Pokroky matematiky, fyziky a astronomie

Roman Barták

Programování s omezujícími podmínkami — na cestě ke svatému grálu

Pokroky matematiky, fyziky a astronomie, Vol. 45 (2000), No. 3, 218--231

Persistent URL: `http://dml.cz/dmlcz/141039`

Terms of use:

© Jednota českých matematiků a fyziků, 2000

Institute of Mathematics of the Academy of Sciences of the Czech Republic provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This paper has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* `http://project.dml.cz`

[8] Hajič, J., Ribarov, K.: *Rule-Based Dependencies.* In *Proceedings of the Workshop on the Empirical Learning of Natural Language Processing Tasks*, 125–136, Praha 1997.
[9] Hajičová, E.: *The Past and the Present of Computational Linguistics at Charles University.* Technická zpráva TR-1996-01, Ústav formální a aplikované lingvistiky, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 1996.
[10] Hajičová, E., Panevová, J., Sgall, P.: *Language Resources Need Annotations To Make Them Really Reusable: The Prague Dependency Treebank.* In *Proceedings of the First International Conference on Language Resources & Evaluation*, 713–718. Granada, Španělsko 1998.
[11] Hajičová, E., Panevová, J., Sgall, P.: *Manuál tektogramatického značkování.* Technická zpráva TR-1999-07, Ústav formální a aplikované lingvistiky, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 1999.
[12] Hladká, B.: *Czech Language Tagging.* Doktorská práce, Ústav formální a aplikované lingvistiky, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 2000.
[13] Kuboň, V., Holan, T., Plátek, M.: *A Grammar-Checker for Czech.* Technická zpráva TR-1997-02, Ústav formální a aplikované lingvistiky, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 1997.
[14] Al-Onaizan, Y., **Cuřín, J.,** Jahr, M., Knight, K., Lafferty, J., Melamed, D., Och, F.-J., Purdy, D., Smith, N. A., Yarowsky, D.: *The Statistical Machine Translation.* In *Technical Report, NLP WS '99.* Center for Language and Speech Processing, Johns Hopkins University, Baltimore, Maryland, USA 1999.

Programování s omezujícími podmínkami — na cestě ke svatému grálu

Roman Barták, Praha

Motto:
„Programování s omezujícími podmínkami představuje jedno z největších přiblížení, jaké kdy informatika udělala k nalezení svatého grálu programování: uživatel zadá problém a počítač ho vyřeší."

E. Freuder, časopis Constraints, duben 1997

Začínajícím programátorům se často zdůrazňuje, že počítač udělá přesně a jen to, co je mu zadáno. Za programování počítačů se pak považuje přesný formální

Mgr. Roman Barták, Dr. (1970), Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, katedra teoretické informatiky a matematické logiky, Malostranské náměstí 2/25, 118 00 Praha 1, e-mail: `bartak@kti.mff.cuni.cz`

Autorova práce je podporována Grantovou agenturou České republiky projektem č. 201/99/D057.

popis postupu, jak daný problém vyřešit, v řeči stroje. Tento způsob programování, nazývaný imperativní nebo také procedurální programování, vychází z myšlenky, že počítači je třeba přesně popsat algoritmus, tj. postup řešení problému. Při strojovém řešení problému tedy musíme sami nejprve vědět, jak daný problém vyřešit. Existuje ovšem i jiný způsob, jak počítač přimět k řešení problémů, a ten nese název deklarativní programování. Deklarativní programování je založeno na myšlence zadávat stroji pouze to, co má řešit, a nestarat se o to, jak konkrétně se má daný problém vyřešit. V následujících odstavcích se podíváme podrobněji na jednoho z mladších a přesto již komerčně velmi úspěšných reprezentantů deklarativního programování, na programování s omezujícími podmínkami. Začněme ale pohledem do historie.

1. Od Prologu k CLP

Logické programování a jeho hlavní implementace v podobě programovacího jazyka Prolog představuje asi nejznámější přístup k deklarativnímu řešení problémů. Ostatně slogan „Neříkejte počítači, jak má problém řešit, ale co má řešit“ najdeme nejčastěji právě ve spojení s programovacím jazykem Prolog. Programování v Prologu je dosti odlišné od všeho, na co jsou tradiční programátoři zvyklí v imperativních (procedurálních) jazycích. Na rozdíl od jazyků, jako je C, C++, Java, Pascal či Fortran, kde je program popsán sekvencí instrukcí, je problém v Prologu formulován pomocí sady pravidel popisujících spíše logický charakter problému než způsob, jak konkrétní problém vyřešit. Při zadání problému potom vestavěný řešící systém sám vybere a poskládá pravidla nutná k jeho vyřešení, přičemž používá mechanismus syntaktického porovnání termů tzv. unifikace a prohledávání s navracením (backtracking). Unifikace je vlastně rovnost nad doménou všech termů nazývanou též Herbrandovo universum, takže například term `f(X)` je unifikovatelný s termem `f(2)` (velkými písmeny se v Prologu tradičně označují proměnné, tj. po dosazení `X = 2` dostaneme syntakticky identické termy), ale term `f(X)` není unifikovatelný s termem `3`.

```
stryc(X,Y) :- bratr(X,Z), rodic(Z,Y).      % strýc je bratrem některého z rodičů
rodic(X,Y) :- otec(X,Y).                   % rodičem je buď otec
rodic(X,Y) :- matka(X,Y).                  % nebo matka
bratr(X,Y) :- sourozenci(X,Y), muz(X).     % bratr je sourozenec, který je mužem
sourozenci(X,Y) :- rodic(Z,X), rodic(Z,Y). % sourozenci mají společného rodiče
```

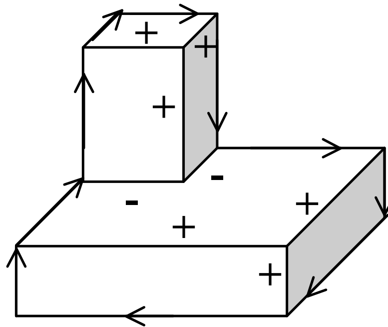
Příklad zadání programu v Prologu (jednoduchá genealogická databáze)

