VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ ALGORITMY V ÚLOZE BOOLEOVSKÉ SPLNITELNOSTI

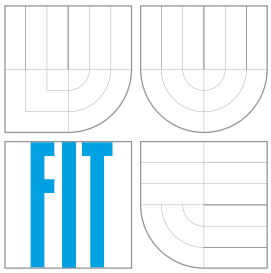
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. SILVESTER SERÉDI

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ ALGORITMY V ÚLOZE BOOLEOVSKÉ SPLNITELNOSTI

EVOLUTIONARY ALGORITHMS IN THE TASK OF BOOLEAN SATISFIABILITY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. SILVESTER SERÉDI
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. LUKÁŠ SEKANINA, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Cílem této diplomové práce je najít heuristiku řešící SAT problém pomocí evolučního algoritmu. Jsou zde uvedeny přístupy k řešení SAT problému a různé varianty k evolučním algoritmům, které jsou relevantní k danému tématu. Následně je popsaná implementace lineárního genetického programování hledající heuristiku pro řešení instancí SAT problému společne s vlastní implementací SAT solveru pracující s výstupem evolučně navrženého programu. Na závěr jsou shrnuty dosažené výsledky

Abstract

The goal of this Master's Thesis is finding a SAT solving heuristic by the application of an evolutionary algorithm. This thesis surveys various approaches used in SAT solving and some variants of evolutionary algorithms that are relevant to this topic. Afterwards the implementation of a linear genetic programming system that searches for a suitable heuristic for SAT problem instances is described, together with the implementation of a custom SAT solver which expoloits the output of the genetic program. Finally, the achieved results are summarized.

Klíčová slova

Genetické programovanie, evolučné algoritmy, zložitosť, SAT problém, NP-úplný problém, SAT solver.

Keywords

Genetic programming, evolutionary algorithms, complexity, SAT problem, NP-complete problem, SAT solver.

Citace

Silvester Serédi: Evoluční algoritmy v úloze booleovské splnitelnosti, diplomová práce, Brno, FIT VUT v Brně, 2013

Evoluční algoritmy v úloze booleovské splnitelnosti

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Lukáše Sekaniny, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Silvester Serédi
24. května 2013

Poděkování

Chtěl bych poděkovat svému vedoucímu diplomové práce, profesorovi Sekaninovi, za jeho pomoc a čas, který mi ochotně věnoval při řešení této práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V teoretickej informatike je pre problémy formálne definovaný pojem trieda zložitosti. Typy zložitostí sa rozdeľujú na priestorovú (koľko pamäte sa spotrebu na riešenie triedy problému) a časovú (koľko diskrétnych krokov treba vykonať na vyriešenie problému). Jedna z tried zložitostí problémov, ktoré majú vysokú časovú náročnosť, ale v bežnom svete vyskytujú pomerne často, sú problémy, ktorých trieda sa nazýva NP-úplná.

Bežné problémy, ktoré vieme veľmi účinne riešiť, radíme do triedy zložitosti P (prípadne tried s ešte nižšími zložitosťami ako napríklad NL). Sú to problémy, ktoré vieme riešiť v čase polynomiálne závislom na veľkosti vstupu. Sú to bežné aritmetické problémy, vyhľadávacie a radiace algoritmy a podobne. Trieda zložitosti, ktorej je P podmnožinou, sa nazýva NP. Otázka či je podmnožina striktná je dlhodobo otvorený problém. NP obsahuje okrem triedy P aj problémy NP-úplné, ktoré sú najťažšie problémy z NP.

Neformálne, u NP-úplného problému platí, že ak by sme dokázali predvídať správny smer riešenia problému, potrebovali by sme k vyriešeniu problému čas polynomiálne závislý od veľkosti vstupu. Keďže však túto schopnosť nemáme, zatiaľ vieme takéto problémy riešiť v najhoršom prípade len v čase exponenciálne závislom od veľkosti vstupu. Táto horná hranica sa však neustále zlepšuje.

Prvý problém, ktorého trieda zložitosť bola určená ako NP-úplná, sa nazýva SAT problém [6]. Úlohou je rozhodnúť, či je možné nájsť také hodnoty premenných pre logický výraz, zapísaný obvykle ako konjukncia diskjunkcií literálov, aby bol pravdivý. Nepýta sa na riešenie, ale len na skutočnosť, či riešenie existuje. Tomuto sa hovorí, že určuje splniteľnosť problému.

Na základe určenia NP-úplnosti SAT problému bola dokázaná NP-úplnosť pre ďalších 21 problémov [19], napríklad problém hľadania kliky v grafe, existencia hamiltonovskej cesty v grafe, knapsack problem, atď. Nasledovne bolo nájdené ohromné množstvo ďalších NP-úplných problémov [14].

Vidíme, že NP-úplný problém sa vyskytuje pomerne často v praxi. U NP-úplných problémov platí, že sa dajú medzi sebou prevádzať v polynomiálnom čase bez toho, aby to malo zásadný vplyv na zložitosť. To znamená, že nájdenie algoritmu, ktorý by riešil ľubovoľný NP-úplný problém v polynomiálnom čase, by znamenalo riešiteľnosť každého NP-úplného problému v polynomiálnom čase. O tom, že riešenie tohto problému je veľmi dôležité svedčí aj odmena jedného milióna dolárov za potvrdenie alebo vyvrátenie rovnosti tried P a NP [5].

Napriek exponenciálnej asymptotickej časovej zložitosti sa riešenie dá nájsť vo veľkom množstve prípadov v rozumnom čase. Mnohé problémy z rôznych domén sa prevádzajú na SAT problém. Vďaka tomu má veľa inštancií spoločné charakteristiky, ktoré sa môžu využiť

pri výpočtoch a tým značne ušetriť čas.

V súčasnej dobe je riešenie SAT problému pomocou evolučných algoritmov pomerne nepreskúmaná, ale sľubná oblasť ([11][13]). Cieľom tejto práce bude ukázať niektoré prístupy k riešeniu SAT problému a uviesť základné typy jedného prístupu z evolučných algoritmov - genetické programovanie. Na základe týchto znalostí sa potom vytvorí algoritmus využívajúci genetické programovanie, ktorý vytvorí heuristiku pre riešenie SAT problému na základe trénovacej množiny. Táto heuristika je testovaná v SAT solveri, ktorý som implimentoval. Vlastný SAT solver bolo nutné použiť kvôli zložitosti začleneniu heuristiky do existujúcich SAT solverov. Dôvodom je optimalizácia stávajúcich solverov na určitý prístup k riešeniu, čím by sa iné varianty prístupov značne znevýhodnili.

V druhej kapitole sú definované pojmy, ktoré sa budú často využívať v tejto práci.

Tretia kapitola ukáže SAT problém a jeho rôzne varianty spolu s ich najlepšie doteraz zistenými zložitosťami.

V princípe existujú dva prístupy k riešeniu SAT problému: úplné a neúplné algoritmy. Úplné algoritmy poskytujú záruku, že výsledok výpočtu bude správny, teda výstupom bude tvrdenie "splniteľný", alebo "nesplniteľný". Neúplné algoritmy túto záruku neposkytujú. Obvykle pracujú s obmedzeným množstvom zdrojov. Prikláňajú sa ku splniteľnosti, tj. formulu vyhlásia za splniteľnú, alebo ohlásia neúspech. Nevyhlasujú nesplniteľnosť. Preto sú v kapitole 4 ukázané rôzne prístupy k úplným algoritmom. Kapitole 5 sa zaoberá neúplnými algoritmami.

Ďalšiu metódu zvyšujúcu účinnosť riešenia oboch prístupov – reštarty a náhodnosť – si ukážeme v kapitole 6.

V siedmej kapitole bude stručný úvod do genetického programovania, spolu s niektorými jeho variantami.

Ôsma kapitola popisuje implementáciu a prácu genetického programovania spolu s popisom implementácie SAT solveru, ktorý pracuje s výstupom evolučne navrhnutého programu.

Nasleduje deviata kapitola, kde sú zhrnuté výsledky experimentov.

Desiata kapitola obsahuje zhrnutie výsledkov dosiahnutých v diplomovej práci.

Kapitola 2

Použité názvoslovie

Výroková logika skúma spôsoby tvorby zložených výrokov z daných jednoduchých prvkov, závislosť pravdivosti (resp. nepravdivosti) zloženého výroku na pravdivosti výrokov, z ktorých je zložený.

Prvotné formuly, ktoré majú úlohu jednoduchých výrokov, značíme písmenami p, q, prípadne p_1, p_2. Zložené výroky vytvárame pomocou logických spojok: neg - negácia, $\wedge$ - konjunkcia, $\vee$ - disjunkcia, $\rightarrow$ - implikácia, $\leftrightarrow$ - ekvivalencia. Každá formula môže mať pravdivú alebo nepravdivú hodnotu.

Prvotné formuly (**premenné**) môžu nadobúdať dve hodnoty: pravdivá (v tejto práci označované aj ako kladná, alebo true) alebo nepravdivá (v tejto práci označované aj ako záporná, alebo false). Ak pracujeme s pravdivou hodnotou literálu premennej x, značíme x. Ak pracujeme s negovanou hodnotou premennej x, značíme to $\neg x$. Každý jeden výskyt premennej s hodnotou sa nazýva **literál**. Spojením literálov spojkou disjunkcie $\vee$ vzniká **klauzula** (**pravidlo**). Spojením klauzúl spojkou konjunkcie $\wedge$ vzniká **formula**. Takýto zápis booleovského výrazu sa nazýva konjunktná normálna forma (**CNF**). Pre zjednodušenie zápisu môže nahradiť zápis $(A \vee B) \wedge (\neg C \vee \neg D)$ zápisom $\{\{A, B\}, \{\neg C, \neg D\}\}$.

Hlavnou témou tejto práce bude **SAT problém**. Tento problém rieši otázku "existuje aspoň jedno priradenie hodnôt také, aby zadaná formula bola pravdivá?". Je to rozhodovací problém, takže výstupom riešenia môže byť len odpoveď "áno"alebo "nie", resp "true"alebo "false". Zvláštnu podmnožinou SAT problému predstavujú **k-SAT problémy**. Sú to inštancie, v ktorých všetky klauzuly obsahujú k literálov.

