je uvedený postup, pri ktorom sieť ohraničení je modifikovaná na stromovú štruktúru vypustením minimálneho počtu hrán. Pre tento jednoduchší problém sa zistí, koľko existuje riešení, obsahujúcich danú skúmanú hodnotu, a tento počet sa použije ako miera zložitosti problému (čím viac riešení, tým je problém jednoduchší)[16].

Sieť ohraničení na obr. 2.2 možno redukovať na stromový tvar vypustením jednej hrany. Existujú štyri také možnosti, ktoré principiálne môžu viesť k rôznym výsledkom. Preto je potrebné nejaké rozhodnutie, ktorá redukcia sa použije. Je možné použiť aj všetky redukcie a o výsledku rozhodnúť hlasovaním.

Ak opäť sa ako prvej bude priraďovať hodnota premennej V_2, tak je možné pre redukovaný problém zistiť nasledujúce počty riešení:

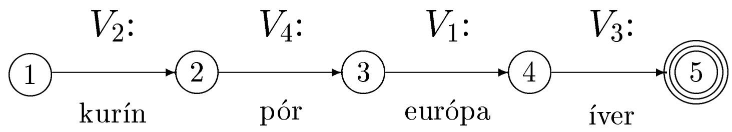
vypustená hrana	priradené hodnoty	počty riešení	
		puška	*kurín*
(V_1,V_2)		0	1
(V_2,V_3)		1	1
(V_3,V_4)		0	1
(V_1,V_4)		0	1

Z uvedeného je zrejmá preferencia hodnoty "kurín". Ak by sa uvažovala iba jedna redukcia na stromový tvar siete ohraničení, tak pri vypustení hrany (V_2,V_3) by mohla byť selektovaná aj hodnota "puška". Po naviazaní prvej premennej sa to isté zopakuje pre druhú premennú. Ak je ňou V_4, tak sa získa (pričom sa uvažuje aj už pridelená hodnota "kurín"):

vypustená hrana	priradené hodnoty	počty riešení	
		eso	*pór*
(V_1,V_2)	*kurín*	0	1
(V_2,V_3)	*kurín*	0	1
(V_3,V_4)	*kurín*	0	1
(V_1,V_4)	*kurín*	0	1

[16] V [41] sa pre tento účel používa minimálna kostra siete ohraničení, kde ohodnotenie každej hrany je dané počtom dvojíc hodnôt príslušných premenných, ktoré sú kompatibilné s ohraničením reprezentovaným danou hranou.

Rovnakým spôsobom je možné zotriediť dynamicky hodnoty v každej doméne pred výberom nejakej hodnoty z tejto domény. Výsledný strom prehľadávania je uvedený na obr. 4.17.



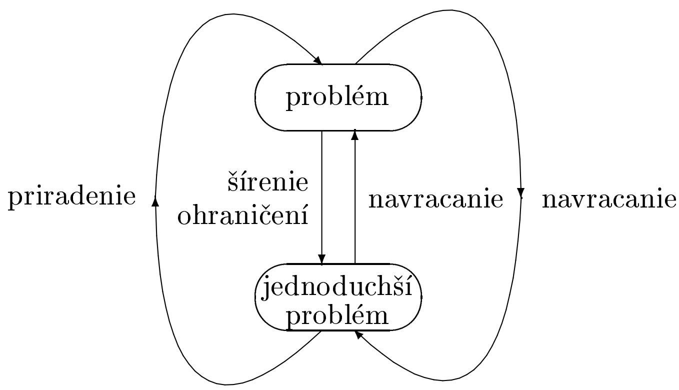
Obr. 4.17: Trasovanie riešenia ilustračného príkladu pomocou spätného navracania pre poradie premenných V_2, V_4, V_1 a V_3.

4.3 Kombinované algoritmy

Redukčné algoritmy zabezpečením prísnej k-konzistencie (kde k je nejaká postačujúca hodnota menšia alebo rovná počtu premenných) umožňujú pohodlne nájsť riešenie. Bohužiaľ ich výpočtová náročnosť zvyčajne presahuje možnosti zdrojov, ktoré sú k dispozícii pre riešenie nejakej konkrétnej úlohy. Na druhej strane prehľadávacie algoritmy sú síce pomerne jednoduché, ale pri "nevhodnej" konfigurácii problému dokážu práve vďaka skúšaniu rozličných nekonzistentných hodnôt vygenerovať obrovské stromy prehľadávania.

Obe triedy uvedených algoritmov majú teda svoje výhody aj nevýhody. A práve ich výhody sa snažia využiť kombinované algoritmy, ktoré spájajú princíp jednoduchého prehľadávania priestoru riešení s metódou zjednodušovania riešeného problému zabezpečovaním nejakého stupňa redukcie ohraničení a domén premenných. Všeobecná schéma takéhoto kombinovaného algoritmu je uvedená na obr. 4.18.

Na aktuálny problém sa aplikuje nejaký redukčný algoritmus. Môže to byť algoritmus pre zabezpečenie plnej (prísnej) k-konzistencie ($1 \leq k \leq n$), nejaký algoritmus, ktorý zabezpečí k-konzistenciu iba čiastočne (negarantuje jej zabezpečenie v celej sieti ohraničení), alebo nejaký substitučný algoritmus. Takýmto spôsobom je vo všeobecnosti pôvodný problém transformovaný na problém jednoduchší. Na tento jednoduchší problém je aplikovaný jeden prehľadávací krok, majúci za následok priradenie hodnoty jednej neviazanej premennej. Po tomto priradení sa problém opäť o niečo zjednoduší ďalšou redukciou atď.



Obr. 4.18: Schéma spojenia prehľadávacieho algoritmu so zabezpečovaním určitého stupňa konzistencie.

Prehľadávacie a redukčné algoritmy je možné rôzne kombinovať. No nie vždy každá kombinácia musí byť účelná. Ak sa kombinuje štandardné chronologické spätné navracanie s redukčnou zložkou, tak výsledok býva pozitívny. Ak sa však uvažujú inteligentné formy prehľadávania, tak ich spojenie s nejakým konkrétnym spôsobom redukcie nemusí vždy priniesť efekt, ale naopak môže celý proces hľadania riešenia spomaliť [17], [111].

Pretože strom prehľadávania bol zobrazovaný takým spôsobom, že priraďovanie hodnôt premenným bolo reprezentované hranami a uzly predstavovali stavy pred resp. po priradení hodnôt, tak jeho rozšírenie pre zobrazovanie činnosti kombinovaných algoritmov je priamočiare – vlastne v každom uzle môže nastať zabezpečenie určitého stupňa redukcie.

Ak v procese redukcie ostane v doméne každej neviazanej premennej iba jedna hodnota a bola zabezpečená minimálne hranová konzistencia siete ohraničení, tak bolo nájdené riešenie. Ak však doména ľubovoľnej premennej sa stane prázdnou (nezávisle od toho, aký stupeň konzistencie je zabezpečovaný), riešenie buď neexistuje alebo niektoré priradenie hodnoty premennej nebolo konzistentné. V oboch prípadoch sa vyvolá navracanie. Podobne ak nejakej premennej je priradená taká hodnota, ktorá spôsobila porušenie nejakého ohraničenia, tak sa tiež vyvoláva navracanie.

Navracanie v kombinovaných algoritmoch je zložitejšie ako v prehľadávacích algoritmoch – je potrebné nielen uvoľniť jednu alebo viac premenných, ale taktiež je potrebné spätne odstrániť redukcie domén a ohraničení, spôsobené redukčnou zložkou algoritmu.

Samotnú redukciu, ktorá môže nastať v nejakom uzle stromu prehľadávania, možno rozdeliť do troch tried v závislosti od toho, ktoré premenné sa zúčastňujú tejto redukcie:

- iba viazané premenné (premenné s priradenou hodnotou)
- iba neviazané premenné (premenné bez priradenej hodnoty), označované aj ako budúce premenné
- viazané aj neviazané premenné.

Zabezpečovanie redukcie iba medzi premennými s priradenými hodnotami nemá zmysel – pretože priradením hodnôt premenným sa vlastne ich domény zredukovali na jednu hodnotu, tak už ďalšia redukcia nie je možná, a súčasne ak by tieto hodnoty neboli konzistentné, museli by spôsobiť porušenie nejakého ohraničenia (a to by automaticky malo za následok navracanie). Ak je teda k viazaných premenných, tak medzi nimi je zaručená prísna k-konzistencia.

Redukcia iba medzi neviazanými premennými zmysluplná je (v koreňovom uzle stromu prehľadávania, keď ešte ani jedna premenná nemá priradenú hodnotu, ani nič iné nie je možné). Ak by sa však iba tento krok zabudoval do prehľadávacieho algoritmu, tak by mal zmysel iba v koreňovom uzle – v žiadnom ďalšom uzle by už nedošlo k žiadnej redukcii domén neviazaných premenných či ohraničení, definovaných nad týmito neviazanými premennými.

Pri treťom type vlastne dochádza k redukcii domén neviazaných premenných a s nimi súvisiacich ohraničení v závislosti na hodnotách priradených premenným. Ak sa takáto redukcia realizuje v každom uzle stromu prehľadávania, tak potom sa zvyčajne uvažuje redukcia iba v súvislosti s posledne priradenou hodnotou (pretože zabezpečenie konzistencie neviazaných premenných voči skôr priradeným hodnotám bolo už realizované v predchádzajúcich uzloch).

Priradenie hodnoty nejakej premennej môže zapríčiniť redukciu domén neviazaných premenných alebo ohraničení, definovaných nad týmito premennými. Následný pokus o redukciu medzi neviazanými premennými môže spôsobiť ďalšiu redukciu. Ak teda sa v jednotlivých uzloch stromu prehľadávania používa zabezpečovanie nejakého stupňa redukcie medzi viazanými a neviazanými premennými, tak má zmysel v týchto uzloch sa snažiť následne zabezpečiť aj redukciu medzi neviazanými premennými navzájom.

Vzájomné spojenie prehľadávania priestoru riešení a zabezpečovania redukcie do jedného kombinovaného algoritmu môže mať napríklad nasledovnú podobu[17]:

```
define method backtracking( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      V[i] := next( D[i], V[i] );
      if ( not ( V[i] ) )
         restore ( i-1 );
         backtracking ( i-1 );
      end if;
      if ( satisfy_all_constraints ( ) )
         check_present ( i );
         check_future ( i+1 );
         if ( empty_domain ( i+1 ) )
            restore ( i );
            backtracking ( i );
         else
            backtracking ( i+1 );
         end if;
      else
         backtracking ( i );
      end if;
   end if;
end method backtracking;
```

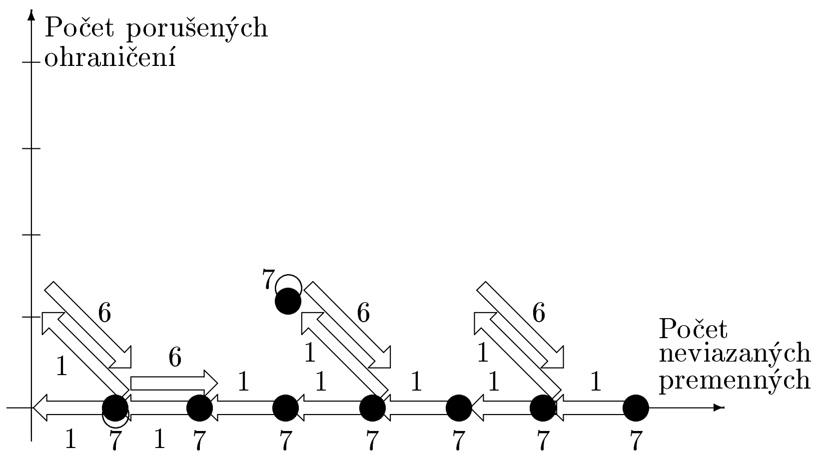
V uvedenej schéme sa používa spätné navracanie pre prehľadávaciu zložku – ale možno tu použiť aj iný prehľadávací algoritmus. Funkcia `check_present` reprezentuje redukčnú zložku algoritmu, kontrolujúcu neviazané premenné voči už priradeným hodnotám, a `check_future` zase redukčnú zložku, pracujúcu iba nad ešte neviazanými premennými. Keďže priraďovanie hodnôt jednotlivým premenným je realizované v určitom poradí, tak v prípade, že toto poradie je vopred známe (čo však nemusí byť – môže byť určované dynamicky), je postačujúce použiť smerové varianty redukčných algoritmov.

Rozhodnutie o tom, ktorý z uvedených typov redukcie sa použije, nemusí byť vopred dané ale môže byť dynamické počas riešenia úlohy. V [114] sa adaptívne určuje, či nejaká hrana bude spracovávaná iba ak jedna z príslušných premenných sa stane viazanou zatiaľ čo druhá ostáva neviazanou

[17]Volanie funkcie `restore(i)` zabezpečí odstránenie všetkých zmien v sieti ohraničení, realizovaných ako výsledok redukcie po poslednom priradení hodnoty premennej V_i.

(teda prípad viazanej a neviazanej premennej), alebo aj v prípade, že obe príslušné premenné sú neviazané. V [60] je zvolený zase iný prístup – stále sa začína tým, že sa využíva aj redukcia medzi neviazanými premennými. Táto sa používa dovtedy, pokiaľ nejaká podmienka je splnená. Ak táto podmienka prestane platiť, tak sa v ďalšom používa už iba redukcia medzi viazanými a neviazanými premennými. Možno použiť rôzne podmienky, napr. hĺbku stromu prehľadávania, meranie úspešnosti orezávania domén, veľkosť domén ap.

Z hľadiska stavového priestoru, hybridné algoritmy používajú širšiu paletu základných typov riešiacich krokov: priradenie, uvoľnenie a redukciu. Typický tvar výslednej trajektórie je na obr. 4.19.



Obr. 4.19: Typická trajektória pre kombinované algoritmy.

Počet redukčných krokov závisí od toho, v ktorých fázach prehľadávania sa šírenie ohraničení používa. Niekedy (napr. pri navracaní k predchádzajúcej premennej) je potrebné aplikovať redukčný krok v opačnom zmysle – je potrebné obnoviť stav siete ohraničení a jednotlivých domén do tvaru pred redukčným krokom (na obrázku je takýto spätný redukčný krok znázornený ako nevyplnený krúžok).

Slepý koniec s následným návratom k jednej z už naviazaných premenných môže byť detekovaný pri takomto type algoritmu v dvoch prípadoch: po aplikácii priraďovacieho kroku alebo po aplikovaní kroku redukčného. V oboch prípadoch prvé použitie uvoľňovacieho kroku končí na vodorovnej osi.

4.3.1 Kontrola budúcich premenných voči aktuálnej

Reprezentantom algoritmov pre zabezpečovanie konzistencie neviazaných premenných vzhľadom na už priradené hodnoty je *algoritmus doprednej kontroly* (forward checking – FC)[18]:

```
define method forward_checking ( i::<integer> )
   for ( j::<integer> from i+1 to n )
      for ( hj::<value> in D[j] )
         if ( not ( satisfy ( C[i,j], list ( V[i], hj ) ) )
            D[j] := remove ( D[j], hj );
         end if;
      end for;
   end for;
end method forward_checking;
```

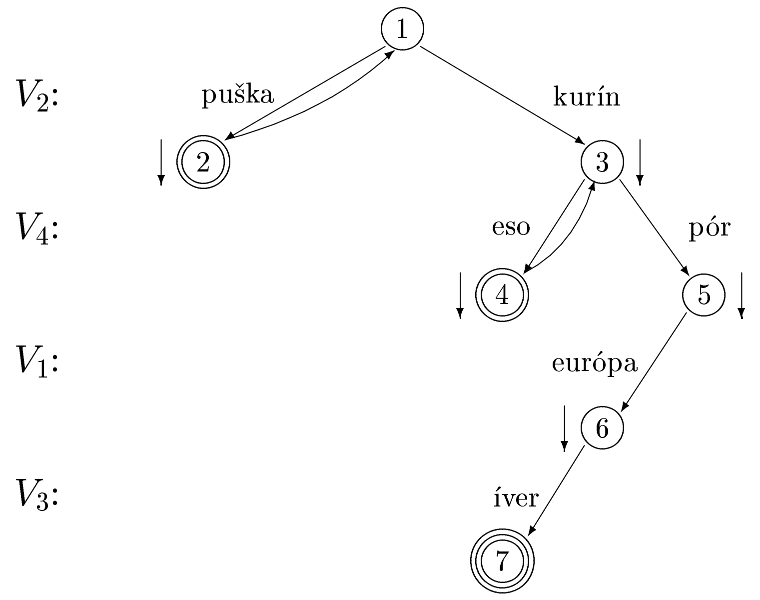
Algoritmus doprednej kontroly zabezpečí hranovú konzistenciu domén všetkých neviazaných premenných voči hodnote tej premennej, ktorá získala svoju hodnotu ako posledná. To znamená, že z domén všetkých neviazaných premenných sú vypustené všetky tie hodnoty, ktoré by po prípadnom priradení týmto premenným viedli k porušeniu binárnych ohraničení, definovaných nad premennou, ktorej práve bola priradená hodnota a jednou z neviazaných premenných.

Ukážka účinnosti tohto algoritmu je ilustrovaná na obr. 4.20. Obrázok zobrazuje strom prehľadávania pre problém podľa obr. 2.2, pričom sa uvažovali premenné v poradí V_2, V_4, V_1 a V_3 a hodnoty v D_4 sa uvažovali v obrátenom (nevýhodnejšom) poradí. Šípka pri uzle znamená, že v danom uzle bol aplikovaný algoritmus pre šírenie ohraničení.

Z obrázku je zrejmé, že dopredná kontrola nebola aplikovaná v koreňovom uzle stromu prehľadávania. To znamená, že pred priradením hodnoty prvej premennej nedošlo k žiadnej redukcii domén premenných. Oba neúspešné pokusy (s následným navracaním) boli detekované práve doprednou kontrolou. Priradenie hodnoty "puška" premennej V_2 spôsobilo, že šírením ohraničení došlo k úprave domén premenných V_1 (vypustená hodnota "žirafa") a V_3 (vypustené obe hodnoty). Prázdnosť domény D_3 signalizovala neexistenciu riešenia a preto bolo vyvolané navracanie. Navracanie v bode 4 bolo zase spôsobené vyprázdnením domén D_1 a D_3.

Ak by sa použil iba samotný algoritmus spätného navracania, tak by vznikol strom prehľadávania, ktorý by mal 19 uzlov a 9 z nich by reprezen-

[18] Je to vlastne funkcia `check_present` zo všeobecného hybridného algoritmu.

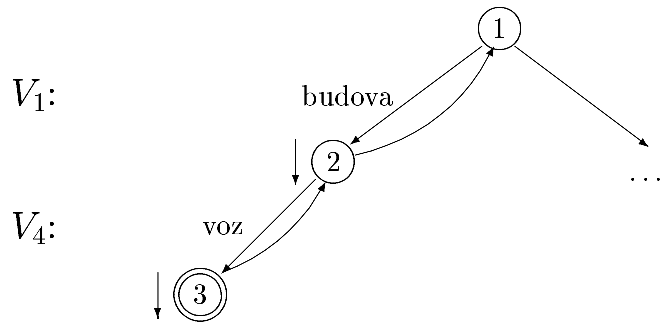


Obr. 4.20: Strom prehľadávania generovaný algoritmom dopřednej kontroly pre ilustračný príklad pre poradie premenných V_2, V_4, V_1 a V_3.

tovalo nutnosť navracania. Doplnenie doprednej kontroly umožnilo vygenerovať strom iba so siedmimi uzlami a len dvoma prípadmi navracania. Z toho vidieť pomerne veľkú účinnosť algoritmu doprednej kontroly – aj keď je veľmi jednoduchý a zabezpečuje hranovú konzistenciu iba medzi niektorými uzlami siete ohraničení, tak jeho opätovné využívanie v každom uzle stromu prehľadávania umožňuje pomerne veľké orezanie priestoru prehľadávania.

Slabinou dopredných schém je to, že robia niekedy zbytočné kontroly. Ak by napríklad v príklade podľa obr. 2.2 boli k dispozícii ešte aj hodnoty "budova" (D_1), "voz" (D_4) a sto ďalších päťpísmenových slov (D_2), tak by časť stromu prehľadávania pri určitom usporiadaní premenných a hodnôt v ich doménach mohla vyzerať podľa obr 4.21.

V tomto prípade pri doprednej kontrole v uzle 2 boli zbytočne testované všetky hodnoty z domény D_2 premennej V_2 voči hodnote "budova" priradenej premennej V_1, ktorá aj tak neskôr bola zamenená inou hodnotou (a teda bolo nutné opäť zisťovať konzistenciu hodnôt z domény D_2).



Obr. 4.21: Časť stromu prehľadávania pre ilustráciu zbytočných kontrol.

Ak pri algoritme doprednej kontroly je premennej V_i priradená nejaká hodnota, tak na túto hodnotu sa kontrolujú domény všetkých premenných V_{i+1} až V_n. Ak však bude neskôr vyvolané navracanie kvôli nemožnosti konzistentného priradenia hodnoty premennej V_j pre $i < j < n$ a následkom toho premennej V_i bude priradená iná hodnota, tak úplná kontrola domén D_{j+1} až D_n voči minulej hodnote premennej V_i bola zbytočná. Toto sa obzvlášť prejaví v prípade veľkého počtu premenných a veľkých domén.

Týmto zbytočným kontrolám sa snaží vyhnúť verzia doprednej kontroly nazvaná minimálna [44], [7]. Kontrola nejakej domény sa robí iba dovtedy (do tej miery), kým sa nájde nejaká konzistentná hodnota v tejto doméne – ostatné hodnoty sa nekontrolujú a ich kontrola sa odloží až do doby, kým bude potrebné vybrať nejakú hodnotu z tejto domény. Tieto dodatočne vykonávané kontroly sa potom uskutočnia v spätnom zmysle voči minulým premenným. Teda všetky testy, ktoré je možné urobiť aj neskôr, sa na tú neskoršiu dobu skutočne odložia (tento princíp sa nazýva "lenivé" vyhodnocovanie).

Cena za to je, že iba prvá hodnota v doménach budúcich premenných je zaručene konzistentná s minulými premennými. Ak bude potrebné vybrať z domény budúcej premennej inú ako prvú hodnotu, tak bude potrebné najprv ju spätne testovať voči minulým premenným (napr. pri navracaní a výbere inej hodnoty alebo ak prvá hodnota sa stala nekonzistentnou a musela byť z domény odstránená). Táto zvýšená réžia je však prevýšená dosiahnutou úsporou kontrol kombinácií hodnôt premenných voči príslušným ohraničeniam, ktorá sa prejaví ako pri použití štandardného spätného navracania, tak aj pri kombinácii minimálnej doprednej kontroly s algoritmom spätného značkovania alebo spätného skoku [84].

4.3.2 Vzájomná kontrola budúcich premenných

Pretože dopredná kontrola zabezpečuje konzistenciu iba medzi niektorými premennými, tak orezávanie priestoru nie je až také veľké aké by mohlo byť pri použití šírenia ohraničení vo väčšom rozsahu. Napr. v uzle 3 na obr. 4.20 následkom doprednej kontroly sa redukovali domény D_1 (na hodnotu "európa") a D_3 (na hodnotu "íver"). Ku kontrole a zmene domény premennej V_4 nedošlo, pretože nie je definované binárne ohraničenie $C^{2,4}$. Ak by sa však do kontroly zahrnula aj v tomto bode neviazaná premenná V_4 a bola porovnaná voči aspoň jednej ďalšej neviazanej premennej (V_1 alebo V_3), tak by z jej domény bola odstránená hodnota "eso" a nedošlo by k zbytočnému generovaniu uzla 4. Z toho je zrejmé, že súčasné použitie nejakého algoritmu pre zabezpečovanie určitého stupňa konzistencie medzi neviazanými premennými ako súčasti kombinovaného algoritmu má zmysel pre zvýšenie stupňa orezávania priestoru prehľadávania.

Úlohu takéhoto šírenia ohraničení často hrá algoritmus pre zabezpečenie hranovej konzistencie alebo nejaká jeho jednoduchšia verzia, ktorá negarantuje toto zabezpečenie pre všetky kontrolované premenné. Táto zjednodušená verzia kontroluje každú hranu (dvojicu domén) presne stanovený počet krát. Má nasledovný tvar:

```
define method check_future ( i::<integer> )
   for ( j::<integer> from ..X.. to ..Y.. )
   for ( hj::<value> in D[j] )
      let delete_flag = #f;
      for ( k::<integer> from ..Z.. to n )
         let consistent_flag = #f;
         for ( hk::<value> in D[k] )
            if ( satisfy ( C[j,k], list ( hj, hk ) )
               consistent_flag := #t;
            end if;
         end for;
         unless ( consistent_flag )
            delete_flag := #t;
         end unless;
      end for;
      if ( delete_flag )
         D[j] := remove ( D[j], hj );
```

```
            end if;
        end for; end for;
    end method check_future;
```

kde hodnoty X, Y (druhý riadok) a Z (piaty riadok) sú rôzne pre rôzne varianty tejto jednoduchšej verzie algoritmu zabezpečovania hranovej konzistencie (navzájom sa líšiace počtom porovnaní dvojíc domén kontrolovaných premenných). Dva také varianty môžu mať nasledovný tvar:

```
define method check_future1 ( i::<integer> )
   for ( j::<integer> from i to n-1 )
   ...
      for ( k::<integer> from j+1 to n )
         ...
   end for; end for;
end method check_future1;

define method check_future2 ( i::<integer> )
   for ( j::<integer> from i to n )
   ...
      for ( k::<integer> from i to n )
         ...
   end for; end for;
end method check_future2;
```

Oba varianty sa líšia iba tým, ktorú premennú s ktorou porovnávajú. Prvý variant je slabší, realizuje približne iba polovicu toľko porovnaní ako druhý. Ak by sa prvý algoritmus použil na sieť ohraničení na obr. 2.2, tak pri poradí premenných V_1, V_2, V_3 a V_4 by síce vylúčil hodnoty "žirafa" a "puška", ale sieť by nebola hranovo konzistentná (pretože v D_3 ostala hodnota "auto" a v D_4 zase hodnota "eso"). Pri opačnom poradí premenných by síce hodnoty "eso" a "auto" boli vylúčené, ale zase by boli ponechané hodnoty "žirafa" a "puška", takže zase by sa nedosiahla hranová konzistencia celej siete. Ak by bol však použitý silnejší variant, tak pri oboch poradiach premenných by boli vylúčené všetky štyri nekonzistentné hodnoty a bola by dosiahnutá hranová konzistencia.

Ani tento silnejší variant však negarantuje dosiahnutie hranovej konzistencie. Ak by na obr. 2.2 hodnota "žirafa" bola v doméne D_1 nahradená hodnotou "mravec" (s príslušnou úpravou ohraničení $C^{1,2}$ a $C^{1,4}$), tak pri poradí uvažovania premenných V_1, V_2, V_3 a V_4 by bola síce dosiahnutá hranová konzistencia celej siete ohraničení, ale pri opačnom poradí premenných by v doméne D_4 ostala nekonzistentná hodnota "eso" – pre dosiahnutie hranovej konzistencie by bol potrebný ešte ďalší prechod cyklom s opätovnou kontrolou domény D_4.

Napriek tomu, že oba varianty nie vždy dosiahnu hranovú konzistenciu, oba dokážu do určitej miery orezať priestor prehľadávania. Použitie prvého jednoduchšieho variantu spolu s dopřednou kontrolou sa označuje ako *čiastočný dopredný pohľad* (partial looking ahead – PLA) a použitie druhého variantu s dopřednou kontrolou zase ako *plný dopredný pohľad* (full looking ahead – FLA)[19].

Svojou schopnosťou orezávania sa niekde medzi týmito dvoma variantmi nachádza prípad dopredného pohľadu, keď sa využije smer kontroly premenných, využívaný pri smerovej hranovej konzistencii (directional arc consistency lookahead – DAC-L). Je to vlastne čiastočný dopredný pohľad, iba poradie kontrolovaných premenných je opačné – od konca k aktuálnej premennej.

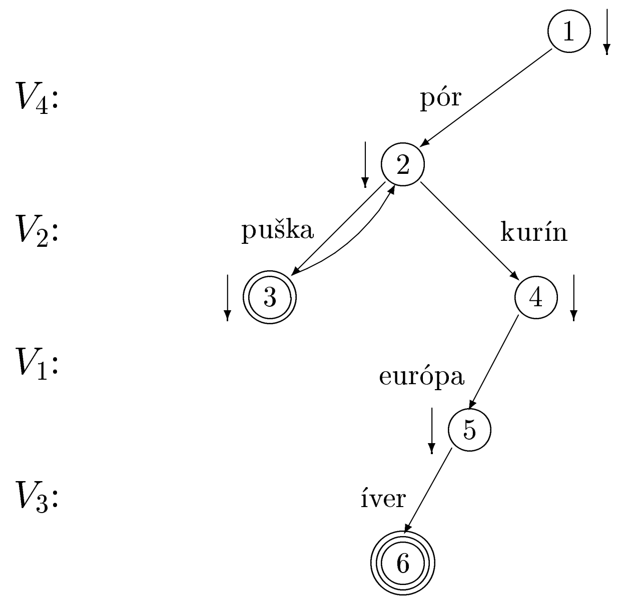
```
define method check_future3 ( i::<integer> )
   for ( j::<integer> from n-1 to i )
   ...
      for ( k::<integer> from j+1 to n )
         ...
   end for; end for;
end method check_future3;
```

Táto verzia vychádza z pojmu podpory. Ak nejaká hodnota má kompatibilnú hodnotu v doméne inej premennej, tak tá ju vlastne podporuje. Akonáhle nejaká hodnota stráca podporu, stáva sa redundantnou a je odstránená čo najskôr, aby zbytočne neposkytovala podporu nejakým iným hodnotám, ktoré by mohli byť tiež odstránené. A práve vďaka inému usporiadaniu kontroly premenných je možné hneď zohľadniť stratu podpory aj pre iné premenné.

Ešte väčší účinok možno dosiahnuť obojsmerným použitím, keď sa toto použije postupne v opačných smeroch (napr. najprv sa kontrolujú voľné premenné pomocou `check_future3` v nejakom usporiadaní a potom opäť v usporiadaní opačnom). Tento prípad reprezentuje algoritmus dopredného pohľadu s obojsmernou hranovou konzistenciou (bidirectional arc consistency lookahead – BDAC-L). V takomto usporiadaní účinok môže byť ešte väčší ako pri plnom dopřednom pohľade [125].

[19] Ak by sa použil algoritmus pre garantované zabezpečenie hranovej konzistencie, získali by sme *skutočne plný dopredný pohľad* (real full looking ahead).

Pre ilustráciu čiastočného dopredného pohľadu možno použiť príklad s dvoma riešeniami, ktorého sieť ohraničení je znázornená na obr. 4.2. Ak pre výber premenných sa použije usporiadanie V_4, V_2, V_1 a V_3 a pre zabezpečovanie konzistencie zase usporiadanie V_4, V_3, V_2 a V_1, tak výsledný strom prehľadávania má tvar podľa obr. 4.22.



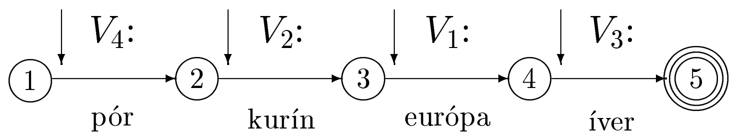
Obr. 4.22: Strom prehľadávania pre ilustračný príklad generovaný algoritmom čiastočného dopredného pohľadu.

V uzle 1 síce bola kontrolovaná hranová konzistencia, ale použité usporiadanie premenných spôsobilo, že nebola objavená nekonzistencia hodnôt "žirafa" a "puška" a preto tieto hodnoty naďalej ostali v doménach premenných. Celkovo domény po skončení kontroly obsahujú desať hodnôt.

Naproti tomu v uzle 2 už mohli byť domény redukované. Najprv vplyvom doprednej kontroly boli vylúčené hodnoty "žirafa" a "medveď" z domény D_1 a hodnota "auto" z D_3. Následne vplyvom kontroly neviazaných premenných navzájom bola hodnota "dekan" vypustená z D_2. Takáto veľká redukcia domén sa nemohla neprejaviť na veľkosti výsledného stromu prehľadávania, ktorý má iba 6 uzlov.

Samozrejme je možný aj taký prístup, ktorý skutočne garantuje dosiahnutie hranovej konzistencie v každom uzle stromu prehľadávania. Takým je napr. algoritmus udržiavania hranovej konzistencie (maintaining arc consistency – MAC) [111] ktorý stále pracuje s hranovo konzistentnou sieťou ohraničení. Ešte pred priradením hodnoty prvej premennej je zabezpečená plná hranová konzistencia. Následné priradenie hodnoty premennej znamená vlastne redukciu domény tejto premennej na jednu hodnotu. Po takomto priradení nasleduje opäť zabezpečenie hranovej konzistencie (ktoré môže byť veľmi efektívne, pretože vlastne ide iba o šírenie redukcie domény jednej premennej medzi doménami ostatných premenných). Efektívnosť je možné ešte zvýšiť využitím faktu, že nie je nutné ustanoviť plnú hranovú konzistenciu pomocou testovania všetkých hodnôt všetkých neviazaných premenných – je možné využiť princíp "lenivosti" [115] (viď algoritmus minimálnej doprednej kontroly v predchádzajúcej kapitole).

Pre rovnaký príklad podľa obr. 4.2 s usporiadaním premenných V_4, V_2, V_1 a V_3 pri použití algoritmu udržiavania hranovej konzistencie možno získať výsledný strom prehľadávania podľa obr. 4.23.