Bohužel Herbrandovo universum je přece jen sémanticky vzdálenější od běžných domén, takže v Prologu například termy `1 + 2` a `3` nejsou unifikovatelné (syntakticky identické), i když běžný rozum by říkal, že jde o tytéž objekty. Právě neinterpretovanost Herbrandova universa vedla v polovině 80. let ke vzniku *logického programování s omezujícími podmínkami* (CLP — Constraint Logic Programming). Nejprve Gallaire [12] a po něm Jaffar a Lassez [16] navrhli obecné schéma CLP(D), kde je Herbrandovo universum nahrazeno konkrétní doménou D (například celými nebo reálnými čísly) a místo unifikace se používá rovnost nad touto doménou. Přirozeně nic nám potom

nebrání používat v CLP programech i další relace/podmínky nad konkrétní doménou, například porovnání.

2. Interdisciplinární začátky

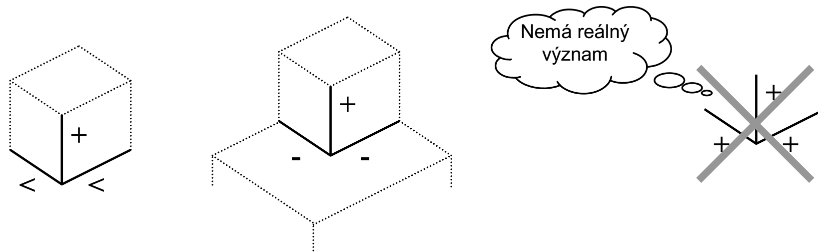
Definice CLP, přestože jde nepochybně o jeden z hlavních kroků vedoucích k programování s omezujícími podmínkami, ovšem nebyla tím úplně prvním použitím omezujících podmínek. Vůbec prvním problémem, který byl formulován pomocí omezujících podmínek, je analýza scény, jejímž cílem je rozpoznat trojrozměrné objekty a vztahy mezi nimi z dvojrozměrného snímku scény. Asi nejdůležitějším krokem v analýze scény je ohodnocení hran (čar) vyskytujících se v obrázku jedním z typů konvexní, konkávní nebo hraniční hrana. Prozkoumat všechna možná ohodnocení hran je vzhledem k jejich počtu přirozeně výpočtově neúnosné (3^N možností pro N hran a 3 hranové typy), a tak nastupuje technika dnes známá jako propagace podmínek (constraint propagation).



Obr. 1. Ohodnocení hran v obrázku tvoří jednu z hlavních částí analýzy scény (+ značí konvexní hrany, − hrany konkávní a šipkou jsou označeny okrajové hrany objektu).

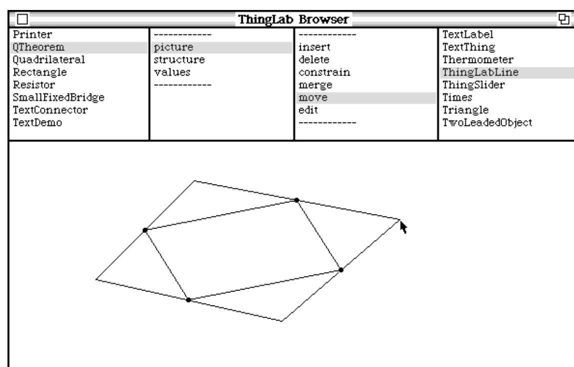
Základní myšlenka ohodnocení hran, jak ji navrhl Waltz [34] a rozpracovali další, je založena na pozorování, že v místech, kde se hrany stýkají, existuje jen omezené množství korektních ohodnocení hran a že hrana zachovává po celé své délce stejný typ (u běžných objektů). Ohodnocování hran se potom provádí tak, že se vybere nějaké přípustné ohodnocení u jednoho z vrcholů a toto ohodnocení se přes hrany propaguje do dalších vrcholů. Znalost typu některých hran u dalších vrcholů vede k omezení počtu povolených ohodnocení v rámci vrcholu, čímž se prohledávaný prostor výrazně redukuje. Pokud se zjistí, že nelze v dalším ohodnocování pokračovat, tj. neexistuje povolené ohodnocení hran v nějakém vrcholu, program se prostě vrátí k poslednímu vrcholu, kde jsou ještě k dispozici další povolená ohodnocení, a zkusí jiné ohodnocení. Tato technika prohledávání s navracením je dnes základem řady řešících algoritmů a v dalším textu se k ní ještě vrátíme.

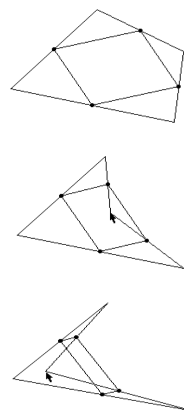
Omezující podmínky se ve svých počátcích objevily i ve zcela jiném kontextu, konkrétně v oblasti *interaktivní grafiky.* Počátkem šedesátých let vyvinul Ivan Sutherland systém Sketchpad [30] pro kreslení a manipulaci geometrických objektů



Obr. 2. Ohodnocování hran využívá pozorování, že pouze relativně malé množství kombinací typů má reálný význam.

na počítačové obrazovce. Na jeho práci potom navázal podobně zaměřený systém ThingLab Alana Borninga [4], těžící z právě vznikající podoby grafických uživatelských rozhraní. Geometrické objekty a jejich vztahy se v těchto systémech popisovaly matematickými formulemi, o jejichž převedení na explicitní tvar, tj. vypočtení hodnot proměnných = zjištění pozic bodů, se staral systém sám. Pokud uživatel změnil polohu nějakého objektu, systém automaticky upravil polohu ostatních objektů tak, aby vztahy mezi objekty zůstaly v platnosti. Techniky vyvinuté v těchto pionýrských systémech se později staly základem metod lokální propagace a kompilace podmínek umožňujících rychlou odezvu řešiče podmínek nezbytnou pro interaktivní prostředí.