Pri riešení SAT problému sa môžme stretnúť s niektorými zvláštnymi prípadmi klauzúl. Klauzula, ktorá má n literálov sa nazýva **n-árna klauzula** .

Ak pri riešení inštancie SAT problému narazíme na klauzulu, ktorá nemá literály (**Nulová klauzula**), znamená to, že pri aktuálnych podmienkach nie je táto inštancia splniteľná. Ďalšia triviálna klauzula je taká, ktorá obsahuje len jeden literál (**Jednotková klauzula**). Takáto klauzula je užitočná pri zjednodušovaní zadania. Keďže musí byť splniteľná každá klauzula a jednotková klauzula obsahuje jeden literál, tento literál určuje hodnotu premennej. Vďaka tomuto je možné splniť niekoľko klauzúl a tým zjednodušiť zadanú inštanciu.

Formálne povedané: Majme výrok s jednotkovou klauzulou A a klauzulou $\neg A \ \vee X$, kde X je ľubovoľná množina literálov. Túto formulu vieme previesť na tvar $A \wedge (A \ \Rightarrow X)$. Vidíme, že ak platí A, musí platiť X, zároveň však platí A. Z týchto dvoch skutočností vyplýva, že musí platiť X, lebo hodnota A je pevne stanovená. Vďaka tomu sa môže A odstrániť z formuly.

Jednotková propagácia označuje postup, v ktorom sa aplikujú jednotkové klauzuly

na formulu, kým existujú klauzule, ktoré obsahujú opačné literály ako sa nachádzajú v jednotkových klauzulách.

Niektoré prístupy používajú zovšeobecnenie jednotkovej propagácie, nazývané **rezolúcia**. Zjednocuje dve klauzuly, ktoré majú spoločnú premennú, ale s opačnými hodnotami. V tomto zjednotení sa spoločná premenná už nevyskytuje. Výsledok rezolúcie je **rezolvent**.

Existuje niekoľko typov SAT problémov, každá má istú zložitosť. Tieto zložitosti majú formálne pomenovania. V tejto práci sa vyskytujú nasledovné:
NL - trieda problémov, ktorá sa dá riešiť nedeterministickým Turingovým strojom v priestore logaritmicky závislom od veľkosti vstupu.
P - trieda problémov, ktorá sa dá riešiť deterministickým Turingovým strojom v čase polynomiálne závislom od veľkosti vstupu.
NP - trieda problémov, ktorá sa dá riešiť nedeterministickým Turingovým strojom v čase polynomiálne závislom od veľkosti vstupu.
PSPACE - trieda problémov, ktorá sa dá riešiť deterministickým Turingovým strojom v priestore polynomiálne závislom od veľkosti vstupu.
NPSPACE - trieda problémov, ktorá sa dá riešiť nedeterministickým Turingovým strojom v priestore polynomiálne závislom od veľkosti vstupu. Je dokázané, že PSPACE = NPSPACE[1].
T-úplný problém - Majme triedu problémov T. Rozhodovací problém C je T-úplný práve vtedy, keď platí [28]:

1. C je v T
2. Každý problém v T sa dá redukovať na C v polynomiálnom čase

Kapitola 3

Zložitosť SAT problému a jeho variantov

Táto kapitola čerpá z [12]. Naivný prístup k riešeniu SAT problému by bol nasledovný: vyskúšali by sme postupne všetky priradenia hodnôt premenných a po každom priradení by sme skontrolovali pravdivosť formuly. Akonáhle by bolo priradenie pravdivé, vyhlásili by sme formulu za splniteľnú. V opačnom prípade za nesplniteľnú. Na určenie nesplniteľnosti by sme museli v najhoršom prípade vyskúšať všetkých 2^n priradení hodnôt, kde n je počet premenných. Časová zložitosť je teda $O(2^n)$.

Prvá objavená netriviálna horná zložitosť pre k-SAT, $k \geq 3$ bola $O(\alpha_k^n, |\phi|)$, kde $|\phi|$ je počet literálov a α_k je ohraničenie funkcie T podobnej Fibonacciho postupnosti

$$T(1) = T(2) = \cdots = T(k-1) = 1 \tag{3.1}$$

$$T(n) = T(n-1) + T(n-2) + \cdots + T(n-k+1) \text{ pre } n \geq k \tag{3.2}$$

Hodnota α_k je $1,681$ pre $k = 3$, $1,8393$ pre $k = 4$, $1,976$ pre $k = 5$. Pre $k = 3$ bola táto hodnota vylepšená na $1,497$. V dnešnej dobe majú najlepšie deterministické algoritmy zložitosť $O((2-\frac{2}{k+1})^n)$. Z čoho vyplýva $O(1,5^n)$, $O(1,6^n)$, $O(1,666^n)$ pre $k = 3$, resp. $k = 4$, $k = 5$. Pre $k = 3$ sa táto hodnota vylepšila na $O(1,473^n)$, neskôr až na $O(1,439^n)$[21].

Pravdepodobnostný prístup mal za výsledok lepšie hodnoty a ich derandomizáciou boli vylepšené stávajúce deterministické algoritmy. Prvý z nich vyvinuli Paturi-Pudlak-Zane [24]. Základom algoritmu je postup, pri ktorom sa každej premennej priradí náhodná hodnota a skontroluje sa pravdivosť. Ak je formula pri danom priradení splnená, končí. Inak sa náhodnej premennej zmení hodnota. Toto sa opakuje r krát. Za ten čas sa buď nájde platné priradenie, alebo sa formula vyhlási za nesplniteľnú. Po $r = 2^{n(1-\frac{1}{k})}$ iteráciách nájde algoritmus spĺňajúce priradenie s vysokou pravdepodobnosťou. Zložitosti sú nasledovné: $O(1,36406^n)$ pre $k = 3$, $O(1,49579^n)$ pre $k = 4$, $O(1,56943^n)$ pre $k = 5$. Pre k väčšie ako 4 je toto momentálne najlepší pravdepodobnostný postup.

Schöning [25] rozvinul Papadimitriouvu ideu, z čoho vznikol druhý algoritmus. Postup je nasledovný: Opakuj nasledujúce r krát: náhodne priraď hodnoty premenným. Ak je formula nepravdivá, opakuj nasledovné v-krát, kde v je počet premenných vo formule: Vyber náhodne neplatnú klauzulu a v nej náhodne vyber literál, ktorým sa klauzula stane pravdivou. Hodnota r určuje pravdepodobnosť, že sa formula vyhodnotí ako nesplniteľná. Pre $k = 3, 4$ a 5 sú základy exponentu $1,3334$, $1,5$ a $1,6$.

Spojením týchto dvoch algoritmov sa hodnota pre $k = 3$ vylepšila na $O(1,32065^n)$ [17] a pre $k = 4$ na $O(1,46928^n)$. Pre formuly bez obmedzenia dĺžky klauzúl je najlepšia dosiahnutá hodnota pre deterministické algoritmy $2^{n(1-\frac{1}{log(2m))})}$.

Ako vidíme, napriek tomu, že sa zatiaľ nenašiel algoritmus s polynomiálnou časovou zložitosťou, postupy sa vylepšujú. Hoci je všeobecný SAT problém výpočtovými zložitý, existuje nemalá skupina inštancií, ktoré sa riešia značne jednoduchšie. Niektoré z nich si ukážeme v nasledujúcej podkapitole. Okrem toho si ukážeme aj problémy, ktoré sú nad NP.

3.1 2-SAT

Jednou z prvých skupín SAT problému, pre ktorú sa podarilo nájsť algoritmus s polynomiálnou časovou zložitosťou, je 2-SAT. Skladá sa z formuly v CNF, kde každá klauzula má nanajvýš dva literály. Pre tento problém môžeme skonštruovať graf implikácií. Potom pomocou jednoduchého vyhľadávania do hĺbky vieme nájsť pravdivé priradenie hodnôt premenným, prípadne naraziť na konflikt a vyhlásiť formulu za nesplniteľnú. K tomuto problému existuje algoritmus s lineárnou časovou zložitosťou. Problém je dokonca riešiteľný v nedeterministickom logaritmickom priestore (je NL-úplný).

3.2 Horn-SAT

Ak majú všetky klauzuly nanajvýš jeden literál kladný, nazývame túto formulu Hornovou formulou. Riešením je jednotková propagácia pre všetky kladné literály, kým nie sú všetky eliminované. Nasledovne sa zostávajúcim negatívnym literálom priradí hodnota "false". Tento problém je P-úplný [7] a riešiteľný v lineárnom čase.

3.3 Premenovateľný Horn-SAT

Ak je možné vo formule zmeniť hodnoty nejakej podmnožiny všetkých premenných tak, aby bola nová formula ekvivalentná s pôvodnou, nazývame formulu premenovateľnou Hornovou formulou. Rozpoznanie takejto formuly a jej riešenie má lineárnu časovú zložitosť.

3.4 Minimálne nesplniteľné formuly

Formula sa nazýva minimálne nesplniteľnou, ak je nesplniteľná a zároveň odstránenie akejkoľvek klauzuly spôsobí, že sa formula stane splniteľnou. Každá minimálne nesplniteľná formula s n premennými musí mať aspoň $n + 1$ klauzúl. Každá premenná sa musí vyskytovať pozitívne a negatívne vo formule. Časovú zložitosť určenia splniteľnosti takýchto formúl sa podarilo zlepšiť z $n^{O(k)}$ na $O(2^k)n^4$ [27].

3.5 k-BRLR

Formulu nazývame k-BRLR, ak všetky rezolventy z nej odvodené majú počet literálov ohraničených k, alebo nulová klauzula je odvoditeľná z rezolventov majúcich maximálne k premenných. Časovú zložitosť určenia splniteľnosti je $O(2^k\binom{n}{k}))$.