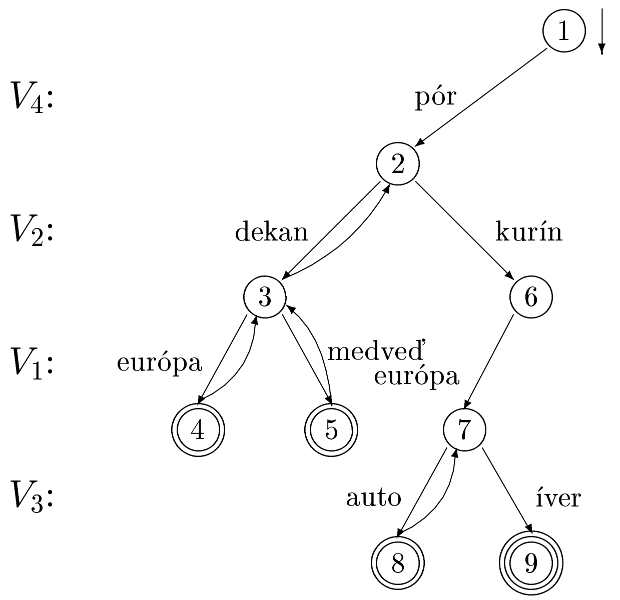


Obr. 4.23: Strom prehľadávania pre ilustračný príklad generovaný algoritmom udržiavania hranovej konzistencie.

Na rozdiel od použitia čiastočného dopredného pohľadu (obr. 4.22) sa zabezpečovanie skutočnej hranovej konzistencie prejaví v zmenšení výsledného stromu prehľadávania. Avšak cenou za to je zvýšenie výpočtových nárokov. V uzle 1 pre dosiahnutie hranovej konzistencie bolo nutné odstrániť hodnoty "žirafa" a "puška" z domén D_1 a D_2. Priradenie hodnoty "pór" premennej V_4 spôsobí nekonzistenciu hodnôt "medveď", "dekan" a "auto", ktoré museli byť vypustené v uzle 2.

Je možné šírenie ohraničení nepoužiť v každom uzle stromu prehľadávania, ale iba v niektorých. Typicky to je jeden uzol – koreňový. Pri tejto stratégii sa zabezpečí nejaká úroveň konzistencie premenných (čím sa priestor prehľadávania redukuje na prijateľnú mieru), a potom sa použije niektorý z prehľadávacích algoritmov. V tomto prípade sa však uprednostňujú tie algoritmy, ktoré garantujú dosiahnutie nejakého stupňa konzistencie – napr. hranovej alebo 3-konzistencie. Takáto stratégia sa nazýva *predspracovanie*.

Keďže pri predspracovaní sa zabezpečí k-konzistencia, pričom typicky $k < n$, tak strom prehľadávania môže mať aj slepé konce, pričom vo všeobecnosti ich má viac ako keby sa táto k-konzistencia zabezpečovala v každom uzle. Ak by sme opäť použili príklad s dvoma riešeniami ako vyššie (s tými istými usporiadaniami premenných a domén), tak pri predspracovaní zabezpečujúcom hranovú konzistenciu by výsledný strom prehľadávania mal tvar podľa obr. 4.24.



Obr. 4.24: Strom prehľadávania pre ilustračný príklad generovaný algoritmom spätného navracania s predspracovaním zabezpečujúcim hranovú konzistenciu.

Aj keď na rozdiel od obr. 4.22 teraz v koreňovom uzle bola väčšia redukcia domén, použitie samotného algoritmu spätného navracania vygenerovalo viac zbytočných uzlov.

Vo všeobecnosti platí, že ak pri predspracovaní je zabezpečený vyšší stupeň konzistencie, tak následné prehľadávanie bude mať menší nanajvýš rovnaký počet navracaní, ako keby bol pri predspracovaní zabezpečený iba nižší stupeň konzistencie. Pri dostatočne vysokom stupni prísnej konzistencie je možné garantovať, že prípadné navracanie bude ohraničené – postačí, ak sa zmenia hodnoty iba niekoľkých posledne naviazaných premenných [53].

Miera vhodného predspracovania je závislá od konkrétneho problému. Pre niektoré problémy zabezpečenie hranovej konzistencie oreže veľa hodnôt a pre niektoré problémy ani jednu (a je teda úplne zbytočné). Závisí to od hustoty ohraničení a ich voľnosti resp. prísnosti [21].

Pre tento účel sa v [126] definuje pojem *m-voľnosti* – binárne ohraničenie je m-voľné, ak každá hodnota z domény jednej premennej tohto binárneho ohraničenia je konzistentná aspoň s m hodnotami z domény druhej premennej daného ohraničenia. Sieť binárnych ohraničení je potom m-voľná, ak každé z ohraničení je m-voľné.

Potom ak sieť binárnych ohraničení je m-voľná a $\| D_i \|$ označuje veľkosť i-tej domény, tak táto sieť je

$$\left\lceil \frac{\max\limits_{i=1,\ldots,n} \| D_i \|}{(\max\limits_{i=1,\ldots,n} \| D_i \|) - m} \right\rceil \tag{4.4}$$

prísne konzistentná. Zabezpečovanie nižšieho stupňa konzistencie v predspracovaní nemá žiadny význam. Tak napr. sieť na obr. 2.2 je iba 0-voľná a teda má zmysel zabezpečovať akýkoľvek stupeň konzistencie.

4.4 Reštrukturalizačné algoritmy

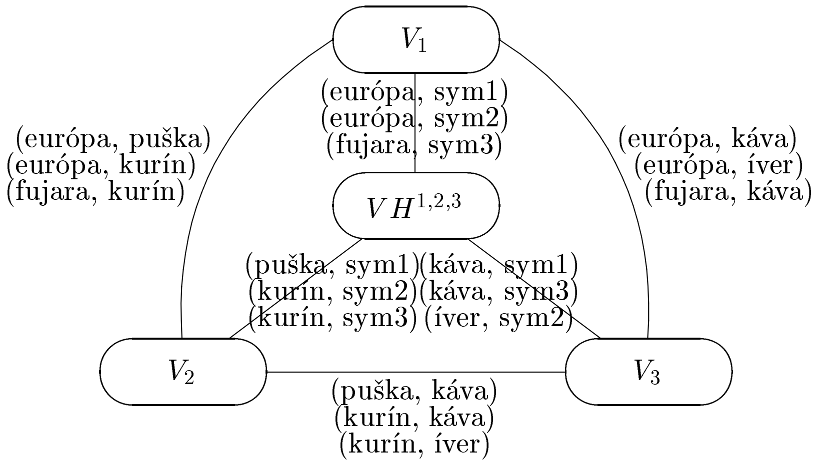
Niekedy je vhodné transformovať problém do inej podoby, čím sa získa sieť ohraničení iného tvaru. Príkladom takejto transformácie je eliminácia ohraničení vyšších árností (niektoré algoritmy predpokladajú prítomnosť iba binárnych ohraničení a nevedia manipulovať s ohraničeniami vyšších árností). Vychádza sa z faktu, že n-árne ohraničenie je možné nahradiť sieťou binárnych ohraničení [96]. Napríklad pre náhradu nejakého ohraničenia $C^{1,2,3}$, definovaného nad nejakými tromi premennými V_1, V_2 a V_3, ktoré pripúšťa iba dve trojice hodnôt

(európa, puška, káva)
(európa, kurín, íver)

by stačilo definovať tri nové ohraničenia $C^{1,2}$, $C^{2,3}$ a $C^{1,3}$, pričom každé z nich by pripúšťalo dve dvojice hodnôt vyskytujúce sa v $C^{1,2,3}$ (teda napr. $C^{1,2}$ by povoľovalo iba "európu" spolu s "puškou" alebo "kurínom"). No nie vždy je takýto postup postačujúci. Ak by $C^{1,2,3}$ obsahovalo tri možné kombinácie hodnôt

(európa, puška, káva)
(európa, kurín, íver)
(fujara, kurín, káva)

tak sieť ohraničení $C^{1,2}$, $C^{1,3}$ a $C^{2,3}$, vytvorených uvedeným postupom, by pripúšťala aj kombináciu ("európa", "kurín", "káva"), pričom táto kombinácia je zakázaná pôvodným ohraničením $C^{1,2,3}$. Vo všeobecnosti pri náhrade nejakého ohraničenia sieťou binárnych ohraničení je potrebné definovať nové pomocné premenné, ktoré netvoria súčasť riešenia. V uvedenom prípade by potom náhradná sieť ohraničení mohla mať tvar napríklad podľa obr. 4.25. V tomto prípade bola zavedená nová premenná $VH^{1,2,3}$ s tromi možnými hodnotami, ktoré tvoria kontext pre hodnoty ostatných premenných (hodnota $sym1$ reprezentuje prvú povolenú kombináciu v $C^{1,2,3}$ atď.).



Obr. 4.25: Sieť binárnych ohraničení.

Tieto novodefinované pomocné premenné sa označujú ako *h-premenné*. Sú to skryté premenné, ktoré netvoria súčasť hľadaného riešenia. Takúto h-premennú je potrebné vytvoriť pre každé viacárne ohraničenie pôvodného problému, ktoré je transformované do siete binárnych ohraničení. Doména h-premennej obsahuje jedinečný identifikátor pre každú kombináciu hodnôt, povolenú pôvodným viacárnym ohraničením. Pri náhrade sa vytvoria binárne ohraničenia medzi zavedenou h-premennou a každou z premenných, nad ktorými bolo nahrádzané ohraničenie definované. Tieto zavedené ohraničenia vlastne hovoria, v ktorých povolených kombináciách pôvodného ohraničenia (hodnoty z domény h-premennej) participuje tá-ktorá hodnota z domény druhej ohraničovanej premennej [8].

Ak je sieť ohraničení transformovaná na nejaký špeciálny tvar, potom je možné použiť špecializované algoritmy, ktorých výkonnosť pre daný tvar siete ohraničení je väčšia ako výkonnosť algoritmov určených pre všeobecný tvar problému s ohraničeniami. Typickým reprezentantom takéhoto špeciálneho tvaru je napríklad stromová štruktúra.

Jeden z možných prístupov je založený na využívaní *duálnej siete ohraničení* na rozdiel od v predchádzajúcich kapitolách používaného tvaru siete ohraničení, ktorý sa označuje ako *primárny*. Pri duálnej reprezentácii uzly siete (označované ako *c-premenné*) reprezentujú jednotlivé ohraničenia pôvodného problému. Dva uzly siete sú spojené hranou vtedy, ak množiny premenných, nad ktorými sú definované zodpovedajúce ohraničenia, majú neprázdny prienik – tie premenné, ktoré tvoria tento prienik, sa označujú ako *zdieľané* premenné[20]. Hrany duálnej siete ohraničení sú označené príslušnými zdieľanými premennými.

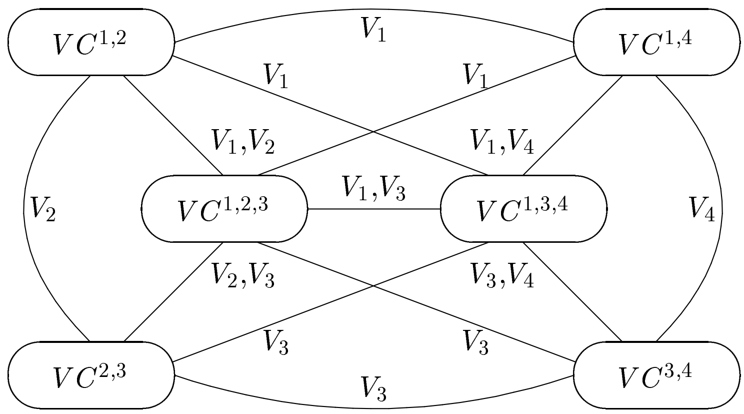
Reprezentáciou problému v takomto duálnom tvare je možné transformovať nebinárny problém na špeciálny binárny problém, v ktorom domény c-premenných sú tvorené k-ticami hodnôt[21] (k závisí od arity ohraničenia reprezentovaného danou c-premennou) a binárne ohraničenia majú charakter rovnosti (zdieľané premenné musia mať rovnakú hodnotu v oboch k-ticiach priradených tým c-premenným, ktoré sú spojené navzájom hranou).

Využitím duálneho princípu je možné problém z obr. 2.2 transformovať do tvaru siete ohraničení so štyrmi uzlami (reprezentujúcimi c-premenné $VC^{1,2}$, $VC^{2,3}$, $VC^{3,4}$ a $VC^{1,4}$) a štyrmi hranami, súvisiacimi so zdieľanými premennými V_1 (medzi c-premennými $VC^{1,2}$ a $VC^{1,4}$), V_2 ($VC^{1,2}$ a $VC^{2,3}$), V_3 ($VC^{2,3}$ a $VC^{3,4}$) a V_4 ($VC^{3,4}$ a $VC^{1,4}$).

[20]Nie je žiadna súvislosť medzi tu používaným pojmom a rovnakým pojmom známym v oblasti logického programovania.

[21]Tieto k-tice sú vytvárané ako kombinácie hodnôt z domén tých premenných, nad ktorými je definované dané ohraničenie, reprezentované nejakou c-premennou. Tieto kombinácie hodnôt musia byť povolené daným ohraničením.

Na prvý pohľad sa takouto transformáciou veľa nezískalo. Ak by sa však uvažovali napríklad aj ternárne ohraničenia $C^{1,2,3}$ a $C^{1,3,4}$ (ktoré možno jednoducho dodefinovať s tým, že každé pripúšťa všetky možné trojice hodnôt, ktoré možno vytvoriť kombinovaním hodnôt z domén príslušných premenných), potom by duálna sieť ohraničení mala tvar podľa obr. 4.26.



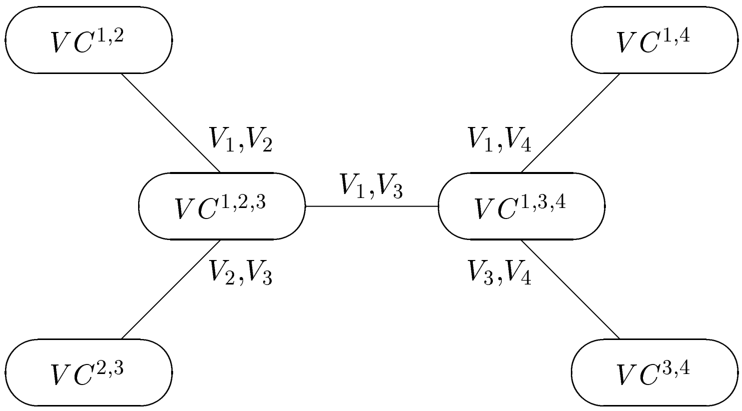
Obr. 4.26: Duálna sieť ohraničení.

Jednotlivé c-premenné reprezentované uzlami tejto duálnej siete ohraničení majú pritom nasledovné domény:

$VC^{1,2}$	:	(európa,puška), (európa,kurín)
$VC^{1,2,3}$	:	(žirafa,puška,auto) (žirafa,puška,íver) (žirafa,kurín,auto) (žirafa,kurín,íver) (európa,puška,auto) (európa,puška,íver) (európa,kurín,auto) (európa,kurín,íver)
$VC^{2,3}$	:	(kurín,íver)
$VC^{1,3,4}$	:	(žirafa,auto,pór) (žirafa,auto,eso) (žirafa,íver,pór) (žirafa,íver,eso) (európa,auto,pór) (európa,auto,eso) (európa,íver,pór) (európa,íver,eso)
$VC^{1,4}$	:	(európa,pór)
$VC^{3,4}$	:	(pór,íver),(eso,auto).

Teraz je možné získanú duálnu sieť ohraničení zjednodušiť vypustením *redundantných* hrán. Vo všeobecnosti ohraničenie je považované za redundantné, ak jeho eliminácia nezmení množinu riešení daného problému. Binárne ohraničenie medzi dvoma uzlami v duálnej sieti ohraničení je považované za redundantné, ak jeho zdieľané premenné sú zároveň zdieľanými premennými každej hrany nejakej alternatívnej cesty medzi tými istými uzlami. Tak napríklad na obr. 4.26 medzi uzlami $VC^{1,2}$ a $VC^{1,4}$ sa nachádzajúca hrana so zdieľanou premennou V_1 je redundantná, pretože premenná V_1 je zdieľanou premennou napr. pozdĺž alternatívnej cesty $VC^{1,2}$–$VC^{1,2,3}$–$VC^{1,3,4}$–$VC^{1,4}$.

Vypustením všetkých redundantných hrán je možné získať tvar duálnej siete ohraničení podľa obr. 4.27.



Obr. 4.27: Zjednodušená (neredundantná) duálna sieť ohraničení.

Takáto stromová štruktúra môže byť priamo riešená nejakým algoritmom profitujúcim z jej tvaru. Pre stromovú štruktúru je vždy možné zostaviť také usporiadanie premenných, ktorého šírka je 1. K tomu je postačujúce, ak v tomto usporiadaní sa rodičovský uzol nachádza pred svojimi potomkami. Keďže šírka takejto siete ohraničení je 1, tak pre zabezpečenie priraďovania hodnôt premenným bez nutnosti navracania je postačujúce zabezpečiť smerovú hranovú konzistenciu [52].

Ak by sa pre sieť na obr. 4.27 použilo napr. usporiadanie c-premenných $VC^{1,2}$, $VC^{1,2,3}$, $VC^{2,3}$, $VC^{1,3,4}$, $VC^{1,4}$ a $VC^{3,4}$, tak po zabezpečení smerovej hranovej konzistencie by domény jednotlivých c-premenných mali tvar:

$$
\begin{array}{lcl}
VC^{1,2} & : & (\text{európa,kurín}) \\
VC^{1,2,3} & : & (\text{európa,kurín,íver}) \\
VC^{2,3} & : & (\text{kurín,íver}) \\
VC^{1,3,4} & : & (\text{európa,íver,pór}) \\
VC^{1,4} & : & (\text{európa,pór}) \\
VC^{3,4} & : & (\text{pór,íver}),(\text{eso,auto}).
\end{array}
$$

Z takéhoto tvaru je už jednoduché zostaviť riešenie duálneho problému (akurát je potrebné zvoliť takú dvojicu pre $VC^{3,4}$, ktorá by mala rovnaké hodnoty premenných zdieľaných s $VC^{1,3,4}$) a z riešenia duálneho problému je potom možné jednoducho získať riešenie problému pôvodného.

Stromovú topológiu je možné využiť aj pre konštrukciu špeciálneho algoritmu pre priraďovanie hodnôt premenným. Ukážky takýchto špecializovaných prehľadávacích algoritmov pre riešenie stromových problémov je možné nájsť v [11]. Jednou z možností je algoritmus prechádzania stromom (tree tracker – TT):

```
define method solve( i::<integer> )
   if ( and ( 0 < i, i <= n ) )
      h := next( D[i], V[i] );
      if ( not ( h ) )
         k := find_parent ( i );
         D[k] := remove ( D[k], V[k] );
         solve ( k );
      else
         V[i] := h;
      end if;
      if ( satisfy_all_constraints ( ) )
         solve ( i+1 );
      else
         solve ( i );
      end if;
   end if;
end method solve;
```

Predpokladá sa usporiadanie premenných podľa prehľadávania do hĺbky (teda šírka použitého usporiadania bude 1). Akonáhle nie je možné naviazať nejakú premennú bez porušenia ohraničenia, na vine je hodnota priradená rodičovskému uzlu – a preto táto hodnota je z domény rodičovského uzla vypustená.

Pri použití pre strom na obr. 4.27 a to isté poradie uzlov ako v predošlom, má naviazanie hodnoty ("európa","puška") na c-premennú $VC^{1,2}$

za následok to, že po pokusoch priradiť hodnotu c-premennej $VC^{1,2,3}$ a následne c-premennej $VC^{2,3}$ budú z domény c-premennej $VC^{1,2,3}$ vypustené trojice ("európa", "puška","auto") a ("európa","puška","íver"). Po návrate k prvej c-premennej $VC^{1,2}$ už doména c-premennej $VC^{1,2,3}$ bude mať iba šesť hodnôt.

Je to vlastne akási kombinácia obyčajného spätného navracania s pamäťou – pamätá si hodnoty, ktoré po priradení vedú k navracaniu a v ďalšom hľadaní už tieto hodnoty nepoužije. Algoritmus síce generuje väčší počet uzlov v strome prehľadávania ako pri použití smerovej hranovej konzistencie, avšak potrebuje menší celkový počet kontrol splnenosti ohraničení.

Iným prístupom je priama redukcia primárnej siete ohraničení vypúšťaním redundantných častí. Centrálnu úlohu tu hrá pojem *redundancie.* Ten v literatúre možno najčastejšie nájsť definovaný rovnakým spôsobom ako pri duálnej reprezentácii – teda funkčne. Nejaká časť siete ohraničení je považovaná za redundantnú, ak jej odstránenie neovplyvní množinu riešení.

Podľa tejto definície napr. binárne ohraničenie medzi dvoma uzlami bude redundantné, ak neohraničuje možné kombinácie hodnôt daných premenných prísnejšie ako indukované (implicitné) binárne ohraničenie medzi tými istými premennými. Inak povedané, ak každá dvojica hodnôt dvoch premenných, ktorá je povolená nejakou množinou ohraničení (ktoré vytvárajú cestu v sieti ohraničení od jednej premennej k druhej premennej prechádzajúcu cez aspoň jeden ďalší uzol), je povolená aj priamym ohraničením medzi danými dvoma premennými, tak toto priame ohraničenie je redundantné [94]. Cieľom takéhoto prístupu je zjednodušiť sieť ohraničení do vhodného výsledného tvaru (napr. stromovej štruktúry).

Pri podrobnejšom skúmaní siete ohraničení podľa obr. 2.2 je možné zistiť, že napr. cesta tvorená ohraničeniami $C^{1,2}$, $C^{2,3}$ a $C^{3,4}$ indukuje ako jedinú prípustnú kombináciu pre ohraničenie $C^{1,4}$ dvojicu ("európa","pór") a teda explicitné vyjadrenie ohraničenia $C^{1,4}$ je možné v danej sieti ohraničení považovať za redundantné. Podobne je redundantným aj ohraničenie $C^{3,4}$ (ale nie spolu s $C^{1,4}$). Na druhej strane ohraničenie $C^{2,3}$ redundantné nie je, pretože je prísnejšie ako indukované ohraničenie medzi premennými V_2 a V_3 (toto pripúšťa aj dvojicu hodnôt ("puška","íver")).

Redundantnou časťou siete môže byť aj uzol – ak takýto uzol nereprezentuje takú premennú, ktorá je súčasťou hľadaného riešenia (také uzly sa označujú ako *skryté* a slúžia pre definovanie problému). Spolu s uzlom sú odstránené aj všetky ohraničenia viažúce sa na tento uzol [110].

Tvar siete ohraničení, ktorý je uvedený na obr. 4.27 je možné využiť aj inak. Jeho štruktúra je taká, že priradenie hodnoty nejakej c-premennej vlastne môže znamenať, že v zmysle riešenia pôvodného problému bola súčasne priradená hodnota aj inej c-premennej (resp. priradenie hodnoty tejto druhej c-premennej už neprispeje k dosiahnutiu riešenia pôvodného problému). Je to vtedy, ak druhá c-premenná reprezentuje ohraničenie definované nad podmnožinou tých premenných z množiny premenných, nad ktorými je definované ohraničenie zodpovedajúce prvej c-premennej. Príkladom je uzol $VC^{1,2,3}$ a uzly $VC^{1,2}$ a $VC^{2,3}$.

Ak by sa všetky uzly tohto druhého typu z duálnej siete ohraničení vypustili, získala by sa zjednodušená stromová štruktúra, ktorá už však nie je použiteľná na priame hľadanie riešenia – zo siete na obr. 4.27 by ostali iba dva uzly $VC^{1,2,3}$ a $VC^{1,3,4}$, ktoré však pripúšťajú všetky možné kombinácie hodnôt premenných, nad ktorými je definované ohraničenie reprezentované týmito c-premennými.

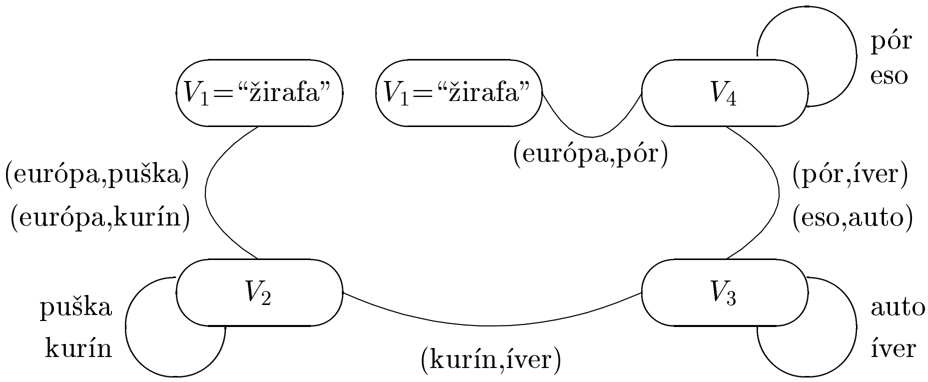
Takúto štruktúru je však možné chápať ako dekompozíciu pôvodného problému na jednoduchšie problémy. Pre uvádzaný prípad by potom stačilo najprv vyriešiť podproblém zahŕňajúci iba premenné V_1, V_2 a V_3 a po jeho vyriešení zase ďalší podproblém, zahŕňajúci premenné V_1, V_3 a V_4. Na tomto princípe je založený algoritmus uvedený v [42] a [43], ktorý dekomponuje pôvodný problém na podproblémy (dané podmnožinami premenných primárneho problému) tak, aby výsledná štruktúra podproblémov mala stromový charakter. Vhodným usporiadaním týchto podproblémov sa dosiahne, že celý problém je riešiteľný takým spôsobom, že navracanie sa môže vyskytnúť iba v rámci riešenia jednotlivých podproblémov. Hľadanie riešenia nejakého podproblému nebude vyžadovať návrat k predchádzajúcemu podproblému.

Iný spôsob dekompozície stavia na fakte, že priradenie hodnoty nejakej premennej môže mať za následok vlastne zmenu štruktúry siete ohraničení [36]. Po priradení hodnoty premennej V_1 z obr. 2.2 sa štruktúra siete zmení na štruktúru uvedenú na obr. 4.28. Tým, že premennej V_1 bola priradená konkrétna hodnota, sa cyklická štruktúra zmenila na lineárnu – priradenie hodnoty malo za následok "rozseknutie" existujúceho cyklu.

Premenné, ktoré ešte nemajú priradenú žiadnu hodnotu vytvárajú stromovú štruktúru a to môže byť využité pre ďalšie riešenie. Takýto spôsob dekompozície je možné využiť dvojako [36]:

- špeciálny algoritmus pre takýto typ dekompozície

    1. nájsť minimálnu množinu *deliacich* premenných, po naviazaní ktorých zvyšná časť premenných vytvára stromovú štruktúru



Obr. 4.28: Modifikovaná sieť ohraničení po priradení hodnoty jednej premennej.

2. vyriešiť priradenie konzistentných hodnôt týmto premenným
3. zvyšnú časť problému riešiť s prihliadnutím na stromovú štruktúru.

- zahrnúť túto stratégiu do prehľadávacieho algoritmu

V prvom type je možná tiež existencia navracania – napr. naviazanie množiny deliacich premenných je také, že zvyšné premenné nie je možné konzistentne naviazať. Pri druhom type je potrebné neustále kontrolovať, či zvyšné nenaviazané premenné nevytvárajú stromovú štruktúru. Ak áno, tak sa prepne zo všeobecného prehľadávacieho algoritmu na efektívny algoritmus pre riešenie sietí ohraničení stromovej topológie.

Iné typy dekompozície môžu byť založené napríklad na obojstranne viazaných komponentoch [53] alebo na využití hypergrafov [66].

Časť III

Logické programovanie ohraničení

Kapitola 5

Základné princípy

Už názov *logické programovanie ohraničení* (constraint logic programming – CLP) sa snaží vyjadriť dve základné zložky tejto technológie – *logické programovanie* (logic programming – LP) a *riešenie ohraničení* (constraint solving – CS).

Logické programovanie [18] používa logiku na vyjadrenie problému a dedukciu na jeho riešenie. Výsledkom je deklaratívny programovací jazyk, kde v princípe programátorovi stačí vyjadriť *čo* sa má urobiť bez toho aby sa musel starať o to *ako* sa má tento cieľ dosiahnuť. Tento prístup je opačný v porovnaní s klasickými procedurálnymi programovacími jazykmi ako sú napr. pascal alebo C. Typickým predstaviteľom skupiny logických programovacích jazykov je prolog (prvú implementáciu vytvoril roku 1972 v Marseille Alain Colmerauer).

To však, čo je z programátorského hľadiska výhodou, sa ukázalo pri riešení reálnych úloh ako určité obmedzenie. Logické programovanie totiž narába iba s objektmi ktoré sú neinterpretované štruktúry (tzv. termy) (t.j. symboly bez vlastného významu) a zhoda medzi nimi sa realizuje jedine syntakticky, a to algoritmom nazývaným *unifikácia.* Každý sémantický objekt musí byť takto explicitne zakódovaný v podobe termov. Číselná rovnosť *5 = 2 + 3* bude takto napríklad unifikačným algoritmom zamietnutá, nakoľko ľavá a pravá strana rovnosti sú syntakticky rozdielne (porovnáva sa konštanta 5 so štruktúrou +(2,3)), napriek tomu že sa sémanticky zhodujú.

Druhým problémom je práve jednotné, ale pomerne jednoduché výpočtové pravidlo, a síce prehľadávanie do hĺbky, ktorého výsledkom je stratégia "generuj a testuj" s dobre známymi problémami s výkonnosťou pri kombinatoricky náročných úlohách.

Oba tieto problémy pomáha preklenúť integrácia metód na riešenie ohraničení. Ohraničenia totiž predstavujú implicitné vyjadrenie vzťahov medzi sémantickými objektmi z nejakej výpočtovej oblasti (napr. prirodzené alebo reálne čísla). Táto špecifikácia umožňuje využitie efektívnejších algoritmov, tak povediac šitých na mieru tej ktorej výpočtovej domény. Tento koncept teoreticky sformulovali Joxan Jaffar a Jean-Louis Lassez v [77]. Jeho základnou myšlienkou je nahradiť unifikáciu ako základnú operáciu v logickom programovaní všeobecnejším procesom riešenia ohraničení na vopred špecifikovanej výpočtovej doméne $\mathcal{X}$. Tento koncept, označovaný CLP($\mathcal{X}$), bol následne realizovaný na doméne reálnych čísel v programovacom jazyku nazývanom CLP($\mathcal{R}$) [71].

Avšak v tej istej dobe sa objavili v Európe prologovsky orientované programovacie jazyky ohraničení ako CHIP [47] a Prolog III [29]. CHIP predstavil okrem domény reálnych čísel aj doménu boolovských premenných a tzv. konečné domény (premenné môžu nadobúdať hodnotu z konečnej množiny vopred definovaných hodnôt). Doménou systému Prolog III boli zoznamy. Tieto systémy boli vybudované na podstatne odlišných prístupoch než systém CLP($\mathcal{R}$).

Pre definíciu programovacieho jazyka je nevyhnutné definovať jeho syntax a sémantiku. Špecifikom logických programovacích jazykov je, že ich možno čítať deklaratívne (deklaratívne vyjadrenie úlohy, t.j. čo sa má dosiahnuť), ale aj procedurálne (t.j. ako bude tento program pochopený a vykonaný zabudovaným deduktívnym mechanizmom). Túto vlastnosť samozrejme preberajú aj jazyky CLP.

CLP systém je determinovaný ohraničujúcou doménou ($\mathcal{X}$) a detailnou procedurálnou sémantikou.

5.1 Ohraničujúce domény CLP jazykov

Trieda CLP jazykov je parametrizovaná ohraničujúcou doménou $\mathcal{X}$. Pre konkrétnu inštanciu CLP jazyka, označovanú CLP($\mathcal{X}$), parameter $\mathcal{X}$ je charakterizovaný štvoricou $(\Sigma, \mathcal{L}, \mathcal{D}, \mathcal{T})$, kde:

Σ je *abeceda* [32] (signature) určujúca vopred definovanú množinu použitých symbolov, t.j. konštánt, predikátových a funkčných symbolov a ich arity,

$\mathcal{L}$ je *predikátový jazyk* (t.j. množina všetkých ohraničení, ktoré je možné vytvoriť predpísaným spôsobom nad Σ),

$\mathcal{D}$ je *interpretačná štruktúra* [32] (t.j. štruktúra nad ktorou má prebiehať výpočet, teda interpretácia predikátového jazyka $\mathcal{L}$)

$\mathcal{T}$ je *teória s jazykom* $\mathcal{L}$ (ľubovoľná množina formúl $\mathcal{T} \subseteq \mathcal{L}$, tzv. axióm, ktorých pravdivosť je daná definitoricky. Ide o axiomatizáciu niektorých vlastností $\mathcal{D}$).

Ohraničujúca doména je potom definovaná ako dvojica $(\mathcal{L}, \mathcal{D})$. V praxi sú najčastejšie používané dve ohraničujúce domény: tzv. konečné domény a aritmetika nad reálnymi (racionálnymi) číslami.