Obr. 3. ThingLab nabízí interaktivní práci s geometrickými objekty (vlevo – grafické rozhraní, vpravo – postupné změny tvaru objektu při tažení jednoho z bodů).

3. Základní řešící techniky

Jak ukazují předchozí odstavce, cest vedoucích ke vzniku *programování s omezujícími podmínkami* (CP – Constraint Programming) bylo hned několik, a tak se nelze divit, že dnešní CP je souhrnem celé řady někdy dosti odlišných řešících technik [17, 21, 31, 32]. Programování s omezujícími podmínkami jako celek je obor věnující se řešení problémů popsaných soustavou omezení. Omezení nebo jinak též omezující podmínka

je libovolná relace, jejíž splnění je při řešení problému vyžadováno. Relace mohou být definovány nad nekonečnými spojitými doménami, např. reálných čísel, potom se pro jejich řešení nejčastěji používají techniky operačního výzkumu (simplexová metoda) nebo numerické matematiky (Newtonova interpolační metoda, Taylorovy rozvoje). Pro CP je ovšem mnohem typičtější práce s konečnými diskrétními doménami, které jsou použity snad v 95 % všech aplikací CP. V následujícím textu se proto soustředíme na techniky řešení podmínek nad konečnými doménami. V takovém případě hovoříme o *problému splňování podmínek* (CSP – Constraint Satisfaction Problem), který je formálně definován jako:

- konečná množina proměnných,
- každé proměnné je přiřazena konečná doména, tj. množina možných hodnot,
- konečná množina podmínek omezujících hodnoty, které mohou proměnné současně nabývat.

Řešením CSP problému je potom ohodnocení všech proměnných, tj. vybrání hodnoty proměnné z příslušné domény tak, že všechny podmínky jsou splněny. Někdy stačí nalézt jediné přípustné řešení, jindy se požaduje nalézt všechna řešení a často se setkáme s hledáním optimálního řešení, kde kvalita řešení je určena objektivní funkcí nad proměnnými problému.

3.1. Systematické prohledávání

Asi nejjednodušším způsobem řešení CSP problému je prohledávání prostoru všech ohodnocení proměnných a hledání takového ohodnocení, které splňuje všechny podmínky. Nejjistější metodou je pak systematické prohledávání takového prostoru, které nám zaručí, že neopomineme žádné řešení.

Metoda, která zabere při řešení libovolného problému s konečně velkým prohledávaným prostorem, se jmenuje *generuj a testuj*. Přestože o její efektivitě lze pochybovat, je určitě dobré si její princip a nedostatky připomenout, protože právě na odstranění nedostatků jsou založeny další, již mnohem efektivnější metody. Tedy, technika generuj a testuj prohledává prostor všech ohodnocení proměnných tak, že systematicky generuje možná ohodnocení a ve druhé fázi testuje, zda dané ohodnocení splňuje všechny podmínky. Pokud tomu tak je, řešení je nalezeno, pokud ne, vygeneruje se další dosud neprověřené ohodnocení. V této obecné metodě lze odhalit asi dva zásadní nedostatky, které mají negativní vliv na efektivitu výpočtu:

- neinformovanost generátoru ohodnocení, tj. pokud testovací fáze selže (nějaká podmínka je nesplněna), generátor slepě vezme další ohodnocení podle pravidla systematického procházení a nebere v úvahu konkrétní důvod selhání testu (často se potom procházejí podobná ohodnocení se stejnou „chybou“),
- příliš pozdní odhalení nekonzistencí, tj. nesplnění podmínky je zjištěno až po ohodnocení všech proměnných.

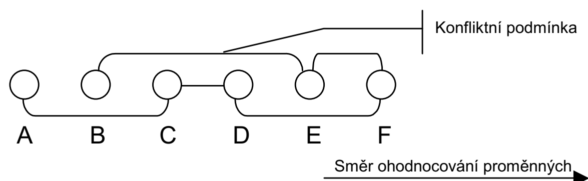
O generátorech, které využívají informací z testovací fáze, budeme hovořit později (nepatří mezi systematické metody). Podívejme se nyní na metodu snažící se odhalit nekonzistence dříve než po ohodnocení všech proměnných. Generuj a testuj provádí test podmínek až poté, co je vygenerováno nějaké úplné ohodnocení proměnných. Test konkrétní podmínky lze ovšem provést již ve chvíli, kdy jsou ohodnoceny proměnné zahrnuté v této podmínce (např. podmínku `X > 5` lze otestovat, jakmile známe hodnotu přiřazenou proměnné `X`). V případě nesplnění podmínky na nějakém částečném ohodnocení proměnných tedy není potřeba ohodnocovat ostatní proměnné, čímž se prohledávaný prostor výrazně zredukuje (např. všechna ohodnocení obsahující `X = 4` jsou při podmínce `X > 5` nekonzistentní a nemusí být dále zkoumána).

Na principu, kdy jsou proměnné ohodnocovány postupně a podmínky jsou testovány okamžitě po ohodnocení všech proměnných svázaných podmínkou, je založena další metoda systematického prohledávání, známá jako *prohledávání s navracením* (backtracking) [26]. Prohledávání s navracením tak vlastně postupně rozšiřuje částečné konzistentní ohodnocení (podmínky nad již ohodnocenými proměnnými jsou splněny) na úplné konzistentní ohodnocení (všechny proměnné jsou ohodnoceny a všechny podmínky jsou splněny). Pokud nějaká podmínka nad částečným ohodnocením proměnných není splněna, zkusí se prostě další hodnota z domény posledně ohodnocené proměnné, a pokud je tato doména již vyčerpána, vrátí se algoritmus k předposlední ohodnocené proměnné atd. Prohledávání s navracením tak zachovává úplnost metody generuj a testuj (pokud řešení existuje, je nalezeno), je ale vždy efektivnější, protože řada nekonzistentních ohodnocení je „uříznuta“ dřívějším testováním podmínek. Přes svoji jednoduchost je prohledávání s navracením velice často využíváno při řešení problémů, jde vlastně o základní řešící mechanismus Prologu. Ani tato technika ovšem není zbavena všech neřestí a postupem času byly identifikovány tři základní nedostatky prohledávání s navracením, jejichž překonání se stalo základem dalších řešících technik:

- thrashing, neboli opakovaný neúspěch kvůli stejnému důvodu,
- redundantní práce,
- stále pozdní odhalení nekonzistence.