3.6 MAX-SAT

MAX-SAT sa pýta aké priradenie hodnôt dokáže splniť maximálne množstvo klauzúl. Tu vidíme rozdiel s bežným SAT problémom, ktorý chce aby boli splnené všetky klauzuly. Táto rodina problémov sa od bežného SAT problému líši tým, že sa nepýta otázku s odpoveďou "áno", alebo "nie". Namiesto toho sa pýta na hodnoty. Teda nie je to rozhodovací problém, ale funkčný problém. Jedná o zovšeobecnenie SAT problému, a bolo dokázané, že patrí do triedy NP-ťažkých problémov. MAX-2-SAT ešte pridáva podmienku, že všetky klauzuly majú nanajvýš 2 literály. Napriek tomuto obmedzeniu, zostáva MAX-2-SAT NP-ťažký. Časová zložitosť výpočtu pre MAX-SAT je ovplyvnená parametrami L (počet literálov vo vstupe), m (počet klauzúl vo vstupe) a n (počet premenných vo vstupe). Momentálne najlepšie hodnoty pre MAX-SAT sú $O(L2^{m/2,36})$ a $O(L2^{L/6,89})$ [2]. Pre MAX-2-SAT je to hodnota $O(m2^{m/5})$ [16].

3.7 Kvantifikované booleovské formuly

SAT problém, už podľa definície, obsahuje na vstupe pri premenných existenčný kvantifikátor. Kvantifikované booleovské formuly (QBF) pridávajú možnosť výskytu všeobecného kvantifikátora vo formule. Sémantika je nasledovná: Všeobecne kvantifikovaná formula ψ je pravdivá vtedy a len vtedy, ak je formula $\psi = \forall x\phi$ pravdivá pre kladnú aj zápornú hodnotu x. Pre existenčne kvantifikované formuly $\psi = \exists x\phi$ platí, že sú pravdivé vtedy a len vtedy, ak existuje aspoň jedna hodnota ($true$ alebo $false$) pre všetky x taká, že je formula pravdivá. Problém booleovskej splniteľnosti pre QBF sa nazýva QSAT. QSAT je PSPACE-úplný. Keďže platí PSPACE = NPSPACE, veľká časť problémov súvisiacich s kvantifikovanými problémami sa dá riešiť v PSPACE. Toto je v kontraste s NP, kde nie je dokázaná ani rovnosť medzi NP a coNP. Dokonca sa predpokladá ich nerovnosť. Podobne ako u SAT, má aj QSAT jednoduchšie varianty, sú nimi Q2-CNF (formula s univerzálnymi kvantifikátormi a maximálne dvoma literálmi v klauzulách) a QHORN (Hornova formula obsahujúca univerzálne kvantifikátory). Pre obidva existujú rýchle algoritmy: Q2-CNF sa dá vyriešiť v lineárnom čase. QHORN má riešenie v $O(n \cdot r)$, kde r je počet premenných s univerzálnym kvantifikátorom. Ďalšia významná skupina je QEHORN. Táto obsahuje formuly, ktoré po odstránení literálov so všeobecnými kvantifikátormi zostanú Hornovými formulami. Táto skupina zostáva PSPACE-úplná.

Kapitola 4

Úplné algoritmy

Táto kapitola čerpá z [8]. V princípe existujú štyri prístupy k úplným algoritmom. Prvý spočíva v postupnej eliminácii premenných, bez toho, aby sa zmenila pravdivostná hodnota formuly. Druhý prístup sa snaží odvodzovať nové pravidlá kým sa nájde rozpor (nesplniteľná formula), alebo je CNF uzatvorená vzhľadom k novým odvodeniam (splniteľná formula). Ďalší prístup prehľadáva splniteľné priradenia. Posledný prístup kombinuje predošlé prístupy. Tento je najviac využívaný v moderných SAT solveroch. Teraz si ukážeme konkrétne algoritmy.

4.1 Riešiteľnosť existenciálnou kvantifikáciou

Podstata existenciálnej kvantifikácie je nasledovná: Výsledok existenciálnej kvantifikácie premennej P vo formule Δ, označený $\exists P\Delta$ je definovaná $\exists P\Delta \stackrel{\text{def}}{=} (\Delta|P) \vee (\Delta|\neg P)$.

Ak máme $\Delta|P$, kde P je literál, vyjadruje táto operácia funkciu $\{\alpha - \{\neg P\}|\alpha \in \Delta, L \notin \alpha\}$. Inak povedané, odstránia sa všetky klauzuly, ktoré obsahujú literál P a z klauzúl sa odstráni literál $\neg P$.

Napríklad majme CNF $\Delta = \{\{\neg A, B\}, \{\neg B, C\}\}$, ktorá v princípe hovorí, že A implikuje B, B implikuje C. Potom máme $\Delta|B = \{\{C\}\}, \Delta|\neg B = \{\{\neg A\ \}\}$, teda $\exists B\Delta = \{\{\neg A, C\}\}$ (tj. A implikuje C). Existenciálna kvantifikácia spĺňa veľa vlastností, ale najdôležitejšia je skutočnosť, že Δ je splniteľná, len ak $\exists P\Delta$ je splniteľná. Teda stačí nahradiť tesť na splniteľnosť formuly Δ testom na splniteľnosť formuly $\exists P\Delta$, ktorá má o jednu premennú menej. To znamená, že môžeme postupne aplikovať existenciálnu kvantifikáciu na všetky premenné, kým nezostane triviálna CNF neobsahujúca ani jednu premennú. Táto triviálna CNF bude teda buď $\{\emptyset\}$, ktorá je nesplniteľná, alebo $\{\}$, ktorá je splniteľná. Tento prístup sa môže implementovať rôznymi spôsobmi. Ukážeme si dva z nich.

4.1.1 DP algoritmus

DP algoritmus[1] [9] využíva vyššie spomínanú existenciálnu kvantifikovatelnosť. Tento prístup postupne existenciálne kvantifikuje všetky premenné po jednom. Jeden z variantov prístupu (po anglicky nazývaná bucket elimination), prebieha v dvoch fázach: Prvá fáza je konštrukcia a naplnenie sady kontajnerov. Druhá fáza spočíva ich následnom spracovaní v nejakom poradí.

Pri danom poradí π konštruujeme a napĺňame kontajnery nasledovne:

[1] Algoritmus je pomenovaný podľa mien svojich tvorcov, M. Davis and H. Putnam

1. Kontajner pre každú premennú P je označený P
2. Kontajnery sú zoradené podľa π
3. Každá klauzula α v CNF je pridaná k prvému kontajneru P, kde sa P nachádza v α.

Napríklad majme CNF
$\Delta = \{\{\neg C, B\}, \{\neg A, C\}, \{\neg B, D\}, \{\neg C, \neg D\}, \{A, \neg C, E\}\}$
a poradie C, B, A, D, E. Po vytvorení kontajnerov máme
$C : \{\neg A, C\}, \{\neg C, \neg D\}, \{A, \neg C, E\}$
$B : \{\neg A, B\}, \{\neg B, D\}$
$A :$
$D :$
$E :$

Spracovanie potom prebieha tak, že sa kontajnery prechádzajú zhora nadol. V každom kontajneri P vytvoríme všetky P-rezolventy a každý vložíme do prvého kontajneru, ktorého premenná sa v danom P-rezolvente nachádza. P-rezolventom nazývame klauzulu, ktorá vznikne zjednotením dvoch klauzúl, ktoré majú spoločnú premennú P, ale s opačnými hodnotami, s tým, že P sa v zjednotení nenachádza.

V našom prípade by sa po spracovaní kontajneru C vytvoril C-rezolvent $\{\neg A, \neg D\}$, ktorý by sa vložil do kontajneru A, potom po spracovaní kontajneru B by sme dostali B-rezolvent $\{\neg A, D\}$, ktorý by sa tiež vložil do kontajneru A. Spracovanie kontajneru A už nevytvorí žiadny A-rezolvent a kontajnery D a E sú prázdne. Máme teda $\exists C, B, A\ \Delta = \{\}$, teda pôvodná CNF je konzistentná. Rôzne poradie premenných má za následok rôznu časovú zložitosť behu algoritmu. Keby sme boli pri predošlej formule použili poradie E, A, B, C, D, nedal by sa vytvoriť žiadny rezolvent. Pri vhodnom poradí premenných je časová zložitosť $O(n \cdot e^w)$, kde n je veľkosť CNF Δ a w je šírka stromu grafu spojitosti: neorientovaný graf, v ktorom sú vrcholmi premenné a medzi dvoma vrcholmi existuje hrana, ak sa premenné, ktoré reprezentujú, nachádzajú v jednej klauzule v Δ. Je dokázané, že na isté triedy sa tento prístup dá aplikovať v polynomiálnom čase [10].

4.1.2 Symbolické SAT solvery

Hlavný problém pozorovaný u DP algoritmu je jeho priestorová náročnosť, keďže môže generovať priveľa nových klauzúl, čo má nepriaznivý vplyv na rýchlosť algoritmu. Táto priestorová zložitosť sa dá zmierniť použitím kompaktnejšej formy zápisu výsledných klauzúl. Jedna z nich sa nazýva binárne rozhodovacie diagramy (Binary Decision Diagrams, BDD), ktorej používanie vedie k triede symbolických SAT algoritmov. Ukážeme si tu základný princíp symbolických SAT algoritmov. Formálne je BDD orientovaný acyklický graf s koreňovým vrcholom a dvoma koncovými stavmi označenými 0 a 1. Každý iný vrchol je označený premennou a má práve 2 synovské vrcholy, označené ako "high"a "low". BDD reprezentuje formulu, ktorej modely sa dajú vymenovať tak, že sa vezme každá cesta z koreňa do uzlu 1. Low hrana z uzlu značí, že sa premennej, ktorú reprezentuje, pridá hodnota false. High hrana značí true. Ak sa na nejakej ceste nejaká premenná nenachádza, je označená ako "don't care", tj. môže mať priradenú ľubovoľnú hodnotu.