Oba príklady spĺňajú nasledovné podmienky [78]:

- Termy a ohraničenia v $\mathcal{L}$ vyhovujú podmienkam predikátovej logiky prvého rádu. To znamená, že je dovolená kvantifikácia premenných, nie však predikátov a funkcií. Napríklad v predikátovej logike prvého rádu nemôže byť vyjadrené tvrdenie "Všetky predikáty majú iba jeden argument". Logika prvého rádu je neprotirečivá (sound - nemožno dokázať nepravdivé tvrdenie) a úplná (complete - pre každé pravdivé tvrdenie existuje dôkaz) [32].

- Binárny predikátový symbol $=$ je obsiahnutý v Σ a je interpretovaný ako identita na $\mathcal{D}$.

- V $\mathcal{L}$ existujú ohraničenia, ktoré sú identicky pravdivé, resp. identicky nepravdivé na $\mathcal{D}$. Napr. $w = w$, kde w je konštanta z Σ je vždy identicky pravdivé ohraničenie.

- Trieda ohraničení $\mathcal{L}$ je uzavretá pod premenovaním premenných, konjunkciou a existenčnou kvantifikáciou.

Konečné domény je možné v duchu predchádzajúcej definície popísať takto:

Σ : $\{\{\in [u,v]\}_{u \leq v}, +, -, *, =, \neq, \leq\}$ pričom $+$, $-$ a $*$ sú binárne funkčné symboly a $=, \neq, \leq$ sú binárne predikátové symboly. Pre každú dvojicu prirodzených čísel u, v platí, že existuje také V_i, pre ktoré platí: $V_i \in [u, v]$. To znamená, že $u \leq V_i \leq v$. Ostatné symboly majú svoj obvyklý význam.

$\mathcal{L}$: ohraničenia generované elementárnymi ohraničeniami, obmedzené tak že každá premenná vystupujúca v nejakom ohraničení c je predmetom intervalového ohraničenia. Typické ohraničenia: $V_i \in [1, 7] \wedge V_j \in [0, 5]$

$\wedge V_i \neq 5 \wedge V_i + V_j \leq 5 \wedge 2 * V_i - V_j < 3$. Výsledná doména premennej V_i je $\{1, 2, 3\}$.

$\mathcal{D}$: množina celých čísel $\mathcal{Z}$.

$\mathcal{T}$: zahŕňa také axiómy, ako napríklad komutatívnosť a asociatívnosť operácie $+$, neutrálnosť nuly voči $+$, ale aj axiómy o usporiadanosti prvkov podľa veľkosti.

Potom $\mathcal{FD} = (\mathcal{D}, \mathcal{L})$ je ohraničujúca doména označovaná ako konečné domény.

Príklad krížovky uvedený v prvej časti tejto knihy možno tiež zahrnúť pod konečné domény, ale v tomto prípade abeceda $\Sigma_{krížovka} = \{$žirafa, európa, puška, kurín, auto, íver, pór, eso, $=, \in, \Leftrightarrow, \vee\}$ kde $\Leftrightarrow$ predstavuje binárny predikátový symbol kompatibility dvojice konštánt z Σ a $\vee$ binárny predikátový symbol pre logickú disjunkciu.

$\mathcal{L}$ je množina unárnych ohraničení daných vymenovaním prvkov z Σ (pre každý riadok resp. stĺpec ide o množinu slov danej dĺžky) a binárnych ohraničení kompatibility nasledovného tvaru ($V_i \in W_i \vee V_j \in W_j$ kde V_i, V_j sú premenné a W_i, W_j sú podmnožiny prípustných konštánt z Σ (dané spoločným písmenom na pozícii, kde sa daný riadok a stĺpec pretínajú).

$\mathcal{D}$ je daná definíciou krížovky, t.j. podľa umiestnenia slov do príslušných riadkov (stĺpcov). $\mathcal{T}$ obsahuje napríklad axiómu: $\forall V_i, V_j, i \neq j : V_i = w \Leftrightarrow V_j \in W \Rightarrow w \notin W$. T.j. to isté slovo sa v krížovke nesmie vyskytovať viackrát.

Aritmetika nad reálnymi (racionálnymi) číslami je charakterizovaná nasledovne:

Σ : $\{0, 1, +, -, *, =, <, \leq\}$, kde 0 a 1 sú konštanty, $+$, $-$ a $*$ binárne funkčné symboly a $=, <, \leq$ binárne predikátové symboly.

$\mathcal{L}$: ohraničenia generované elementárnymi ohraničeniami. Typické ohraničenia: $\exists V_i, V_j$: $5 * V_i + 3 * V_j < 5 \wedge V_i * V_i \leq V_j$

$\mathcal{D}$: množina reálnych čísiel $\Re$. $\mathcal{D}$ interpretuje symboly z Σ tak ako je to v aritmetike obvyklé.

$\mathcal{T}$: okrem axióm platných pre $\mathcal{FD}$ sem patrí aj distributívnosť a neutrálnosť jednotky voči $*$.

Potom $\mathcal{R} = (\mathcal{L}, \mathcal{D})$ je ohraničujúca doména aritmetiky nad reálnymi číslami. Existujú však aj ďalšie varianty tejto ohraničujúcej domény, ako napr.:

- ak v $\mathcal{L}$ pripustíme použitie funkčného symbolu $*$ len medzi premennou a konštantou, nie medzi dvoma premennými, potom výsledná ohraničujúca doména $\mathcal{R}_{lin}$ je ohraničujúcou doménou lineárnej aritmetiky nad reálnymi číslami (nepripúšťa ohraničenia typu $V_i * V_i < V_j$).
- Ak je doména navyše obmedzená len na racionálne čísla, potom nová výsledná ohraničujúca doména je $\mathcal{Q}_{lin}$.
- Ak jediným binárnym predikátovým symbolom v Σ bude $=$ (t.j. povolené ohraničenia sú len rovnosti), dostaneme ohraničujúcu doménu $\mathcal{R}_{lineqn}$ resp. $\mathcal{Q}_{lineqn}$ pre reálne, resp. racionálne čísla.
- Ak rozšírime $\mathcal{L}$ a pripustíme aj negované rovnosti (t.j. predikátový symbol $\neq$), potom výsledná ohraničujúca doména bude $\mathcal{R}_{linneq}$ resp. $\mathcal{Q}_{linneq}$ pre reálne, resp. racionálne čísla. Tieto ohraničujúce domény povoľujú ohraničenia typu $V_i + 2V_j < 2 \wedge V_i \neq V_j$.

Pre príklad numerickej doplňovačky z prvej časti knihy nám stačí definovať $\Sigma = \{0, 1, +, =, <, \}$, s významom ako je uvedené vyššie. $\mathcal{L}$ sú ohraničenia generované elementárnymi ohraničeniami. Typické ohraničenia: $\exists V_i, V_j : V_i < u \wedge v < V_j \wedge V_i + u + V_j = v$ kde $u, v \in \Re\}$. $\mathcal{D}$ a $\mathcal{T}$ sú rovnaké ako v prípade všeobecnej ohraničujúcej domény $\mathcal{R}$.

5.2 Syntax a sémantika

Program je v logickom programovaní aj v logickom programovaní ohraničení tvorený množinou klauzúl. Klauzula v logickom programovaní vyzerá nasledovne:

$$H \leftarrow B_1, \ldots, B_i.$$

kde $H, B_1, \ldots, B_i$ sú termy. Pričom *term* má tvar $p(t_1, \ldots, t_j)$, kde p je j-árny funktor (postupnosť znakov začínajúca malým písmenom) a $t_1, \ldots, t_j$ sú termy. Vo všeobecnosti *term* môže byť konštanta (číslo alebo atóm[1]), premenná (postupnosť znakov začínajúca veľkým písmenom alebo znakom "_") alebo štruktúra. Štruktúra s nulovým počtom argumentov sa nazýva *atóm* (zodpovedá približne textovým konštantám procedurálnych jazykov) a v logických programoch začína malými písmenami.

Deklaratívne táto klauzula znamená implikáciu

[1]Pozor, v (C)LP sa pojem atóm používa odlišne než v matematickej logike.

$$H \text{ platí vtedy, ak platí } B_1 \wedge \ldots \wedge B_i.$$

pričom všetky prípadné premenné v tomto výraze sú univerzálne kvantifikované.

Cieľom nazývame klauzulu bez hlavy, t.j.

$$\leftarrow B_1, \ldots, B_i.$$

Napríklad hlavný predikát programu na riešenie úlohy krížovka definovanej v prvej časti knihy by mohol vyzerať takto:

```
krizovka(V1,V2,V3,V4) :-
        slovo(V1,6), slovo(V2,5),
        slovo(V3,4), slovo(V4,3),
        kompatibilne(V1,2,V2,2), kompatibilne(V2,4,V3,1),
        kompatibilne(V1,5,V4,1), kompatibilne(V3,4,V4,3).
```

Pričom predikát `slovo(S,Dlzka)` generuje slová `S` zadanej dĺžky `Dlzka` a predikát `kompatibilne(S1,P1,S2,P2)` testuje zhodu písmen v slove `S1` na pozícii `P1` a v slove `S2` na pozícii `P2`.

Cieľ ktorý vedie k nájdeniu riešenia krížovky by vyzeral v rámci dialógu s LP systémom nasledovne:

```
krizovka(V1,V2,V3,V4).
```

V CLP programe sú naviac možné aj klauzule tvaru:

$$H \leftarrow c_1, \ldots, c_j, B_1, \ldots, B_i.$$

kde $c_1, \ldots, c_j$ sú ohraničenia.

Ohraničenia majú syntakticky takú podobu ako termy, ale na rozdiel od klasických termov musí ku každému ohraničeniu existovať nástroj na jeho riešenie. V klasickom logickom programe ide vlastne len o unifikáciu ako jediný typ "ohraničenia".

Analogicky *cieľom* nazývame klauzulu bez hlavy, t.j.

$$\leftarrow c_1, \ldots, c_j, B_1, \ldots, B_i.$$

Riešenie príkladu doplňovačky (program aj cieľ pre jeho aktivovanie) pomocou knižnice konečných domén jazyka ECL^iPS^e (teda za pomoci CLP) možno nájsť v 6.1.1.

Odpoveď na zadaný cieľ je v CLP konjunkcia ohraničení s výbornou vlastnosťou, že všetky ich riešenia (t.j. všetky priradenia hodnôt ich premenným ktoré spĺňajú tieto ohraničenia) sú tiež riešeniami pôvodného cieľa. Originálnosť tohoto vyjadrenia je v tom, že umožňuje konečnú reprezentáciu aj nekonečného počtu riešení (napr. výsledná odpoveď v tvare $V_i = 1, V_j > 3$ pre doménu reálnych čísel zahrňuje nekonečne mnoho prípustných hodnôt V_j, spĺňajúcich zadaný cieľ).

Procedurálna sémantika je definovaná pomocou stavového priestoru ako systém prechodov medzi stavmi. Stav je pritom definovaný usporiadanou trojicou $< A, C, G >$, kde:

A je množina termov a ohraničení ktoré je potrebné spracovať,

C je množina aktívnych ohraničení (t.j. tých, ktoré bude daný systém riešenia ohraničení riešiť v nasledujúcom prechode),

G je množina pasívnych (alebo uspaných) ohraničení (t.j. ohraničenia ktoré nemôžu byť spracované hneď alebo ktorých spracovanie by neprinieslo žiaden pokrok).

Okrem toho ešte existuje špeciálny stav označovaný ako *neúspech* (fail) ktorý zodpovedá situácii kedy riešenie zadanej množiny atómov a ohraničení neexistuje.

Táto definícia svojou všeobecnosťou pokrýva správanie sa väčšiny najznámejších CLP systémov. Aby bolo možné jednoznačne definovať procedurálnu sémantiku konkrétneho CLP systému, je potrebné podrobnejšie definovať:

1. *Výpočtové pravidlo*, ktoré v danom kroku vyberá typ prechodu a tiež vhodný prvok A (ak je to nutné) v každom stave.

2. Možné *prechody*.

Prechody môžu byť vo všeobecnosti nasledovných typov:

$< A \cup B_k, C, G > \rightarrow_r < A \cup B^{1,\ldots,i}, C, G \cup (B_k = H) >$ – ak je výpočtovým pravidlom vybraná B_k, pričom B_k je term, $H \leftarrow B^{1,\ldots,i}$ je pravidlo z CLP programu premenované na nové premenné podľa B_k. H a B_k majú rovnaký funktor a aritu. Výraz $B_k = H$ znamená konjunkciu rovností medzi zodpovedajúcimi argumentmi B_k a H.

$< A \cup B_k, C, G > \rightarrow_r$ *fail* – ak je výpočtovým pravidlom vybraná B_k, pričom B_k je term, pre každé pravidlo $H \leftarrow B^{1,\ldots,i}$ z CLP programu, H a B_k majú rôzne funktory, alebo rôznu aritu.

$< A \cup c, C, G > \rightarrow_c < A, C \cup c, G >$ – ak c je ohraničenie vybrané výpočtovým pravidlom.

$< A, C, G > \rightarrow_i < A, C', G' >$ – ak $(C', G') =$ *infer(C, G)*

$< A, C, G > \rightarrow_s < A, C, G >$ – ak *consistent(C)*.

$< A, C, G > \rightarrow_s fail$ – ak $\neg$*consistent(C)*.

Existujú teda 4 typy prechodov:

$\rightarrow_r$	vyplýva z tradičnej rezolúcie v logickom programe,
$\rightarrow_c$	zavádza do aktuálnej množiny ohraničení nové ohraničenie,
$\rightarrow_s$	testuje, či aktívne ohraničenia sú konzistentné
$\rightarrow_i$	odvodzuje nové aktívne ohraničenia a (prípadne) modifikuje existujúce pasívne ohraničenia.

Z vyššie popísanej procedurálnej sémantiky vyplýva, že ak chceme definovať konkrétnu inštanciu CLP jazyka, musíme okrem výpočtového pravidla a možných prechodov definovať naviac aj:

- predikát *consistent(C)* a
- funkciu *infer(C, G)*.

Predikát *consistent(C)* je test konzistencie C. Tento môže byť definovaný ako úplný test konzistencie (typický pre nekonečné domény) t.j. zabezpečí globálnu konzistenciu C po každom jeho spustení. Ale mnoho reálnych CLP systémov má len neúplný (čiastočný) test konzistencie. Ide najmä o systémy pre konečné domény. Napr. systém CAL, ktorý pracuje s doménou reálnych čísel, konzistenciu testuje na doméne komplexných čísel.

Funkcia *infer(C, G)* odvodí z aktuálnej množiny ohraničení novú množinu aktívnych ohraničení C' a pasívnych ohraničení G'. Pritom sa vyžaduje, aby platilo že konjunkcia $C' \wedge G'$ je splniteľná vtedy a len vtedy ak bola splniteľná aj konjunkcia $C \wedge G$. Prolog napríklad nepozná pasívne ohraničenia, takže preň platí: *infer(C, G)* $= (C \cup G, \emptyset)$.

Táto formulácia signalizuje, že existuje široká paleta implementácií CLP systémov. V CLP($\mathcal{R}$) [71] sú nelineárne ohraničenia pasívne a *infer* jednoducho presúva (linearizovanú verziu) takýchto ohraničení z G do C' keď sa dané nelineárne ohraničenie stane lineárnym v kontexte C a vymaže takéto ohraničenie z G. Napríklad ak

$$G: V_i * V_j = V_k \wedge V_k * V_j = 2$$
$$C: V_i = 4 \wedge V_k \leq 0,$$

potom *infer(C, G)* $= (C', G')$ kde

$$G': V_k * V_j = 2$$
$$C': V_i = 4 \wedge V_k \leq 0 \wedge 4 * V_j = V_k.$$

T.j. aktívne ohraničenie $(V_i = 4) \in C$ po dosadení do G spôsobilo že z nelineárneho ohraničenia $(V_i * V_j = V_k) \in G$ sa stalo lineárne a preto ho bolo možné presunúť medzi aktívne ohraničenia, t.j. $(4 * V_j = V_k) \in C'$.

V CLP jazykoch ako je CHIP alebo ECL^iPS^e funkcia *infer* vykonáva menej zrejmé operácie založené na istom stupni hranovej konzistencie berúc do úvahy len hranice domén. Napríklad na doméne $\mathcal{FD}$ ak

$$G: V_i = V_j + 1$$
$$C: 2 \leq V_i \leq 5 \wedge 0 \leq V_j \leq 3,$$

potom *infer(C, G)* $= (C', G')$ kde

$$G': G$$
$$C': 2 \leq V_i \leq 4 \wedge 1 \leq V_j \leq 3.$$

Keďže podľa ohraničenia z G je V_i o jednotku väčšie ako V_j a zároveň podľa druhého ohraničenia z C môže V_j nadobúdať najviac hodnotu 3, potom V_i nemôže byť väčšie ako 4 (preto je zodpovedajúco upravené prvé ohraničenie z C). Podobne podľa ohraničenia z G je V_j o jednotku menšie ako V_i a zároveň podľa prvého ohraničenia z C môže V_i nadobúdať minimálne hodnotu 2, potom V_j nemôže byť menšie ako 1 (preto je zodpovedajúco upravené druhé ohraničenie z C).

BNR-Prolog ako aj *ria* knižnica v ECL^iPS^e sú CLP jazyky využívajúce intervalovú aritmetiku nad reálnymi číslami. Intervaly sú aktívne ohraničenia a ostatné ohraničenia sú pasívne. Funkcia *infer* potom opakovane vypočítava menšie intervaly pre každú z premenných na základe ohraničení v G až do chvíle, keď už nie je možné odvodiť žiadne ďalšie zúženie intervalu niektorej premennej.

5.3 Základné technológie implementácie CLP

Formálna špecifikácia triedy CLP jazykov z predchádzajúcej kapitoly dáva možnosť využiť pre jednotlivé ohraničujúce domény $\mathcal{X}$ efektívne algoritmy na riešenie ohraničení tak povediac šité na mieru tej-ktorej ohraničujúcej doméne (často veľmi dobre sa hodiacej na určitú reálnu aplikačnú oblasť).

Najdôležitejšou novinkou CLP jazykov oproti LP je ich schopnosť spracovávať ohraničenia. Na rozdiel od logického programovania je potrebné v CLP naviac v každom kroku zabezpečovať konzistentnosť aktuálnej množiny ohraničení. V zásade sú možné dva prístupy [91].

Prvým z nich je použiť efektívny inkrementálny algoritmus, ktorý je schopný zabezpečiť konzistentnosť riešením ohraničení a ich uchovávaním v nejakej efektívnej podobe (tzv. normálna forma). Jeho inkrementálnosť spočíva v tom, že ak v niektorom kroku pribudnú nové ohraničenia, algoritmus je schopný zabezpečiť konzistentnosť tejto rozšírenej množiny ohraničení s menšími nákladmi (t.j. napr. bez toho, aby musel prebehnúť celý výpočet odznova). Inak povedané, riešenie je upravené tak, aby okrem existujúcej množiny ohraničení spĺňalo aj v danom kroku novo zadané ohraničenia. Typickým príkladom je simplexový algoritmus, resp. jeho upravené varianty, používané na doméne reálnych (racionálnych) čísel pre riešenie sústavy lineárnych rovníc a nerovníc. Preto sa táto technológia nazýva *inkrementálne lineárne riešenie.*

Dôležité je si uvedomiť, že za najdôležitejšie kritériá pre výber algoritmu sú považované:

- dobré priemerné správanie sa algoritmu v priemernom prípade (čo je dôležitejšie ako jeho zložitosť v najhoršom prípade),
- inkrementálnosť (v zmysle výpočtovej náročnosti).

Kvôli tomuto cieľu je udržovaná tzv. *vyriešená forma* reprezentácie pre splnené ohraničenia. Ide o takú formu reprezentácie ohraničení, v ktorej je ich splniteľnosť evidentná. *Výsledková forma* reprezentácie ohraničení je potom taká ich vyriešená forma z ktorej je evidentná aj množina všetkých riešení daných ohraničení.

Pre ohraničujúcu doménu $\mathcal{R}_{lineqn}$ je najpriamejším algoritmom Gaussova eliminačná metóda (v najhoršom prípade kvadratická zložitosť). Pre všeobecnejšiu ohraničujúcu doménu $\mathcal{R}_{lin}$ sú známe polynomiálne algoritmy ale tie sa nepoužívajú v praktických CLP systémoch. Namiesto toho sa ako základ využíva simplexový algoritmus napriek jeho exponenciálnej zložitosti v najhoršom prípade, nakoľko má lepšie priemerné správanie.

Rozdiely medzi rôznymi implementáciami riešiacich algoritmov na báze simplexovej metódy v praktických CLP systémoch sú v špecifických realizáciách tohoto algoritmu. Napríklad systém CLP($\mathcal{R}$) používa pohyblivú rádovú čiarku na reprezentáciu reálnych čísel. Naproti tomu systémy CHIP alebo Prolog III používajú úplnú presnosť racionálnych čísiel (ohraničujúca doména je preto $\mathcal{Q}$). Iný príklad: CLP($\mathcal{R}$) spracúva rovnice a nerovnice oddelene využívajúc rýchly Gaussov eliminačný algoritmus pre rovnice, pri stálych stratách v dôsledku komunikácie medzi oboma časťami CLP systému.

Keďže simplexový algoritmus pracuje pre nezáporné čísla a neostré nerovnice, v CLP systémoch musí byť rozšírený. V $\mathcal{R}_{lin}$ sú lineárne rovnice často reprezentované v parametrizovanej podobe, t.j. $V_1 = le_1(V^{i,\dots,j})$, $\cdots$, $V_n = le_n(V^{i,\dots,j})$ kde le_i označuje lineárny výraz obsahujúci premenné z $V^{i,\dots,j}$ ale žiadna premenná V_k $(k = 1, \dots, n)$ na ľavej strane rovnice sa nevyskytuje na pravej strane rovnice v lineárnom výraze $le_k(V^{i,\dots,j})$. Pre $\mathcal{R}_{lin}$ simplexový algoritmus reprezentuje ohraničenia vo forme matice $u \times v$ ako $\tilde{U}V^{1,\dots,n} = \tilde{V}$ kde $\tilde{U}$ obsahuje identickú podmaticu rozmeru $n \times n$ definujúcu bázické premenné a všetky čísla v stĺpci vektora $\tilde{V}$ sú nezáporné.

Pri implementovaní CLP systémov pre ohraničujúcu doménu $\mathcal{R}_{linneq}$ vzniká problém pretože simplexový algoritmus nie je schopný riešiť ohraničenia v podobe nerovností. Neformálne povedané problém spočíva v detekovaní viazaných premenných (skrytých konštánt), t.j. premenných ohraničených tak, že môžu nadobúdať iba jednu hodnotu. Jeden pokus vyriešiť tento problém efektívne možno nájsť v [76]. Je tam prezentovaná syntaktická vyriešená forma ohraničení zabraňujúca výskytu spomínaných skrytých konštánt. Naviac je v tejto práci navrhnutý inkrementálny algoritmus, ktorý využíva špeciálne vlastnosti novej vyriešenej formy. Tento algoritmus je implementovaný v systéme CLP($\mathcal{R}$).

Druhou možnosťou je neposkytnúť síce úplné riešenie v každom kroku (čo by bolo časovo veľmi náročné), ale uspokojiť sa s určitým rizikom nekonzistentnosti (t.j. zabezpečiť len nejaký nižší stupeň konzistencie, najčastejšie hranovú konzistenciu) v priebehu riešenia, profitujúc z neskorších zjednodušení ohraničení (napr. naviazaním premenných v priebehu výpočtu). To je práve prípad redukčných techník ktoré boli popísané ako súčasť kombinovaných algoritmov (viď kap. 4.3.2). Tieto techniky sa v literatúre o CLP nazývajú často techniky *lokálneho šírenia ohraničení* a používajú sa najmä pre konečné domény. Preto sa tento prístup zvykne označovať ako *doménová technológia.*

Algoritmy na riešenie ohraničení sa tu aktivujú prostredníctvom zmien v doménach premenných, ktorých sa toto ohraničenie týka. Pri tomto prístupe sa najskôr šírením ohraničení značne zredukujú pôvodné domény premenných. Už samotné šírenie ohraničení vedie často k takému zredukovaniu domén jednotlivých premenných, ktoré s dostatočnou presnosťou popisuje riešenia úlohy, prípadne nájde jediné existujúce riešenie.

Vo väčšine prípadov je však potrebné odštartovať prehľadávanie tak, že sa jednotlivým premenným postupne priradzujú hodnoty z ich redukovaných domén. Pritom každé takéto priradenie môže vyvolať ďalšie zmeny v doménach niektorých iných premenných. To vyvoláva ďalšie šírenie prostredníctvom definovaných ohraničení aj na ostatné premenné, čo vedie v konečnom dôsledku k postupnej redukcii priestoru prehľadávania. Pre tento proces je charakteristická aktívna úloha definovaných ohraničení.

Aby sme vyriešili množinu ohraničení C obsahujúcich množinu premenných $V_1, \ldots, V_n$ ktoré nadobúdajú hodnoty z konečných domén D obsahujúcich po u prvkoch, môžme postupne skúšať každú možnú n-ticu hodnôt premenných $(V_1, \ldots, V_n)$ čo je spolu u^n možných n-tíc. Všeobecný algoritmus ktorý toto robí (viac alebo menej inteligentne) je nejaká podoba prehľadávania so spätným navracaním (BT), kde sa v každom okamihu v určitom poradí naviaže jedna premenná na hodnotu zo svojej domény a k spätnému navracaniu dôjde vtedy, ak nejaká premenná nemôže byť konzistentným spôsobom naviazaná.

Algoritmus takéhoto typu má v najhoršom prípade exponenciálnu zložitosť vzhľadom na počet premenných vystupujúcich v ohraničeniach. Jeden spôsob ako vylepšiť toto správanie je použitie techník (algoritmov) lokálneho šírenia ohraničení, napr. pre zabezpečenie nejakého stupňa konzistencie. Tieto techniky sa nedívajú na celú sieť ohraničení, ale len lokálne na jej malú podmnožinu.

V niektorých špeciálnych prípadoch takéto techniky lokálneho šírenia ohraničení môžu nájsť presné riešenie problému bez nutnosti prehľadávania. Ale vo všeobecnosti je u konečných domén stále potrebný neúplný mechanizmus na riešenie ohraničení, t.j. množina ohraničení tu nie je reprezentovaná vo vyriešenej forme. Táto množina môže dovoľovať niektoré kombinácie hodnôt, ktoré nie sú (globálne) konzistentné, keďže nemohli byť odhalené algoritmom lokálneho šírenia ohraničení. Preto je nevyhnutná prehľadávacia procedúra pre nájdenie presného riešenia (t.j. n-tice globálne konzistentných hodnôt premenných).

Kapitola 6

Jazyk ECL^iPS^e

ECL^iPS^e (ECL^iPS^e Common Logic Programming System) [3] je systém založený na prologu, ktorého hlavným cieľom je slúžiť ako platforma integrujúca rôzne rozšírenia logického programovania, najmä logické programovanie ohraničení. Jadro ECL^iPS^e je efektívnou implementáciou štandardného prologu (edinburgského typu) ako je popísaný v základných knihách o prologu [25]. Systém je postavený na inkrementálnom kompilátore ktorý kompiluje zdrojové texty v ECL^iPS^e do kódu podobného WAM (Warenov abstraktný stroj) [4] a emulátore tohoto abstraktného kódu.

ECL^iPS^e bol pôvodne vyvinutý v Európskom výskumnom ústave počítačového priemyslu (European Computer-Industry Research Centre - ECRC) v Mníchove a je ďalej vyvíjaný a podporovaný v IC-Parc (Centrum pre plánovanie a riadenie zdrojov pri Imperial College v Londýne) s podporou ICL a Európskej únie v rámci ESPRIT projektu CHIC-2.[1]

ECL^iPS^e možno nainštalovať v niekoľkých konfiguráciách na rôznych platformách (vrátane Linuxu a 32-bitových verzií operačného systému MS Windows). Aktivuje sa potom jedným z nasledujúcich príkazov:

- *eclipse* – základná varianta ECL^iPS^e s používateľským rozhraním vo forme príkazového riadku.

- *tkeclipse* – ECL^iPS^e s grafickým vývojárskym rozhraním.

[1]Najnovšie informácie o tomto systéme možno získať na web stránke systému ECL^iPS^e: *http://www.icparc.ic.ac.uk/eclipse* alebo zaslaním elektronickej pošty na adresu *eclipse-request@icparc.ic.ac.uk*. Existuje aj diskusná skupina používateľov ECL^iPS^e. Príspevky do nej možno posielať na adresu *eclipse-users@icparc.ic.ac.uk* a žiadosti o prihlásenie resp. odhlásenie do/z diskusnej skupiny na adresu *majordomo@icparc.ic.ac.uk*.

- *peclipse* – paralelný variant ECL^iPS^e.

Po aktivácii (v príkazovom riadku) sa systém prihlási nasledovnou hlavičkou:

```
% eclipse
ECLiPSe Constraint Logic Programming System [kernel]
Copyright Imperial College London and ICL
Certain libraries copyright Parc Technologies Ltd
GMP library copyright Free Software Foundation
Version 5.0.0, Thu Jun 22 15:42 2000
[eclipse 1]:
```

Hranaté zátvorky obsahujú zoznam (v predchádzajúcom jednoprvkový "kernel") udávajúci aktuálnu konfiguráciu bežiaceho systému, t.j. rozšírenia systému, ktoré sú v systéme momentálne zavedené. Po hlavičke nasleduje výzva `[eclipse 1]:`, ktorý používateľovi napovedá, že systém čaká v cykle na najvyššej úrovni (top level) na cieľ od používateľa. Názov aktuálneho modulu je *eclipse*.

V ďalšom budeme pri vysvetľovaní predikátov používať zaužívané označovanie predikátov v tvare *funktor/arita*, t.j. názov predikátu (funktor) a počet jeho argumentov (arita). Pri vysvetľovaní významu predikátov bude názov každého argumentu začínať jedným z nasledujúcich znakov:

- + bude označovať vstupný argument, t.j. taký, ktorý musí byť v čase volania naviazaný,
- - bude označovať výstupný argument, t.j. taký, ktorý nesmie byť naviazaný v čase volania predikátu,
- ? bude označovať vstupný alebo výstupný argument, t.j. taký, ktorý môže, ale nemusí byť v čase volania naviazaný.

Predikát *help*/0 podáva všeobecnú nápoveď o interakcii so systémom a *help*/1 slúži na vyvolanie nápovede o konkrétnom zabudovanom predikáte. Práca so systémom sa ukončí volaním predikátu *halt*/0. Na konci každého cieľa je nutné napísať bodku tak, ako je to typické v logickom programovaní.

Program uložený v súbore (viacerých súboroch) je možné nakompilovať do aktuálneho modulu pomocou predikátu *compile*/1, pričom argumentom je názov súboru, resp. zoznam s názvami súborov.[2]

[2]existuje aj predikát *compile*/2 pre kompiláciu do zadaného modulu.

```
[eclipse 1]: compile('file1.pl').
file1.pl  compiled traceable 2396 bytes in 0.00 seconds

yes.
```

Ak má súbor(y) s programom príponu *.pl* alebo *.ecl*, stačí uviesť jeho názov (ich názvy oddelené čiarkami) bez prípony v hranatých zátvorkách. Napríklad:

```
[eclipse 2]: [file1].
file1.pl  compiled traceable 2396 bytes in 0.00 seconds

yes.
[eclipse 3]: [file2,file3,file4].
file2.pl       compiled traceable 796 bytes in 0.00 seconds
file3.pl       compiled traceable 3152 bytes in 0.00 seconds
file4.pl       compiled traceable 2396 bytes in 0.00 seconds

yes.
```

Samotný výpočet sa spúšťa volaním príslušného predikátu, napr. ak je v našom programe definovaný predikát $krizovka/1$ na riešenie doplňovačky, potom sa jeho výpočet aktivuje nasledovne:

```
[eclipse 4]: krizovka(L).

L = [europa, kurin, iver, por]     More? (;)

no (more) solution.
```

Odpoveďou systému je naviazanie premennej L na nájdené riešenie a ponuka hľadať ďalšie riešenie vo forme otázky `More? (;)`. Ak používateľ chce pokračovať v hľadaní ďalšieho riešenia, stlačí "`;`" ináč ľubovoľný iný kláves. V tomto prípade viac riešení neexistuje, preto po stlačení "`;`" je odpoveďou systému informácia `no (more) solution.` V prípade stlačenia ľubovoľného iného klávesu by bolo odpoveďou `yes.`

Systém ECL^iPS^e je možné aktivovať aj s viacerými zaujímavými prepínačmi [3] ktoré umožňujú napríklad okamžité nakompilovanie zvoleného súboru po spustení ECL^iPS^e, alebo napr. aj okamžité volanie zadaného cieľa po aktivovaní ECL^iPS^e. Tak napr. kombináciou týchto prepínačov možno

ECL^iPS^e využiť v dávkovom režime a pomocou skriptových súborov riadiť aj zložitejšie a rozsiahle testy (výpočty) bez nutnosti priameho riadenia používateľom. Výstup je pritom možné samozrejme presmerovať do súboru.