Všechny tyto problémy si můžeme přiblížit na příkladu binární podmínky `B > E`. Předpokládejme, že domény obou proměnných se skládají z prvků `1`,…,`5` a že proměnné jsou ohodnocovány v abecedním pořádku (k proměnným `B`, `E` uvažujme ještě proměnné `A`, `C`, `D` a `F`). Dále předpokládejme, že jsme proměnnou `B` ohodnotili `1` a poprvé na řadě při ohodnocování je proměnná `E`. Přirozeně žádnou hodnotu z domény proměnné `E` nelze vybrat tak, aby podmínka `B > E` byla splněna, což vyvolá navracení. Tradiční, tzv. chronologický backtracking tedy změní hodnotu proměnné `D` (tato proměnná byla ohodnocena těsně před `E`) a opět se snaží ohodnotit proměnnou `E`. To se opakuje tak dlouho, dokud se postupným navracením nedostaneme k proměnné `B` (po vyčerpání domén proměnných `C` a `D`). Protože se zahazuje informace o důvodu neúspěchu a tento neúspěch je opakovaně (a zbytečně) opakován, hovoříme o tzv. thrashingu. Problému thrashingu se lze vyhnout tak, že se každý neúspěch při pokusu přiřadit hodnotu proměnné analyzuje a tato analýza se použije při navracení. Metoda, která tuto techniku realizuje, se příznačně nazývá *backjumping* [11] a v našem příkladě

by doporučila skočit rovnou na proměnnou B. Tím se vyhneme řadě zbytečných pokusů, na druhou stranu je třeba říci, že analýza neúspěchu něco stojí a jeden krok backjumpingu je proto výpočtově náročnější než u backtrackingu. Z tohoto důvodu se někdy používá jednodušší analýza, tzv. grafem (podmínek) řízený backjumping, který v případě konfliktu doporučuje návrat k nejbližší „provázané" (přes podmínku) proměnné (v našem příkladě by to shodou okolností byla též proměnná B).



Obr. 4. Backjumping při navracení skáče na konfliktní proměnnou a zlepšuje tak tradiční chronologický backtracking. Při neúspěchu ohodnocení proměnné E doporučí vrátit se k proměnné B, protože změny hodnot proměnných C a D konflikt podmínky neodstraní.

Předchozí příklad může posloužit i k ukázce redundantní práce při backtrackingu. Pokud je proměnné B již přiřazena hodnota 2, podaří se proměnnou E ohodnotit, a to pouze hodnotou 1, ostatní hodnoty nejsou konzistentní. Jestliže je nyní při ohodnocování proměnné F vyvolán backtracking až k proměnné D, podmínka B > E je opětovně testována, aby byla nalezena správná hodnota proměnné E. Tento test je ovšem zbytečný, stačí si pamatovat, že jediná správná hodnota pro E je 1 a ostatní hodnoty jsou s B nekonzistentní. To přirozeně platí jen do té doby, dokud není při navracení změněna hodnota proměnné B, potom je nutné test opakovat. Pamatování si nekompatibilních hodnot je základem metody s názvem *backchecking*, která byla dále zdokonalena do podoby *backmarkingu* [14], jenž si pamatuje i provedené úspěšné testy. Podobně jako backjumping mohou tyto metody ušetřit spousty zbytečného prohledávání a testů podmínek, přirozeně za cenu větší výpočtové náročnosti jednoho kroku.

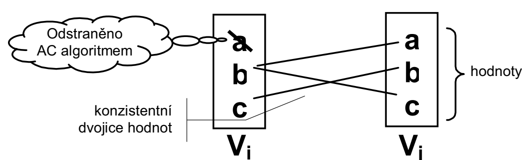
Popsané metody zlepšení prohledávání s navracením mají společný název inteligentní backtracking, se kterým se opět můžeme setkat i u implementací logického programování. V CP patří tyto metody do kategorie tzv. look-back (pohled zpět) algoritmů, pro které je typická analýza nekonzistence a doporučení dalšího postupu podle výsledku.

Pozorný čtenář si jistě všiml, že u prohledávání s navracením je zmíněn ještě jeden nedostatek, a to příliš pozdní odhalení nekonzistence. U obou předchozích vylepšení se vždy předpokládalo, že inkriminovaná podmínka B > E je testována až při ohodnocení proměnné E. Pokud bychom podmínku aktivně použili již při přiřazení hodnoty proměnné B, okamžitě bychom zjistili, že hodnota 1 je pro tuto proměnnou nevhodná (nelze najít žádnou hodnotu pro E tak, aby podmínka platila). Pokud pro B zvolíme hodnotu 2, můžeme E okamžitě přiřadit hodnotu 1 atd. Tento princip používá pohledu dopředu a je podrobněji rozebrán v následující kapitole.