V praxi sa používajú Ordered BDD (OBDD), v ktorých sa premenné vyskytujú v rovnakom poradí na každej ceste. Okrem toho sa zvyknú pridať ďalšie pravidlá: Žiadny uzol nemá identické synovské uzly a v grafe sa nenachádzajú izomorfné podgrafy. Za týchto podmienok sú OBDD v kanonickej forme, tj. pri danom poradí premenných má každá formula

unikátny OBDD. Premenná X sa v OBDD dá existenciálne kvantifikovať tak, že sa vytvorí formula vzniknutá po nastavení premennej X, potom na $\neg X$. Výsledok sa potom dá rozdeliť, aby sa z neho získal výsledok kvantifikácie. Celý proces môže byť teda implementovaný kvadratickom čase vzhľadom k veľkosti OBDD. Na vyriešenie SAT sa môže CNF previesť na OBDD a existenciálne kvantifikovať všetky premenné. Naivný prístup je však nepraktický, lebo výsledný OBDD bude veľmi veľký. Aby sa dosiahlo zmenšenia priestorovej zložitosti, využíva sa prístup, kde pre existenciálnu kvantifikáciu premennej P sa spoja všetky OBDD obsahujúce danú premennú, a aplikuje sa existenciálna kvantifikácia. Výsledkom je jeden OBDD. Toto sa nazýva skorá kvantifikácia. Výsledné OBDD potom nahradí všetky OBDD, z ktorých sa vznikol a proces pokračuje. Ak sa vytvorí prázdny OBDD, algoritmus vyhlási CNF za nesplniteľnú. Ak sa podarí kvantifikovať všetky premenné, CNF je splniteľná . Je zrejmé, že poradie kvantifikácie premenných má veľký vplyv na výkon algoritmu. Hľadanie najmenšej dočasnej OBDD je však NP-ťažké. Preto sa skúma viacero heuristík.

4.2 Splniteľnosť pomocou odvodzovacích pravidiel

4.2.1 Stålmarckov algoritmus

Stålmarckov algoritmus je algoritmus na určenie, či je zadaná výroková formula (nie nutne v CNF) tautológia. Na určenie splniteľnosti CNF môžeme skontrolovať, či je jej negácia tautológiou. Ukážeme si základný princíp tohto algoritmu.

Na začiatku vykoná algoritmus predspracovanie, v ktorom prevedie implikácie na disjunkcie, odstráni dvojité negácie a aplikuje jednoduché odvodzovacie pravidlá. Ak sa z výslednej formuly dá určiť pravdivostná hodnota formuly, končí. Inak pokračuje. Výsledná formula je prevedená do podoby trojíc. Každá má podobu $p \Leftrightarrow (q \otimes r)$, kde $\otimes$ je buď konjunkcia, alebo ekvivalencia. p je booleovská premenná, r a q sú literály. Počas tohto procesu sa môžu vytvoriť nové premenné reprezentujúce rôzne podformuly. Po tomto prevode sa očakáva, že zadaná formula je nepravdivá. Algoritmus sa bude snažiť nájsť rozpor, čo bude znamenať, že neznegovaná pôvodná formula bola tautológia.

Algoritmus používa sadu pravidiel ("simple rules"), aby získal ďalšie pravidlá. Aplikácia pravidiel, dokým to je možné, sa nazýva 0-saturácia. Ak sa po 0-saturácii dá určiť pravdivosť, algoritmus končí. Inak prechádza k pravidlu nazývanému "dilemma rule".

"Dilemma rule"je spôsob dedukcie nových záverov o formule predpokladmi, ktoré umožňujú zjednodušenie. Funkcia je nasledovná: Majme formulu Δ, ktorá je po 0-saturácii. Pre každú booleovskú premennú v v Δ, algoritmus 0-saturuje $\Delta|v$ a $\Delta|\neg v$, aby získal závery Γ_v a $\Gamma_{\neg v}$. Potom ich prienik je pridaný do Δ. Inak povedané, algoritmus vyskúša obidve hodnoty premennej a uloží závery odvoditeľné z oboch vetví. Tieto závery platia nezávisle od hodnoty v. Toto sa opakuje, kým sa nedá určiť pravdivostná hodnota formuly. Opakovaná aplikácia "dilemma rule"sa nazýva 1-saturácia . 1-saturácia sa ešte dá rozšíriť tak, že sa naraz aplikuje na všetkých kombináciách n-premenných. Toto sa nazýva n-saturácia. Ak necháme n narásť na dostatočné veľkú hodnotu, máme zaručené, že nájdeme spĺňajúce priradenie, alebo rozpor. Keďže n začína na hodnote 0, má tento algoritmus tendenciu nájsť krátke dôkazy o splniteľnosti, resp. nesplniteľnosti.

4.2.2 HeerHugo

HeerHugo je SAT solver inšpirovaný Stålmarckovým algoritmom. Hoci HeerHugo neslúži na dokazovanie tautológií, mnohé techniky tohto algoritmu sú podobné Stålmarckovmu

algoritmu.

Vstupná formula je najprv prevedená na 3-CNF. Potom sa aplikuje sada základných odvodzovacích pravidiel zvaných "simple rules". Hoci sa tieto pravidlá volajú rovnako ako v predošlom algoritme, sú medzi nimi rozdiely. Pravidlá v HeerHugo sa zdajú byť silnejšie. Obsahujú jednotkovú propagáciu, subsumpciu a obmedzenú formu rezolúcie.

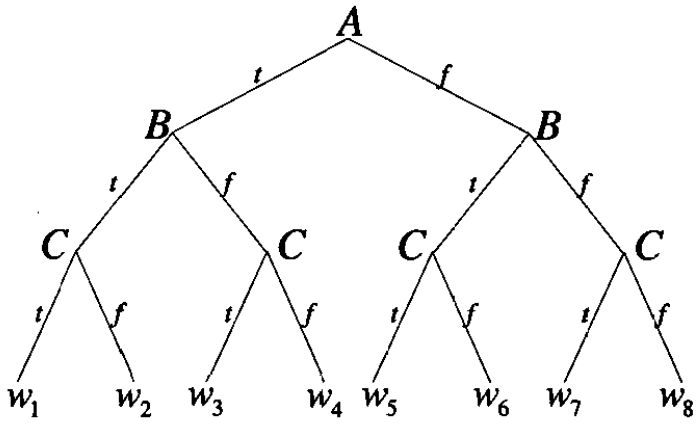
Rezolúcia sa vykonáva podobne ako v DP s tým rozdielom, že v HeerHugo sa vykoná len, ak výsledkom bude formula s menším počtom literálov. Za následok sa neustále pridávajú a uberajú pravidlá z báze znalostí.

Počas spracovania simple rules sa, podobne ako v Stålmarckovom algoritme, dá určiť nesplniteľnosť, ak sa narazí na rozpor. V opačnom prípade sa pokračuje použitím "branch/merge rule", ktoré je v princípe zhodné so Stålmarckovým "dilemma rule". Algoritmus tiež končí akonáhle sa dá určiť nejaký záver z pravidiel.

Úplnosť je dosiahnutá zväčšovaním počtu premenných použitých v branch/merge rule, podobne ako pri n-saturácii.

4.3 Splniteľnosť hľadaním - DPLL algoritmus

DPLL algoritmus vznikol ako odpoveď na priestorové nároky DP algoritmu 4.1. Prácou DPLL algoritmu je prehľadávanie stromu do hĺbky, kde každá úroveň predstavuje premennú a každá vetva predstavuje pravdivostné ohodnotenie premennej (tzv. "search tree"na obrázku 4.1). Prechod týmto stromom teda určuje priradenie hodnôt jednotlivým premenným.



Obrázek 4.1: "Search tree"s poradím premenných A, B, C (obrázok prevzatý z [8])

Pri každom nastavení hodnoty sa skontroluje pravdivosť zadaného CNF. Algoritmus teda dokáže určiť pravdivé alebo nepravdivé ohodnotenie aj na vnútorných uzloch.

4.3.1 Termination tree

Jeden zo spôsobov, ako určiť množstvo práce vykonanej spomínaným algoritmom, je použitie tzv. "termination tree": podmnožina "search tree", ktorá bola prejdená počas prehľadávania. Na obrázku 4.2 vidíme príklad "termination tree"pre CNF

$$\Delta = \{\{\neg A, B\}, \{\neg B, \neg C\}, \{C, \neg D\}\}$$

Algoritmus DPLL(CNF Δ, depth d). Vracia sadu literálov, alebo $UNSATISFIABLE$.

if $\Delta = \{\}$ then
  return $\{\}$
else if $\{\} \in \Delta$ then
  return $UNSATISFIABLE$
else if $L = DPLL(\Delta|P_{d+1}, d+1) \neq UNSATISFIABLE$ then
  return $L \cup P_{d+1}$
else if $L = DPLL(\Delta|\neg P_{d+1}, d+1) \neq UNSATISFIABLE$ then
  return $L \cup \neg P_{d+1}$
else
  return $UNSATISFIABLE$

Tabulka 4.1: DPLL algoritmus (prevzaté z algoritmu 3.3 v [8])

Na tomto obrázku vidíme, že výraz je pravdivý pri hodnotách $A \wedge B \wedge \neg C \wedge \neg D$. Takisto vidíme celú cestu hodnôt premenných, ktoré boli vyskúšané. Tieto stromy tiež môžu pomôcť pri určovaní zložitosti problému, ktorá sa dá odhadnúť z výšky a šírky stromu.

4.4 Splniteľnosť kombináciou vyhľadávania a odvodzovania

Moderné SAT algoritmy vychádzajú z DPLL. Vylepšenia, ktoré však implementujú, upravujú správanie DPLL tak, že je lepšie k nim pristupovať ako k súhre vyhľadávania a odvodzovania. Úspech SAT solverov implementujúcich toto správanie (GRASP, ReLsat) spôsobila rozmach moderných SAT solverov ako BerkMin, JeruSAT, MiniSAT, PicoSAT, Rsat, Siege, TiniSAT, zChaff. Aby sme pochopili ich úspech, treba si ukázať hlavné obmedzenie základného DPLL algoritmu.