Veľmi príjemná je práca v režime *tkeclipse*, t.j. s grafickým vývojárskym prostredím (viď. obr. 6.1) ktorý poskytuje viacero okien, menu a tlačidiel na interakciu používateľa so systémom. Toto prostredie pozostáva z dvoch hlavných komponent:

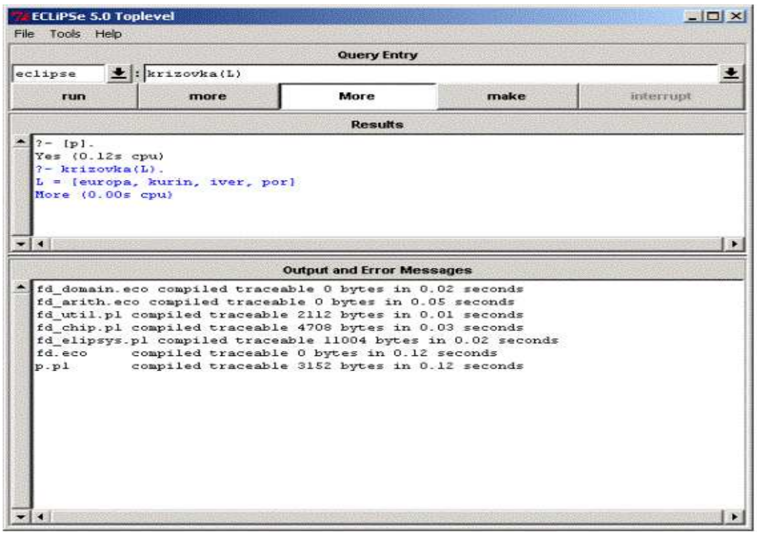
- Grafická verzia systému (top-level) pre komunikáciu používateľa so systémom. Ide o grafické okno s poľom pre výber aktuálneho modulu (vľavo hore), hneď za ním je pole pre vstup cieľov zadaných používateľom (prvý riadok zhora), niekoľko tlačidiel zodpovedajúcich základným operáciám pre riadenie výpočtu. Systém má výstup rozdelený do dvoch polí, v strednej časti okna sa zobrazujú výsledky a v spodnej časti okna výstupné a chybové správy.

- Súprava vývojárskych nástrojov pre podporu vývoja (aj rozsiahlych) aplikácií v ECL^iPS^e. Ide o položku menu Nástroje (tools), ktorá obsahuje napríklad nástroje na správu aktuálnych súborov s programom, krokovanie výpočtu, sledovanie štatistík o priebehu výpočtu, zmeny globálnych nastavení, zobrazovanie premenných a ohraničení pri behu CLP programov a mnohé ďalšie možnosti.

Jazyk ECL^iPS^e poskytuje mnoho zaujímavých rozšírení už v rámci samotného prologu. Napríklad systém modulov užitočný najmä pri vývoji veľkých aplikácií, rôzne predikáty pre programovania cyklov, polia a globálne premenné, podpora paralelizmu (v konfigurácii *peclipse* - nie je dostupné na všetkých platformách), makrá, ošetrovanie udalostí a prerušení, bohaté ladiace nástroje a ďalšie. Podrobnejšie sa o nich možno dozvedieť v manuáli používateľa [3].

Jedným z rozšírení je aj možnosť lepšieho zápisu štruktúr v ECL^iPS^e (nie je to povinné, je možné používať štruktúry v klasickom prologovskom zápise) umožňujúca pomenovať jednotlivé argumenty a odkazovať sa potom na ne cez tieto mená. Zdrojové texty sú potom čitateľnejšie a ľahšie sa upravujú, bez negatívneho vplyvu na efektívnosť, nakoľko sú implementované vo forme makier. Štruktúru s pomenovanými argumentami je možné deklarovať definovaním predlohy (template) príkazom *struct*/1. Príkladom môže byť nasledovná štruktúra s názvom kniha, ktorá má 4 argumenty.

```
:- local struct( kniha(autor, nazov, rok, vydavatel) ).
```



Obr. 6.1: $TkECL^iPS^e$ - grafická verzia systému ECL^iPS^e.

Deklarácia *local* znamená, že štruktúra bude známa len v rámci aktuálneho modulu. V prípade potreby globálnej platnosti štruktúry sa použije deklarácia *global*.

Štruktúry s názvom *kniha*/4 môžu byť potom ďalej zapisované takýmto spôsobom: `kniha with []` alebo `kniha with rok:2000` a podobne.

Takýto zápis vnútri programu bude makrom expandovaný do definitívnej podoby nasledovným spôsobom:

```
[eclipse 1]: local struct(kniha(autor,nazov,rok,vydavatel)).

yes.
[eclipse 2]: X = kniha with [].

X = kniha(_170, _171, _172, _173)
yes.
[eclipse 3]: Y = kniha with rok:2000.

Y = kniha(_173, _175, 2000, _176)
yes.
```

```
[eclipse 4]: Y = kniha with nazov:'Úlohy s ohraničeniami'.

Y = kniha(_173, 'Úlohy s ohraničeniami', _175, _176)
yes.
[eclipse 5]: Z = kniha with [nazov:'Úlohy s ohraničeniami',
 autor:'M.Mach, J.Paralič', rok:2000, vydavatel:elfa].

Z = kniha('M.Mach, J.Paralič', 'Úlohy s ohraničeniami',
 2000, elfa)
yes.
```

Poradové číslo argumentu takto deklarovanej štruktúry si potom nie je nutné pamätať, pretože ho možno získať použitím zápisu termu v tvare `NazovPola of NazovStruktury`. Napr. pre prístup k argumentu *rok* vyššie definovanej štruktúry *kniha*/4 možno použiť zabudovaný predikát *arg*/3 nasledovným spôsobom: `arg(rok of kniha, B, Y)`. Argument B je vstupný a musí to byť štruktúra typu *kniha*/4. Argument Y je výstupný a naviaže sa naň hodnota argumentu *rok* štruktúry B. Pritom tento zápis bude makrom najskôr prevedený do vnútornej formy `arg(3, B, Y)`.

```
[eclipse 6]: Z = kniha with [nazov:'Úlohy s ohraničeniami',
autor:'M.Mach, J.Paralič', rok:2000, vydavatel:elfa],
arg(rok of kniha, Z, Rok).

Z = kniha('M.Mach, J.Paralič', 'Úlohy s ohraničeniami',
 2000, elfa)
Rok = 2000
yes.
```

Štruktúry môžu byť deklarované tak, že ich argumentmi sú opäť štruktúry. V takom prípade takáto vnútorná štruktúra dedí štruktúru mien argumentov podľa svojej definície. Majme nasledujúci program uložený v súbore *struct.ecl*:

```
:- local struct(osoba(meno,adresa,vek)).
:- local struct(zamestnanec(o:osoba,plat)).

jano(Zam,Meno,Vek,Plat,Adresa,Osoba) :-
        Zam = zamestnanec with
              [meno:jano,vek:32,plat:10000,adresa:tu],
```

```
        arg(meno of zamestnanec, Zam, Meno),
        arg(vek of zamestnanec, Zam, Vek),
        arg(plat of zamestnanec, Zam, Plat),
        arg(adresa of zamestnanec, Zam, Adresa),
        arg(o of zamestnanec, Zam, Osoba).
```

Štruktúra *zamestnanec*/2 obsahuje argument *o* ktorý je štruktúrou typu *osoba*/3. Mená argumentov štruktúry *osoba* možno teraz používať, akoby boli súčasťou štruktúry *zamestnanec*. Pri použití syntaxe pre zápis štruktúr pomocou operátorov *with* a *of* sa možno na štruktúru *zamestnanec* pozerať ako na vnorenú, alebo ako na plochú podľa toho, čo je v danej situácii vhodnejšie.

```
[eclipse 7]: [struct].
struct.ecl compiled traceable 400 bytes in 0.00 seconds

yes.
[eclipse 8]: jano(Z,M,V,P,A,O).

Z = zamestnanec(osoba(jano, tu, 32), 10000)
M = jano
V = 32
P = 10000
A = tu
O = osoba(jano, tu, 32)
yes.
[eclipse 9]: Zam = zamestnanec with [meno:marian,plat:12000].

Zam = zamestnanec(osoba(marian, _191, _192), 12000)
yes.
```

Z pohľadu CLP najdôležitejším rozšírením je *atribútovaná premenná* (nazývaná tiež metaterm). Na tomto základe je vybudovaný napríklad aj veľmi dôležitý mechanizmus "corouting" (uspávania a budenia cieľov - viď. knižnice *suspend_simple.pl* a *suspend.pl*). Keď sa ECL^iPS^e prepne do tohoto módu príkazom `coroutine` tak ciele ktoré v čase volania nemôžu byť vyhodnotené kvôli tomu, že nejaká premenná je neviazaná, budú automaticky uspané až do okamihu, keď sa táto premenná naviaže. V tom okamihu budú automaticky opäť vyhodnotené.

Napríklad v prvej z nasledujúcich ukážok došlo k chybe kvôli tomu, že premenná X nebola v čase volania naviazaná, a teda cieľ $X > 3$ nemohol byť vyhodnotený (ako v klasickom prologu). Ak však ECL^iPS^e prepneme do módu uspávania a budenia cieľov (druhá ukážka), to isté volanie v tretej ukážke už neskončí chybovou hláškou, ale informáciou o pozastavení vyhodnotenia cieľa $X > 3$, ktorý sa týmto spôsobom uspal.

```
[eclipse 1]: X > 3.
instantiation fault in X > 3
[eclipse 2]: coroutine.

yes.
[eclipse 3]: X > 3.

X = X

Delayed goals:
        X > 3
yes.
[eclipse 4]: X > 3, X = 4.

X = 4
yes.
[eclipse 5]: X > 3, X = 2.

no (more) solution.
```

Posledné dve ukážky potom demonštrujú správanie sa uspaného cieľa po jeho zobudení v prípade neskoršieho naviazania premennej X. V štvrtej ukážke naviazanie premennej X na hodnotu 4 spôsobí úspešné vyhodnotenie zobudeného cieľa $X > 3$ a výpočet končí úspechom. Naproti tomu v piatej ukážke po unifikácii premennej X s dvojkou zobudený cieľ $X > 3$ neuspeje a s ním aj celé volanie.

6.1 Konečné domény

Knižnica *fd.pl* [22] implementuje ohraničenia na konečných doménach ktorých prvky môžu byť celočíselné, atomické (napr. atómy, reťazce, čísla v pohyblivej rádovej čiarke, atď.) a zložené termy bez výskytu premenných v nich (napr. *f(a, b)*). Knižnicu je možné nakompilovať príkazom `lib(fd)`.

Doménová premenná je atribútovaná premenná s atribútmi definovanými v knižnici konečných domén. Doménová premenná môže byť naviazaná len na hodnotu z danej konečnej množiny. Unifikácia s termom mimo tejto množiny neuspeje. Doména môže byť asociovaná s premennou použitím predikátu (operátora) `::/2` v tvare $?Vars$ `::` $?Domain$, kde $?Vars$ je premenná alebo zoznam premenných s asociovanou doménou $?Domain$. Táto môže byť tvorená intervalom celých čísel $Min..Max$ alebo ako zoznam takýchto intervalov a/alebo atómov resp. termov bez výskytu premenných. Napríklad:

```
[eclipse 6]: lib(fd).
...

yes.
[eclipse 7]: X::1..10.

X = X{[1..10]}
yes.
[eclipse 8]: Y :: [biela, cierna].

Y = Y{[biela, cierna]}
yes.
[eclipse 9]: Z :: [1..3,6..8].

Z = Z{[1..3, 6..8]}
yes.
[eclipse 10]: U::[biela,farba(cierna),1.23].

U = U{[1.23, biela, farba(cierna)]}
yes.
[eclipse 11]: [A,B,C] :: [1..3,5,7..9].

A = A{[1..3, 5, 7..9]}
B = B{[1..3, 5, 7..9]}
C = C{[1..3, 5, 7..9]}
yes.
```

Doménové premenné môžu vystupovať v troch typoch ohraničení: aritmetických, logických a symbolických. Doménová premenná sa nazýva *celočíselná* ak jej doména obsahuje len celé čísla. Len celočíselné doménové premenné sú akceptované v aritmetických ohraničeniach definujúcich relácie usporiadanosti a v racionálnych termoch (viď. nižšie).

Lineárny term je lineárna celočíselná kombinácia celočíselných doménových premenných. Predikáty definujúce ohraničenia akceptujú lineárne termy aj v nekanonickej podobe, akceptujúc operátory $+, -, *$, napr. $5 * (3 + (4 - 6) * Y - X * 3)$. Ak ohraničujúci predikát zistí premennú ktorá nemá doménu, priradí jej preddefinovanú doménu $-10000000..10000000$. Pozor, aritmetické operácie nad lineárnymi termami sa vykonávajú s celými číslami s jednoduchou presnosťou bez testovania pretečenia !

Racionálny term je term skonštruovaný z celých čísel a celočíselných doménových premenných použitím aritmetických operácií $+, -, *, /$. Každý podvýraz vo forme $VarA/VarB$ musí mať ako výsledok celé číslo. Systém nahrádza takýto podvýraz novou premennou X a pridá nové ohraničenie `VarA #= VarB * X`. Podobne všetky podvýrazy vo forme $VarA * VarB$ sú nahradené novou premennou X a novým ohraničením `X #= VarA * VarB`, takže vnútorná reprezentácia je stále vo forme lineárneho termu.

Úplný zoznam *aritmetických ohraničení* možno nájsť v prílohe V. Napríklad aritmetické ohraničenie na rovnosť `?T1 #= ?T2` znamená že hodnota racionálneho termu `T1` je rovná hodnote racionálneho termu `T2`.

Logické ohraničenia možno použiť na kombinovanie jednoduchších ohraničení do zložitejších logických ohraničujúcich výrazov[3]. Príkladom logického ohraničenia môže byť logická disjunkcia `+E1 #\/ +E2`, ktorá znamená že aspoň jeden z ohraničujúcich výrazov `E1` a `E2` je pravdivý. Akonáhle sa jeden z výrazov `E1` alebo `E2` stane nepravdivý, druhý z dvojice je zadaný. Zoznam a popis všetkých zabudovaných logických ohraničení v ECL^iPS^e je uvedený v prílohe V.

Okrem aritmetických a logických ohraničení sú v knižnici konečných domén zabudované ešte aj *vyhodnocovacie ohraničenia*, ktoré asociujú pravdivostnú hodnotu nejakého ohraničenia s booleovskou premennou. Táto môže potom vystupovať v ďalších zložitejších ohraničeniach (meta-ohraničeniach). Napríklad v prípade rovnosti ide o vyhodnocovacie ohraničenie `#=(?T1, ?T2, ?B)`, ktoré znamená že `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` je rovná hodnote racionálneho termu `T2`, ináč `B` sa rovná 0. Zoznam ostatných vyhodnocovacích ohraničení možno opäť nájsť v prílohe V.

[3]Tieto ohraničenia sú systémom predspracované a transformované na postupnosť vyhodnocovacích ohraničení (viď. ďalej) a aritmetických ohraničení.

Existuje ešte ďalšia dôležitá skupina ohraničení zabudovaných v knižnici konečných domén ECL^iPS^e. Ide o tzv. *symbolické ohraničenia* ktoré vyjadrujú nejaké zložitejšie vzťahy medzi doménovými premennými. V ďalšej časti tejto kapitoly budú pri riešení úloh použité nasledovné dve:

- `alldistinct(?List)` – Všetky prvky zoznamu *List* (doménové premenné a termy bez výskytu premenných) sú po dvojiciach rôzne.

- `element(?Index, +List, ?Value)` – *Index*-ty prvok zoznamu *List* termov bez výskytu premenných má hodnotu *Value*. *Index* a *Value* môžu byť alebo obyčajné premenné, potom sa im asociuje doména, alebo doménové premenné. Kedykoľvek sa zmení doména *Index* alebo *Value*, predikát sa zobudí a upraví zodpovedajúco domény oboch premenných.

Okrem týchto je ešte niekoľko ďalších uvedených v prílohe V.

Pre definovanie vlastných ohraničení je nevyhnutné poznať niektoré základné informácie o implementácii knižnice konečných domén v ECL^iPS^e (*fd.pl*). Každá doménová premenná má atribút *fd*, ktorý je definovaný práve v knižnici *fd.pl* nasledovne:

```
fd with [domain:D, min:Mi, max:Ma, any:A]
```

Tento atribút ukladá informáciu o aktuálnej doméne **domain** a tri rôzne zoznamy pozastavených (uspaných) cieľov, a síce:

- **min** – pozastavené ciele ktoré sa majú zobudiť pri zmene dolnej hranice domény danej premennej,

- **max** – pozastavené ciele ktoré sa majú zobudiť pri zmene hornej hranice domény danej premennej,

- **any** – pozastavené ciele ktoré sa majú zobudiť keď sa doména danej premennej zmenší, nezáleží pritom ako (stačí ak ubudne nejaký prvok v strede domény).

Mená týchto zoznamov pozastavených cieľov možno použiť v predikáte *suspend*/3 pri uspávaní cieľa pre stanovenie vhodnej podmienky budenia.

Okrem preddefinovaných ohraničení vo forme predikátov v knižnici konečných domén si môže používateľ definovať svoje vlastné ohraničenia a ich správanie. Na to je potrebné poznať ďalšiu skupinu predikátov z knižnice konečných domén, ktoré pristupujú priamo k doménam premenných resp.

s nimi narábajú. Tie predikáty, ktoré bude potrebné použiť v ilustračných príkladoch, budú vysvetlené na príslušnom mieste ako komentár[4] priamo v kóde programu.

Postup vytvárania používateľom definovaného ohraničenia možno ilustrovať na príklade definície nového ohraničenia $X^2 + Y^2 \leq C$, kde X a Y sú doménové premenné a C je celé číslo. Cieľom je dosiahnuť také správanie, aby sa predikát (ohraničenie) zobudil pri zmene domény niektorej zo zainteresovaných premenných a upravil podľa možnosti doménu druhej premennej. Pre tento účel možno definovať predikát $sq(?X, ?Y, +C)$ ktorý toto ohraničenie implementuje:[5]

```
:- lib(fd).

sq(A, B, C) :-    % A*A + B*B <= C
    dvar_domain(A, DomA),
        % zisti doménu premennej A a ulož ju do DomA
    dvar_domain(B, DomB),
        % zisti doménu premennej B a ulož ju do DomB
    dom_range(DomA, MinA, MaxA),
        % zisti dolnú a hornú hranicu domény DomA
    dom_range(DomB, MinB, MaxB),
        % zisti dolnú a hornú hranicu domény DomB
    orezavanie(MinA,B,MaxB,C,NewMaxB2),
        % uprav hornú hranicu domény premennej B
    orezavanie(MinB,A,MaxA,C,NewMaxA2),
        % uprav hornú hranicu domény premennej A
    (NewMaxA2 + NewMaxB2 =< C ->
            % ak je súčet štvorcov nových horných
            % hraníc premenných A a B menší alebo rovný C
        true
             % potom je ohraničenie splnené
    ;        % ináč
        suspend(sq(A, B, C), 3, (A, B)->min)
            % uspi toto ohraničenie (s prioritou 3) tak, aby
```

[4]Komentár sa v ECL^iPS^e označuje znakom % s platnosťou od výskytu znaku do konca riadku.

[5]v programe je použitý v ECL^iPS^e definovaný operátor $->$. Jeho význam je nasledovný: výraz ($Podmienka -> A1; A2$) realizuje vetvenie typu "Ak je splnená *Podmienka*, vykonaj $A1$, ináč $A2$."

```
    ),      % sa budilo pri zmene dolnej hranice domény
            % premennej A alebo B
    wake.   % spusti proces šírenia ohraničení

orezavanie(MinX,Y,MaxY,C) :-
    MinX2 is MinX*MinX,
    MaxY2 is MaxY*MaxY,
    (MinX2 + MaxY2 > C ->
            % zisti, či horná hranica Y je konzistentná
        NewMaxY is fix(sqrt(C - MinX2)),
            % ak nie, vypočítaj novú hodnotu hornej hranice Y
        dvar_remove_greater(Y, NewMaxY)
            % odstráň z domény premennej Y hodnoty
            % väčšie ako NewMaxY
    ;       % ináč
        NewMaxY = MaxY % sa horná hranica domény Y nemení
    ),
    NewMaxY2 is NewMaxY*NewMaxY.
```

Správanie takto definovaného ohraničenia možno pochopiť z niekoľkých nasledovných ukážok. V prvej ohraničenie $sq/3$ spôsobilo redukciu domén oboch premenných a bolo opätovne uspané. V druhej ukážke ohraničenie $sq/3$ zistilo, že pri aktuálnych doménach premenných X a Y neexistuje riešenie, ktoré by ho spĺňalo a výpočet preto skončil neúspechom. Konečne v tretej ukážke bolo ohraničením $sq/3$ nájdené za daných podmienok jediné prípustné riešenie.

```
[eclipse 12]: [sq].
sq.pl      compiled traceable 3228 bytes in 0.01 seconds

yes.
[eclipse 13]: [X,Y] :: 1..10, sq(X,Y,20).

X = X{[1..4]}
Y = Y{[1..4]}

Delayed goals:
        sq(X{[1..4]}, Y{[1..4]}, 20)
yes.
```

```
[eclipse 14]: X :: 1..10, Y :: 5..15, sq(X,Y,25).

no (more) solution.
[eclipse 15]: X :: 1..10, Y :: 5..15, sq(X,Y,26).

X = 1
Y = 5
yes.
```

6.1.1 Schéma riešenia úloh s konečnými doménami

CLP programy na riešenie úloh s ohraničeniami nad konečnými doménami majú svoju typickú štruktúru. Ak $V_1, \ldots, V_n$ sú stavové premenné, ktorých hodnoty popisujú hľadané riešenie, možno všeobecnú schému CLP programu pre riešenie takýchto úloh zapísať nasledovne:

```
    najdi_vsetky(V1, ..., Vn) :-
        ohranicenia(V1, ..., Vn),
            % definovanie premenných a stanovenie ohraničení
        prehladavanie(V1, ..., Vn).
            % prehľadávanie - generovanie konkrétnych riešení
```

Ohraničenia možno rozdeliť v zásade do troch skupín:

Stanovenie počiatočných domén premenných – Tento krok v sebe zahŕňuje hľadanie vhodného modelu úlohy v zmysle premenných, ktorých hodnoty budú riešením zadanej úlohy. Pri stanovení počiatočných domén premenných ide v podstate o základné unárne ohraničenia premenných. Správna voľba premenných môže výrazne zredukovať priestor prehľadávania v priebehu ďalšieho výpočtu.

Základná množina ohraničení – Ide o ohraničenia ktoré sú nutné a postačujúce pre formuláciu danej úlohy. Vo všeobecnosti môžu existovať rôzne množiny ohraničení s týmito vlastnosťami a jednotlivé množiny môžu viesť k rôzne efektívnemu výpočtu. To vyplýva práve zo skutočnosti, že CPL systémy pre konečné domény využívajú neúplné algoritmy riešenia ohraničení a orezávací efekt rôznych typov ohraničení môže byť rôzny.

Redundantné ohraničenia – Ide o ohraničenia, ktoré nemenia výsledky ani význam množiny ohraničení z prvej skupiny. Vo všeobecnosti by takéto ohraničenia viedli k spomaleniu výpočtu v prípade úplného algoritmu na riešenie ohraničení. Avšak u systémov s neúplným algoritmom riešenia ohraničení môžu viesť k zlepšenému šíreniu ohraničení. To je možné preto, lebo im zodpovedajúca informácia v základnej množine ohraničení môže byť pasívna.

Z druhej kapitoly tejto knihy vyplýva, že príklad doplňovačky je možné reprezentovať ako úlohu s ohraničeniami definovanými nad konečnými doménami. Premenné pritom predstavujú jednotlivé riadky resp. stĺpce do ktorých sa majú vložiť slová príslušnej dĺžky (množina slov, ktoré môžu byť vložené do daného riadku resp. stĺpca, tvorí doménu zodpovedajúcej premennej). Keďže počet slov ktoré sú k dispozícii je konečný, je možné aplikovať všeobecnú schému riešenia na naprogramovanie a vyriešenie tejto úlohy za pomoci knižnice konečných domén jazyka ECL^iPS^e.

V tomto príklade máme 4 premenné, ktoré je z dôvodov menšieho počtu argumentov hlavného predikátu účelné zaradiť do zoznamu $L = [V1, V2, V3, V4]$. Pri použití redukovanej siete ohraničení podľa obr. 2.2 bude zápis ohraničení najúspornejší (charakter ohraničení je pritom stále rovnaký, mení sa len ich počet podľa miery konzistencie, ktorú sieť ohraničení má). Domény stavových premenných sú v tomto prípade dané vymenovaním hodnôt (slová príslušnej dĺžky), napr. doména premennej V_1 je definovaná `V1 :: [zirafa, europa]`.

Pri takto definovaných doménach je možné z aritmetických ohraničení použiť len ohraničenie na rovnosť `#=`. Všetky ohraničenia sú binárne, keďže riadok a stĺpec sa môžu pretínať iba raz. Pre vyjadrenie kompatibility dvojíc slov v jednotlivých poliach doplňovačky je možné použiť logické ohraničenia, najmä operátor logickej ekvivalencie `<=>`. Napríklad ohraničenie C^{23} možno zapísať `V2 #= kurin #<=> V3 #= iver`.

V prípade, že pre jednu z premenných vystupujúcich v binárnom ohraničení existuje viacero alternatívnych hodnôt, je nutné použiť pre zápis ohraničenia aj logický súčet `#\/`. Napríklad ohraničenie C^{12} pripúšťa pre hodnotu "europa" premennej V_1 dve alternatívne hodnoty premennej V_2 ("puska" a "kurin"), čo možno zapísať nasledovne:

```
V1 #= europa #<=> (V2 #= puska #\/ V2 #= kurin)
```

V prípade že existuje viacero alternatívnych hodnôt pre obe premenné vystupujúce v ohraničení, potom je použitie logického súčtu také, ako v prípade ohraničenia C^{34}, t.j.

```
((V3 #= iver #<=> V4 #= por) #\/ (V3 #= auto #<=> V4 #= eso))
```

Informácie z ohraničení zadaných v prvom kroku sú ďalej šírené až do chvíle, kým už nedochádza k žiadnej ďalšej zmene v doméne niektorej z premenných. V dôsledku neúplnosti použitého algoritmu riešenia ohraničení sa v tejto fáze nemusia odhaliť všetky nekonzistentnosti ohraničení a veľmi často táto fáza ešte nevedie k naviazaniu premenných na konkrétne hodnoty. Aby sa zabezpečila globálna konzistentnosť a vygenerovalo sa riešenie, v ktorom sú všetky premenné naviazané, je nevyhnutné doplniť šírenie ohraničení prehľadávaním.

Prehľadávacia fáza pozostáva z priradzovania hodnôt premenným z ich domén (labeling). Procedúra priradzovania hodnôt premenným vyberie premennú, ktorú v nasledujúcom kroku naviaže a vyberie hodnotu z jej domény na ktorú ju naviaže. V ECL^iPS^e táto procedúra vyzerá nasledovne:

```
labeling([]).
labeling([H|T]) :-  % vstupom je zoznam premenných
    indomain(H),    % zabudovaný predikát indomain/1 naviaže
        % premennú, ktorá je jeho argumentom na prvú hodnotu
        % z jej domény, pri spätnom navracaní potom na druhú
        % atď. kým nevyčerpá všetky hodnoty z jej domény.
    labeling(T).
```

Obyčajne naviazanie jednej premennej okamžite spustí ďalšiu vlnu šírenia ohraničení (najskôr tých v ktorých táto premenná priamo vystupuje), čo vedie k ďalšej redukcii domén premenných alebo k detekcii nekonzistentnosti. V takom prípade sa vyberie iná hodnota z domény premennej (predikát *indomain*/1). Ak je doména premennej už prázdna, výpočet končí s neúspechom.

Proces výberu premenných počas prehľadávania ako aj hodnôt z ich domén môže byť riadený dynamicky. Implementácia rôznych inteligentných stratégií priradzovania hodnôt premenným je možnosťou pre použitie rôznych heuristík. Jedna z najznámejších je "first-fail" čo znamená v našom prípade najskôr výber premennej s najmenšou doménou (resp. najviac ohraničenú premennú). Voľba správnej stratégie priradzovania hodnôt premenným môže výrazne ovplyvniť výslednú efektívnosť programu.

ECL^iPS^e má k dispozícii niekoľko zabudovaných predikátov pre realizáciu základných heuristík pre výber premennej na priradenie hodnoty [6], napr.

[6] Podrobný popis spôsobov implementácie rôznych prehľadávacích techník v ECL^iPS^e možno nájsť v [118].

deleteff(?Var, +List, -Rest) vyberie zo zoznamu $+List$ premennú $?Var$ s najmenšou doménou. $-Rest$ je zoznam zvyšných premenných. Druhou možnosťou je predikát *deleteffc(?Var, +List, -Rest)* ktorý vyberie najohraničenejšiu premennú (t.j. okrem veľkosti domény berie do úvahy aj počet ohraničení nad danou premennou) alebo *deletemin(?Var, +List, -Rest)* ktorý vyberie premennú s najmenšou dolnou hranicou domény.

Využitie takýchto heuristík je potom možno naprogramovať nasledovne:

```
my_labeling([]).
my_labeling(List) :-
    deleteffc(H,List,T),
    indomain(H),
    labeling(T).
```

V prípade príkladu doplňovačky vzhľadom na jeho malý rozmer nie je nutné používať žiadne heuristiky na zefektívnenie fázy prehľadávania, ale stačí využiť v knižnici konečných domén definovaný predikát $labeling/1$, t.j. použiť volanie `labeling(L)`, kde `L` je vyššie spomínaný zoznam premenných V_1 až V_4.

V príklade slovnej doplňovačky nás zaujímajú všetky riešenia, takže nepotrebujeme použiť optimalizáciu. Výsledný program preto vyzerá nasledovne:

```
:- lib(fd).

krizovka(L) :-
    L = [V1, V2, V3, V4],
        % Stanovenie počiatočných domén premenných
    V1 :: [zirafa, europa],
    V2 :: [puska, kurin],
    V3 :: [auto, iver],
    V4 :: [por, eso],
        % Základná množina ohraničení (po redukcii)
    V1 #= europa #<=> (V2 #= puska #\/ V2 #= kurin), % C12
    V2 #= kurin #<=> V3 #= iver,                     % C23
    V1 #= europa #<=> V4 #= por,                     % C14
    ((V3 #= iver #<=> V4 #= por) #\/
     (V3 #= auto #<=> V4 #= eso)),                   % C34
        % Prehľadávanie (hľadanie všetkých riešení)
    labeling(L).
```

Odpoveďou systému je potom nájdenie jediného existujúceho riešenia.

```
[eclipse 2]: krizovka(L).

L = [europa, kurin, iver, por]     More? (;)
no (more) solution.
```

6.1.2 Optimalizácia

V prípade že sa požaduje nie ľubovoľné riešenie, ale je zadaná aj kriteriálna funkcia podľa ktorej sa posudzuje kvalita jednotlivých riešení, pristupuje k prehľadávaniu ešte aj optimalizácia. Úlohou je nájsť optimálne riešenie v zmysle nejakého kritéria (t.j. je potrebné minimalizovať alebo maximalizovať nejakú funkciu nákladov $cost(V_1, ..., V_n)$). V takomto prípade prehľadávacia procedúra je riadená nejakým vyšším, optimalizačným predikátom, napr. $min\_max/2$.

```
najdi_optimum(X1, ..., Xn) :-
    ohranicenia(X1, ..., Xn),
    min_max(prehladavanie(X1, ..., Xn), Naklady).
        % hladanie optimálneho riešenia v zmysle
        % funkcie nákladov Naklady
```

Väčšina CLP systémov ponúka aj optimalizačné predikáty založené na metóde vetvenia a hraníc. Základnou myšlienkou, ktorú tieto predikáty využívajú, je hľadať riešenia obvyklým spôsobom, po jeho nájdení pridať nové ohraničenie že riešenie musí byť lepšie ako doteraz najlepšie nájdené riešenie a opäť spustiť prehľadávanie.

V jazyku ECL^iPS^e sú implementované viaceré predikáty na báze tohoto prístupu. Základné dva sú *min_max(+Goal, ?C)* a *minimize(+Goal, ?Term)* kde C je lineárny term vyjadrujúci funkciu nákladov a *+Goal* je cieľ, ktorý je vzhľadom na ňu optimalizovaný (ide stále o minimalizáciu). V prípade *minimize* je druhým argumentom celočíselná doménová premenná $?Term$.

Metóda vetvenia a hraníc používa pre orezávanie tzv. hornú a dolnú hranicu nákladov, t.j. predpokladá, že optimálne riešenie leží niekde medzi týmito hranicami. Na tomto mieste si je dôležité uvedomiť, že okrem spôsobu a efektívnosti prehľadávania samotnej metódy vetvenia a hraníc môže celkový čas potrebný na nájdenie riešenia výrazne ovplyvniť aj voľba týchto hraníc na začiatku prehľadávania. Čím užší je inicializačný interval, tým menší priestor prehľadávania musí spracovať metóda vetvenia a hraníc. Efektívna voľba týchto hraníc je silne závislá od konkrétnych znalostí o riešenej úlohe. Pre rozsiahle úlohy je obvykle nevyhnutné venovať určitý čas na výpočet čo možno najlepšieho odhadu pre dolnú a hornú hranicu. Tento čas sa obvykle mnohonásobne vráti v ďalšej etape výpočtu, keď nastúpi metóda vetvenia a hraníc.

Zadanie hornej $+High$ a dolnej hranice $+Low$ nákladov umožňujú viacargumentové verzie predikátov, napr. *min_max(+Goal, ?C, +Low, +High, +Percent)* a *minimize(+Goal, ?Term, +Low, +High, +Percent)*. Tolerancia presnosti priblíženia sa k optimu je zadaná argumentom *+Percent* (t.j. ak zadáme 0 požadujeme presné optimálne riešenie). Existujú aj ďalšie analogické predikáty, napríklad s možnosťou zadať maximálny čas riešenia úlohy. Podrobnejšie v [22].