3.2. Propagace podmínek

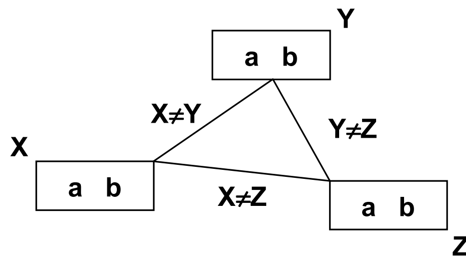
Bohužel mnoho odborníků, zvláště z tradičních teoretických disciplín, zůstává u prvního pohledu na CP a považuje tento přístup k řešení problémů pouze za jednoduchou enumeraci, jak byla prezentována v předchozí kapitole. Pravdou je, že ohodnocení proměnných je často jedním z kroků řešení podmínek, hlavní síla CP je ovšem ukryta jinde. Již Waltzův algoritmus ohodnocování hran ve scéně byl založen na technice propagace podmínek (constraint propagation), která výrazně omezuje prohledávaný prostor. Tato propagace není z obecného pohledu nic jiného než aktivní využití omezujících podmínek spojením tzv. konzistenčních technik s prohledáváním.

Konzistenční techniky [19, 20] jsou způsobem řešení podmínek založeným na vyřazování nekonzistentních hodnot z domén proměnných. Například u podmínky `B > E`, kdy počáteční domény obou proměnných jsou `1`,…,`5`, můžeme okamžitě vyřadit hodnotu `1` z domény proměnné `B` a `5` z domény pro `E`. Důvodem vyřazení je neexistence kompatibilní hodnoty v propojené proměnné. Po zmenšení domény se přirozeně může porušit konzistence další podmínky (např. `F > E`), a tak je potřeba testy konzistence podmínek provádět opakovaně, dokud se doména některé proměnné zmenšuje. Tento postup je základem konzistenčního algoritmu AC-1, který patří do třídy algoritmů udržujících tzv. *hranovou konzistenci* (arc consistency). Tato konzistenční technika dostala svůj název od reprezentace CSP problému formou grafu podmínek, kde vrcholy představují proměnné a hrany podmínky. Obecně jde sice o multi-graf, lze ovšem snadno nahlédnout, že libovolný CSP problém lze převést na ekvivalentní binární CSP, kde se vyskytují pouze binární podmínky (viz např. [2]). AC algoritmů byla vyvinuta celá řada, mezi nejznámější patří AC-3, který provádí jen nezbytné konzistenční testy, a AC-4, pracující přímo s dvojicemi konzistentních hodnot.



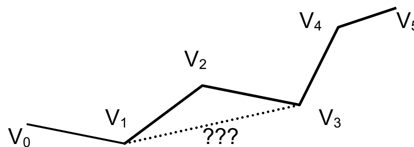
Obr. 5. Hranová konzistence umožňuje vyřadit nekonzistentní hodnoty vzhledem k jedné podmínce. Hodnota `a` z domény proměnné V_i je odstraněna, protože pro ni neexistuje kompatibilní hodnota v doméně proměnné V_j.

Zajištění hranové konzistence ovšem na úplné vyřešení problému obecně nestačí (viz obrázek 6), a tak vznikly další konzistenční metody. Přirozeným pokračováním hranové konzistence je *konzistence po cestě* (path consistency), která se zabývá zajištěním konzistence hodnot proměnných po nějaké cestě v grafu podmínek. Formální definice říká, že binární CSP je konzistentní po cestě, pokud pro každou dvojici hodnot dvou proměnných `X` a `Y`, která splňuje všechny binární podmínky mezi těmito proměnnými, existuje hodnota pro každou proměnnou na cestě mezi proměnnými `X` a `Y` tak, že všechny binární podmínky na této cestě jsou splněny. V [24] Montanari ukázal, že CSP je konzistentní po cestě, právě když je konzistentní po cestách délky 2, čehož



Obr. 6. Ani po zajištění hranové konzistence nemusí být vyřazeny všechny nekompatibilní hodnoty. Graf podmínek na obrázku je hranově konzistentní a přesto nemá řešení.

s úspěchem využívají algoritmy PC (path consistency) pro zajištění konzistence po cestě. Praxe ovšem ukázala, že navýšení počtu vyřazených nekonzistentních hodnot z domén proměnných není proti AC algoritmům tak výrazné, aby se vyplatily výrazně vyšší výpočtové nároky PC algoritmů. Navíc ani PC algoritmy nemohou obecně zajistit úplné vyřešení problému, tj. vyřazení všech nekonzistentních hodnot (do problému z obrázku 6 stačí přidat další proměnnou spojenou nerovnostmi a o jeden prvek zvětšit domény).



Obr. 7. Konzistence po cestě se zajímá pouze o podmínky podél cesty, tj. o podmínky mezi proměnnými V_i a V_{i+1}. Splnění podmínky mezi proměnnými V_1 a V_3 se nevyžaduje.

V posilování konzistenčních technik lze pokračovat dále až po úplné vyřešení CSP problému pomocí silné N-konzistence (definice viz [9, 31]), kde N je počet proměnných v problému. Existují sice algoritmy pro dosažení této úrovně konzistence, jejich výpočtová složitost je ovšem srovnatelná se složitostí algoritmů systematického prohledávání, jejichž implementace je neporovnatelně jednodušší.

Jako zajímavá cesta řešení CSP problémů se nakonec ukázalo již naznačené spojení systematického prohledávání s konzistenčními technikami. Princip spojení je poměrně jednoduchý, po přiřazení hodnoty proměnné se spustí konzistenční algoritmus, který z domén ostatních proměnných vyřadí nekompatibilní hodnoty. Právě přiřazená hodnota se tak vlastně přes podmínky propaguje do dalších proměnných, odtud název *propagace podmínek* (constraint propagation). Nejčastěji se při propagaci používají AC algoritmy, případně jejich slabší verze (directed AC). Podle síly použitého konzistenčního algoritmu hovoříme o *dopředné kontrole* (forward checking), kdy se propagace provádí pouze přes podmínky obsahující právě ohodnocovanou proměnnou, nebo o *pohledu dopředu* (look ahead), kdy se provádí úplná hranová konzistence mezi dosud neohodnocenými proměnnými. Pokud se při propagaci vyprázdní doména nějaké proměnné, víme okamžitě, že současné částečné ohodnocení je nekonzistentní (resp. nelze

je rozšířit na úplné konzistentní ohodnocení) a je možné vyvolat navracení. Případné nekonzistence se v tomto případě odhalí dříve než při prostém backtrackingu, čímž se ušetří zbytečné prohledávání. Podobně jako u již zmíněných vylepšení backtrackingu i zde platíme zvýšenou složitostí jednotlivých kroků. V konečném důsledku se ale tato práce navíc (vyřazování nekonzistentních hodnot z dosud neohodnocených proměnných) většinou vyplatí a propagace podmínek tvoří základ většiny algoritmů pro řešení systémů podmínek.