4.4.1 Chronologický backtracking

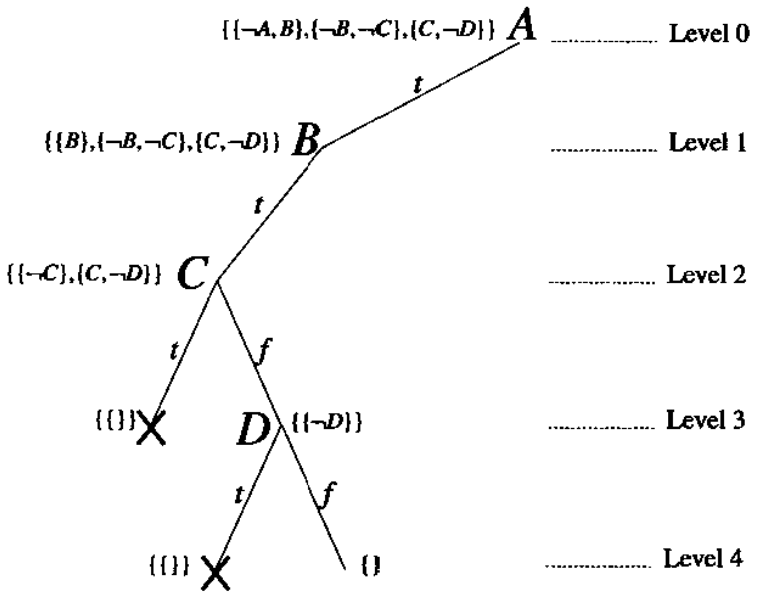
DPLL je založené na chronologickom backtrackingu (v preklade chronologické spätné vyhľadávacie). To znamená, že na úrovni l sa pokúsi priradiť obe hodnoty a ak je pri oboch rozpor, vráti sa na úroveň $l-1$, odvolá všetky priradenia na vyššej úrovni a pokúsi sa priradiť inú hodnotu ako je aktuálna (ak ešte nejaká hodnota nebola vyskúšaná). Ak už žiadna hodnota nezostáva, vracia sa na úroveň $l-2$, kde sa predošlý postup opakuje. Ak sa dostane až na úroveň 0 a už viac možností na vyskúšanie nezostáva, vyhlasuje sa CNF za nesplniteľnú.

Proces návratu na nižšiu úroveň je známy ako backtracking. Ak sa vracia na úroveň l len po vyskúšaní všetkých možností na $l+1$, je to známe ako chronologický backtracking, čo je základný princíp funkcionality DPLL. Problémom tohto prístupu je, že neberie do úvahy príčinu konfliktu, ktorá spôsobuje backtracking.

Príklad: Majme poradie premenných A, B, C, X, Y, Z a CNF

$$\Delta = \{\{A, B\}, \{B, C\}, \{\neg A, \neg X, Y\}, \{\neg A, X, Z\}, \{\neg A, \neg Y, Z\}, \{\neg A, X, \neg Z\}, \{\neg A, \neg Y, \neg Z\}\}$$

Na obrázku 4.3 vidíme "termination tree"pre uvedenú CNF. Vidíme, že ak sa nastaví A na true, každé priradenie hodnôt ostatným premenným vedie ku konfliktu, tj. $\Delta \wedge A$ je

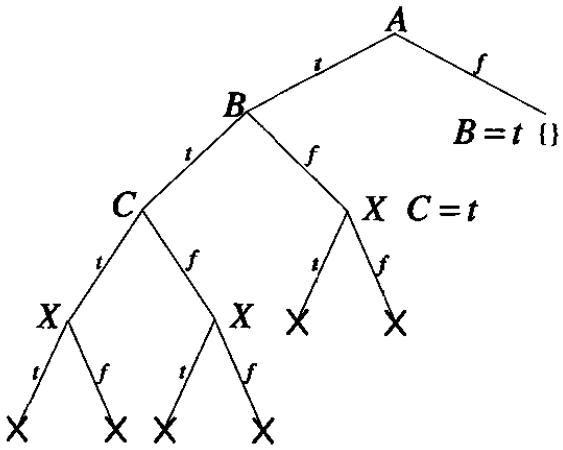


Obrázek 4.2: "Termination tree"s poradím premenných A, B, C, D (obrázok prevzatý z [8])

nekonzistentná. Vidíme, že k tomuto zisteniu musela byť priradená hodnota premenným B, C a X.

Ďalším príkladom by bola skutočnosť, že hoci $\Delta|A, B, C, X$ a $\Delta A, B, C, \neg X|$ obsahujú rozpor, čo implikuje, že $\Delta|A, B, C$ je v rozpore, algoritmus toto nedokáže určiť. Miesto toho sa pokúsi priradiť inú hodnotu C, kde zase zistí, že obidva podstromy obsahujú rozpor.

Podstatou je, že hoci neustále nachádza rozpory, snaží sa ich obísť tým, že priraďuje iné hodnoty premenným, ktoré nemajú na konflikty vplyv (menovite B, C). Je zrejmé, že vo väčších stromoch toto môže spôsobiť ohromné zdržanie, keďže sa algoritmus môže neustále snažiť vyhnúť konfliktu nastavovaním nepodstatných premenných.



Obrázek 4.3: "Termination tree"s poradím premenných A, B, C, X (obrázok prevzatý z [8])

4.4.2 Nechronologický backtracking

Tento prístup sa snaží riešiť problém chronologického backtrackingu. Je to metóda, pomocou ktorej sa dá rýchlo dostať z časti grafu, ktoré nemá žiadne riešenie. Tento proces dokáže aplikovať backtracking na nejakú úroveň bez toho, aby bolo potrebné vyskúšať všetky možnosti priradení hodnôt premenným. Ako sme si ukázali v predošlej podkapitole, konflikt môže mať na svedomí premenná na veľmi nízkej úrovni a kompletné prehľadávanie by bolo zbytočné.

Nechronologický backtracking sa dá vykonať po identifikácii všetkých priradení, ktoré prispeli ku konfliktu. Táto sada priradení sa nazýva konfliktná sada. Potom sa pokúsi negovať hodnotu poslednej premennej, ktorej zmena prispela ku konfliktu. To je v kontraste s chronologickým backtrackingom, ktorý sa neustále snaží zmeniť hodnotu posledne nastavenej premennej. Pri návrate na úroveň l sa zrušia všetky priradenia premenných vyššie ako l.

Príklad: V predošlom prípade sa priradí $A = true$, $B = true$, $C = true$, $X = true$. Jednotková propagácia potom odvodí $Y = true$ (z klauzuly $\{\neg A, \neg X, Y\}$) a $Z = true($ z $\{\neg A, \neg Y, X\})$. Klauzula $\{\neg A, \neg Y, \neg Z\}$ sa potom stane prázdnou klauzulou. V tomto prípade bude konfliktná sada $\{A = true, X = true, Y = true, Z = true\}$. B a C sa nezúčastňujú konfliktu. Nechronologický backtracking sa pokúsi zmeniť posledne nastavenej premennej hodnotu, ktorou je v tomto prípade X. Pri $X = false$ bude znovu konflikt s konfliktnou sadou $\{A = true, X = false, Z = true\}$. Keďže už boli vyskúšané všetky hodnoty pre X, bude treba zmeniť hodnotu A. Tu vidno výhodu nechronologického backtrackingu: nemenili sa hodnoty B a C a stačili len 2 konflikty, aby sa prišlo na to, že pri $A = true$, nie je CNF konzistentná.

Nechronologický backtracking čiastočne zabraňuje algoritmu, aby opakoval rovnaký konflikt. Avšak akonáhle algoritmus prejde backtrackingom všetky premenné z konfliktnej sady, môže nastať znovu rovnaký konflikt.

Toto sa dá riešiť tak, že sa umožní pridávanie klauzúl. Napríklad v predošlom prípade je implikované $\{\neg A, \neg X\}$. Ak by táto klauzula bola vo formule prítomná od začiatku, zistil by sa konflikt hneď ako sa nastaví A na true. Takto získané nové klauzuly sa nazývajú konfliktné klauzuly. Pridanie takejto klauzuly umožní skoršiu detekciu konfliktu a zabráni opakovaniu rovnakej chyby v budúcnosti. Učenie z konfliktov má pôvod z úspechov riešenia problémov typu Constraint Satisfaction Problem pomocou nich.

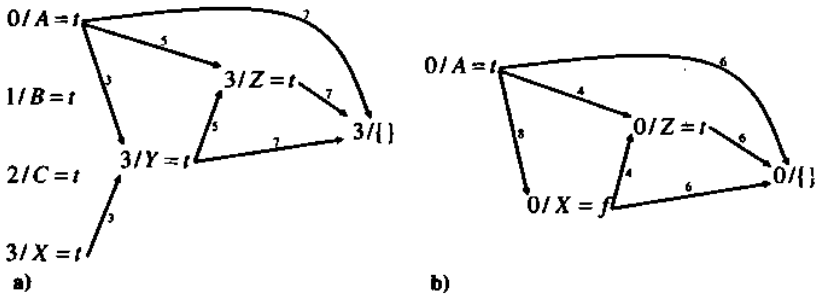
4.4.3 Nechronologický backtracking a konfliktné klauzuly

Konfliktné klauzuly sú súčasťou moderných SAT solverov. V tejto podkapitole si ukážeme ako sa dajú získať. Proces získavanie takýchto klauzúl sa nazýva konfliktová analýza, ktorá pracuje s implikačným grafom.

4.4.4 Implikačný graf

Na obrázku 4.4 vidíme dva implikačné grafy. Obrázok (a), ktorý odpovedá príkladu z obrázku 4.3, ukazuje výsledný implikačný graf vzniknutý po nastavení A, B, C a X na true.

Každý uzol má značenie $l/V = v$, kde V je názov premennej, ktorej hodnota bola nastavená na v, na úrovni l. Hodnoty sa premenným nastavujú buď rozhodnutím, alebo implikáciou. Premenná sa nastavuje na hodnotu implikáciou, ak je hodnota nastavená kvôli jednotkovej propagácii, inak je nastavená rozhodnutím.