Dôvod, prečo sa v CLP jazykoch (ako napr. CHIP [2], ECL^iPS^e [22] alebo cc(FD) [74]) používa práve tento prístup, je jednoduchý. Metóda vetvenia a hraníc je úplná metóda prehľadávania a výborne sa hodí do prostredia kde je možné spätné navracanie, ktoré je typickým sprievodným javom prehľadávania do hĺbky u logických programovacích jazykov.

V princípe existujú dve stratégie metódy vetvenia a hraníc (sú implementované napr. v CLP jazykoch CHIP a ECL^iPS^e) [99]:

1. **MIN_MAX** – Vychádzajúc zo známej počiatočnej hodnoty hornej a dolnej hranice nákladov, ktoré sa požadujú od riešenia N^{max} a N^{min} používa ohraničenie $N^{min} \leq N \leq N^{max}$ (ide o celé čísla) na orezávanie priestoru prehľadávania. Akonáhle nájde riešenie s nákladmi $N_i < N^{max}$, prehľadávanie zastaví a začne opäť odznova, ale s novým ohraničením $N^{min} \leq N \leq N_i - 1$. Ak sa nenájde žiadne ďalšie riešenie, alebo ak $N_i = N^{min}$ potom posledne nájdené riešenie je optimálne.

2. **MINIMIZE** pracuje veľmi podobne s tým rozdielom, že po nájdení riešenia nezačína prehľadávať od začiatku, ale pokračuje na mieste, kde sa práve nachádza. Na jednej strane MINIMIZE v porovnaní s predchádzajúcim MIN_MAX nemusí odznova prehľadávať tú časť priestoru, ktorú už predtým prehľadal do nájdenia posledného riešenia. Na druhej strane sa pri tejto metóde objavuje u mnohých úloh jav nazývaný "trashing", t.j. prípad keď je generovaných množstvo riešení s podobnými nákladmi (pretože sú topologicky blízko seba v priestore prehľadávania) a konvergencia k optimálnemu riešeniu sa tak veľmi spomalí.

Tento postup je možné vylepšiť niekoľkými spôsobmi.

1. Ak sa uspokojíme so suboptimálnym riešením v rámci vopred zadanej presnosti E (číslo od 0 do 1), potom je možné oba vyššie uvedené postupy upraviť nasledovne. Po nájdení nového riešenia sa novou hornou hranicou stane $N_i(1-E)-1$ (namiesto pôvodnej $N_i - 1$). Ak sa nenájde žiadne lepšie riešenie, potom vieme že optimálne riešenie leží v intervale $< N_i(1-E), N_i >$. Tento prístup čiastočne predchádza javu nazvanému "trashing".

2. Paralelným prehľadávaním priestoru prehľadávania. Ďalšie vylepšenia boli dosiahnuté využitím paralelizmu na prehľadávanie s viacerými hranicami pre náklady súčasne, ako boli navrhnuté a testované v [99]. Táto stratégia predchádza tak trashingu u MINIMIZE ako aj zbytočnému opätovnému prehľadávaniu častí priestoru u MIN_MAX.

3. V jazyku ECL^iPS^e boli implementované aj logaritmické verzie [104] vyššie uvedených postupov MIN_MAX a MINIMIZE. Podstata oboch metód, tzv. **logaritmický min_max** i **logaritmický minimize** (podrobnejší popis programov možno nájsť v [119]) spočíva v delení intervalu ohraničeného aktuálnou dolnou a hornou hranicou nákladov na polovicu, t.j. $N^{max\_new} = (N^{min}+N^{max})/2$ a v hľadaní riešenia s takto získanou novou hranicou N^{max_new}. Ak neexistuje riešenie s menšími nákladmi, potom sa táto hodnota stáva novou dolnou hranicou, t.j. $N^{max\_new} + 1 \leq N \leq N^{max}$. Ak sa takéto riešenie nájde (povedzme s nákladmi $N_i < N^{max\_new}$) potom sa jeho hodnota kriteriálnej funkcie stáva novou hornou hranicou, t.j. $N^{min} \leq N \leq N_i - 1$ a postup sa opakuje až kým sa interval nezúži na jednotku.

Rozdiel medzi logaritmickým min_max a logaritmickým minimize je analogický rozdielu ich nelogaritmických predchodcov. Pre účinnejšie riešenie úloh s ťažkým dôkazom optimálnosti[7] je naviac algoritmus logaritmický min_max vylepšený o dva typy špeciálnych situácií, kedy sa jeho stratégia mení na stratégiu MIN_MAX. Jednou z nich je situácia, keď sa trikrát po sebe nezmení horná hranica, potom v nasledujúcom kroku už neposúva dolnú hranicu, ale len hornú zníži o 1 (pre jeden nasledujúci krok) teda použije stratégiu MIN_MAX. Druhá situácia nastáva, keď sa veľkosť intervalu daná dolnou a hornou hranicou zníži pod hodnotu 6. Potom sa stratégia trvalo prepne na MIN_MAX (na zvyšných nanajvýš 5 krokov).

[7]Fáza prehľadávania od nájdenia optimálneho riešenia až po prehľadanie zvyšného priestoru za účelom dokázania že lepšie riešenie už neexistuje.

6.2 Nekonečné domény

Rozšírenie CLP schopností jazyka ECL^iPS^e vo verzii 5.0[8] aj na oblasť nekonečných domén je zabezpečená viacerými spôsobmi, z ktorých každý je realizovaný samostatnou knižnicou:

- **eplex** – táto knižnica umožňuje použitie externého systému založeného na báze simplexového algoritmu resp. zmiešaného celočíselného programovania. V súčasnosti sú podporované dva známe systémy na riešenie úloh lineárneho, celočíselného a zmiešaného programovania, a síce *CPLEX* a *XPRESS-MP*. Pre použitie tejto knižnice je nevyhnutná platná licencia na aspoň jeden zo spomínaných systémov, alebo je možné použiť funkcionalitu študentskej verzie XPRESS-MP[9]. Kvôli vlastnostiam lineárneho programovania neumožňuje táto knižnica definovanie ostrých nerovníc $<, >$ a nerovnosti $\neq$.

- **ria** – knižnica implementujúca intervalovú aritmetiku (šírenie hraníc intervalov) nad reálnymi aj celočíselnými premennými. Ohraničenia pritom nemusia byť lineárne, je k dispozícii pomerne veľká množina základných matematických funkcií. Rovnako ako eplex, neumožňuje definovanie ostrých nerovníc $<, >$ a nerovnosti $\neq$.

  V tejto knižnici sa využívajú dva rôzne algoritmy: hranová konzistencia a tzv. "squash" algoritmus ktorý lepšie šíri ohraničenia, ale je časovo náročnejší. Pozor, tento mechanizmus nikdy nenaviaže premenné, t.j. nedá presné riešenie (neumožňuje ani optimalizáciu). Odpoveďou sú len spresnené intervaly v rámci ktorých *môže* ležať riešenie. Spravidla na konci výpočtu totiž zostáva ešte väčšie množstvo uspaných ohraničení. Systém preto nezaručuje existenciu riešenia vo výsledných intervaloch, ale len neexistenciu riešenia mimo nich.

- **range** – knižnica nie je v pravom slova zmysle ďalší mechanizmus, ale implementuje hranice pre reálne i celočíselné intervaly. Táto knižnica slúži ako základ pre implementáciu aritmetických algoritmov na riešenie ohraničení. Môže slúžiť aj ako mechanizmus na komunikáciu viacerých algoritmov. Definuje potrebné atribúty a základné predikáty na narábanie s týmito atribútmi.

[8]Táto verzia bola zverejnená koncom júna 2000.

[9]stačí do konfiguračného súboru *eplex_lic_info.ecl* zapísať riadok *licence(_Default, xpress, '11', '//c/eclipse/lib/i386_nt', 0).* kde tretím argumentom je cesta do adresára so súborom *express11.dll* štandardne dodávaným s ECL^iPS^e vo verzii 5.0.

- **clpq, clpr** – Tieto dve knižnice sú najstaršou, samostatnou CLP implementáciou systému pre spĺňanie ohraničení nad racionálnymi resp. reálnymi číslami v rámci ECL^iPS^e. Nevyžadujú žiadne externé systémy na riešenie ohraničení a optimalizáciu, vedia spracovať ostré nerovnice $<, >$ a nerovnosti $\neq$.

V nasledujúcom budú na riešenie úloh s ohraničeniami nad nekonečnými doménami používané už len knižnice $clp(Q, R)$ [75], ktoré sú inštanciami základnej CLP schémy definovanej v [79]. Tieto implementácie sa vyrovnajú ostatným $CLP(\mathcal{R})$ systémom. Riešia lineárne rovnice nad racionálnymi alebo reálnymi číslami, podporujú pasívne riešenie nelineárnych rovníc (pozastavujú ich riešenie kým sa nestanú lineárnymi v dôsledku naviazania niektorých premenných), obsahujú rozhodovací algoritmus pre lineárne nerovnice, dovoľujú lineárne nerovnosti a poskytujú aj lineárnu optimalizáciu.

Pre prácu s nekonečnými doménami je nutné najskôr nakompilovať príslušnú knižnicu. Pre prácu s racionálnymi číslami je to knižnica *clpq* a pre prácu s reálnymi číslami *clpr*. Rozhranie oboch knižníc ponúka početnú množinu základných ohraničujúcich operátorov a niektoré iné dôležité predikáty.[10]

Ohraničenia môžu byť tvorené relačnými ohraničujúcimi operátormi a spájané čiarkami vyjadrujúcimi ich konjunkciu. Relačné ohraničujúce operátory sú nasledovné:

- `?E1 =:= ?E2` alebo `?E1 = ?E2` – rovnica
- `?E1 < ?E2` alebo `?E1 > ?E2` – ostrá nerovnica
- `?E1 =< ?E2` alebo `?E1 >= ?E2` – neostrá nerovnica
- `?E1 \= ?E2` – nerovnosť

Argumentmi (`?E1, ?E2`) vo všetkých prípadoch môžu byť aritmetické výrazy, pričom za výraz sa (okrem ďalších) považuje:

- premenná (prologovská premenná)
- číslo (v pohyblivej rádovej čiarke, alebo celé číslo)
- `+?E3` unárne plus

[10] Uvádzame len tie, ktoré budú využívané v nasledujúcich príkladoch. Úplný zoznam možno nájsť v [75].

- `-?E4` unárne mínus
- `?E3 + ?E4` súčet výrazov
- `?E3 - ?E4` rozdiel výrazov
- `?E3 * ?E4` súčin výrazov
- `?E3 / ?E4` podiel výrazov
- abs(?E3) absolútna hodnota výrazu
- sin(?E3) trigonometrický sínus výrazu
- exp(?E3,?E4) mocnina výrazu

Pozor, ohraničujúce aritmetické operátory knižníc *clpq* a *clpr* vyzerajú rovnako ako klasické prologovské testy! Na to, aby boli spracované algoritmami na riešenie ohraničení nad nekonečnými doménami, je nutné vložiť všetky ohraničenia do zložených zátvoriek. Napr.:

```
[eclipse 1]: lib(clpr).
...

yes.
[eclipse 2]: X > 3.
instantiation fault in X > 3
[eclipse 3]: {X > 3}.

X = X

% Linear constraints:
{
  X > 3.0
}
yes.
```

Ďalšia ukážka demonštruje správanie sa systému v prípade naviazania premennej obsiahnutej v aktuálnej množine ohraničení:

```
[eclipse 4]: {X > 3}, X = 2.0.

no (more) solution.
```

```
[eclipse 5]: {X > 3}, X = 4.0.

X = 4.0
yes.
```

Ďalšie dôležité predikáty poskytované knižnicami *clpq* a *clpr* sú:

- `entailed(+C)` uspeje, ak dané ohraničenie `C` je implikované aktuálnou množinou ohraničení. Aktuálna množina ohraničení (constraint store) ostáva pritom bez zmeny. Napríklad

  ```
  [eclipse 6]: {A=<5}, entailed(A=\=6).

  A = A

  % Linear constraints:
  {
    A =< 5.0
  }
  yes.
  [eclipse 7]: {A=<5}, entailed(A=\=4).

  no (more) solution.
  ```

- `inf(+E1,-Inf)` Vypočíta infimum lineárneho výrazu `E1` a unifikuje ho s `Inf`. Neúspech indikuje neohraničenosť zdola.

- `sup(+E1,-Sup)` Vypočíta supremum lineárneho výrazu `E1` a unifikuje ho so `Sup`. Neúspech indikuje neohraničenosť zhora.

  ```
  [eclipse 8]: { 2*X +   Y =< 16,
                   X + 3*Y =< 15,
                   X + 2*Y =< 11,
                   Z = 30*X + 50*Y
               }, sup(Z, Sup).

  X = X
  Y = Y
  Z = Z
  Sup = 310.00000000000006
  ```

```
% Linear constraints:
{
  Z = 50.000000000000014 * Y + 30.000000000000004 * X,
  Y + 0.33333333333333326 * X =< 5.0000000000000027,
  Y + 0.5 * X =< 5.5000000000000036,
  Y + 1.9999999999999956 * X =< 15.999999999999979
}
yes.
```

- `minimize(+E1)` Vypočíta infimum lineárneho výrazu `E1` a dá ho do rovnosti s týmto výrazom, t.j. ako by sme definovali `minimize(E1) :- inf(E1, E1).`

- `maximize(+E1)` Vypočíta supremum lineárneho výrazu `E1` a dá ho do rovnosti s týmto výrazom. Napríklad

```
[eclipse 9]: { 2*X +   Y =< 16,
                 X + 3*Y =< 15,
                 X + 2*Y =< 11,
                 Z = 30*X + 50*Y
             }, maximize(Z).

X = 7.0
Y = 2.0000000000000009
Z = 310.00000000000006
yes.
```

Zo všetkých možných riešení vyberie to s najväčšou hodnotou pre Z.

6.2.1 Schéma riešenia úloh s nekonečnými doménami

Podobne ako v prípade konečných domén, aj CLP programy na riešenie úloh s ohraničeniami nad nekonečnými doménami majú svoju typickú štruktúru. Táto je dosť podobná štruktúre pre úlohy s ohraničeniami nad konečnými doménami. Hlavný rozdiel je v tom, že v tomto prípade odpadáva fáza prehľadávania v takej podobe, ako tomu bolo u konečných domén. Môže sa vyskytnúť len vo forme optimalizácie, ak je táto požadovaná.

Ak *V1, ..., Vn* sú stavové premenné ktorých hodnoty popisujú hľadané riešenie, možno všeobecnú schému CLP programu pre riešenie úloh s ohraničeniami nad nekonečnými doménami zapísať nasledovne:

```
najdi_vsetky(V1, ..., Vn) :-
    ohranicenia(V1, ..., Vn).
        % definovanie premenných a stanovenie ohraničení
```

Riešenie teda spočíva v spustení výpočtu šírením ohraničení, ktorý spravidla vedie k redukcii počiatočných domén premenných a efektívnejšiemu vyjadreniu množiny ohraničení (ak nie je ich splniteľnosť zabezpečená pre všetky hodnoty z domén premenných).

Ďalší rozdiel je v spôsobe zápisu ohraničení. Všetky relačné ohraničenia definované v knižniciach *clpq* resp. *clpr* musia byť uvedené v zložených zátvorkách. Ostatné (t.j. používateľom definované) sú umiestnené mimo zátvoriek. Odpadá tiež explicitné stanovenie počiatočných domén premenných, nakoľko tieto je možné zadať vo forme relačných ohraničení.

V prípade numerickej doplňovačky podľa obr. 2.3 ide o úlohu s ohraničeniami nad nekonečnými doménami. Premenné V_{12}, V_{14}, V_{23}, V_{34} označujúce voľné políčka doplňovačky podľa indexov riadkov a stĺpcov ktorých sú priesečníkom, majú byť reálne čísla.

Unárne ohraničenia vyplývajúce z usporiadanosti čísiel v doplňovačke sú dané susednými políčkami a je možné ich zapísať pomocou relačného ohraničujúceho operátora $>$, resp. $<$ keďže vyžadujeme rastúce hodnoty čísiel v políčkach doplňovačky v smere zľava doprava a zhora nadol. Tak napríklad pre políčko ktorému zodpovedá premenná V_{12} máme 4 susedné políčka. V riadku 1 ktorý ho pretína leží naľavo hodnota 0.8 a napravo hodnota 2.4, čo zodpovedá zápisu ohraničení vo forme `{0.8 < V12, V12 < 2.4}`.

Podobne v stĺpci 2 ktorý ho pretína je nad ním hodnota 1.7 a pod ním 3.2. Takže analogicky zapíšeme ďalšie dve unárne ohraničenia nasledovne: `{1.7 < V12, V12 < 3.2}`. Z uvedených štyroch unárnych ohraničení zjavne stačí vybrať dve, ktoré najviac zúžia doménu premennej V_{12}, nakoľko všetky ohraničenia musia platiť súčasne (konjunkcia).

Okrem unárnych ohraničení pre všetky premenné riešenie musí spĺňať aj podmienku rovnakého súčtu 20 vo všetkých riadkoch aj stĺpcoch. V prípade danej doplňovačky pôjde o binárne ohraničenia nakoľko v každom riadku, resp. stĺpci sú práve dve voľné políčka. Tieto ohraničenia možno zapísať vo forme rovníc. Pre prvý riadok bude ohraničenie vyzerať nasledovne: `{0.8 + V12 + 2.4 + 2.6 + V14 + 8.8 = 20}`.

Požiadavka usporiadanosti čísiel v jednotlivých políčkach v horizontálnom ($V_{12} < V_{23}$, $V_{12} < V_{14}$) aj vertikálnom smere ($V_{23} < V_{34}$, $V_{14} < V_{34}$) je implicitne obsiahnutá v unárnych ohraničeniach.
Výsledný program (uložený v súbore *krizovka.ecl*) teda bude vyzerať nasledovne:

```
:- lib(clpr).

krizovka1(Cisla) :-
    Cisla = [V12,V14,V23,V34],
    {
     1.7 < V12, V12 < 2.4,
     2.6 < V14, V14 < 8.1,
     3.2 < V23, V23 < 3.6,
     8.1 < V34,
     0.8 + V12 + 2.4 + 2.6 + V14 + 8.8 = 20,
     1.7 + V12 + 3.2 + V23 + 9.7 = 20,
     V23 + 3.6 + 4.5 + V34 = 20,
     V14 + 8.1 + V34 = 20
    }.
```

Po nakompilovaní zdrojového textu programu a zadaní zodpovedajúceho cieľa je odpoveď systému nasledovná:

```
[eclipse 10]: [krizovka].
krizovka.ecl compiled traceable 4956 bytes in 0.00 seconds

yes.
[eclipse 11]: krizovka1(L).

L = [V12, V14, V23, V34]

% Linear constraints:
{
  V12 < 2.2,
  V14 = 5.3999999999999986 - V12,
  V23 = 5.4 - V12,
  V34 = 6.5 + V12,
  V12 > 1.8000000000000003
}
```

```
yes.
```

To znamená, že všetky riešenia musia vyhovovať výslednej (vyriešenej) forme ohraničení. Pri bližšom pohľade na túto výslednú upravenú množinu zadaných ohraničení možno vidieť, že premenná V_{12} musí ležať približne v intervale $< 1.8, 2.2 >$ a všetky ostatné premenné sú priamo závislé len od hodnoty V_{12}. Znamená to teda, že riešení je nekonečne veľa. Konkrétne riešenie je možné získať zadaním hodnoty jednej premennej, najlepšie práve nezávislej premennej V_{12}. Napríklad:

```
[eclipse 12]: krizovka1([V12,V14,V23,V34]), V12 = 2.0.

V14 = 3.3999999999999986
V23 = 3.4000000000000004
V34 = 8.5
V12 = 2.0
yes.
```

Podobne ako v prípade konečných domén aj teraz mnohé úlohy vyžadujú optimalizáciu, t.j. nájdenie najlepšieho riešenia vzhľadom na zadanú optimalizačnú funkciu. V knižniciach *clpq* resp. *clpr* sú za týmto účelom k dispozícii predikáty *minimize*/1 resp. *maximize*/1. Všeobecná schéma programu na riešenie optimalizovanej úlohy s ohraničeniami nad nekonečnými doménami potom vyzerá nasledovne:

```
najdi_min(X1, ..., Xn) :-
    ohranicenia(X1, ..., Xn),
    minimize(Naklady).
        % hľadanie optimálneho riešenia v zmysle
        % funkcie nákladov Naklady
```

Analogicky by vyzeral program pre nájdenie maxima kriteriálnej funkcie. Ak sa v úlohe numerickej doplňovačky definuje funkcia nákladov ako vážený súčet doplnených čísiel[11], potom výsledný program bude vyzerať nasledovne:

```
krizovka2(Cisla, Sucet) :-
    Cisla = [V12,V14,V23,V34],
    Sucet = 5*V12 + 3*V14 + 2*V23 + V34,
```

[11] Ak by sme uvažovali len ich súčet, dostali by sme konštantu približne rovnú 17.3 nakoľko $V_{12} + V_{14} + V_{23} + V_{34} = V_{12} + 5.4 - V_{12} + 5.4 - V_{12} + 6.5 + V_{12} = 17.3$.

```
    {
     1.7 < V12, V12 < 2.4,
     2.6 < V14, V14 < 8.1,
     3.2 < V23, V23 < 3.6,
     8.1 < V34,
     0.8 + V12 + 2.4 + 2.6 + V14 + 8.8 = 20,
     1.7 + V12 + 3.2 + V23 + 9.7 = 20,
     V23 + 3.6 + 4.5 + V34 = 20,
     V14 + 8.1 + V34 = 20
    },
    maximize(Sucet).
```

Jeho nakompilovaním a spustením však možno zistiť, že optimálne riešenie neexistuje, čo je spôsobené ostrými nerovnosťami v unárnych ohraničeniach.

```
[eclipse 13]: krizovka2(C,S).

no (more) solution.
```

Je však možné zistiť hornú (a analogicky aj dolnú) hranicu kriteriálnej funkcie ku ktorej riešenia môžu konvergovať použitím predikátu *sup/2* (*inf/2*).

```
krizovka3(Cisla, Max) :-
    Cisla = [V12,V14,V23,V34],
    Sucet = 5*V12 + 3*V14 + 2*V23 + V34,
    {
     1.7 < V12, V12 < 2.4,
     2.6 < V14, V14 < 8.1,
     3.2 < V23, V23 < 3.6,
     8.1 < V34,
     0.8 + V12 + 2.4 + 2.6 + V14 + 8.8 = 20,
     1.7 + V12 + 3.2 + V23 + 9.7 = 20,
     V23 + 3.6 + 4.5 + V34 = 20,
     V14 + 8.1 + V34 = 20
    },
    sup(Sucet, Max).
```

Výsledok bude v tomto prípade nasledovný:

```
[eclipse 14]: krizovka3(Cisla,Max).
```

```
Cisla = [V12, V14, V23, V34]
Max = 35.699999999999996

% Linear constraints:
{
  V12 < 2.2,
  V14 = 5.3999999999999986 - V12,
  V23 = 5.4 - V12,
  V34 = 6.5 + V12,
  V12 > 1.8000000000000003
}
yes.
```

Nech sa úloha numerickej doplňovačky trocha zovšeobecní a síce tak, že sa bude požadovať rovnosť súčtu hodnôt po riadkoch i stĺpcoch, ale nepredpíše sa jej presná hodnota. Úloha bude mať takto o jednu premennú viac. Bude to práve spomínaný súčet premenných. Program teda bude veľmi podobný.

```
krizovkaX(Cisla) :-
    Cisla = [V12,V14,V23,V34, X],
    {
     0.8 < V12, V12 < 2.4, 1.7 < V12, V12 < 3.2,
     2.6 < V14, V14 < 8.8, V14 < 8.1,
     V23 > 3.2, V23 < 3.6, V23 < 9.7,
     V34 > 4.5, V34 > 8.1,
     0.8 + V12 + 2.4 + 2.6 + V14 + 8.8 = X,
     1.7 + V12 + 3.2 + V23 + 9.7 = X,
     V23 + 3.6 + 4.5 + V34 = X,
     V14 + 8.1 + V34 = X
    }.

[eclipse 15]: krizovkaX(L).

L = [V12, V14, V23, V34, X]

% Linear constraints:
{
  V12 < 2.4,
  V14 < 3.5999999999999983,
```

```
  V23 = V14,
  V34 = 6.5 + V12,
  X = 14.600000000000001 + V14 + V12,
  V12 > 1.7,
  V14 > 3.1999999999999984
}
yes.
```

Riešenie úlohy v tomto prípade obsahuje už aj explicitné vyjadrenie rovnosti premenných V_{23} a V_{14} ktoré bolo v predchádzajúcom prípade vyjadrené iba implicitne.

Vo výsledkoch programov využívajúcich knižnicu *clpr*, teda pracujúcu s reálnymi číslami reprezentovanými v pohyblivej rádovej čiarke, dochádza k numerickej nepresnosti. Možno ju pozorovať napr. vo výsledku ktorý sme dosiahli pre $V_{12} = 2.0$. Ostatné premenné mali nasledovné hodnoty: $V_{14} = 3.3999999999999986$, $V_{23} = 3.4000000000000004$ a $V_{34} = 8.5$. Zdá sa, že V_{14} by sa mohlo rovnať V_{23}. Ak si chceme overiť túto hypotézu, musíme použiť úplnú presnosť, ktorú nám u nekonečných domén poskytuje knižnica *clpq* pracujúca s racionálnymi číslami.

Program je samozrejme nutné upraviť v tom zmysle, že reálne čísla nahradíme racionálnymi zlomkami a namiesto knižnice *clpr* nakompilujeme knižnicu *clpq*.

```
:- lib(clpq).

krizovka1Q(Cisla) :-
    Cisla = [V12,V14,V23,V34],
    {
     4/5 < V12, V12 < 12/5, 17/10 < V12, V12 < 16/5,
     13/5 < V14, V14 < 44/5, V14 < 81/10,
     V23 > 16/5, V23 < 18/5, V23 < 97/10,
     V34 > 9/2, V34 > 81/10,
     4/5 + V12 + 12/5 + 13/5 + V14 + 44/5 = 20,
     17/10 + V12 + 16/5 + V23 + 97/10 = 20,
     V23 + 18/5 + 9/2 + V34 = 20,
     V14 + 81/10 + V34 = 20
    }.
```

Množina riešení bude teraz popísaná analogickou vyriešenou formou ohraničení s úplnou presnosťou racionálnych čísel.

```
[eclipse 16]: krizovka1Q(L).

L = [V12, V14, V23, V34]

% Linear constraints:
{
  V12 < 11 / 5,
  V14 = 27 / 5 - V12,
  V23 = 27 / 5 - V12,
  V34 = 13 / 2 + V12,
  V12 > 9 / 5
}
yes.
```

Z výsledku je už teraz jasné, že hodnoty premenných $V14$ a $V23$ sú rovnaké. Pre získanie konkrétneho riešenia pri použití racionálnych čísel nie je možné zadávať ohraničenia mimo zloženú zátvorku tak, ako u reálnych čísel. Konkrétnu hodnotu je preto potrebné zadať priamo v argumente predikátu $krizovka1Q/1$.

```
[eclipse 17]: krizovka1([2,V14,V23,V34]).

V14 = 17 / 5
V23 = 17 / 5
V34 = 17 / 2
yes.
```

Časť IV

Riešené úlohy s ohraničeniami

Kapitola 7

Úlohy nad konečnými doménami

7.1 Niekoľko jednoduchých úloh na úvod

Prvú úlohu možno nazvať "hmyz". V krabici sa nachádzajú osemnohé pavúky a šesťnohé chrobáky. Dohromady majú N nožičiek. Úlohou je zistiť, koľko je v krabici pavúkov a koľko chrobákov.

Najskôr je potrebné stanoviť premenné a ich počiatočné domény. Cieľom je určiť počet pavúkov (premenná V_1) a počet chrobákov (premenná V_2), takže sú potrebné dve celočíselné premenné. Pri stanovení počiatočnej domény existuje viacero možností.

- Doménu nestanoviť, len zabezpečiť nezápornosť oboch premenných.
- Počet jedincov bude určite nezáporný, takže dolná hranica domény bude 0. Horná hranica bude dostatočne veľké celé číslo, napr. práve zadaný počet nôh, t.j. `N`.
- Pri snahe vyjadriť maximum informácie v unárnych ohraničeniach (t.j. už pri definícii domén) možno zo zadaného počtu nôh vypočítať pomocou operácie celočíselného delenia maximálny teoreticky možný počet pavúkov (`P is N // 8`), resp. chrobákov (`CH is N // 6`).

Pre prvú alternatívu môže program (nech je uložený v súbore s názvom *hmyz.pl*) vyzerať takto:

```
:- lib(fd).
```

```
hmyz(L,N) :-
        % L je zoznam stavových premenných.
        % V hlave pravidla sa nachádza preto,
        % aby sa vo výsledku objavili ich hodnoty.
        % N (počet nôh) musí byť zadané
    L = [P,CH],
        % Stanovenie premenných, až tu sa rozpíše
        % zoznam L na konkrétne premenné
        % Nasledujú definície ohraničení
    P #>= 0,   % počet musí byť nezáporné číslo
    CH #>= 0,
    8*P + 6*CH #= N.
        % Základné binárne ohraničenie pre počet pavúkov
        % a chrobákov pri zadanom počte nôh
```

Takýto program bez prehľadávania (labeling) je postačujúci v prípadoch, ak existuje len jediné riešenie, napr.

```
[eclipse 1]: [hmyz].
...
hmyz.pl compiled traceable 376 bytes in 0.12 seconds

yes.
[eclipse 2]: hmyz(L,34).

L = [2, 3]
yes.
[eclipse 3]: hmyz(L,46).

L = [P{[2..5]}, CH{[1..5]}]

Delayed goals:
        -46 + 6 * CH{[1..5]} + 8 * P{[2..5]}#=0
yes.
```

Ako vidno z výpisu dialógu so systémom ECL^iPS^e, ak sa zadá taký počet nôh, že existuje iba jedno riešenie (napr. $N = 34$), program ho nájde len na základe ohraničení (t.j. fáza lokálneho šírenia ohraničení bola postačujúca), bez nutnosti prehľadávania. V prípade, že riešenie nie je jednoznačné (napr. $N = 46$), odpoveďou systému sú premenné so zúženými doménami[1], tak ako ich dokázalo zúžiť šírenie ohraničení, plus uspané ohraničenia (v našom prípade len jedno) v normálnej forme.

Ak sa do predchádzajúceho programu doplní aj prehľadávanie napr. pomocou zabudovaného predikátu *labeling/1*, t.j.

```
:- lib(fd).

hmyz1(L,N) :-
    L = [P,CH],
    P #>= 0,
    CH #>= 0,
    8*P + 6*CH #= N,
    labeling(L).
```

Potom systém postupne nájde všetky existujúce riešenia.

```
[eclipse 4]: hmyz1(L,46).

L = [2, 5]     More? (;)

L = [5, 1]
yes.
```

Ďalšou zaujímavou skupinou úloh sú kryptogramy, napr. $DONALD + GERALD = ROBERT$. Cieľom je nájsť také priradenie cifier písmenám, aby po ich substitúcii za písmená uvedený súčet platil.

V tomto prípade sú opäť pomerne jednoznačné premenné, a síce sú nimi práve zadané písmená D, O, N, A, L, G, E, R, B, T. Keďže ich je potrebné nahradiť ciframi, ich počiatočnou doménou budú teda všetky cifry, čo možno v ECL^iPS^e vyjadriť ako interval 0..9. Súčet možno v ECL^iPS^e vyjadriť jediným lineárnym ohraničením po vynásobení jednotlivých cifier príslušným násobkom desiatich. Naviac musí platiť, že navzájom rôzne písmená predstavujú rôzne cifry, takže musia byť po dvojiciach rôzne. Toto možno jednoducho vyjadriť priamo pomocou zabudovaného ohraničenia *alldifferent/1*. Nakoniec ešte nemožno zabudnúť na to, že prvá platná cifra v každom čísle (slove zadaného kryptogramu) nemôže byť nula. V tomto prípade sa to týka premenných D, G, R. Keďže neide o optimalizačný problém, na prehľadávanie opäť stačí použiť zabudovaný predikát *labeling*.

[1]Zúžené domény obsahujú tak validné kombinácie (2 pavúky, 5 chrobákov), ako aj neprípustné (3 pavúky, 4 chrobáky).

```
dgr(Zoznam) :-
    Zoznam = [D, O, N, A, L, G, E, R, B, T],
    Zoznam :: 0..9,
    D*100000 + O*10000 + N*1000 + A*100 + L*10 + D +
    G*100000 + E*10000 + R*1000 + A*100 + L*10 + D #=
    R*100000 + O*10000 + B*1000 + E*100 + R*10 + T,
    alldifferent(Zoznam),
    D #\= 0, G #\= 0, R#\= 0,
    labeling(Zoznam).
```

Tento kryptogram má práve jedno riešenie. Odmerať čas výpočtu, ktorý potrebuje procesor na vyhodnotenie daného volania, možno použitím zabudovaného predikátu *time/1*, ktorý je definovaný v knižnici *util*. V tomto prípade pôjde o necelú sekundu.

```
[eclipse 5]: lib(util).
util.pl    compiled traceable 6620 bytes in 0.01 seconds

yes.
[eclipse 6]: time(dgr(L)).

Success, time = 0.761

L = [5, 2, 6, 4, 8, 1, 9, 7, 3, 0]
yes.
```

Tento čas je možné podstatne skrátiť zmenou reprezentácie úlohy. Ohraničenie vyjadrujúce súčet totiž obsahuje príliš veľa premenných a algoritmy hranovej konzistencie počas fázy šírenia ohraničení nedokážu dobre využiť takéto ohraničenia na dedukciu vedúcu k zmenšeniu domén. V konečnom dôsledku sa tým predlžuje fáza prehľadávania a znižuje pozitívny efekt CLP.