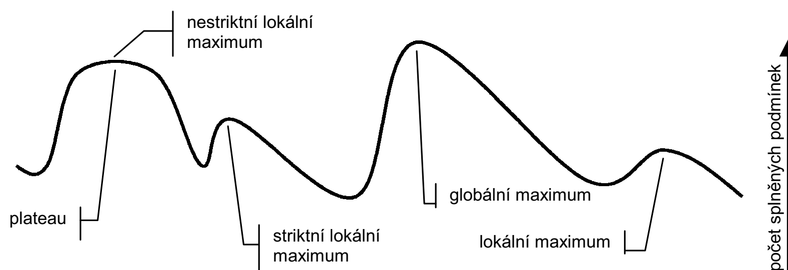
3.3. Lokální prohledávání

Kromě tradičního spojení propagace podmínek a enumerace proměnných používá CP i metody tzv. lokálního prohledávání. Lokální prohledávání svým způsobem navazuje na metodu generuj a testuj, výsledky testu jsou zde ovšem použity při generování dalšího úplného ohodnocení proměnných. Společným znakem algoritmů lokálního prohledávání je, že začínají z nějakého náhodně vygenerovaného úplného ohodnocení proměnných a toto ohodnocování lokálně zlepšují, tj. změnou hodnoty proměnné postupně odstraňují nekonzistence podmínek. Protože další generované ohodnocení proměnných se od současného ohodnocení liší jen lokálně (v hodnotě jedné proměnné), hovoříme o lokálním prohledávání.

Asi nejznámějším algoritmem lokálního prohledávání je *metoda největšího stoupání*, jinak také nazývaná horolezecká metoda (hill climbing) [26]. Tato metoda zkouší postupně změnit hodnotu každé proměnné a ze všech vygenerovaných ohodnocení (každé se liší od aktuálního ohodnocení v právě jedné proměnné) vybere to, které splňuje největší počet podmínek. Pokud je takových ohodnocení více, vybere mezi nimi náhodně.

Protože počet ohodnocení, které musí metoda největšího stoupání v každém kroku prozkoumat, je poměrně velký (přesně $(D_1 - 1) + \cdots + (D_N - 1)$, kde D_i je velikost domény i-té proměnné a N je počet proměnných), byla navržena úspornější *metoda minimalizace konfliktů* (min-conflicts) [22]. Tato metoda nejprve náhodně zvolí libovolnou konfliktní proměnnou, tj. proměnnou, která je součástí nějaké nesplněné podmínky, a potom vybere její novou hodnotu minimalizující počet nesplněných podmínek (odtud minimalizace konfliktů, čtenář jistě sám nahlédne, že minimalizace počtu nesplněných podmínek je ekvivalentní maximalizaci počtu splněných podmínek). Minimalizace konfliktů tedy v každém kroku prozkoumává pouze $D_i - 1$ ohodnocení, kde i je index vybrané konfliktní proměnné.

Obě právě popsané metody se tedy snaží lokálně zlepšovat úplné ohodnocení, dokud nenaleznou ohodnocení splňující všechny podmínky. Při tomto lokálním prohledávání se ovšem mohou dostat do situace, kdy každé další ohodnocení není lepší, resp. je horší než současné ohodnocení, které ale není řešením (nesplňuje všechny podmínky). Hovoříme potom o lokálním optimu, resp. o striktním lokálním optimu. Jednoduchým způsobem, jak se z lokálního optima dostat, je algoritmus restartovat, tj. vygenerovat nové náhodné ohodnocení proměnných a začít prohledávání s ním. Existují ovšem sofistikovanější metody, které se více drží principu lokálních kroků. Mezi nimi je asi



Obr. 8. Lokální prohledávání prochází prostor všech ohodnocení proměnných a hledá v něm globální optimum, tj. ohodnocení splňující nejvíce podmínek (resp. všechny podmínky).

nejpopulárnější *metoda náhodné procházky* (random walk) [28], doporučující s jistou pravděpodobností p použít krok navržený hlavním algoritmem (největší stoupání nebo minimalizace konfliktů) a se zbytkovou pravděpodobností $1 - p$ vybrat další ohodnocení náhodně změnou hodnoty jedné proměnné (i přesto, že toto ohodnocení může být horší než současné ohodnocení). Pravděpodobnost p je parametrem algoritmu a její nastavení umožňuje čas od času, ale ne zase příliš často, sejít z doporučené trasy prohledávání a vydat se jinou, snad lepší cestou.

Jiný způsob „vyskočení“ z lokálního optima a zabránění cyklení přes stejně dobrá ohodnocení nabízí technika *tabu seznamu* (tabu search) [13]. Tabu seznam je speciální krátkodobá paměť, ve které je uloženo několik posledních zkoumaných ohodnocení, resp. jejich charakteristický atribut (zpravidla se pamatuje pouze měněná proměnná a její původní hodnota). Velikost této paměti, tj. délka tabu seznamu, je opět dána jako parametr algoritmu. Jak již napovídá název, tabu seznam popisuje ohodnocení, která jsou v následujícím kroku zakázána. Algoritmus tak musí vybírat další ohodnocení mimo tabu seznam, což mu umožňuje dostat se z lokálního optima (může se vzít i ohodnocení horší než současné) a nedostat se při tom do cyklu opakujících se ohodnocení.