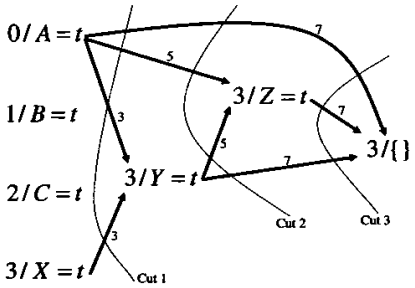


Obrázek 4.4: Príklady implikačných grafov (obrázok prevzatý z [8])

Ak sa použije implikácia na nastavenie hodnoty, použije klauzulu a niekoľko premenných na nastavenie. V tomto prípade sa v grafe vytvorí hrana z každej premennej, ktorá je v klauzule, do premennej, ktorá je touto implikáciou nastavená. Číslo na hrane určuje klauzulu, ktorá je použitá. V grafe na obrázku 4.4a je premenná Y nastavená na true na treťej úrovni klauzulou 3 a priradeniami $A = true$, $X = true$, preto z nich smeruje hrana do Y. Hodnota 3 nad hranou značí, že priradenie bolo spôsobené 3. klauzulou. V grafe sa môže nachádzať zvláštny uzol označený $\{\}$, do ktorého idú hrany z premenných z klauzuly, z ktorej vznikla nulová klauzula.

4.4.5 Získanie konfliktnej klauzuly

Proces získania konfliktnej klauzuly spočíva v získaní konfliktnej sady z implikačného grafu a jej prevod na klauzulu. Konfliktná sada sa dá získať nasledovne: každý rez implikačným grafom definuje konfliktnú sadu, kým rozdeľuje rozhodovacie premenné (koreňové uzly) a nulovú klauzulu (listový uzol). Každý uzol, z ktorého vychádzajúca hrana pretína rez, bude v konfliktnej sade. Na obrázku 4.5 sú znázornené niektoré rezy vedúce ku konfliktným sadám



Obrázek 4.5: Príklady rezov grafu (obrázok prevzatý z [8])

$\{A = true, X = true\}$, $\{A = true, Y = true\}$, $\{A = true, Y = true, Z = true\}$.

Keďže môže naraz vzniknúť viacero konfliktných sád, treba nejakým spôsobom určiť najzaujímavejšiu. V praxi sa všetky moderné SAT solvery snažia zvoliť rezy, ktoré obsahujú práve jednu premennú, ktorá bola nastavená na aktuálnej úrovni. V prípade obrázku 4.5a je to $\{Y = true, A = true\}$ a $\{X = true, A = true\}$. V prípade obrázku 4.5b $\{A = true\}$.

Akonáhle sa nájde vhodný rez, konfliktná klauzula sa dá získať negáciou priradení v konfliktnej sade. Konfliktná klauzula sa nazýva vynucovacia, keď práve jedna premenná bola

nastavená na aktuálnej úrovni rozhodnutím. Moderné SAT solvery sa učia výhradne vynucovacie.

4.4.6 Pridávanie konfliktných klauzúl a backtracking

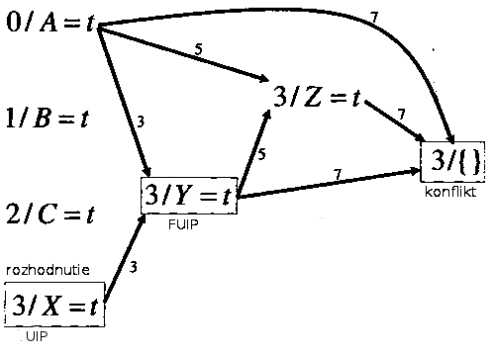
Pridávanie konfliktných klauzúl do CNF sa nazýva učenie klauzúl (clause learning).

Po objavení konfliktu a pridaní novej klauzuly treba určiť, ako ďalej pokračovať. V predošlom príklade sme narazili na konflikt tým, že sme priradili X hodnotu true. To znamená, že minimálne toto rozhodnutie treba odvolať. Moderné SAT solvery však zrušia všetky vykonané priradenia po vynucovacej úrovni. Vynucovacia úroveň je druhá najvyššia úroveň v konfliktnej klauzule. V našom prípade je vynucovacia úroveň 0, lebo A malo priradenú hodnotu na úrovni 0, X na úrovni 3. V prípade, že pracujeme s jednotkovou klauzulou, je vynucovacia úroveň o 1 nižšia ako úroveň, na ktorej bola získaná konfliktná sada (čo vyplýva z definície). Dôvodom návratu na túto úroveň je to, že je to najnižšia úroveň, na ktorej sa dá odvodiť nová implikácia z novovytvorenej klauzuly. Kvôli tomuto backtrackujú moderné SAT solvery na túto úroveň, pridajú novú klauzulu, aplikujú jednotkovú propagáciu a pokračujú v hľadaní. Takýto druh backtrackingu sa nazýva far-backtracking.

SAT solvery, ktoré aplikujú učenie klauzúl, dokážu nájsť exponenciálne kratšie dôkazy ako tie, ktoré ho nepoužívajú.

4.4.7 Unikátne implikačné body

Častou úpravou konfliktných klauzúl je zavedenie myšlienky unikátneho implikačného bodu (UIP, Unique implication point). UIP na nejakej úrovni znamená také priradenie hodnoty na danej úrovni, ktorým prechádza každá rozhodovacia premenná na tej úrovni. V predošlom prípade by to bolo $3/Y = true$ a $3/X = true$. .



Obrázek 4.6: UIP pre implikačný graf na obrázku 4.4a (obrázok vytvorený na základe obr. 3.9 z [8])

Zvláštnu pozíciu má prvý UIP (FUIP, first implication point), ktorým je UIP nachádzajúci sa najbližšie ku konfliktu. Tento býva často používaný v moderných SAT solveroch, lebo vynucovacie klauzuly vytvorené pomocou FUIP majú tendenciu vytvárať kratšie klauzuly, ktoré potenciálne dovoľujú viac zmenšiť prehľadávaný stavový priestor. Táto schéma využívania UIP sa nazýva 1UIP schéma.

4.4.8 Hľadanie 1UIP

Hovoríme, že uzol x dominuje uzol y, keď každá cesta z počiatočného uzlu do uzlu y musí prechádzať uzlom x. Uzol x striktne dominuje y, ak x dominuje y a $x \neq y$. Každý uzol môže byť dominovaný viacerými uzlami. Tieto uzly sa nazývajú dominátory. Každý dominovaný uzol má jeden unikátny "najbližší"dominátor. Keďže každý uzol má len jeden takýto dominátor, vytvorením grafu zo všetkých dominátorov dostaneme strom.

Všetky UIP sa dajú účinne nájsť pomocou Lengauer-Tarjan algoritmu v čase (n^2) ([22]). Tento algoritmus vytvorí strom dominátorov pre daný graf. Na nájdenie 1UIP treba v tomto strome nájsť dominátor uzlu, ktorý predstavuje konfliktnú premennú.

4.4.9 Úplnosť učenia klauzúl a far backtrackingu

Algoritmy využívajúce far backtracking a učenie klauzúl z FUIP nevylučujú opakované priradenie rovnakých hodnôt premenným, ktoré znovu povedú ku konfliktu. Avšak pridanie novej vynucovacej klauzuly do CNF po každom nájdenom konflikte znamená, že zakaždým bude nový konflikt v inom kontexte. Je dokázané, že tento algoritmus je úplný.

4.4.10 Mazanie konfliktných klauzúl

Pridávanie nových klauzúl je užitočné, lebo zvyšuje účinnosť jednotkovej propagácie. Môže však nastať situácia, keď počet nových klauzúl výrazne presahuje počet pôvodných klauzúl, a tým je spomaľovaný celý algoritmus, keďže ich jednotková propagácia musí spracovávať viac. Existujú aspoň 2 metódy ako toto riešiť. Prvou je možnosť, že nové konfliktné klauzuly subsumujú staré, čím sa zmenší ich počet, alebo sa skrátia. Druhou je premazávať konfliktné klauzuly na základe nejakej heuristiky (vek, aktivita, dĺžka, apod.).

4.5 SAT solvery s predvídaním

Táto podkapitola čerpá z [18]. Rozdiel medzi SAT solvermi s predvídaním a konfliktom riadenými SAT solvermi je ten, že konfliktom riadené SAT solvery vykonávajú hľadanie jedným smerom, kým nenarazia na konflikt. SAT solvery s predvídaním v každom kroku pozerajú krok (prípadne viac) dopredu a na základe nejakej heuristiky ohodnotia dôsledky každého rozhodnutia a vyberú si to výhodnejšie z nich.

Takéto SAT solvery majú náskok oproti konfliktom riadeným vo formulách, ktoré majú nízku **hustotu** (tj. nízky pomer klauzúl ku premenným) alebo malý **polomer** (najdlhšia najkratšia cesta v rezolučnom grafe - graf, v ktorom uzly sú klauzuly a medzi nimi existuje hrana, ak majú konfliktnú premennú), vid obrázok 4.7

Základom práce predvídacích algoritmov je priradenie hodnôt premenným a opakované aplikovanie jednotkovej propagácie, ak priradením hodnoty vznikne jednotková klauzula. Toto priradenie sa značí $\mathcal{F}[x_{decision} = 0]$, ak sa premennej x priradí false, alebo $\mathcal{F}[x_{decision} = 1]$, ak sa jej priradí true.

Základný algoritmus je v tabuľke 4.2, ktorá používa predvídaciu procedúru LookAhead uvedenú v tabuľke 4.3.

Predvídacia procedúra sa pokúsi určiť účinok priradenia hodnoty true premennej x. Týmto sa pokúsi odhadnúť dôležitosť premennej x. Myšlienka za vykonaním predvídacej procedúry je tá, že dokáže pravdepodobne lepšie predpovedať účinok nastavenia hodnoty ako nejaký odhad na základe nejakej štatistickej hodnoty vo formule.

Algoritmus DPLL s predvídaním DPLL(CNF $\mathcal{F}$).
Vracia $SATISFIABLE$, alebo $UNSATISFIABLE$.