Preto je z hľadiska efektívnosti výpočtu výhodnejšie formulovať ohraničenia podľa možnosti tak, aby obsahovali menší počet premenných, lebo tieto skôr povedú k redukcii domén (pri znížení počtu neviazaných premenných na dve alebo jednu). V úlohách kryptogramov je to možné dosiahnuť zmenou formulácie ohraničení tak, že súčet sa reprezentuje ako konjunkcia súčtov pre jednotlivé rády s prihliadnutím na prenosy. V prípade spomínaného kryptogramu ide o 5 rádov (5 cifier v každom čísle zakódovanom písmenami), takže je nutné zaviesť ďalších 5 premenných $P1, \ldots, P5$ s doménami $[0, 1]$. $P1$ bude znamenať prenos z prvého rádu do druhého atď.

```
dgr1(Zoznam) :-
    Zoznam = [D, O, N, A, L, G, E, R, B, T],
    Zoznam :: 0..9,
    Prenosy = [P1, P2, P3, P4, P5],
    Prenosy :: [0,1],
    2*D #= T + 10*P1,
    2*L + P1 #= R + 10*P2,
    2*A + P2 #= E + 10*P3,
    N + R + P3 #= B + 10*P4,
    O + E + P4 #= O + 10*P5,
    D + G + P5 #= R,
    alldifferent(Zoznam),
    D #\= 0, G #\= 0, R#\= 0,
    labeling(Zoznam).
```

Volanie predikátu *dgr1/1* dá nasledovnú odpoveď:

```
[eclipse 7]: time(dgr1(L)).

Success, time = 0.0899999999999999

L = [5, 2, 6, 4, 8, 1, 9, 7, 3, 0]
yes.
```

V tomto prípade je výpočet viac ako osemkrát rýchlejší. Je to práve zásluhou formulácie úlohy pomocou ohraničení nižšej árnosti. Tieto vedú k lepšiemu orezávaniu domén najmä počas fázy prehľadávania. Prejaví sa to v menšom počte návratov (t.j. nekonzistentných priradení hodnôt premenným). Počet návratov je možné sledovať po úprave predikátu *labeling/1*, ktorá je uvedená v [118]. V prípade prvej reprezentácie (dgr/1) bude potrebných 1020 návratov na nájdenie jediného riešenia, kým pri použití druhej reprezentácie (dgr1/1) iba 39 !

7.2 Investičná úloha

Operačný výskum a logistika sú nevyčerpateľným zdrojom zaujímavých kombinatorických úloh. Mnohé z nich boli vyriešené pomocou jazykov CLP ([47], [72]).

Všetky doteraz uvedené problémy viedli prirodzene na úlohu nájdenia jedného, resp. všetkých riešení. Nasledujúca úloha z oblasti logistiky si vyžaduje už optimalizáciu. Ide o tzv. *investičnú úlohu.*

Podnik hodlá uskutočniť 5 investičných akcií označených poradovými číslami 1 až 5 s nákladmi (v peňažných jednotkách, PJ) a zodpovedajúcimi ziskami uvedenými v tabuľke 7.1. Avšak investičný limit podniku na daný rok činí $130PJ$ a teda podnik nemôže uskutočniť všetky akcie. Je potrebné vybrať tie investičné akcie, ktoré prinesú podniku maximálny priemerný ročný zisk.

Investičná akcia	1	2	3	4	5
Náklady	40	70	30	50	20
Priemerný ročný zisk	4	5	2	3	1

Tabuľka 7.1: Vstupné údaje pre investičný problém

Prirodzenými premennými v tejto úlohe sú investičné akcie. O každej potenciálnej investičnej akcii je potrebné rozhodnúť, či sa v danom roku zrealizuje alebo nie (t.j. zodpovedajúca premenná môže nadobúdať dve možné hodnoty). V tomto konkrétnom prípade teda ide o 5 dvojhodnotových premenných. Ohraničenie je len jedno, týkajúce sa neprekročenia úhrnných investičných nákladov, ktoré má podnik pre daný rok k dispozícii.

Ak premenné budú nadobúdať hodnotu 0 v prípade, že sa investičná akcia nevyberie a 1 v prípade, že sa vyberie, potom úhrnné investičné náklady možno vypočítať násobením premennej s nákladmi pre danú akciu a ich súčtom pre všetky akcie. Nakoniec ostáva ešte optimalizácia riešenia. V tomto prípade teda nestačí použiť len predikát *labeling/1* pre prehľadávanie, ale je nutné ho vložiť do optimalizačného predikátu, napr. $min\_max/2$, spolu s kriteriálnou funkciou. Tou je v tomto prípade zisk.

Zisk je možné vypočítať podobne ako náklady, ale násobením premenných ziskom pre jednotlivé akcie a ich spoločným súčtom. Keďže optimalizačné predikáty v ECL^iPS^e zabezpečujú stále minimalizáciu a zisk je potrebné maximalizovať, pre dosiahnutie želaného efektu maximalizácie zisku je nutné v kriteriálnej funkcii uvažovať záporné hodnoty ziskov.

```
:- lib(fd).

investicny(L) :-
    L = [X1,X2,X3,X4,X5], % päť premenných
```

```
    L :: [0,1], % Domény dvojhodnotové, 0 alebo 1
    40*X1 + 70*X2 + 30*X3 + 50*X4 + 20*X5 #<= 130,
        % ohraničenie úhrnných investičných nákladov
    Zisk #= -4*X1 - 5*X2 - 2*X3 - 3*X4 - X5,
        % kriteriálna funkcia je celkový dosiahnutý zisk
    min_max(labeling(L), Zisk).
        % optimalizácia metódou vetvenia a medzí
```

Priebeh riešenia je potom nasledovný.

```
[eclipse 2]: investicny(L).
Found a solution with cost 0
Found a solution with cost -1
Found a solution with cost -3
Found a solution with cost -4
Found a solution with cost -5
Found a solution with cost -6
Found a solution with cost -8
Found a solution with cost -9
Found a solution with cost -10

L = [1, 1, 0, 0, 1]
yes.
```

Optimálne riešenie s maximálnym ziskom $10PJ$ je teda možné dosiahnuť výberom investičných akcií s poradovými číslami 1, 2 a 5.

Aby sa dosiahla všeobecná použiteľnosť programu na riešenie investičných úloh ľubovoľného rozsahu, je užitočné upraviť zdrojový text programu tak, aby stačilo zadať rozmer úlohy a potrebné parametre investičných akcií by sa načítali z nejakého zdroja údajov, napr. zo štruktúry typu *investicny/4* ktorú vráti volanie predikátu *data/1*.

```
:- local struct (investicny(pocet,investicie,zisky,limit)).

data(X) :-
    X = investicny with [ pocet:5,
                          investicie:[40,70,30,50,20],
                          zisky:[4,5,2,3,1],
                          limit:130
                        ].
```

```
ries_investicny :-
    data(X),
        % vstup údajov
    arg(pocet of investicny,X, N),
    gen_premenne(N,[0,1],L),
        % definícia premenných
    arg(investicie of investicny, X, Investicie),
    vytvor_zoznam_nakladov(Investicie, L, LOC),
    arg(zisky of investicny, X, zisky),
    vytvor_zoznam_ziskov(zisky, L, LOP),
    arg(limit of investicny, X, Max),
    sum(LOC) #<= Max,
        % ohraničenie investičných nákladov
    Zisk #= sum(LOP),
        % definícia kriteriálnej funkcie
    min_max((labeling(L),write(L),write(' ')), Zisk).
        % optimalizácia s výpisom každého nájdeného riešenia
```

Tento program využíva tri pomocné predikáty na tvorbu vstupných štruktúr ktorých veľkosť závisí od zadaných údajov, známych až v čase behu programu. Prvý z nich (*gen_premenne(+N, +Domena, -Zoznam)*) je určený na vytvorenie potrebného počtu ($+N$) stavových premenných a inicializáciu ich domén na $+Domena$. Premenné vráti vo forme zoznamu $-Zoznam$.

```
gen_premenne(0,_,[]) :- !.
gen_premenne(N,Domena,[X|T]) :-
    X :: Domena,
    N1 is N-1,
    gen_premenne(N1,Domena,T).
```

Ďalšie dva pomocné predikáty sú navzájom veľmi podobné, vytvárajú zoznam ktorého prvky budú tvoriť súčtový výraz. Tento výraz je ďalej možné použiť ako jednu stranu aritmetického ohraničenia alebo na vyjadrenie kriteriálnej funkcie.

V oboch prípadoch sú prvé dva argumenty vstupné, prvý je zoznam číselných konštánt, druhý zoznam doménových premenných a tretí je potom výstupný zoznam jednotlivých členov súčtu, ktorý je potom možné veľmi jednoducho vytvoriť zabudovaným makrom *sum/1* (viď. hlavný program).

```
vytvor_zoznam_ziskov([], [], []).
vytvor_zoznam_ziskov([P|RP], [Var|RV], [-Var*P|RLOP]) :-
    vytvor_zoznam_ziskov(RP, RV, RLOP).

vytvor_zoznam_nakladov([], [], []).
vytvor_zoznam_nakladov([C|RC], [Var|RV], [Var*C|RLOC]) :-
    vytvor_zoznam_nakladov(RC, RV, RLOC).
```

Dosiahnuté výsledky sú prirodzene zhodné, akurát pribudol výpis každého nájdeného riešenia doplnením volania predikátu *write/1* do vnútra optimalizačného predikátu *min_max/2*.

```
[eclipse 7]: ries_investicny.
[0, 0, 0, 0, 0] Found a solution with cost 0
[0, 0, 0, 0, 1] Found a solution with cost -1
[0, 0, 0, 1, 0] Found a solution with cost -3
[0, 0, 0, 1, 1] Found a solution with cost -4
[0, 0, 1, 1, 0] Found a solution with cost -5
[0, 0, 1, 1, 1] Found a solution with cost -6
[0, 1, 0, 1, 0] Found a solution with cost -8
[1, 0, 1, 1, 0] Found a solution with cost -9
[1, 1, 0, 0, 1] Found a solution with cost -10
yes.
```

7.3 Rozvrhovanie

Pri úlohách časového rozvrhovania je voľba premenných jednoznačná. Pre každú úlohu (operáciu) ktorej začiatok je potrebné naplánovať v čase je nutné zaviesť aspoň jednu premennú ktorá bude reprezentovať čas začiatku tejto úlohy (operácie). V prípade premenlivej dĺžky trvania niektorých úloh (operácií) potrebujeme ešte jednu premennú pre každú takúto úlohu (operáciu), ktorá bude označovať čas jej trvania, prípadne čas jej ukončenia.

Ďalšie dodatočné informácie nutné k reprezentácii danej úlohy (operácie) sa obvykle spolu s vyššie uvedenými premennými dávajú do spoločnej štruktúry. Ide najmä o informácie typu: identifikačné označenie danej úlohy, na ktorom stroji má byť vykonaná (ktorý zdroj je potrebný pre jej uskutočnenie), trvanie úlohy a pod.

Vo všeobecnosti môže byť úloha komplikovaná ďalšími skutočnosťami, ako napr. viaceré alternatívne zdroje pre vykonanie jednej úlohy (operácie).

V takom prípade potrebujeme pre túto úlohu ešte jednu premennú, ktorá bude reprezentovať použitý zdroj. Jej počiatočná doména bude obsahovať množinu všetkých zdrojov, ktoré pre túto úlohu (operáciu) prichádzajú do úvahy.

Základ ale tvoria obvykle premenné reprezentujúce začiatky jednotlivých úloh (operácií), ktoré je potrebné rozvrhnúť v čase. Ich počiatočné domény sú preto závislé od požadovanej presnosti rozvrhu (hodiny, smeny, dni a pod.). Vzhľadom na túto presnosť je potom možné diskretizovať interval rozvrhu a stanoviť tak konečné domény pre jednotlivé premenné napr. ako interval $0..Limit$. $Limit$ možno najjednoduchšie stanoviť ako súčet dĺžok trvaní všetkých úloh ktoré je nutné rozvrhnúť, prípadne zväčšený o predpísané časové odstupy medzi úlohami.

V niektorých prípadoch, ak je požadovaná presnosť vysoká a reprezentovaný interval dlhý, je možné použiť aj reprezentáciu pomocou reálnych premenných. Obe hranice je však často možno ďalej priblížiť a tým zúžiť počiatočné domény premenných (a teda skúmanú časť priestoru prehľadávania) rôznymi heuristikami na odhad hornej a dolnej hranice nákladov.

Ohraničenia vyskytujúce sa pri úlohách rozvrhovania možno rozdeliť v zásade do dvoch skupín:

1. precedenčné ohraničenia,
2. disjunktné ohraničenia.

Precedenčné ohraničenia vyjadrujú jednoznačné časové relácie medzi dvojicami premenných (začiatky úloh). Tieto je možné rozčleniť na niekoľko základných typov, ako to definoval už [5]. Pre prehľad niektorých typov časových relácií a spôsobu ich reprezentácie v CLP (na príklade jazyka ECL^iPS^e) predpokladajme nech `A` a `B` sú operácie, pričom premenné so zodpovedajúcimi názvami budú označovať čas začiatku príslušnej operácie. Trvania operácií budú pevne zadané a označené T_A resp. T_B.

- `A` po `B` - počiatočný čas úlohy `A` musí byť väčší nanajvýš rovný času ukončenia úlohy `B`:

  ```
  A #>= B + TB
  ```

- `A` pred `B` - čas ukončenia úlohy `A` musí byť menší nanajvýš rovný počiatočnému času úlohy `B`:

  ```
  A + TA #=< B
  ```

- A nasleduje po B - počiatočný čas úlohy A musí byť rovný času ukončenia úlohy B:

  ```
  A #= B + TB
  ```

- A prekrýva B - počiatočný čas úlohy B musí byť väčší ako počiatočný čas úlohy A a čas ukončenia úlohy B musí byť väčší ako čas ukončenia úlohy A:

  ```
  B #> A,
  B + TB #> A + TA
  ```

- A je obsiahnutá v B - počiatočný čas úlohy B musí byť menší ako počiatočný čas úlohy A a čas ukončenia úlohy B musí byť väčší ako čas ukončenia úlohy A:

  ```
  B #< A,
  B +TB #> A + TA
  ```

- A ukončuje B - čas ukončenia úlohy A je rovný času ukončenia úlohy B a počiatočný čas úlohy A je väčší ako počiatočný čas úlohy B:

  ```
  A + TA #= B + TB,
  A #> B
  ```

  Uvedené časové relácie medzi dvoma úlohami (operáciami) A, B môžu byť ešte doplnené o presné časové odstupy. Napríklad ak úloha B môže začať najskôr 3 časové jednotky po skončení úlohy A, možno to zapísať ako numerické ohraničenie

  ```
  B #>= A + TA + 3
  ```

Iný charakter majú ohraničenia vyjadrujúce zdieľanie zdrojov viacerými úlohami. Takéto ohraničenia nazývame *disjunktné.* Tieto sa používajú dvojakým spôsobom.

1. Ako nezávislé alternatívy (body výberu), ktoré sa skúmajú v priebehu prehľadávania.

2. Ako ohraničenia zabezpečujúce väčšiu mieru šírenia informácie v nich obsiahnutej. V tomto smere existuje niekoľko alternatív s rôznou mierou získanej informácie a časovej náročnosti [104].

Prvý spôsob použitia je záväzný a slúži pri hľadaní riešenia vlastne tým, že vyberá postupne čiastkové usporiadania jednotlivých dvojíc úloh zdieľajúcich ten istý zdroj. Druhý spôsob je ľubovoľný a priamo závislý od charakteru a veľkosti konkrétneho rozvrhovacieho problému, ktorý je potrebné riešiť. Na základe výsledkov mnohých testov, ktoré boli vykonané s rôznymi implementovanými spôsobmi reprezentácie disjunktných ohraničení v ECL^iPS^e [104] na rozvrhovacích úlohách rôznej veľkosti a charakteru, možno povedať, že pre úlohy malého rozsahu (menej ako 50 operácií)[2] je úplne postačujúce použiť len prvý spôsob reprezentácie disjunkcií ako nezávislých alternatív. Čas potrebný na šírenie dômyselnejších spôsobov reprezentácie je totiž dlhší než prehľadanie celého priestoru.

S narastajúcou zložitosťou úloh sa zväčšuje aj význam druhej skupiny reprezentácií disjunktných ohraničení. Pre stredné úlohy (50 - 100 operácií) ešte úplne postačuje menej náročné šírenie ohraničení pomocou hranovej B-konzistencie[3] prípadne úplnej hranovej konzistencie. Tá sa však ukázala menej účinná pre také úlohy časového rozvrhovania, kde rozhodujúci význam má redukcia domén na ich hraniciach, nie vo vnútri.

Pre veľké rozvrhovacie úlohy (stovky operácií) je prakticky nevyhnutné použiť ohraničenie s najlepším šírením *disjunctive/3*[4]. Naviac tu hrá už veľmi významnú rolu aj počiatočné zúženie intervalu trvania rozvrhu pomocou rôznych heuristík.

Vo väčšine prípadov je cieľom nájsť najkratší možný rozvrh, takže optimalizačným kritériom sa stáva čas ukončenia poslednej úlohy (operácie). Najlepšie je pridať jednu premennú (napr. `K`), ktorú si možno predstaviť ako fiktívnu úlohu nulovej dĺžky, ktorá musí byť vykonaná ako posledná. Takže pre každú úlohu (operáciu) `X`, ktorú je potrebné rozvrhnúť, je nutné pridať aj ohraničenie `X + TX #< K`.

Potom ako optimalizačnú funkciu je možné použiť práve hodnotu `K`. Nakoľko ide tiež o doménovú premennú s rovnakou počiatočnou doménou ako

[2]Zložitosť úlohy nie je daná len počtom operácií, ktoré je potrebné rozvrhnúť, ale aj ohraničeniami medzi nimi. Preto aj uvedené počty úloh sú len orientačné.

[3]Hranová B-konzistencia zabezpečuje nižší stupeň konzistencie, ako (úplná) hranová, nakoľko uvažuje len dolné a horné hranice domén premenných, nie všetky hodnoty z ich domén.

[4]Toto symbolické ohraničenie v sebe integruje efektívne algoritmy z oblasti operačného výskumu vyvinuté pre úlohy typu job-shop [104].

majú všetky ostatné premenné reprezentujúce počiatočné časy úloh, je potrebné minimalizovať jej minimálnu možnú hodnotu (t.j. spodnú hranicu jej domény).

Vzhľadom na vyššie uvedené ohraničenia časov ukončenia všetkých úloh, ktoré sú viazané na hodnotu premennej `K`, je možné jej znižovaním priamo ovplyvňovať všetky zainteresované premenné a tým aj dĺžku výsledného rozvrhu. To je aj spôsob akým sa hľadá optimálne riešenie. Ak bol nájdený rozvrh s dĺžkou `L1` (`K` = `L1`), potom sa v nasledujúcom cykle hľadá riešenie s dĺžkou `L2 < L1`.

Okrem deterministického výpočtu najnižšej hodnoty `L1` (napríklad ako súčet dĺžok trvania všetkých úloh) znamená často veľký prínos rýchle heuristické vygenerovanie prvého (suboptimálneho) riešenia, čím sa dosiahne pomerne nízka hodnota `L1` (horná hranica `K`) hneď na začiatku výpočtu.

Význam tohoto kroku narastá so zložitosťou riešenej rozvrhovacej úlohy. Čím zložitejšia úloha, tým väčší priestor prehľadávania a tým časovo náročnejší každý krok vedúci k skráteniu výsledného rozvrhu (znižovanie hodnoty optimalizačnej funkcie). Z toho nutne vyplýva, že čím bližšie bude prvé vygenerované riešenie k skutočnému optimálnemu riešeniu, tým väčšiu šanca že bude optimálne riešenie nájdené v prijateľnom čase resp. že bude nájdené také riešenie, ktoré bude dostatočne blízke optimu.

Vygenerovať počiatočné riešenie je možné pomerne ľahko deterministickým algoritmom [23]. Rozvrh sa vytvára postupne tak, že sa vyberajú úlohy (operácie) jedna za druhou a každá z nich začne tak skoro, ako je to len možné.

V každom kroku je k dispozícii množina úloh, z ktorých je ešte možné vybrať nasledujúcu. Na začiatku sú v tejto množine všetky úlohy (operácie). V každom kroku sa vyberie jedna z týchto úloh a priradí sa jej najskorší možný začiatok. Táto úloha sa potom vyberie zo spomínanej množiny. Celý postup sa potom opakuje tak dlho, až kým sa všetky úlohy z množiny nevyberú. Ťažisko algoritmu teda leží v pravidle, podľa ktorého sa vyberá úloha z množiny ešte nerozvrhnutých úloh.

Podľa výsledkov experimentov popísaných podrobnejšie v [104], možno doporučiť pre tento krok novo navrhnutú heuristiku EST+, ktorá preferuje úlohy (operácie) s najskorším možným časom začiatku a v prípade zhody u viacerých úloh vyberie tú, ktorá z nich má najdlhšiu zvyškovú prácu (t.j. súčet trvaní úloh, ktoré ešte musia byť vykonané za vybranou úlohou).

Okrem odhadu hornej hranice, čo predstavuje nájdenie čo možno najlepšieho prvého riešenia, môže prispieť k zúženiu priestoru prehľadávania aj dobrý odhad dolnej hranice. Tu ide o odhad doby trvania, pod ktorú sa určite nedá už rozvrh stihnúť. Pre odhad dolnej hranice `K` sa používa klasický postup, ktorý spočíta dĺžky trvaní jednotlivých úloh (operácií) pre jednotlivé zdroje (stroje) a pre jednotlivé výrobky. Za dolnú hranicu sa potom vyberie najdlhší z nich. Znamená to, že rozvrh nemôže byť kratší ako súčet trvaní všetkých úloh na tom zdroji, ktorý bude najviac využívaný, resp. ako súčet trvaní všetkých operácií toho výrobku, ktorý ho má najväčší.

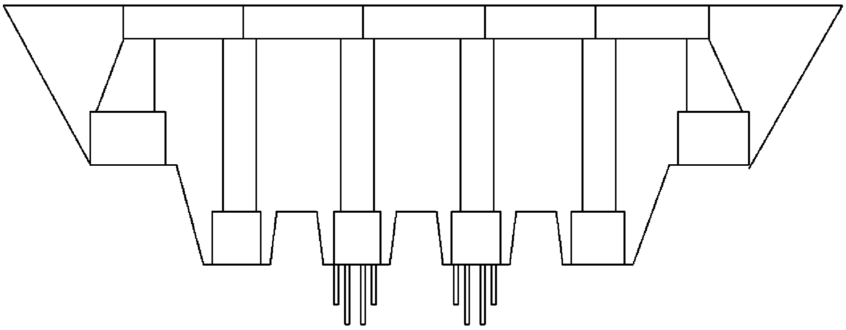
Poslednou etapou riešenia úloh rozvrhovania je iteratívne vylepšovanie najlepšieho nájdeného riešenia tým, že sa snažíme nájsť lepšie (t.j. kratší rozvrh). Obvyklý postup implementovaný v jazykoch CLP postupne znižuje hornú hranicu buď po jednotkách (ak hľadáme optimálny rozvrh), alebo po určitých krokoch (ak hľadáme rozvrh s určitou toleranciou vzdialenosti od optimálneho riešenia). Podrobnejšie bol tento postup popísaný v predchádzajúcej časti.

Novo navrhnuté postupy logaritmický min_max a logaritmický minimize viedli v drvivej väčšine uskutočnených testov k omnoho rýchlejšiemu nájdeniu optimálneho riešenia. Naviac, aj u úloh s náročným dôkazom optimálnosti sú veľmi účinné v tom zmysle, že v každom kroku podávajú aj informáciu o maximálnej možnej vzdialenosti najlepšieho nájdeného riešenia od optima a u oboch metód je počet medzikrokov vedúcich od počiatočného heuristicky nájdeného riešenia k optimu malý v dôsledku delenia intervalu nákladov na polovicu.

V tejto časti je uvedený náčrt programu pre nájdenie minimálneho časového rozvrhu prác na stavbe päťsegmentového mostu (viď obr. 7.1). Tento projekt pozostáva zo 46 úloh (pre každú je definované jej trvanie a zdroje potrebné pre jej vykonanie) a množiny ohraničení (úplnú definíciu problému možno nájsť v [72]).

Okrem precedenčných ohraničení (ako napr. že výkop pre daný pilier musí predchádzať betonárskym prácam) musia byť splnené aj disjunktné ohraničenia vyplývajúce z ohraničeného počtu zdrojov, ktoré musia byť zdieľané pri viacerých úlohách (napríklad že jediná betonárska čata ktorá je k dispozícii môže v jednom okamihu pracovať maximálne na betónovaní jedného piliera) a ešte skupina dodatočných ohraničení (ako napr. že medzi ukončením daného debnenia a zodpovedajúcim betónovaním základov môžu uplynúť nanajvýš 4 dni).

Každú úlohu `I` je možné reprezentovať štruktúrou `[NazovI, TrvanieI, StartI]`, s názvom úlohy (`NazovI`), dĺžkou jej trvania (`TrvanieI`) a dňom začiatku jej vykonávania (`StartI`). Začiatok celého projektu bude v čase 0 a jeho ukončenie v dni `Koniec` (rovná sa ukončeniu poslednej z vykonávaných úloh).



Obr. 7.1: Náčrt pätsegmentového mosta.

Úlohou je teda nájsť také priradenia hodnôt premenným `StartI` u jednotlivých úloh, aby boli splnené všetky vyššie uvedené ohraničenia a aby sa minimalizovalo trvanie celého projektu `Koniec`. Základná schéma programu, ktorý toto zabezpečí, je nasledovná.

```
most(L, Koniec) :-
    zadanie_ulohy(L),
    definuj_domeny(L, Koniec),
    stanov_ohraničenia(L),
    min_max(vyber(L, Koniec), Koniec).
```

Najskôr sa načítajú vstupné údaje týkajúce sa úloh, ktoré majú byť vykonané v rámci projektu a na ich základe sa vytvorí zoznam štruktúr `L` jednotlivých úloh (predikát `zadanie_ulohy(L)`).

V ďalšom kroku sa priradia premenným označujúcim začiatok vykonávania úlohy (v štruktúre vyššie označené ako `StartI`) počiatočné domény, t.j. počiatočný rozsah, v rámci ktorého sa určite bude vyskytovať skutočný čas započatia tej ktorej úlohy vo výslednom rozvrhu. Túto doménu možno na začiatku jednoznačne ohraničiť na interval ohraničený zdola nulou (začiatok projektu) a zhora hodnotou súčtu dĺžok trvania všetkých úloh (`Max_trvanie`). To zodpovedá najjednoduchšej možnej realizácii projektu ako postupnosti všetkých úloh za sebou, bez akejkoľvek súbežnosti. Stanovenie domény bude v CLP programe dané `StartI :: 0..Max_trvanie`.

Predikát `definuj_domeny(L, Koniec)` slúži pre inicializáciu domén počiatočných časov pre jednotlivé úlohy v zozname `L`, ako aj domény premennej reprezentujúcej celkovú dĺžku projektu `Koniec`.

Na takto inicializované doménové premenné možno teraz aplikovať ohraničenia z definície úlohy. V rámci predikátu `stanov_ohraničenia(L)` sa definujú precedenčné a dodatočné ohraničenia. Skutočnosť, že úloha `[NazovI, StartI, TrvanieI]` musí byť vykonaná ešte pred úlohou `[NazovJ, StartJ, TrvanieJ]` možno v CLP vyjadriť ako `StartJ #>= StartI + TrvanieI`.

U disjunktných ohraničení je situácia trocha iná, nakoľko ich určenie sa nedá urobiť deterministicky. Ak totiž máme rozvrhnúť dve úlohy (povedzme so začiatočnými časmi `SI`, resp. `SJ` a dĺžkami trvaní `TI`, resp. `TJ`), ktoré musia zdieľať spoločný zdroj, potom máme dve možnosti. Alebo bude v rozvrhu najprv prvá úloha, a potom druhá nemôže začať skôr, kým neskončí prvá, alebo opačne. Túto skutočnosť možno reprezentovať nasledovným nedeterministickým predikátom

```
disjunktivne(SI, SJ, TI, TJ) :-
    SI #>= SJ + TJ.
disjunktivne(SI, SJ, TI, TJ) :-
    SJ #>= SI + TI.
```

ktorý je súčasťou definície predikátu `stanov_disjunkcie(L)` z nasledujúcej definície `vyber(L, Koniec)` (viď. definícia hlavného programu `most/2`).

```
vyber(L, Koniec) :-
    stanov_disjunkcie(L),
    indomain(Koniec)
```

Predikát `vyber(L, Koniec)` okrem stanovenia disjunktných ohraničení naviac spúšťa aj prehľadávanie tým, že zvolí hodnotu z aktuálnej domény premennej `Koniec`.

Nakoniec je potrebné nájsť optimálne riešenie, čo je v tomto prípade riešenie s minimálnou dĺžkou projektu `Koniec`. Túto funkciu zabezpečuje predikát `min_max(vyber(L, Koniec), Koniec)` v hlavnom programe `most/2`, ktorý pre daný cieľ (jeho prvý argument, t.j. v našom prípade `vyber(L, Koniec)`) nájde také riešenie, pre ktoré hodnota druhého argumentu (v našom prípade `Koniec`) je minimálna.

7.4 Distribučná úloha

Zovšeobecnená distribučná úloha spočíva v optimálnom pridelení u - distribučných miest k v - odberateľom tak, aby sa vyhovelo požiadavkám odberateľov pri minimálnych nákladoch.

Výrobný podnik dodáva tovar svojim zákazníkom (každý z v zákazníkov má svoju požiadavku na veľkosť dodávky $b_j, 1 \leq j \leq v$) a má k dispozícii u miest kde je možné vybudovať distribučné centrum. Pre zriadenie každého distribučného centra sú potrebné nejaké fixné náklady ($f_i, 1 \leq i \leq u$). Okrem toho sú tu samozrejme aj prepravné náklady na prepravu požadovaných množstiev tovarov z distribučných centier k zákazníkom. Nech c_{ij} predstavuje jednotkové prepravné náklady medzi i-tym distribučným centrom a j-tym zákazníkom. Úlohou je určiť počet a umiestnenie distribučných centier tak, aby sa minimalizovali celkové náklady (t.j. fixné aj prepravné).

Táto úloha môže byť formulovaná ako úloha celočíselného programovania. Ak je k dispozícii balík na riešenie úloh celočíselného programovania (ako napr. Visual XPRESS MP, CPLEX a pod.), potom stačí úlohu preformulovať do množiny celočíselných premenných a množinu lineárnych rovníc, resp. nerovníc nad týmito premennými. V prípade zovšeobecnenej distribučnej úlohy je potrebné definovať $u * (v + 1)$ premenných. Pre každú z u možných lokalít máme $v + 1$. Jedna binárna premenná s doménou $[0, 1]$ bude označovať či sa daná lokalita vyberie pre distribučné centrum, alebo nie, ostatné premenné označujú ktorým zákazníkom bude dané distribučné centrum dodávať tovar.

Veľkosť priestoru prehľadávania bude teda $2^{u(v+1)}$ čo vedie v prípade reálnych úloh (napr. 20 možných lokácií a 80 zákazníkov ako tomu bolo v úlohe popísanej v [73]) na obrovské úlohy celočíselného programovania. A to je veľmi častý prípad pri tomto prístupe k riešeniu. Všeobecnou formuláciou sa strácajú cenné informácie o špecifikách úlohy a to vedie k veľkej redundancii pri prehľadávaní.

Ďalšou možnosťou ako úlohu riešiť je navrhnúť špeciálny program v procedurálnom programovacom jazyku. Autori v [73] pritom vychádzali z uvažovania o možných lokalitách pre distribučné centrum. Ak už boli raz vybrané miesta, kde sa distribučné centrá zriadia, potom priradenie zákazníkov distribučným centrám je už triviálnou úlohou (jednoducho pre každého zákazníka sa vyberie najbližšie distribučné centrum). Priestor prehľadávania bol v tomto prípade už len 2^u.

Tento prístup je veľmi vhodný, pretože umožňuje využiť špeciálne vlastnosti úlohy. Situácia je ale odlišná keď vezmeme do úvahy samotné programovanie. Jadro programu spomínaného v [73] obsahovalo 2000 riadkov v Pascale a vyžiadalo si niekoľko mesiacov vývoja. Zmena heuristiky, alebo pridanie nového ohraničenia by spôsobilo značné programátorské nároky. Už len pridanie disjunktných ohraničení medzi lokalitami (napríklad že len jedna z dvoch môže byť vybraná vo výslednom riešení) vyžaduje komplexné zmeny v programe. Pridanie kapacitných ohraničení pre distribučné centrá by si vyžiadalo kompletnú zmenu programu.

Autori v [73] prezentujú ako ďalšiu alternatívu použitie CLP, konkrétne jazyka CHIP. Podobné riešenie možno realizovať aj v jazyku ECL^iPS^e.

```
distribucny1(O, DC):-
    ohranic(O, DC, Naklady),
    min_max((generuj(DC,O),nl,write(DC),nl,write(O)),Naklady).
```

Prvý predikát (`ohranic`) vo svojom tele definuje nasledovné:

- Zoznamy doménových premenných. `DC` je zoznam dvojhodnotových (0, resp. 1) premenných V_{1i} $(1 \leq i \leq u)$ pre jednotlivé lokality (potenciálne distribučné centrá). `O` je zoznam doménových premenných V_{2j} $(1 \leq j \leq v)$ s celočíselnými doménami v intervale $1..v$ pre zákazníkov (odberateľov). Nakoniec sú to aj dodatočné doménové premenné V_{3j} pre každého zákazníka reprezentujúce prepravné náklady spojené s dodávkou tovaru zákazníkovi j.