Díky svému stochastickému charakteru nemohou metody lokálního prohledávání zaručit úplnost, tj. nalezení řešení, pokud existuje, případně provedení důkazu, že řešení neexistuje. Aplikace těchto metod a nastavení jejich parametrů tak podobně jako u genetických algoritmů (i ty lze aplikovat na řešení CSP) připomíná alchymii, a proto je někteří rigorózně uvažující vědci nemají příliš v oblibě. Pravdou ovšem je, že pokud se takovou metodu podaří aplikovat v praxi, zvláště na řešení velkých problémů, výsledky bývají často velice dobré, ať už pokud jde o kvalitu řešení (u optimalizačních problémů), nebo o čas výpočtu.

4. Oblasti aplikací

Po představení základních řešících technik je určitě zajímavé ukázat si některé konkrétní oblasti, kde lze CP prakticky uplatnit. Vzhledem k obecnosti popisu problému pomocí omezujících podmínek je záběr CP, pokud jde o reálné aplikace, skutečně

široký a sahá od analýzy struktury DNA přes rozvrhování a konfiguraci až třeba k animaci lidské tváře nebo ke geografickým informačním systémům.

Asi nejčastěji nachází CP uplatnění v různých přiřazovacích a rozvrhovacích problémech, jejichž kombinatorický charakter je pro CP zvláště vhodný. S aplikacemi založenými na omezujících podmínkách se můžeme setkat při alokaci stojánek pro letadla [7], přiřazení odbavovacích pultů na letištích [6] nebo při alokaci kotvišť ve velkých překladištích [27]. CP najdeme i v pozadí přípravy rozvrhů směn v nemocnicích [5] nebo přiřazení posádek k vlakovým spojům [8]. Technik CP se používá při rozvrhování výroby vojenských i obchodních letadel [3], při plánování vrtů na ropných plošinách [18] nebo při optimalizaci těžby dřeva [1]. Úspěšnou aplikací je také optimální rozmísťování základových stanic u bezdrátových telekomunikačních sítí [10]. Z nových oblastí, kde se začíná CP objevovat, lze jmenovat například návrh (authoring) multimediálních děl nebo analýzu běhu programů.

5. Vývojová prostředí pro CP

Řešící technologie popsané v předchozích odstavcích jsou samozřejmě k dispozici v řadě vývojových prostředí, takže uživatel se může plně soustředit na modelování vlastního problému pomocí podmínek. Dnes již snad všechny implementace jazyka Prolog obsahují v nějaké formě knihovny pro řešení podmínek. Jmenovat můžeme SICStus Prolog od SICS, rozšířený zvláště (ale ne pouze) v akademickém prostředí, IF Prolog od IF Computer, zaměřený spíše na komerční aplikace, systém ECLiPSe od IC-Parc, implementující nejnovější výsledky výzkumu v CP nebo CHIP firmy Cosytec, nabízející asi nejlepší implementaci tzv. globálních podmínek (tyto podmínky zastupují sadu jednoduchých podmínek, např. nerovností, a použitím speciálních algoritmů umožňují dosáhnout lepší propagace). Opomíjeni ovšem nejsou ani uživatelé ostatních programovacích jazyků, globální podmínky systému CHIP jsou například dostupné formou knihoven CHIP C a CHIP C++ příznivcům jazyků C a C++. Firma ILOG nabízí C++ knihovnu ILOG Solver poskytující vše potřebné pro práci s omezujícími podmínkami, k dispozici jsou i nadstavby pro tvorbu aplikací v konkrétní oblasti, například ILOG Scheduler pro řešení rozvrhovacích problémů. Zajímavé je, že hlavní dodavatelé balíků pro řešení omezujících podmínek jsou soustředěni v Evropě, především pak ve Francii. Japonské a americké společnosti, např. i2 Technologies, dávají přednost tvorbě aplikací pro koncové zákazníky před poskytováním vývojových prostředí.

6. Trendy

Programování s omezujícími podmínkami se ukazuje jako velice slibný nástroj pro řešení složitých problémů. Zvláště je oceňován jeho deklarativní přístup k zadání problému. Přirozená formulace problému použitím omezujících podmínek totiž umožňuje rychlou tvorbu prototypů a snadné úpravy při změnách zadání. Uživatel tak

dostává řešení svého problému mnohem rychleji (řeč je o času vývoje software), což je v dnešním dynamickém světě velkou devizou. Deklarativní programování tak, jak je nabízí CP, navíc umožňuje běžným uživatelům přímo se podílet na tvorbě aplikace, a to i bez znalostí vestavěných řešících technik.

Jedním z témat současného výzkumu je právě pro rostoucí zapojení běžných uživatelů do formulace problému oblast modelování [33]. Deklarativní formulace problému pomocí omezujících podmínek totiž zásadním způsobem ovlivňuje, jak bude problém řešen, a drobná změna modelu (použitých proměnných a podmínek) se může výrazně projevit v efektivitě jeho řešení. Důležité jsou proto nástroje umožňující ladění modelů a poskytující vizuální pohled na prohledávání [29], které tvoří základ mnoha řešících algoritmů. Porozumění a kontrola prohledávání je zatím poměrně málo vyvinutou částí CP, a proto se tomuto tématu věnuje náležitá pozornost.

Na poli řešících technologií se jako žhavé téma výzkumu ukazuje integrace různých technologií formou hybridních algoritmů [15] a spolupráce řešících algoritmů [23]. Programování s omezujícími podmínkami se stále více sbližuje s metodami operačního výzkumu a techniky vyvinuté v kombinatorické optimalizaci jsou integrovány do CP například formou globálních omezení.

Shrnuto a podtrženo, svatého grálu programování, kdy uživatel pouze zadá, co po stroji požaduje, zatím dosaženo nebylo. Programování s omezujícími podmínkami se ale tomuto vysněnému cíli hodně přibližuje.