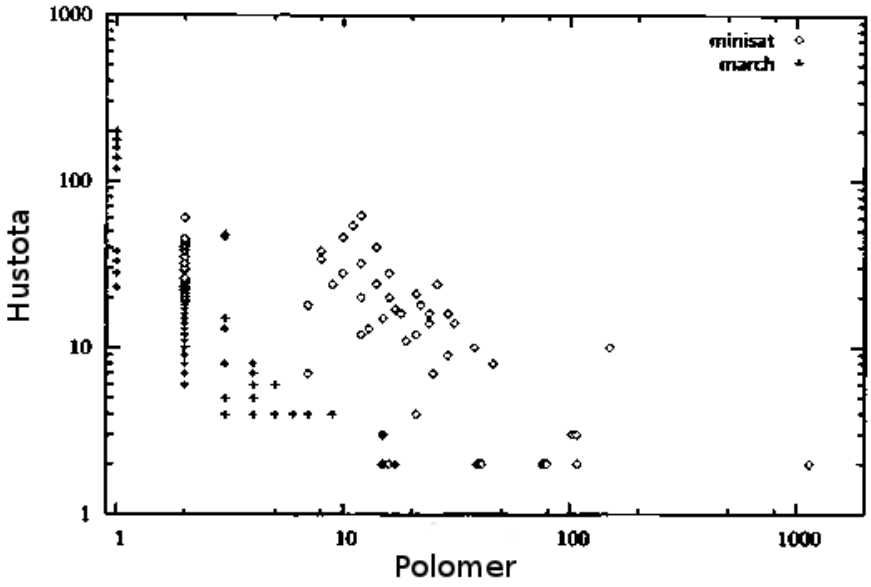
if $\{\} \in \Delta$ then
  return $SATISFIABLE$
$< \mathcal{F}; x_{decision} >$:= LookAhead($\mathcal{F}$)
if $\{\} \in \mathcal{F}$ then
  return $UNSATISFIABLE$
else if žiadna $x_{decision}$ sa nezvolí then
  return DPLL($\mathcal{F}$)
B := SmerováHeuristika($x_{decision}, \mathcal{F}$)
if DPLL($\mathcal{F}[x_{decision} = B]$) = $SATISFIABLE$ then
  return $SATISFIABLE$
return DPLL($\mathcal{F}[x_{decision} = \neg B]$)

Tabulka 4.2: DPLL so zapojením predvídania (prevzaté z algoritmu 5.2 v [18])

LookAhead(CNF $\mathcal{F}$). Vracia najlepšie ohodnotenú premennú

$\mathcal{P}$:= PreSelect($\mathcal{F}$)
repeat
  for all $x_i \in \mathcal{P}$ do
    $\mathcal{F}$:= predspracovanie $(\mathcal{F}, x_i)$
    if $\{\} \in \mathcal{F}[x = 0]$ and $\{\} \in \mathcal{F}[x = 1]$ then
      return $< \mathcal{F}[x = 0]; * >$
    else if $\{\} \in \mathcal{F}[x = 0]$ then
      return $\mathcal{F} := \mathcal{F}[x = 1]$
    else if $\{\} \in \mathcal{F}[x = 1]$ then
      return $\mathcal{F} := \mathcal{F}[x = 0]$
    else
      $H(x_i)$ = RozhodovaciaHeuristika($\mathcal{F}, \mathcal{F}[x = 0], \mathcal{F}[x = 1]$)
    until (nič zaujímavého sa nezistí)
return $< \mathcal{F}; x_i$ s najlepším ohodnotením $H(x_i) >$

Tabulka 4.3: Základná predvídacia procedúra (prevzaté z algoritmu 5.3 v [18])



Obrázek 4.7: Porovnanie solverov minisat (konfliktom riadený SAT solver) a march (SAT solver s predvídaním). Vidíme jasný rozdiel vo výkonoch v jednotlivých oblastiach (obrázok vytvorený na základe obr. 5.2 z [18])

Týmto sa dostávame k dvom hlavným súčastiam predvídacej procedúry. Prvou je rozhodovacia heuristika. Tá je zložená z rozdielovej alebo vzdialenostnej heuristiky (skrátene DIFF) a z heuristiky, ktorá kombinuje dve DIFF hodnoty (MixDIFF).

DIFF heuristika meria redukciu formuly spôsobenou predvídaním. Čím je redukcia väčšia, tým väčšia hodnota výstupu heuristiky. Táto hodnota sa dá merať rôznymi spôsobmi, ako napríklad počet voľných premenných, alebo počet novovytvorených klauzúl. Výslednou hodnotu získame MixDIFF heuristikou, ktorá skombinuje DIFF($\mathcal{F}, \mathcal{F}[x = 0]$) a DIFF($\mathcal{F}, \mathcal{F}[x = 1]$). Ich súčin sa považuje za efektívnu heuristiku.

Druhá súčasť predvídacej procedúry je detekcia konfliktných literálov. Ak predvídanie na nejakej premennej pri nejakej hodnote zlyhá, je nastavená na opačnú hodnotu. Ak je v konflikte aj opačná hodnota, zlyháva predvídacia procedúra.

Jednou z vyvinutých vylepšení predvídacej architektúry bola snaha zníženie ceny predvíacej procedúry tak, že sa vykonáva len na podmnožine voľných premenných. Táto množina (značená $\mathcal{P}$) je zvolená PreSelect procedúrou. Ak je však táto množina príliš malá, môže sa znížiť celkový výkon algoritmu, lebo sa nájde menej konfliktných literálov a je možné, že sa zvolí menej efektívna rozhodovacia premenná. Okrem toho je možné pridať niekoľko ďalších rozhodovacích heuristík, ktoré sú však nad rámec tejto práce. V nasledujúcich podkapitolách si ukážeme niektoré z používaných heuristík na určenie vplyvu nastavenia premennej.

4.5.1 Rozdielové heuristiky

Heuristika klauzulovej redukcie

Táto heuristika, implementovaná v OKsolveri, používa len váhy γ_k. Nazýva sa heuristika klauzulovej redukcie (clause reduction heuristic CRH, známa aj ako MNC). Jej tvar je nasledovný

$$CRH(x_i) := \sum_{k \geq 2} \gamma_k \cdot |\mathcal{F}[x_i = 1]_k \backslash \mathcal{F}| \tag{4.1}$$

Hodnoty pre k menšie ako 7 sú výsledky optimalizácie OKsolveru pre náhodné k-SAT formuly. Hodnoty pre k vyššie ako 6 sú získané lineárnou regresiou.

Heuristika vážených dvojíc

Autormi tejto heuristiky (weighted binaries heuristic, WBH) sú Li a Anbulagan. Je implementovaná v solveri satz. Každej premennej (kladnej aj zápornej) je priradená váha na základe jej výskytu vo formule. $\#(\neg x)$ značí počet výskytov literálu x. Každej novovytvorenej binárnej klauzule je priradená váha ako súčet váh jej dvoch literálov, z ktorých je zložená. Súčet $\#(\neg x) + \#(\neg y)$ vyjadruje počet klauzúl, na ktorých môže byť vykonaná rezolúcia pomocou $x \vee y$.

Váha výskytu premennej v klauzule veľkosti k je 5^{3-k}. Táto hodnota bola získaná po optimalizácii satz solveru na náhodných SAT problémoch.

$$w_{WBH}(x_i) := \sum_{k \geq 2} \gamma_k \cdot \#_k(x_i), \text{ kde } \gamma_k := 5^{3-k} \tag{4.2}$$

$$WBH(x_i) := \sum_{(x \wedge y) \in \{\mathcal{F}[x_i = 1]_2 \backslash \mathcal{F}_2\}} (w_{WBH}(\neg x) + w_{WBH}(\neg y)) \tag{4.3}$$

Heuristika hľadania opory

Táto heuristika (backbone search heuristic, BSH) bola inšpirovaná konceptom opory formuly, ktorou je sada premenných, ktorá má pevnú pravdivostnú hodnotu vo všetkých priradeniach spĺňajúce maximálny počet klauzúl. Premenným je priradená váha, podobne ako u WBH, podľa veľkosti klauzúl výskytu.

Najvýraznejší rozdiel medzi WBH a BSH je, že WBH sčíta váhy zložiek, BSH ich násobí.

$$w_{BSH}(x_i) := \sum_{k \geq 2} \gamma_k \cdot \#_k(x_i), \text{ kde } \gamma_k := 2^{3-k} \tag{4.4}$$

$$BSH(x_i) := \sum_{(x \wedge y) \in \{\mathcal{F}[x_i = 1]_2 \backslash \mathcal{F}_2\}} (w_{BSH}(\neg x) \cdot w_{BSH}(\neg y)) \tag{4.5}$$

.

Je zrejmé, že BSH je výpočtovo náročnejšia ako WBH, teda celkový trvanie behu algoritmu môže urýchliť len v prípade, že poskytuje lepšiu heuristiku. Toto zatiaľ platí len u náhodných formúl.

Existuje procedúra pre BSH, ktorá priradí váhy všetkým novým klauzulám, nie len binárnym. Toto poskytuje vysoký výkon u náhodných k-SAT problémov, kde k je väčšie ako 3. Táto heuristika, nazvaná Backbone Search Renormalized Heuristic (BSRH), sa skladá z 2 ďalších aspektov: menšie klauzuly sú ťažšie a w_{BSH} hodnoty sú renormalizované tak, že sa vydelia priemernou váhou všetkých literálov $\mathcal{F}[x_i = 0] \backslash \mathcal{F}$.

4.5.2 Smerové heuristiky

Tieto heuristiky sa používajú na určenie hodnoty, ktorá sa má priradiť premennej ako prvá, keď sa nejaká premenná na priradenie zvolí. Teoretická sila smerovej heuristiky je veľmi vysoká: ak by bola schopná zakaždým zvoliť priradenie, pre ktoré by bola formula splniteľná, dalo by sa nájsť riešenie bez backtrackingu. Napriek teoretickej sile, je zložité sformulovať ideálnu heuristiku. Ukážeme si niektoré používané v predvídacích SAT solveroch. Tieto sa môžu navzájom dopĺňať.