- Ohraničenia medzi týmito premennými. Pre stanovenie prepravných nákladov pre jednotlivých zákazníkov je použité symbolické ohraničenie *element(?I, +L, ?El)* (platí vtedy a len vtedy ak $?I$-ty prvok v $+L$ je rovný $?El$). Takže pre každého zákazníka $(j = 1, \ldots, v)$ je potrebné definovať jedno ohraničenie určujúce s ním spojené prepravné náklady:

  $$element(V_{2j}, [c_{1j}, \cdots, c_{kj}, \cdots, v_{uj}], V_{3j}).$$

  Okrem toho je ešte nutné aj ohraničenie ktoré zabezpečí logické prepojenie premenných zo zoznamov `DC` a `O`. Toto prepojenie spočíva v tom, že ak raz daná lokalita nie je vybraná pre distribučné centrum, potom žiaden zákazník z nej nesmie odoberať tovar. Pre tento účel je definovaný nasledovný predikát *odstran/2*.

```
odstran(DC, O):-
        % nastav aktuálnu pozíciu zodpovedajúcu lokalite na 1
    odstran(DC,O,1).

odstran([],_,_).
odstran([H|T],O,N):-
        % odstráň z doménových premenných O hodnotu N v
```

```
        % prípade ak daná lokalita je zamietnutá, t.j. H = 0
    o(H,O,N),
        % prejdi na v poradí ďalšiu lokalitu
    N1 is N+1,
        % rekurzívne pokračuj ďalej
    odstran(T,O,N1).

o(0,O,N):-  % ak bola lokalita zamietnutá
        % odstráň jej poradové číslo z domén
        % premenných v O
    dvar_zoz(O,N).
o(1,_,_).   % ak bola lokalita vybraná, nerob nič

dvar_zoz([],_).
dvar_zoz([O|T],N):-
        % zabudovaný predikát pre odstránenie
        % prvku N z domény premennej O
    dvar_remove_element(O,N),
        % rekurzívne aj pre ostatné premenné z T
    dvar_zoz(T,N).
```

Druhý predikát v hlavnom programe (*min_max/2*) realizuje optimalizáciu počas prehľadávania zabezpečeného generovaním hodnôt v rámci predikátu *generuj/2*, ktorý je definovaný nasledovne.

```
generuj(X,Y):-
    labeling(X),
    labeling(Y).
```

Pritom kriteriálna funkcia je definovaná ako nasledovný súčet:

$$f_1 * V_{11} + ... + f_u * V_{1u} + V_{31} + ... + V_{3n}.$$

Úplný zdrojový text univerzálneho programu (s oddelenou údajovou časťou) okrem predikátov, ktoré už boli definované vyššie je nasledovný:

```
:-lib(fd).

:- local struct( distribucny(lokality,
                 fixne_naklady,
```

```
                 zakaznici,
                 dodavky,
             % nasleduje deklarácia matice s rozmermi v, u
             % pre uloženie vzdialeností zákazníkov od
             % jednotlivých lokalít pre distribučné centrum
                 vzdialenosti: dim(zakaznici,lokality)) ).

data(Data) :-
    Data = distribucny with [lokality:3,
                 fixne_naklady:[300,200,400],
                 zakaznici:5,
                 dodavky:[50,70,30,80,60],
                 vzdialenosti: []([](5,9,2),
                          [](3,6,4),
                          [](8,1,6),
                          [](4,3,8),
                          [](2,5,3))
                ].

distribucny1(O, DC):-
    ohranic1(O, DC, Naklady),
    min_max((generuj(DC,O),nl,write(DC),nl,write(O),
                          write(' ')),  Naklady).

ohranic1(O, DC, Naklady) :-
    data(Data), % načítanie vstupných údajov
        % zistenie počtu M alternatívnych lokalít
        % pre distribučné centrá
    arg(lokality of distribucny, Data, M),
        % vygenerovanie M binárnych doménových premenných
        % a ich uloženie do zoznamu DC
    gen_premenne(M, [0,1], DC),
        % zistenie poctu N zákazníkov
    arg(zakaznici of distribucny, Data, N),
        % vytvorenie N doménových premenných s doménami
        % 1..N a ich uloženie do zoznamu O
    gen_premenne(N, 1..M, O),
        % načítanie matice vzdialeností do V
    arg(vzdialenosti of distribucny, Data, V),
```

```
        % vygenerovanie ohraničení pre prepravné náklady
    gen_ohranicenia(O, V, PN, 1, M),
        % načítanie vektora fixných nákladov do F
    arg(fixne_naklady of distribucny, Data, F),
        % vytvorenie zoznamu násobkov fixných nákladov
        % a binárnych premenných pre všetky lokality
    vytvor_zoznam_nakladov(F,DC,LOC_F),
        % načítanie vektora požiadaviek zákazníkov do D
    arg(dodavky of distribucny, Data, D),
        % vytvorenie zoznamu prepravných nákladov
    vytvor_zoznam_nakladov(D,PN,LOC_T),
        % uspanie ohraničenia pre prepojenie zoznamov
        % premenných DC a O
    suspend(odstran(DC,O), 4, DC -> inst),
        % definovanie celkových nákladov
    Naklady #= sum(LOC_F) + sum(LOC_T).

gen_ohranicenia([],_,[],_,_).
gen_ohranicenia([Oi|RO],V,[PNi|RPN],I,M) :-
        % do X vlož I-ty riadok matice vzdialeností,
        % t.j. vzdialenosti od I-teho zákazníka
        % ku všetkým lokalitám
    X is V[I,1..M],
        % ohraničenie pre výber prepravných nákladov
    element(Oi,X,PNi),
        % rekurzívne aj pre ostatných zákazníkov
    J is I + 1,
    gen_ohranicenia(RO,V,RPN,J,M).
```

Po niekoľkých iteráciách metóda vetvenia a medzí implementovaná v zabudovanom predikáte *min_max/2* nájde optimálne riešenie.

```
[eclipse 5]: distribucny1(O,DC).

[0, 0, 1]
[3, 3, 3, 3, 3] Found a solution with cost 1780

[0, 1, 0]
[2, 2, 2, 2, 2] Found a solution with cost 1640
```

```
[0, 1, 1]
[3, 2, 2, 2, 3] Found a solution with cost 1570

[0, 1, 1]
[3, 3, 2, 2, 2] Found a solution with cost 1550

[0, 1, 1]
[3, 3, 2, 2, 3] Found a solution with cost 1430

[1, 1, 0]
[1, 1, 2, 2, 1] Found a solution with cost 1350

O = [1, 1, 2, 2, 1]
DC = [1, 1, 0]
yes.
```

Znamená to, že optimálne riešenie danej distribučnej úlohy je výber lokalít s poradovými číslami jedna a dva (zoznam DC) a priradenie zákazníkov 1, 2 a 5 prvému distribučnému centru a zákazníkov 3 a 4 druhému distribučnému centru (zoznam O).

V [73] je uvedené porovnanie riešenia tejto úlohy troma odlišnými prístupmi, typickými pre riešenie úloh z oblasti operačného výskumu, resp. logistiky. Ide o nasledovné tri prístupy:

- všeobecné nástroje ako napr. balíky na riešenie úloh lineárneho, resp. celočíselného programovania,

- napísanie špeciálneho programu v tradičnom, procedurálnom programovacom jazyku a

- použitie deklaratívneho programovacieho jazyka vyššej úrovne so zabudovanými prístupmi z oblasti umelej inteligencie (CLP prístup).

Čo sa týka efektivity, CLP prístup podľa výsledkov v [73] bol porovnateľný s druhým prístupom (bol 5-10 krát pomalší). Úlohy reálnej veľkosti bol schopný riešiť rádovo v minútach. Ale jeho najväčším prínosom je pružnosť. Program ktorý navrhli autori v jazyku CHIP mal 2 strany a trvalo len niekoľko dní aby bol problém pochopený, naprogramovaný a odladený.

Zmena programu je pritom jednoduchá. Pridanie vyššie uvedeného disjunktného ohraničenia na lokality znamená len pridanie ohraničenia vo forme lineárnej nerovnosti medzi príslušnými dvojicami lokalít (napr. $V_{1i}\#\#V_{1k}$).

Podobne pridanie kapacitného ohraničenia pre distribučné centrá tak isto nie je problémom doplniť. Stačí rozšíriť základnú údajovú štruktúru `distribucny` o ďalší argument obsahujúci kapacity distribučných centier v jednotlivých lokalitách. Napr. *kapacity:[130,80,90]* a doplniť k ohraničeniam ďalšie, ktoré budú strážiť maximálne kapacity distribučných centier.

Aby však bolo zabezpečené šírenie ohraničení na kapacity, je potrebné zaviesť pomocné binárne premenné V_{4ij} $(1 \leq i \leq u)$, $(1 \leq j \leq v)$, ktoré budú reprezentovať tú skutočnosť, či i-te distribučné centrum dodáva j-temu zákazníkovi (hodnota 1 ak áno, resp. 0 ak nie). Z pohľadu reprezentácie je najvhodnejšie vytvoriť pole takýchto binárnych premenných.

```
gen_pole_binarnych_premennych(N,M,O) :-
    dim(O,[N,M]), % vytvor pole premenných s rozmermi M a N
                  % a ulož ho do O
        (for(I,1,N), param(O,M) do % cyklus pre I rovné
                  % 1 až N, hodnoty O a M sa predávajú ako
                  % parametre do tela cyklu
            (for(J,1,M), param(O,I) do % cyklus pre J rovné
                  % 1 až M, hodnoty O a I sa predávajú ako
                  % parametre do tela cyklu
                O[I,J] :: [0..1] % stanovenie binárnej domény
            )
        ).
```

Takéto pomocné premenné dokonca umožnia efektívnejšie formulovať aj pôvodný problém (bez obmedzenia kapacít distribučných centier) tým, že sa nepoužije ohraničenie $odstran/2$ v rámci predikátu $ohranic/3$, ktoré sa šíri až v okamžiku naviazania premennej pre distribučné centrum.

V pôvodnom programe ($distribucny1/3$ sa premenuje na $distribucny2/3$ a analogicky $ohranic1/3$ sa premenuje na $ohranic2/3$) dôjde len k nasledujúcim zmenám:

```
ohranic2(O, DC, Naklady) :-
    ...
    arg(kapacity of distribucny, Data, K),
    ...
```

```
    gen_pole_binarnych_premennych(N,M,OO),
    gen_ohranicenia(O, V, PN, 1, M, OO),
    gen_kapacitne_ohranicenia(1,K,DC,OO,D,N),
        ...
```

Predikát *gen_ohranicenia*/5 je potrebné doplniť tak, aby novo definované premenné boli previazané s ostatnými premennými (ide o dva typy ohraničení - viď. komentár v nasledujúcom zdrojovom texte programu). Počet jeho argumentov sa teda zvýši o 1 (pole pomocných binárnych premenných). Potrebné dodatky v predikáte *gen_ohranicenia*/6 a definícia nového predikátu *gen_kapacitne_ohranicenia*/6 ohraničenia môžu vyzerať nasledovne:

```
gen_ohranicenia([],_,[],_,_,_).
gen_ohranicenia([Oi|RO],V,[PNi|RPN],I,M,OO) :-
    X is V[I,1..M],
    element(Oi,X,PNi),
        % v prípade, že sa danému odberateľovi sa priradí
        % distribučné centrum s poradovým číslom Oi,
        % zodpovedajúca binárna premenná tiež musí byť 1
    suspend(OO[I,Oi] #= 1, 4, Oi -> inst),
        % každému odberateľovi môže byť priradené práve
        % jedno distribučné centrum, takže súčet danému
        % odberateľovi zodpovedajúcich pomocných binárnych
        % premenných je 1
    sum(OO[I,1..M]) #= 1,
    J is I + 1,
    gen_ohranicenia(RO,V,RPN,J,M,OO).

gen_kapacitne_ohranicenia(_,[],[],_,_,_).
gen_kapacitne_ohranicenia(I,[KI|K],[DCI|DC],OO,D,N) :-
        % do X vlož zoznam pomocných binárnych premenných
        % pre danú lokalitu (možné distribučné centrum)
        % s poradovým číslom I
    X is OO[1..N,I],
        % zo zoznamu X a zoznamu kapacít D vytvor zoznam
        % z toho vyplývajúcich požiadaviek na dané
        % distribučné centrum
    vytvor_zoznam_nakladov(D,X,LOC),
        % zaveď ohraničenie na maximálnu kapacitu
```

```
        % distribučného centra I
    sum(LOC) #<= KI,
        % nasledujúce ohraničenie zabezpečuje prepojenie
        % pomocných binárnych premenných s binárnymi
        % premennými pre distribučné centrá ak sa totiž
        % nejaké distribučné centrum nevyberie (t.j. DCI
        % bude 0), potom aj všetky jemu zodpovedajúce
        % pomocné binárne premenné musia mať hodnotu 0
    sum(OO[1..N,I]) #<= N*DCI,
    J is I+1,
    gen_kapacitne_ohranicenia(J,K,DC,OO,D,N).
```

Po uskutočnení popísaných zmien v programe pre zohľadnenie kapacít distribučných centier pri hodnotách zadaných tak ako je spomenuté vyššie, sa riešenie úlohy zmení nasledovne:

```
[eclipse 6]: distribucny2(O, DC).

[1, 1, 1]
[1, 1, 3, 2, 3]Found a solution with cost 1960

[1, 1, 1]
[3, 1, 3, 2, 1]Found a solution with cost 1750

O = [3, 1, 3, 2, 1]
DC = [1, 1, 1]
yes.
```

To znamená že v tejto verzii úlohy s obmedzenými kapacitami distribučných centier je nutné zriadiť distribučné centrá vo všetkých troch lokalitách, pričom zákazníkom 2 a 5 bude dodávať distribučné centrum s poradovým číslom 1, zákazníkom 1 a 3 distribučné centrum 3 a zákazníkovi 4 distribučné centrum 2.

Z porovnania troch typických prístupov k riešeniu úloh z operačného výskumu a logistiky na príklade distribučnej úlohy [73]) vyplýva pre jednotlivé prístupy nasledovné:

1. Ak je úloha prirodzene formulovateľná ako úloha celočíselného programovania, potom najjednoduchšie je zvoliť práve tento prístup. Je najjednoduchšie ho naprogramovať a dáva aj rozumnú efektívnosť pre nie príliš veľké úlohy.

2. Pre všetky úlohy, u ktorých je ťažké získať a využiť matematickú formuláciu úlohy, je prístup CLP najvhodnejší. Slúži ako pružný a efektívny nástroj na riešenie takýchto úloh. Je špeciálne vhodný ako prototypovací nástroj.

3. Pre úlohy, u ktorých ich špecifické vlastnosti umožňujú napr. využitie efektívnych relaxačných metód (obchodný cestujúci, dopravný problém), resp. úlohy väčších rozmerov, kde efektivita je kritická, je najefektívnejší program v tradičnom procedurálnom jazyku.

Kapitola 8

Úlohy nad nekonečnými doménami

8.1 Zmiešavací problém

Výrobca kŕmnych zmesí pre hovädzí dobytok sa snaží vyrobiť zmes s minimálnymi nákladmi. Kŕmna zmes môže obsahovať dve aktívne zložky (tabuľka 8.2) a výplň. Jeden kilogram kŕmnej zmesi musí obsahovať minimálne množstvá zo všetkých sledovaných živín uvedených v tabuľke 8.1.

Živina	A	B	C	D
[g]	90	50	20	2

Tabuľka 8.1: Predpísané množstvá jednotlivých živín na 1 kilogram kŕmnej zmesi.

	A	B	C	D	cena/kg
Prísada 1 [g/kg]	100	80	40	10	40
Prísada 2 [g/kg]	200	150	20	0	60
Výplň	0	0	0	0	0

Tabuľka 8.2: Nutričné hodnoty a ceny za 1 kg prísady.

V tejto úlohe sú 3 premenné zodpovedajúce množstvám jednotlivých prísad, t.j. 2 rôzne prísady a výplň, ktoré budú zmiešané na 1 kg výslednej kŕmnej zmesi. Znamená to teda 3 nezáporné reálne hodnoty, ktorých súčet musí byť 1, pričom je nutné splniť minimálne požadované množstvá živín vo výslednej zmesi.

Tieto možno vyjadriť jednoducho lineárnymi nerovnicami pre každú sledovanú živinu. Optimalizácia je v prípade nekonečných domén vyčlenená mimo sekciu ohraničení a možno ju preto skúmať aj mimo programu, priamo ako súčasť otázky v ECL^iPS^e.

```
:- lib(clpr).

zmiesavaci(Prisady,Naklady) :-
        % premenné zodpovedajúce zložkám kŕmnej zmesi
    Prisady = [P1,P2,F],
        % funkcia nákladov (cena za kilogram kŕmnej zmesi)
    Naklady = P1*40 + P2*60,
    {
        % požiadavka nezápornosti pre jednotlivé zložky
     P1 >= 0,
     P2 >= 0,
     F >= 0,
        % základné ohraničenie - súčet zložiek
     P1 + P2 + F = 1, % na 1 kg kŕmnej zmesi
     P1*100 + P2*200 >= 90, % Živina A
     P1*80 + P2*150 >= 50, % Živina B
     P1*40 + P2*20 >= 20, % Živina C
     P1*10 >= 2 % Živina D
    }.
```

Otázkou na takto definovaný predikát možno v odpovedi systému ECL^iPS^e získať vlastne vnútornú formu zadaných ohraničení.

```
[eclipse 2]: zmiesavaci(L,Naklady).

L = [P1, P2, F]
Naklady = P1 * 40 + P2 * 60

% Linear constraints:
{
  P1 = 1.0 - F - P2,
  F + P2 =< 0.8,
```

```
  F + 0.5 * P2 =< 0.5,
  F - P2 =< 0.1,
  P2 >= 0.0,
  F >= 0.0
}
yes.
```

Po pridaní optimalizácie (v tomto prípade minimalizácie nákladov), systém nájde jediné optimálne riešenie, v ktorom je zastúpených $36{,}\bar{6}\%$ prísady 1, $26{,}\bar{6}\%$ prísady 2 a $36{,}\bar{6}\%$ výplne.

```
[eclipse 3]: zmiesavaci(L,Naklady), minimize(Naklady).

L = [0.3666666666667, 0.26666666666666, 0.3666666666667]
Naklady = 0.3666666666667 * 40 + 0.26666666666666 * 60
yes.
```

Systém však nevyčísli celkové náklady, ale ponechá ich v tvare lineárneho výrazu. Pre získanie celkovej hodnoty nákladov je potrebné použiť operátor *is/2* a novú (pomocnú) premennú.

```
[eclipse 4]: zmiesavaci(L,Naklady), minimize(Naklady),
             Min_naklady is Naklady.

L = [0.3666666666667, 0.26666666666666, 0.3666666666667]
Min_naklady = 30.666666666668
Naklady = 0.3666666666667 * 40 + 0.26666666666666 * 60
yes.
```

8.2 Finančné plánovanie

Ďalší typ úlohy s ohraničeniami nad reálnymi premennými sa týka rozdeľovania bankových finančných prostriedkov na jednotlivé typy pôžičiek. Banka ponúka 4 druhy osobných pôžičiek svojim zákazníkom za nasledovné úrokové sadzby:

- 1. hypotéka 14%
- 2. hypotéka 20%
- výbava domácnosti 20%

- osobné výdavky 10%

Banka má maximálne predpovedateľné pôžičkové kapacity 250 mil. peňažných jednotiek (PJ) a je ďalej obmedzená týmito pravidlami:

1. prvé hypotéky musia byť aspoň 55 % zo všetkých hypoték a aspoň 25 % všetkých pridelených pôžičiek,
2. druhé hypotéky nesmú presiahnuť 25 % všetkých pridelených pôžičiek,
3. aby sa predišlo všeobecnej nespokojnosti a zavedeniu nových daní, priemerná úroková sadzba zo všetkých pôžičiek nesmie prekročiť 15 %.

V tejto úlohe prirodzene vystupujú štyri reálne nezáporné premenné zodpovedajúce objemu finančných prostriedkov vyčlenených na jednotlivé typy pôžičiek v danom roku. Súčet ich hodnôt nesmie presiahnuť maximálne predpovedateľné pôžičkové kapacity.

Okrem toho sú zadané tri, resp. štyri ohraničenia (prvé ohraničenie v sebe obsahuje vlastne dve, ohraničenie objemu pôžičiek na prvú hypotéku voči druhej a voči všetkým pôžičkám). Všetky tieto ohraničenia sa dajú zapísať ako lineárne nerovnosti.

```
:- lib(clpr).

financny(Pozicky, Zisk) :-
        % premenné zodpovedajú úhrnnému množstvu financií
        % vyčlenených na jednotlivé typy pôžičiek
    Pozicky = [H1, H2, VD, OV],
        % zisk možno vyčísliť na základe úrokových sadzieb
    Zisk = 0.14*H1 + 0.2*H2 + 0.2*VD + 0.1*OV,
    {
        % ohraničenie maximálnych pôžičkových kapacít
     H1 + H2 + VD + OV =< 250,
        % prirodzená požiadavka nezápornosti
     H1 >= 0, H2 >= 0, VD >= 0, OV >= 0,
        % ohraničenie pomeru prvých a druhých hypoték
     0.45*H1 >= 0.55*H2,
        % ohraničenie pomeru prvých hypoték
        % ku všetkým pôžičkám (prvé ohraničenie)
     0.75*H1 >= 0.25*H2 + 0.25*VD + 0.25*OV,
        % ohraničenie pomeru druhých hypoték
```

```
        % ku všetkým pôžičkám (druhé ohraničenie)
     0.75*H2 =< 0.25*H1 + 0.25*VD + 0.25*OV,
        % ohraničenie na priemernú úrokovú sadzbu
        % (tretie ohraničenie)
     -0.01*H1 + 0.05*H2 + 0.05*VD - 0.05*OV =< 0
    },
        % Optimalizácia
    maximize(Zisk).
```

Táto úloha má nekonečne veľa optimálnych riešení (optimálne riešenia ležia na úsečke v priestore riešení). Možno to vidieť z výpisu, ktorý napriek maximalizácii zisku v programe, stále nevedie na jediné riešenie, ale popisuje všetky riešenia množinou ohraničení v normalizovanom tvare.

```
[eclipse 5]: financny(Pozicky, Zisk).

Pozicky = [H1, H2, VD, OV]
Zisk = 0.14 * H1 + 0.2 * H2 + 0.2 * VD + 0.1 * OV

% Linear constraints:
{
  H2 = 125.0 - VD - 0.4 * H1,
  OV = 125.00000000003 - 0.60000000000009 * H1,
  H1 =< 208.33333333337,
  VD + 0.40000000000013 * H1 =< 125.00000000003,
  H1 >= 62.500000000028,
  VD >= 0.0,
  VD + 0.40000000000013 * H1 >= 62.500000000014,
  VD + 1.21818181818189 * H1 >= 125.0000000001
}
yes.
```

Pre získanie konkrétneho riešenia je potrebné zvoliť hodnoty dvoch nezávislých premenných ľubovoľne tak, aby vyhovovali ohraničeniam. Napríklad:

```
[eclipse 6]: financny([H1,H2,VD,OV], Zisk),
              H1 = 62.5, H2 = 51.1, Max is Zisk.

VD = 48.9
OV = 87.500000000014
```

```
H1 = 62.5
H2 = 51.1
Max = 37.5
Zisk = 0.14 * 62.5 + 0.2 * 51.1 + 0.2 * 48.9 +
       0.1 * 87.500000000014
yes.
```

Pri zmene hodnoty $H1$ a ponechaní hodnoty $H2$ výsledná hodnota kriteriálnej funkcie (zisk) sa nezmení.

```
[eclipse 7]: financny([H1,H2,VD,OV], Zisk),
             H1 = 75.0, H2 = 51.1, Max is Zisk.

VD = 43.9
OV = 80.000000000014
H1 = 75.0
H2 = 51.1
Max = 37.5
Zisk = 0.14 * 75.0 + 0.2 * 51.1 + 0.2 * 43.9 +
       0.1 * 80.000000000000014
yes.
```

Časť V

Prílohy

Matematické symboly

Táto príloha obsahuje zoznam matematických symbolov, používaných v texte pre označenie jednotlivých parametrov, hodnôt, ap. Pripojené krátke popisy indikujú významy daných symbolov. V tých prípadoch, keď v tejto časti uvedený popis nie je dostatočným vysvetlením významu symbolu, je možné jeho podrobnejšiu definíciu nájsť pomocou registra, umiestneného na konci textu.

Symbol	Krátky popis
$\Vert\,.\,\Vert$	kardinalita množiny
$\lceil . \rceil$	zaokrúhlenie nahor
$\prod$	súčin
Σ	abeceda
A	množina štruktúr a ohraničení
B_i	štruktúra v tele pravidla (C)LP programu
$B^{1,\ldots,i}$	konjunkcia štruktúr B_1 až B_i v tele pravidla CLP programu
c	ohraničenie (všeobecné)
C	množina ohraničení
$C^{i,\ldots,j}$	ohraničenie definované nad premennými V_i až V_j
C_i^{all}	všetky ohraničenia definované nad premennou V_i
C^{free}	ohraničenia definované nad voľnými premennými

Symbol	Krátky popis
CS_i	konfliktná množina prislúchajúca premennej V_i
D	množina domén premenných
D_i	doména premennej V_i
D_i^r	redukovaná doména premennej V_i
f	funkcia (všeobecná)
f_h	hodnotiaca funkcia
G	množina pasívnych (uspaných) ohraničení
h	hodnota premennej (všeobecná)
h_i	hodnota priradená premennej V_i
h_j	hodnota priradená premennej V_j
h_k	hodnota priradená premennej V_k
H	štruktúra v hlave pravidla (C)LP programu
i	index (prirodzené číslo z nejakého intervalu)
j	index
k	index
$le_k(V^{i,\ldots,j})$	lineárny výraz obsahujúci premenné z $V^{i,\ldots,j}$
m	počet definovaných ohraničení
m_{MAX}	maximálne možný počet ohraničení
n	počet premenných
N	hodnota kriteriálnej funkcie (nákladov)
N_i	hodnota kriteriálnej funkcie nájdeného riešenia
N^{min}	dolná hranica hodnôt kriteriálnej funkcie
N^{max}	horná hranica hodnôt kriteriálnej funkcie
o_x	os x
o_y	os y
p_c	prísnosť ohraničenia c

Symbol	Krátky popis
P	priestor prehľadávania
$sym0$, $sym1$, ...	symboly hodnôt
S	stavový priestor
u	číselná konštanta
$\tilde{U}$	matica
v	číselná konštanta
V	množina premenných
$\tilde{V}$	matica
V_i	i-ta premenná
$V^{i,\ldots,j}$	podmnožina premenných V_i až V_j
V^{free}	množina voľných (neviazaných) premenných
$VC^{i,\ldots,j}$	c-premenná reprezentujúca ohraničenie $C^{i,\ldots,j}$
$VH^{i,\ldots,j}$	h-premenná reprezentujúca ohraničenie $C^{i,\ldots,j}$
w	ľubovoľná konštanta
w_i	i-ta konštanta
W	množina konštánt
W_i	podmnožina konštánt
$\mathcal{D}$	interpretačná štruktúra
$\mathcal{FD}$	konečné domény
$\mathcal{L}$	predikátový jazyk
$\mathcal{Q}$	ohraničujúca doména aritmetiky nad racionálnymi číslami
$\mathcal{R}$	ohraničujúca doména aritmetiky nad reálnymi číslami
$\mathcal{T}$	teória
$\mathcal{X}$	ohraničujúca doména

Symbol	Krátky popis
$\mathcal{Z}$	množina celých čísiel
$\Re$	množina reálnych čísiel

Slovensko-anglický slovník

Následkom toho, že v slovenskom jazyku nie je dostupných veľa publikácií z oblasti pokrytej prezentovaným textom je to, že slovenské podoby jednotlivých odborných termínov nie sú jednoznačne ustálené alebo všeobecne známe. Aby sme umožnili čitateľovi zorientovať sa v nami použitej terminológii a uľahčili mu konfrontovať nami predkladané informácie s informáciami prezentovanými v iných publikáciách, je v tejto prílohe uvedený krátky slovensko-anglický slovník najdôležitejších odborných termínov z oblasti reprezentácie a riešenia úloh s ohraničeniami, ktoré boli použité v texte.

	Slovenský termín	Anglický termín
A	algoritmus	algorithm
	aktuálne riešenie	current solution
Á	árnosť	arity
C	cesta	path
	cieľ	goal
	– uspaný	suspended goal
D	doména	domain
	– konečná	finite domain
	– nekonečná	infinite domain
	– ohraničujúca	constraint domain
	dopredný pohľad	looking ahead
	dynamická zmena prehľadávania	dynamic search rearrangement

	Slovenský termín	Anglický termín
H	heuristika	heuristics
	hodnota	value
	– lepkavá	sticking value
	hrana	arc
K	konfliktná množina	conflict set
	kontrola	checking
	– dopredná	forward checking
	– ohraničení	consistency check
	– spätná	backchecking
	konzistencia	consistency
	– hranová	arc consistency
	– po ceste	path consistency
	– prísna	strong consistency
	– smerová	directional consistency
	– uzlová	node consistency
	konzistenčný algoritmus	consistency enforcing algorithm
	kostra grafu	spanning tree
L	lenivé vyhodnocovanie	lazy evaluation
	logické programovanie	logic programming
M	množina	set
	– minimálna	minimum set
N	náhrada podľa okolia	neighbourhood substitution
	naviazať	instantiate
	NP-úplnosť	NP-completeness
O	ohraničenie	constraint
	– binárne	binary constraint
	– explicitné	explicit constraint
	– indukované	induced constraint
	– n-árne	n-ary constraint
	– unárne	unary constraint
	orezanie	pruning

	Slovenský termín	Anglický termín
P	podgraf	subgraph
	podmienka	condition
	– nutná	necessary condition
	– postačujúca	sufficient condition
	podpora	support
	podstrom	subtree
	porušenie ohraničení	constraint violation
	prahová hodnota	threshold value
	predspracovanie	preprocessing
	prehľadávanie	search
	– do hĺbky	deep-first search
	– slepé	blind search
	premenná	variable
	– budúca	future variable
	– deliaca	cutting variable
	– neviazaná	free variable
	– skrytá	hidden variable
	– viazaná	bound variable
	– voľná	free variable
	– zdieľaná	shared variable
	priestor prehľadávania	search space
	priradenie	assignment
	– spĺňajúce	satisfying assignment
	prísnosť ohraničení	constraint tightness
	problém s ohraničeniami	constraint satisfaction problem
	– binárny	binary constraint satisfaction problem
	– optimalizačný	constraint satisfaction optimization problem
	programovanie ohraničení	constraint programming
	– logické	constraint logic programming
R	redundancia	redundancy
	– hrán	arc redundancy
	riešenie	solving
	– ohraničení	constraint solving
	riešiteľnosť	tractability

	Slovenský termín	Anglický termín
S	schéma	scheme
	– dopredná	look-ahead scheme
	– spätná	look-back scheme
	sieť ohraničení	constraint network
	– duálna	dual constraint network
	– primárna	primal constraint network
	slepý koniec	deadend
	spätné navracanie	backtracking
	– dynamické	dynamic backtracking
	spätné značkovanie	backmarking
	spätný skok	backjumping
	– riadený konfliktom	dependency-directed backjumping
	– založený na grafe	graph-based backjumping
	spĺňanie ohraničení	constraint satisfaction
	stavový priestor	state space
	strom prehľadávania	search tree
	stupeň	degree
	– maximálny	maximum degree
Š	šírenie ohraničení	constraint propagation
	šírka	width
	– grafu	graph width
	– minimálna	minimum width
	– usporiadania	ordering width
	– uzla	node width
	štruktúra	structure
	– stromová	tree structure
T	topológia siete	network topology
U	učenie	learning
	– hlboké	deep learning
	– plytké	shallow learning
	unifikácia	unification
	usporiadaná sieť	ordered net

	Slovenský termín	Anglický termín
	usporiadanie	ordering
	– domén	domain ordering
	– dynamické	dynamic ordering
	– premenných	variable ordering
	– statické	static ordering
	uvoľnenie	relaxation
	uzol	node
	– koreňový	root node
	– rodičovský	parent node
V	voľnosť	looseness
	vyprázdnenie domény	domain wipe-out
Z	zameniteľnosť	interchangeability
	– čiastočná	partial interchangeability
	– plná	full interchangeability
	– podľa okolia	neighbourhood interchangeability
	zaznamenávanie nevhodností	no-good recording
	značkovanie	labeling
	– konzistentné	consistent labeling

Zoznam algoritmov

V texte je možné nájsť roztrúsené popisy niekoľkých algoritmov, súvisiacich s riešením problémov s ohraničeniami. Jednotlivé algoritmy sú pritom uvedené na rozličnej hladine detailnosti – pre niektoré je k dispozícii zápis v tvare kódu (zvyčajne neúplný, funkčnosť niektorých funkcií je uvedená iba slovne resp. názvy funkcií sú samovysvetľujúce), zatiaľ čo iné sú popísané iba slovne. Podobne činnosť niektorých algoritmov je ilustrovaná na príklade, zatiaľ čo iné sú uvedené iba v teoretickej podobe.