L i t e r a t u r a

[1] ADHIKARY, J., HASLE, G., MISUND, G.: *Constraint Technology Applied to Forest Treatment Scheduling.* In: Proc. of Practical Application of Constraint Technology (PACT97), London, UK, 1997.

[2] BARTÁK, R.: *On-line Guide to Constraint Programming.* Prague, 1998, `http://kti.mff.cuni.cz/~bartak/constraints/`

[3] BELLONE, J., CHAMARD, A., PRADELESS, C.: *PLANE: An Evolutive Planning System for Aircraft Production.* In: Proc. of Practical Application of Prolog (PAP92), London, UK, 1992.

[4] BORNING, A.: *The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory.* In: ACM Transactions on Programming Languages and Systems 3 (4): 252–387, 1981.

[5] CHAN, P., HEUS, K., WEIL, G.: *Nurse Scheduling with Global Constraints in CHIP: GYMNASTE.* In: Proc. of Practical Application of Constraint Technology (PACT98), London, UK, 1998.

[6] CHOW, K. P., PERETT, M.: *Airport Counter Allocation using Constraint Logic Programming.* In: Proc. of Practical Application of Constraint Technology (PACT97), London, UK, 1997.

[7] DINCBAS, M., SIMONIS, H.: *APACHE — A Constraint Based, Automated Stand Allocation Systems.* In: Proc. of Advanced Software Technology in Air Transport (ASTAIR91), London, UK, 1991.

[8] FOCACCI, F., LAMMA, E., MELLO, P., MILANO, M.: *Constraint Logic Programming for the Crew Rostering Problem.* In: Proc. of Practical Application of Constraint Technology (PACT97), London, UK, 1997.

[9] FREUDER, E. C.: *Synthesizing Constraint Expressions.* In: Communications ACM 21 (11): 958–966, ACM, 1978.

[10] Frühwirth, T., Brisset, P.: *Optimal Placement of Base Stations in Wireless Indoor Telecommunication.* In: Proc. of Principles and Practice of Constraint Programming (CP98), Pisa, Italy, 1998.

[11] Gaschnig, J.: *Performance Measurement and Analysis of Certain Search Algorithms.* CMU-CS-79-124, Carnegie-Mellon University, 1979.

[12] Gallaire, H.: *Logic Programming: Further Developments.* In: IEEE Symposium on Logic Programming, Boston, IEEE, 1985.

[13] Glover, F., Laguna, M.: *Tabu Search.* In: Modern Heuristics for Combinatorial Problems, Blackwell Scientific Publishing, Oxford, 1993.

[14] Haralick, R. M., Elliot, G. L.: *Increasing tree search efficiency for constraint satisfaction problems.* In: Artificial Intelligence 14: 263–314, 1980.

[15] Jacquet-Lagreze, E.: *Hybrid Methods for Large Scale Optimisation Problems: an OR Perspective.* In: Proc. of Practical Application of Constraint Technology (PACT98), London, UK, 1998.

[16] Jaffar, J., Lassez, J. L.: *Constraint Logic Programming.* In: Proc. The ACM Symposium on Principles of Programming Languages, ACM, 1987.

[17] Jaffar, J., Maher, M. J.: *Constraint Logic Programming — A Survey.* J. Logic Programming, 19/20: 503–581, 1996.

[18] Johansen, B. S., Hasle, G.: *Well Activity Scheduling — An Application of Constraint Reasoning.* In: Proc. of Practial Application of Constraint Technology (PACT97), London, UK, 1997.

[19] Kumar, V.: *Algorithms for Constraint Satisfaction Problems: A Survey.* AI Magazine 13 (1): 32–44, 1992.

[20] Mackworth, A. K.: *Consistency in Networks of Relations.* In: Artificial Intelligence 8 (1): 99–118, 1977.

[21] Marriot, K., Stuckey. P.: *Programming with Constraints: An Introduction.* The MIT Press, Cambridge, Mass., 1998.

[22] Minton, S., Johnston, M. D., Laird, P.: *Minimising conflicts: a heuristic repair method for constraint satisfaction and scheduling problems.* In: Artificial Intelligence 58 (1–3): 161–206, 1992.

[23] Monfroy, E.: *Solver Collaboration for Constraint Logic Programming.* PhD Thesis, l'Universite Henri Poincare-Nancy I, 1996.

[24] Montanary, U.: *Networks of constraints: Fundamental properties and applications to picture processing.* In: Information Science 7: 95–132, 1974.

[25] Nadel, B.: *Tree Search and Arc Consistency in Constraint Satisfaction Algorithms.* In: Search in Artificial Intelligence, Springer-Verlag, New York, 1988.

[26] Nilsson, N. J.: *Principles of Artificial Intelligence.* Tioga, Palo Alto, 1980.

[27] Perett, M.: *Using Constraint Logic Programming Techniques in Container Port Planning.* In: ICL Technical Journal, May: 537–545, 1991.

[28] Selman, B., Kautz, H.: *Domain-independent extensions to GSAT: Solving Large Structured Satisfiability Problems.* In: Proc. IJCAI-93, 1993.

[29] Simonis, H., Aggoun, A.: *Search Tree Debugging.* Technical Report, COSYTEC SA, 1997.

[30] Sutherland I.: *Sketchpad: a man-machine graphical communication system.* In: Proc. IFIP Spring Joint Computer Conference, 1963.

[31] Tsang, E.: *Foundations of Constraint Satisfaction.* Academic Press, London, 1995.

[32] van Hentenryck, P.: *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, Mass., 1989.

[33] van Hentenryck, P., Michel, L., Deville, Y.: *Numerica: A Modeling Language for Global Optimization.* The MIT Press, Cambridge, Mass., 1997.

[34] Waltz, D. L.: *Understanding line drawings of scenes with shadows.* In: *Psychology of Computer Vision*, McGraw-Hill, New York, 1975.