Táto kapitola uvádza ich zoznam spolu s odkazom na miesto v texte, kde sú popísané podrobnejšie. Pre názvy jednotlivých algoritmov sú uvedené aj anglické ekvivalenty pre ľahšiu orientáciu v cudzojazyčnej literatúre.

Skratka	Názov algoritmu	Kód	Ilustr.	Strana
AC	arc consistency (hranová konzistencia)	A	A	33
BCH	backchecking (spätná kontrola)	N	A	60
BDACL	bidirectional arc consistency lookahead (dopredný pohľad s obojsmernou hranovou konzistenciou)	N	N	86
BJ	backjumping (spätný skok)	N	A	55
BM	backmarking (spätné značkovanie)	N	A	60

Skratka	Názov algoritmu	Kód	Ilustr.	Strana
BT	backtracking (spätné navracanie)	A	A	50
DAC	directed arc consistency (smerová hranová konzistencia)	A	A	44
DACL	directed arc consistency lookahead (dopredný pohľad so smerovou hranovou konzistenciou)	A	N	86
DBT	dynamic backtracking (dynamické spätné navracanie)	N	A	59
DDBJ	dependency-directed backjumping (spätný skok riadený konfliktmi)	A	A	57
DL	deep learning (hlboké učenie)	N	A	63
FC	forward checking (dopredná kontrola)	A	A	81
FLA	full looking ahead (plný dopredný pohľad)	A	A	86
GBBJ	graph-based backjumping (spätný skok založený na grafe)	N	A	58
GW	graph width (šírka grafu)	A	A	42
KC	k-consistency (k-konzistencia)	N	N	38
MAC	maintaining arc consistency (udržiavanie hranovej konzistencie)	N	A	87
MFC	minimal forward checking (minimálna dopredná kontrola)	N	N	83
NC	node consistency (uzlová konzistencia)	A	N	32
NGR	no-good recording (zaznamenávanie nevhodností)	N	A	64

Zabudované ohraničenia v ECL^iPS^e

Aritmetické ohraničenia sú nasledovné:

- `?T1 #\= ?T2` alebo `?T1 ## ?T2` – hodnota racionálneho termu `T1` sa nerovná hodnote racionálneho termu `T2`.
- `?T1 #< ?T2` – hodnota racionálneho termu `T1` je menšia ako hodnota racionálneho termu `T2`.
- `?T1 #<= ?T2` – hodnota racionálneho termu `T1` je menšia alebo rovná hodnote racionálneho termu `T2`.
- `?T1 #= ?T2` – hodnota racionálneho termu `T1` je rovná hodnote racionálneho termu `T2`.
- `?T1 #> ?T2` – hodnota racionálneho termu `T1` je väčšia ako hodnota racionálneho termu `T2`.
- `?T1 #>= ?T2` – hodnota racionálneho termu `T1` je väčšia alebo rovná hodnote racionálneho termu `T2`.

Logické ohraničenia sa zvyknú zapisovať vo forme operátorov. Operátor je funktor deklarovaný pomocou predikátu `op(+Vazba, +Specifikacia, +Nazov)` a umožňuje zápis štruktúry bez zátvoriek, pričom funktor a argumenty sa oddeľujú medzerami. Čím vyššia je hodnota čísla `+Vazba` tým nižšia je priorita daného operátora voči susedným operátorom. `+Specifikacia` definuje o aký typ operátora s názvom `+Nazov` ide, pričom f určuje pozíciu operátora, x pozíciu jedného argumentu a y jedného argumentu, alebo rekurzívne vytvorenej štruktúry vytvorenej operátorom s názvom `+Nazov`.

Logické operátory sú deklarované s nasledovnými prioritami:

```
:- op(750, fy, #\+).
:- op(760, yfx, #/\).
:- op(770, yfx, #\/).
:- op(780, yfx, #=>).
:- op(790, yfx, #<=>).
```

Logické ohraničenia sú nasledovné:

- `#\+ +E1` – uspeje ak výraz `E1` je nepravdivý. Ide teda o logickú negáciu pravdivostnej hodnoty ohraničujúceho výrazu `E1`.
- `+E1 #/\ +E2` – Oba ohraničujúce výrazy `E1` a `E2` sú pravdivé. Toto je ekvivalentné s logickou konjunkciou ohraničujúcich výrazov `E1, E2`.
- `+E1 #\/ +E2` – Aspoň jeden z ohraničujúcich výrazov `E1` a `E2` je pravdivý. Akonáhle sa jeden z výrazov `E1` alebo `E2` stane nepravdivý, druhý z dvojice je zadaný.
- `+E1 #=> +E2` – Ohraničujúci výraz `E1` implikuje ohraničujúci výraz `E2`. Ak sa `E1` stane pravdivým, `E2` je zadaný. Ak `E2` sa stane nepravdivým, potom je zadaná negácia `E1`.
- `+E1 #<=> +E2` – Ohraničujúci výraz `E1` je ekvivalentný ohraničujúcemu výrazu `E2`. Ak jeden z výrazov sa stane pravdivým, druhý z nich je zadaný. Ak sa jeden z výrazov stane nepravdivým, negácia druhého z nich je zadaná.

Vyhodnocovacie ohraničenia asociujú pravdivostnú hodnotu nejakého ohraničenia s booleovskou premennou. V knižnici konečných domén sú definované tieto:

- `?B isd +Expr` – `B` má hodnotu 1 vtedy a len vtedy ak výraz `Expr` je pravdivý, ináč má hodnotu 0. Tento predikát je ohraničujúcou analógiou predikátu $is/2$ – vezme ohraničujúci výraz, transformuje všetky jeho podvýrazy na volania predikátov s aritou o jednotku vyššou a skombinuje výsledné booleovské hodnoty tak, aby dali `B`.

  Napr. `B isd X #= Y` je ekvivalentné volaniu `#=(X, Y, B)`.
- `#<(?T1, ?T2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` je menšia ako hodnota racionálneho termu `T2`, ináč sa rovná 0.

- `#<=(?T1, ?T2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` je menšia alebo rovná hodnote racionálneho termu `T2`, ináč sa rovná 0.
- `#=(?T1, ?T2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` je rovná hodnote racionálneho termu `T2`, ináč sa rovná 0.
- `#\=(?T1, ?T2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` sa nerovná hodnote racionálneho termu `T2`, ináč sa rovná 0.
- `#>(?T1, ?T2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` je väčšia ako hodnota racionálneho termu `T2`, ináč sa rovná 0.
- `#>=(?T1, ?T2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak hodnota racionálneho termu `T1` je väčšia alebo rovná hodnote racionálneho termu `T2`, ináč sa rovná 0.
- `#/\(+E1, +E2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak oba ohraničujúce výrazy `E1` a `E2` sú pravdivé, ináč sa rovná 0.
- `#\/(+E1, +E2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak aspoň jeden z ohraničujúcich výrazov `E1` a `E2` je pravdivý, ináč sa rovná 0.
- `#<=>(+E1, +E2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak ohraničujúci výraz `E1` je ekvivalentný ohraničujúcemu výrazu `E2`, ináč sa rovná 0.
- `#=>(+E1, +E2, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak ohraničujúci výraz `E1` implikuje ohraničujúci výraz `E2`, ináč sa rovná 0.
- `#\+(+E1, ?B)` – `B` sa rovná 1 vtedy a len vtedy ak `E1` je nepravdivý, ináč sa rovná 0.

Symbolické ohraničenia vyjadrujú zložitejšie vzťahy medzi doménovými premennými. Ide napríklad o tieto:

- `alldistinct(?List)` – Všetky prvky zoznamu *List* (doménové premenné a termy bez výskytu premenných) sú po dvojiciach rôzne.
- `atmost(+Number, ?List, +Val)` – Najviac *Number* počet prvkov zoznamu *List* doménových premenných a termov bez výskytu premenných je rovných hodnote *Val*.

- `element(?Index, +List, ?Value)` – *Index*-ty prvok zoznamu *List* termov bez výskytu premenných má hodnotu *Value*. *Index* a *Value* môžu byť alebo obyčajné premenné, potom sa im asociuje doména, alebo doménové premenné. Kedykoľvek sa zmení doména *Index* alebo *Value*, predikát sa zobudí a upraví zodpovedajúco domény oboch premenných.

- `outof(?Var, +List)` – Doménová premenná *Var* sa nerovná ani jednému prvku v zozname *List*.

- `#(?Min, ?CstList, ?Max)` – Operátor kardinality, tak ako bol popísaný napr. v [74]. *CstList* je zoznam ohraničení (ohraničujúcich výrazov). Tento operátor zabezpečí, že aspoň *Min* a najviac *Max* z nich bude splnených.

Literatúra

[1] Affane, M.S. – Bennaceur, H.: A labeling arc consistency method for functional constraints. In: Freuder, E.C. (ed.): Principles and Practice of Constraint Programming - CP'96 (LNCS 1118). Springer Verlag, Heidelberg, 1996, 16–30.

[2] Aggoun, A. – Beldiceanu, N.: Overview of the CHIP compiler system. In: Proc. of the 8th International Conference on Logic Programming. Paris, June 1991, 775–789.

[3] Aggoun, A. et al.: ECL^iPS^e user manual, release 4.2. ©International Computers Limited and ECRC GmbH 1992-1995, ©International Computers Limited and Imperial College London 1996-1999.

[4] Ait-Kaci, H.: Warren's abstract machine. A tutorial reconstruction. Logic Programming Series, The MIT Press, Cambridge, MA, 1991.

[5] Allen, J.F.: Maintaining knowledge about temporal intervals. Communications ACM, vol. 26, 1983, no. 11, 832–843.

[6] Allen, J.A. – Minton, S.: Selecting the right heuristic algorithm: runtime performance predictors. In: McCalla, G. (ed.): Advances in Artificial Intelligence (LNAI 1081). Springer Verlag, Heidelberg, 1996, 41–53.

[7] Bacchus, F. – Grove, A.: On the forward checking algorithm. In: Montanari, U. – Rossi, F. (eds.): Principles and Practice of Constraint Programming - CP'95 (LNCS 976). Springer Verlag, Heidelberg, 1995, 292–309.

[8] Bacchus, F. – Van Beek, P.: On the conversion between non-binary and binary constraint satisfaction problems. In: Proc. of the 15th National

Conference on Artificial Intelligence AAAI'98. Madison, Wisconsin, 1998. http://www.cs.ualberta.ca/~vanbeek/papers.html

[9] Bacchus, F. – Van Run, P.: Dynamic variable ordering in CSPs. In: Montanari, U – Rossi, F. (eds.): Principles and Practice of Constraint Programming - CP'95 (LNCS 976). Springer Verlag, Heidelberg, 1995, 258–275.

[10] Barták, R.: Constraint programming: in pursuit of the holy grail. In: Proc. of the Conference WDS'99, Praha, 1999. http://kti.mff.cuni.cz/~bartak/downloads/WDS99.pdf

[11] Bayardo, R.J.Jr. – Miranker, D.P.: An optimal backtrack algorithm for tree-structured constraint satisfaction problems. Artificial Intelligence, vol. 71, 1994, 159–181.

[12] Bellone J. – Chamard A. – Fischler A.: Constraint logic programming decision support systems for planning and scheduling aircraft manufacturing at Dassault Aviation. In: Proc. of the Third Int. Conference on the Practical Application of Prolog. Paris, 1995, 63-67.

[13] Berlandier, P.: A symbiotic approach to arc and path consistency checking. In: Pinto-Ferreira, C. – Mamede, N.J. (eds.): Progress in Artificial Intelligence (LNAI 990). Springer Verlag, Heidelberg, 1995, 107–114.

[14] Bessiere, Ch.: Arc-consistency and arc-consistency again. Artificial Intelligence, vol. 65, 1994, 179–190.

[15] Bessiere, Ch. – Freuder, E.C. – Regin, J-Ch.: Using constraint metaknowledge to reduce arc consistency computation. Artificial Intelligence, vol. 107, 1999, 125–148.

[16] Bessiere, Ch. – Regin, J-Ch.: Using bidirectionality to speed up arc-consistency processing. In: Meyer, M. (ed.): Constraint Processing (LNCS 923). Springer Verlag, Heidelberg, 1995, 157–169.

[17] Bessiere, Ch. – Regin, J-Ch.: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In: Freuder, E.C.: (ed.) Principles and Practice of Constraint Programming (LNCS 1118). Springer Verlag, Heidelberg, 1996, 61–75. http://www.lirmm.fr/~bessiere/publis.html

[18] Bieliková, M. – Návrat, P.: Funkcionálne a logické programovanie. STU Bratislava, Fakulta elektrotechniky a informatiky, 2000.

[19] Bitner, J.R. – Reingold, E.M.: Backtrack programming techniques. Communications of the ACM, vol. 18, 1975, no. 11, 651–656.

[20] Borning, A.: The programming language aspects of ThingLab, a constraint oriented simulation laboratory. ACM Transactions on Programming Languages and Systems, vol.3, 1981, no. 4, 252–387.

[21] Borrett, J.E. – Tsang, E.P.K.: Observations on the usefulness of arc consistency preprocessing. Technical report CSM-236. Dept. of Computer Science, University of Essex, Colchester, UK, 1995. ftp://ftp.essex.ac.uk/pub/csp/CSM-236.ps.Z

[22] Brisset, P. et al.: ECL^iPS^e library manual, release 4.2, ©International Computers Limited and ECRC GmbH 1992-1995, ©International Computers Limited and Imperial College London 1996-1999.

[23] Caseau, Y. – Laburthe, F.: Disjunctive scheduling with task intervals. LIENS Technical Report 95-25. Laboratoire d'Informatique de I'Ecole Normale Superieure, 1995.

[24] Chen, Y.: Arc consistency revisited. Information Processing Letters, vol. 70, 1999, 175-184.

[25] Clocksin, W.F. – Mellish, C.S.: Programming in Prolog. Springer-Verlag, Heidelberg, 1981.

[26] Cooper, M.C.: An optimal k-consistency algorithm. Artificial Intelligence, vol. 41, 1989/90, 89–95.

[27] Cooper, M.C.: Fundamental properties of neighbourhood substitution in constraint satisfaction problems. Artificial Intelligence, vol. 90, 1997, 1–24.

[28] Crabtree, I. B.: Resource scheduling - comparing simulated annealing with constraint programming. In: Proc. of the Seventh Conference on Practical Application of Constraint Technology. Paris, 1995, 55–62.

[29] Cras, J.Y.: A review of industrial constraint solving tools. AI Perspective Series, Oxford, United Kingdom, 1993.

[30] Creemers, T. et al: Constraint-based maintenance scheduling on an electrical power-distribution network. In: Proc. of the Third International Conference on the Practical Application of Prolog. Paris, 1995, 135–144.

[31] Csontó, J. – Paralič, J.: A look at CLP – theory and application. Applied Artificial Intelligence, vol. 11, 1997, no. 1, 59–70.

[32] Csontó, J. – Sabol, T.: Umelá inteligencia. Elektrotechnická fakulta Technickej univerzity v Košiciach, ISBN 80-7099-104-6, 1991.

[33] David, J. M. – Leong, C. T.: Constraint-based applications in production planning; Examples from the automotive industry. In: Proc. of the Seventh Conference on Practical Application of Constraint Technology. Paris, 1995, 37–52.

[34] Davis, E.: Constraint propagation with interval labels. Artificial Intelligence, vol. 32, 1987, 281–331.

[35] Debruyne, R. – Bessiere, Ch.: Some practicable filtering techniques for the constraint satisfaction problem. In: Proc. of IJCAI'97. Nagoya, Japan, 412–417. http://www.lirmm.fr/~bessiere/publis.html

[36] Dechter, R.: Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artificial Intelligence, vol. 41, 1989/90, 273–312.

[37] Dechter, R.: From local to global consistency. Artificial Intelligence, vol. 55, 1992, 87–107.

[38] Dechter R.: Constraint networks. In: Encyclopedia of Artificial Intelligence, 2nd ed. Wiley and Sons, 1992, 276–285. ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/survey.ps

[39] Dechter, R. – Meiri, I.: Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In: Proc. of the 11th Int. Joint Conference on Artificial Intelligence IJCAI'89. Detroit, Michigan, 1989. 271–277.

[40] Dechter, R. – Meiri, I.: Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. Artificial Intelligence, vol. 68, 1994, 211–241.

[41] Dechter, R. – Pearl, J.: Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, vol. 34, 1988, 1–38.

[42] Dechter, R. – Pearl, J.: Tree-clustering schemes for constraint-processing. In: Proc. of the 7th National Conference on Artificial Intelligence AAAI'88. Saint Paul, Minesota, 1988, 150–154.

[43] Dechter, R. – Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence, vol. 38, 1989, 353–366.

[44] Dent, M.J. – Mercer, R.E.: Minimal forward checking. In: Proc. of the 6th IEEE Int. Conference on Tools with Artificial Intelligence. New Orleans, 1994. 432–438.
ftp://csd.uwo.ca/pub/csd-technical-reports/374/tai94.ps.Z

[45] Deville, Y. – Barette, O. – Van Hentenryck, P.: Constraint satisfaction over row-convex constraints. Artificial Intelligence, vol. 109, 1999, 243–271.

[46] Deville, Y. – Van Hentenryck, P.: An efficient arc consistency algorithm for a class of CSP problems. In: Proc. of the 12th Int. Joint Conference on Artificial Intelligence IJCAI'91. Sydney, Australia, 1991, 325–330.

[47] Dincbas M. et al: The constraint logic programming language CHIP. In: Proc. of the International Conference on Fifth Generation Computer Systems. ICOT, Japan, 1988.

[48] Dincbas, M. – Simonis, H. – Hentenryck, P.: Solving a cutting-stock problem in constraint logic programming. In: Proc. of the 5th International Conference on Logic Programming. MIT Press, 1988, 43–58.

[49] Dvořák, J. – Šeda, M.: Scheduling and lot sizing in a flow shop. In: Proc. of the Scientific Conference on Artificial Intelligence in Industry AIII'98. Stará Lesná, Slovensko, 1998, 91–100.

[50] Faltings, B.: Arc-consistency for continuous variables. Artificial Intelligence, vol. 65, 1994, 363–376.

[51] Freuder, E.C.: Synthesizing constraint expressions. Communications of the ACM, vol. 21, 1978, no. 11, 958–966.

[52] Freuder, E.C.: A sufficient condition for backtrack-free search. Journal of the ACM, vol. 29, 1982, no. 1, 24–32.

[53] Freuder, E.C.: A sufficient condition for backtrack-bounded search. Journal of the ACM, vol. 32, 1985, no. 4, 755–761.

[54] Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Proc of the Ninth National Conference on Artificial Intelligence AAAI'91. Anaheim, California, 1991, 227–233.

[55] Freuder, E.C.: The many path to satisfaction. In: Meyer, M. (ed.): Constraint Processing (LNCS 923). Springer Verlag, Heidelberg, 1995, 103–119.

[56] Freuder, E.C.: Using metalevel constraint knowledge to reduce constraint checking. In: Meyer, M. (ed.): Constraint Processing (LNCS 923). Springer Verlag, Heidelberg, 1995, 171–183.

[57] Frost, D. – Dechter, R.: In search of the best constraint satisfaction search. In: Proc of the Twelfth National Conference on Artificial Intelligence AAAI'94. Seatle, WA, 1994, 301–306. ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/search-for-best-search.ps

[58] Frost, D. – Dechter, R.: Dead-end driven learning. In: Proc of the Twelfth National Conference on Artificial Intelligence AAAI'94. Seatle, WA, 1994, 294–300. ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/dead-end-driven-learning.ps

[59] Frost, D. – Dechter, R.: Look-ahead value ordering for constraint satisfaction problems. In: Proc. of the International Joint Conference on Artificial Intelligence IJCAI'95. Montreal, Canada, 1995, 572–578. ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/look-ahead-value-ordering.ps

[60] Frost, D. – Dechter, R.: Looking at full looking ahead. In: Freuder, E.C. (ed.): Principle and Practice of Constraint Programming (LNCS 1118). Springer Verlag, Heidelberg, 1996, 569–540.

[61] Gaschnig, J.: Performance measurement and analysis of certain search algorithms. Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[62] Gent, I.P. et al.: An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In: Freuder, E.C. (ed):

Principles and Practice of Constraint Programming - CP96 (LNCS 1118). Springer Verlag, Heidelberg, 1996, 179–193.

[63] Ginsberg, M.L.: Dynamic backtracking. Journal of Artificial Intelligence Research, vol. 1, 1993. 25–46.

[64] Gomes, C.P. – Selman, B. – Kautz, H.: Boosting combinatorial search through randomization. In: Proc. of the 15th National Conference on Artificial Intelligence AAAI'98. Madison, WI, 1998. http://simon.cs.cornell.edu/home/selman/papers-ftp/papers.html

[65] Guesgen, H-W. – Hertzberg, J.: Some fundamental properties of local constraint propagation. Artificial Intelligence, vol. 36, 1988, 237–247.

[66] Gyssens, M. – Jeavons, P.G. – Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. Artificial Intelligence, vol. 66, 1994, 57–89.

[67] Han, Ch-Ch. – Lee, Ch-H.: Comments on Mohr and Henderson's path consistency algorithm. Artificial Intelligence, vol. 36, 1988, 125–130.

[68] Haralick, R.M. – Elliot, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, vol. 14, 1980, 263–313.

[69] Haralick, R.M. – Shapiro, L.G.: The consistent labeling problem: part I. IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 1, 1979, no. 2, 173–184.

[70] Haralick, R.M. – Shapiro, L.G.: The consistent labeling problem: part II. IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 2, 1980, no. 3, 193–203.

[71] Heinze N. – Michaylov S. – Stuckey P.: CLP($\mathcal{R}$) and some electrical engineering problems. Journal of Automated Reasoning, vol. 9, 1992, 231–260.

[72] Hentenryck, P.: Constraint satisfaction in logic programming (Logic Programming Series). The MIT Press, Cambridge, MA, 1989.

[73] Hentenryck, P. – Carillon, J.P.: Generality versus specificity: an experience with AI and OR techniques. In: Proc. of the Seventh National Conference on Artificial Intelligence AAAI'88. Saint Paul, Minesota, 1988, 660–664.

[74] Hentenryck, P. – Saraswat, V. – Deville, Y.: Design, implementations, and evaluation of the constraint language cc(FD). Technical Report CS-93-02. CS Department, Brown University, 1993.

[75] Holzbaur, C.: OFAI clp(q,r) manual, edition 1.3.3. Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

[76] Imbert, J.L. – Hentenryck, P.: On the handling of disequations in CLP over linear rational. In: Benhamou, F. – Colmerauer, A. (eds.): Constraint Logic Programming. Selected Research. MIT Press, 1993, 49–72.

[77] Jaffar, J. – Lassez, J. L.: Constraint logic programming. In: Proc. of the 14th Symposium POPL'87. Munich, Jan. 1987, 111–119.

[78] Jaffar, J. – Maher, M. : Constraint logic programming: A survey. Journal of Logic Programming, vol. 19, 1994, 503–581.

[79] Jaffar, J. – Michaylov, S.: Methodology and implementation of a CLP system. In: Proc. of the 4th International Conference on Logic Programming. MIT Press, Melbourne, 1987, 196–217.

[80] Jeavons, P. – Cohen, D. – Cooper, M.: A substitution operation for constraints. In: Borning, A. (ed.): Principles and Practice of Constraint Programming (LNCS 874). Springer Verlag, Heidelberg, 1994, 1–9.

[81] Jeavons, P. – Cohen, D. – Cooper, M.: Constraints, consistency and closure. Artificial Intelligence, vol. 101, 1998, 251–265.

[82] Kondrak, G. – Van Beek, P.: A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence, vol. 89, 1997, 365–387.

[83] Kumar, V.: Algorithms for constraint satisfaction problems: a survey. The AI Magazine, vol. 13, 1992, no. 1. 32–44.
http://www-users.cs.umn.edu/~kumar/

[84] Kwan, A. – Tsang, E.: Minimal forward checking with backmarking. Technical report CSM-260. Dept. of Computer Science, University of Essex, Colchester, UK, 1996, 28 p.
ftp://ftp.essex.ac.uk/CSP/papers/CSM-260.ps.Z

[85] Liu, B.: An improved generic arc consistency algorithm and its specializations. In: Foo, N. – Goebel,R. (eds.): PRICAI'96: Topics in artificial intelligence (LNAI 1114). Springer Verlag, Heidelberg, 1996, 264–275.

[86] Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence, vol. 8, 1977, 99–118.

[87] Mackworth, A.K. – Freuder, E.C.: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence, vol. 25, 1985, 65–74.

[88] Mackworth, A.K. – Freuder, E.C.: The complexity of constraint satisfaction revisited. Artificial Intelligence, vol. 59, 1993, 57–62.

[89] Mach, M.: A generic knowledge based problem solving method for parameter design. In: Proc. of the 12th Int. Conference on Process Control and Simulation ASRTP'96. Zlatá Idka, 1996, 449–454.

[90] Mach, M.: Experimental analysis of CSE. ENCODE Project report TR-Encode-TUKE-2-97. Technická univerzita v Košiciach, február 1997. 105 s.

[91] Mayoh, B. – Tyungu, E. – Uustalu, T.: Constraint satisfaction and constraint programming: A brief lead-in. In: B. Mayoh, E. Tyungu, J. Penjam (eds.): Constraint Programming. Springer Verlag, Heidelberg, 1994.

[92] McAllester, D.: Constraint satisfaction search. Lecture Notes for 6.824, Artificial Intelligence. 1992.
http://www.ai.mit.edu/people/dam/notes/csp.ps

[93] Meier, M.: Controlling pollution in the Venice lagoon. ECRC Today, December 1994.

[94] Meiri, I. – Dechter, R. – Pearl, J.: Uncovering trees in constraint networks. Artificial Intelligence, vol. 86, 1996, 245–267.

[95] Mohr, R. – Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence, vol. 28, 1986, 225–233.

[96] Montanari, U.: Networks of constraints: fundamental properties and applications to picture processing. Information Sciences, vol. 7, 1974, 95–132.

[97] Montanari, U. – Rossi, F.: Constraint relaxation may be perfect. Artificial Intelligence, vol. 48, 1991, 143–170.

[98] Montanari, U. – Rossi, F.: Finite domain constraint solving and constraint logic programming. In: Benhamou, F. – Colmerauer, A. (eds.): Constraint Logic Programming, Selected Research. MIT Press, 1993, 201–221.

[99] Mudambi, S. – Prestwich, S.: Improved branch and bound in constraint logic programming. In: Proc. of the 1st International Conference on Principles and Practice of Constraint Programming, CP'95. Cassis, France, September 1995, 533–548.

[100] Nudel, B.: Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics. Artificial Intelligence, vol. 21, 1983, 135–178.

[101] Paltrinieri, M.: On the design of constraint satisfaction problem. In: Borning, A. (ed.): Principles and Practice of Constraint Programming (LNCS 874). Springer Verlag, Heidelberg, 1994, 299–311.

[102] Pánik, M.: Rozšírenie expertného systému CLIPS o mechanizmus spĺňania ohraničení. Diplomová práca, Katedra kybernetiky a umelej inteligencie Technickej univerzity v Košiciach, 1997.

[103] Paralič, J.: Constraint logic programming - An overview. Rigorózna práca. Univ. of Technology in Kosice, Slovakia & Internal Technical Report ECRC Munich, Germany, 1995.

[104] Paralič, J.: Riešenie úloh rozvrhovania logickým programovaním ohraničení. Dizertačná práca. Technická univerzita v Košiciach, máj 1997.

[105] Paralič, J. – Sabol. T. – Mach, M. – Hatala, M.: Configuration design as constraint satisfaction. In: Proc. of the Int. Conference microCAD'96. Miskolc-Egyetemváros, 1996, 64–68.

[106] Paralič, J. et al.: Constraint satisfaction engine for the ENCODE workbench. In: Proc. of the 8th Int. DAAAM Symposium. Dubrovnik, 1997, 255–256.

[107] Prosser, P.: Forward checking with backmarking. In: Meyer, M. (ed.): Constraint Processing (LNCS 923). Springer Verlag, Heidelberg, 1995, 185–203.

[108] Puget, J.F.: A C++ Implementation of CLP. Technical Report 94-01. ILOG S.A., Gentilly, France, 1994.

[109] Regin, J.-Ch.: A filtering algorithm for constraints of difference in CSPs. In: Proc. of the 12th National Conference on Artificial Intelligence AAAI'94. Seattle, Washington, 1994, 362–367.

[110] Rossi, F.: Redundant hidden variables in finite domain constraint problems. In: Meyer, M. (ed.): Constraint Processing (LNCS 923). Springer Verlag, Heidelberg, 1995, 205–223.

[111] Sabin, D. – Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: Borning, A. (ed.): Principles and Practice of Constraint Programming (LNCS 874). Springer Verlag, Heidelberg, 1994, 10–20.

[112] Sabol, T.: Parallel solution of a system of declarative constraints using local heuristics. Trans. of the Technical University of Košice, vol. 4, 1994, no. 3-4, 249–254.

[113] Sadeh, N. – Fox, M.S.: Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. Artificial Intelligence, vol. 86, 1996, 1–41.

[114] Sakkout, H.E. – Wallace, M.G. – Richards, E.B.: An instance of adaptive constraint propagation. In: Freuder, E.C. (ed.): Principles and Practice of Constraint Programming CP'96 (LNCS 1118). Springer Verlag, Heidelberg, 1996, 164–178.

[115] Schiex, T. – Regin, J-C. – Gaspin, Ch. – Verfaillie, G.: Lazy arc consistency. In: Proc. of the Thirteenth National Conference on Artificial Intelligence AAAI'96. Portland, Oregon, 1996, 216–221.
http://www-bia.inra.fr/T/schiex/Doc/publisE.html

[116] Schiex, T. – Verfaillie, G.: Nogood recording for static and dynamic constraint satisfaction problems. Int. Journal of Artificial Tools, vol. 3, 1994, no. 2, 187–207.
http://www-bia.inra.fr/T/schiex/Doc/publisE.html

[117] Schiex, T. – Verfaillie, G.: Stubbornness: a possible enhancement for backjumping and nogood recording. In: Proc. of the 12th ECAI Conference. Amsterdam, 1994, 165–169.
http://www-bia.inra.fr/T/schiex/Doc/publisE.html

[118] Schimpf, J. – Shen, K. – Wallace, M.: Tutorial on search in ECL^iPS^e. Imperial College, London, June 2000, 31 p.

[119] Schmotzer, M.: Analýza a návrh nových prostriedkov pre riešenie úloh časového rozvrhovania v prostredí logického programovania ohraničení. Diplomová práca. Katedra kybernetiky a umelej inteligencie, Technická univerzita Košice, 1997.

[120] Smith, B.M.: A tutorial on constraint programming. Research report 95.14. School of Computer Studies, University of Leeds, Leeds, UK, 1995, 21 p. http://www.scs.leeds.ac.uk/bms/papers.html

[121] Smith, B.M.: Succeed-first or fail-first: a case study in variable and value ordering. Research report 96.26. School of Computer Studies, University of Leeds, Leeds, UK, 1996, 11 p. http://www.scs.leeds.ac.uk/bms/papers.html

[122] Smith, B.M. – Grant, S.A.: Trying harder to fail first. Research report 97.45. School of Computer Studies, University of Leeds, Leeds, UK, 1997, 19 p. http://www.scs.leeds.ac.uk/bms/papers.html

[123] Stallman, R.M. – Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence, vol. 9, 1977, 135–196.

[124] Sutherland, I.: Sketchpad: a man-machine graphical communication system. In: Proc. IFIP Spring Joint Computer Conference, 1963.

[125] Tsang, E.: No more "partial" and "full looking ahead". Artificial Intelligence, vol. 98, 1998, 351–361.

[126] Van Beek, P.: On the inherent level of local consistency in constraint networks. In: Proc of the Twelfth National Conference on Artificial Intelligence AAAI'94. Seatle, WA, 1994, 368–373.

[127] Van Beek, P. – Dechter, R.: On the minimality and global consistency of row-convex constraint networks. Journal of the ACM, vol. 42, 1995, no. 3, 543–561.

[128] Van Beek, P. – Dechter, R.: Constraint tightness and looseness versus local and global consistency. Journal of the ACM, vol. 44, 1997, no. 4, 549–566.

[129] Van Hentenryck, P. – Deville, Y. – Teng, Ch.: A generic arc-consistency algorithm and its specializations. Artificial Intelligence, vol. 57, 1992, 291–321.

[130] Waltz, D.: Understanding line drawings of scenes with shadows. In: Winston, P.H. (ed.): The psychology of computer vision. McGraw Hill, New York, 1975, 19–91.

[131] Weigel, R. – Bliek, Ch. – Faltings, B.V.: On reformulation of constraint satisfaction problems (extended version). In: Proc. of SARA'98 Conference, 1998. http://liawww.epfl.ch/~weigel/papers.html

[132] Zdráhal, Z. – Motta, E. – Stutt, A.: Optimising revision procedures for configuration design. Technical Report TR-112. Open University, UK, 1994.

[133] Zweben, M. – Eskey, M.: Constraint satisfaction with delayed evaluation. In: Proc. of the 11th Int. Joint Conference on Artificial Intelligence IJCAI'89. Detroit, Michigan, 1989, 875–880.

Register

Autori:	Marián Mach, Ján Paralič
Názov:	Úlohy s ohraničeniami: od teórie k programovaniu
Vydanie:	prvé
Náklad:	100
Rozsah:	217 strán
Vydavateľ:	elfa s.r.o., Letná 9, Košice
Sadzba:	$\LaTeX$
Formát:	b5
Tlač:	elfa s.r.o., Letná 9, Košice
Vytlačené:	December 2000

ISBN 80-88964-48-2