- knfcs vyberá pravdivostnú hodnotu, ktorá sa vyskytuje častejšie pri danej premennej.
- march vyberá pravdivostnú hodnotu, ktorá má menší rozdiel Diff $(\mathcal{F}, \mathcal{F}[x = v])$, kde v je true alebo false.
- OKsolver vyberá podformulu s najmenšou pravdepodobnosťou, že náhodné priradenie zneplatní náhodnú formulu rovnakej veľkosti
- posit vyberá pravdivostnú hodnotu, ktorá sa vyskytuje častejšie v najkratších klauzulách zadanej formuly
- satz vždy začína priradením true

Tieto heuristiky sa dajú rozdeliť na 2 skupiny:

1. Odhad podformuly, ktorá bude mať najnižšiu časovú zložitosť. Táto stratégia vychádza z toho, že sa ťažko predvída, či je podformula splniteľná. Preto, ak obidve podformuly majú rovnakú očakávanú pravdepodobnosť nájdenia riešenia, nájde sa správne riešenie skôr v časti, ktorá bude vyžadovať menej času. Konfliktom riadené solvery môžu využívať tento prístup, keďže im to umožní skôr naraziť na konflikt.
2. Odhad podformuly, ktorá bude splniteľná s najväčšou pravdepodobnosťou.

Ak je formula splniteľná, je najlepšou možnosťou vetva, ktorá má najvyššiu pravdepodobnosť byť splniteľná. Iné podformuly môžu mať nesprávne riešenie a len zdržujú.

Vo všeobecnosti sú tieto prístupy v protiklade: ak sa od podformuly očakáva, že bude časovo menej zložitá, bude viac obmedzená, takže bude splniteľná s menšou pravdepodobnosťou.

4.5.3 Preselect heuristika

Výber podmnožiny voľných premenných v každom uzle DPLL stromu môže mať výrazný vplyv na celkový výkon algoritmu. V tejto podkapitole si ukážeme dve často využívané heuristiky slúžiace na obmedzenie výberu voľných premenných.

1. $prop_z$: Táto heuristika je používaná v solveroch satz a knfcs. Vyberá premenné podľa ich výskytu v binárnych klauzulách. Bola vyvinutá pre vyšší výkon v ťažkých náhodných k-SAT problémoch. Pri koreni stromu vyberá všetky voľné premenné. Od istej veľkosti stromu začne vyberať len tie premenné, ktoré sa v binárnych klauzulách vyskytujú kladne aj záporne. Minimom je 10 premenných. Táto heuristika je efektívna na náhodné problémy, ale jej účinok v štruktúrovaných inštanciách nie je úplne jasný. V niektorých testoch vyberá stále všetky premenné, v iných vyberá o niečo viac ako je spodná hranica.

2. Aproximácia redukcie klauzúl: Táto redukcia, použitá v march, je založená na počítaní novovytvorených klauzúl (clause reduction approximation, CRA). Používa vzťah

$$CRA(x) := (\sum_{x \wedge y \in \mathcal{F}} \#_{>2}(\neg y_i)) \cdot (\sum_{\neg x \wedge y \in \mathcal{F}} \#_{>2}(\neg y_i)) \tag{4.6}$$

   Aproximáciou sa to nazýva preto, lebo sú do výpočtu zahrnuté aj klauzuly, ktoré budú po zvolenom priradení splnené. Je zrejmé, že táto heuristika je zložitejšia ako predošlá, lebo zakaždým počíta všetky klauzuly.

   Premenné s najvyššou hodnotou sú zvolené v PreSelecte. V praxi sa vyberá 10% s najvyššou hodnotou.

3. Adaptívne hodnotenie: Optimálne množstvo preselectovaných premenných úzko súvisí s počtom konfliktných premenných. Ak bolo v konflikte veľa literálov, zdala sa byť väčšia sada optimálnou. Označme počet literálov v konflikte na uzle i ako $\#failed_i$. Priemerný počet literálov v konflikte použitých ako indikátor pre maximálnu veľkosť sady na uzle n nazveme $RankAdapt_n$:

$$RankAdapt_n := L + \frac{S}{n} \sum_{n=1}^{n} \#failed_i, \tag{4.7}$$

   kde L je spodná hranica $RankAdapt_n$, S je parameter označujúci dôležitosť konfliktných literálov. V march_eq sú tieto hodnoty $L := 7, S := 5$.

   Pri tejto heuristike hrozí, že sa vyberú všetky premenné do sady, keď je veľa konfliktných literálov.

Kapitola 5

Neúplné algoritmy

Táto podkapitola čerpá z [20]. Neúplný algoritmus v kontexte SAT solverov je taký, ktorý garantuje, že nájde spĺňajúce priradenie, alebo vyhlási neúspešnosť. Obvyklá implementácia sa prikláňa ku splniteľnosti. To znamená, že po spotrebovaní maximálneho množstva povolených zdrojov (obvykle čas, môže byť aj miesto apod.) vyhlási splniteľnosť, alebo vyhlási neúspech. Nikdy nevyhlási nesplniteľnosť. O takomto algoritme sa hovorí, že má jednostrannú chybu. V princípe by sa mohol algoritmus prikláňať ku nesplniteľnosti, teda buď ohlásiť nesplniteľnosť alebo neúspech. Môže byť neúplný aj oboma smermi, teda mať obojstrannú chybu.

Narozdiel od úplných metód, obvykle založených na vetveniach a backtrackingu, neúplné metódy sú obvykle založené na stochastickom lokálnom vyhľadávaní (SLS). Takéto metódy majú výkonnostný náskok oproti DPLL metódam v množstve domén. Od počiatku 90. rokov 20. storočia sa vložila veľa úsilia do rozvoja SLS metód pre SAT. Boli vyskúšané aj hybridné prístupy, ktoré kombinovali DPLL a SLS. V tejto kapitole si ukážeme základné prístupy.

V rámci tejto kapitoly bude vhodné pre ilustratívnosť si predstaviť formulu F s n premennými a m klauzulami ako terén v priestore $\{0,1\}^n \times \langle 0, m\rangle$, kde 2^n možných priradení predstavuje body$\{0,1\}^n$ a hodnota z intervalu $\langle 0, m\rangle$ vyjadruje počet klauzúl, ktoré sú nesplnené aktuálnymi priradenými hodnotami. Riešenia v tomto teréne sa nachádzajú presne na bodoch, kde je výška nulová. Hľadá sa teda globálne minimum v rámci tohto terénu. Je zrejmé, že ak by terén neobsahoval lokálne minimá, bol by hladový prístup optimálny. Avšak formuly ich majú veľmi často, takže tento problém treba riešiť.

5.1 Hladové vyhľadávanie a sústredený náhodný prechod

5.1.1 GSAT

GSAT je solver využívajúci lokálne vyhľadávacie metódy. Je zvláštny tým, že napriek svojej pomernej jednoduchosti nielenže dokáže riešiť SAT problémy, ale v mnohých prípadoch dokonca nájde riešenie skôr ako systematické metódy.

Základný prístup je taký, že sa najprv vytvorí náhodné priradenie hodnôt premenným. Potom sa neguje hodnota na premennej, ktorá spôsobí najväčší pokles v počte nesplnených klauzúl. Toto pokračuje, kým nie je dosiahnutý istý počet negácií, prípadne sa nájde priradenie, ktoré spĺňa formulu. Ak sa dosiahne zadaný počet negácií, opakuje sa celý postup. Toto celé sa opakuje niekoľkokrát.

GSAT na začína typicky rýchlym poklesom v počte nesplnených klauzúl, kým sa nedostane na priradenie, ktorá má vlastnosť, že po zmene priradených hodnôt sa nezmení počet nesplnených klauzúl. Experimentálne sa dokázalo, že GSAT trávi väčšinu času v takýchto priradeniach. Takisto sa ale prišlo na to, že v praxi má väčšina takýchto priradení "východ", cez ktorý sa GSAT dostane k priradeniu s menším počtom nesplnených klauzúl. Intuitívne, ak má formula veľa premenných, je malá pravdepodobnosť, že algoritmus narazí na lokálne minimum. V praxi toto znamená, že GSAT sa málokedy zastaví v lokálnom minime, hoci môže potrvať značnú dobu, kým sa minimum nájde. Toto bolo motiváciou ku snahe nájsť úpravy, ktoré by urýchlili výpočet. Jeden z najúspešnejších prístupov bolo zavedenie šumu, ktorý spôsobí občasné zvýšenie počtu nesplnených premenných, čo tvorí základ metódy zvanej WalkSat

5.1.2 Walksat

Walksat prekladá hladové vyhľadávanie GSAT-u s náhodnou negáciou premennej. Okrem toho zameriava vyhľadávanie tým, že vyberá premennú z nesplnenej klauzuly. Táto zdanlivo jednoduchá idea umožnila škálovanie algoritmu na formuly s viac ako niekoľkými stovkami premenných. Ak je možné negovať hodnotu premennej bez toho, aby sa zneplatnila akákoľvek aktuálne platná klauzula, algoritmus ju neguje. Inak s istou pravdepodobnosťou prehodí náhodnú premennú, alebo vyberie premennú, ktorá zníži počet konfliktných klauzúl. Walksat zavádza hodnotu šumu, značenú p, kde $p \in \langle 0, 1 \rangle$, ktorá určuje pravdepodobnosť náhodného priradenia. Empiricky bolo zistené, že pre rôzne triedy formúl je ideálna iná hodnota p. Napríklad pre 3-SAT je to 0.57. Walksat s touto hodnotou šumu škáluje takmer lineárne s hustotou až k hodnote okolo $4,2$, čo však nedosahuje najproblematickejšiu oblasť, ktorá má hustotou $4,26$ (tento pojem je bližšie vysvetlený v podkapitole 4.2).

5.1.3 Ďalšie varianty

Medzi ďalšie vylepšenia, ktoré boli vyskúšané, patrí adaptívne nastavovanie šumu v adaptnovelty, zavedenie jednotkovej propagácie (UnityWalk), zavedenie rezolúcie, využitie štruktúry problému na jej riešenie. Okrem toho sa zistilo, že vhodné kódovanie problému tiež môže urýchliť riešenie stochastickými solvermi. Úprava, kde sa váhy neupravovali aditívne, ale multiplikatívne, dosiali ďalšie urýchlenie. Z prístupov, ktoré upravujú váhy, sú najznámejšie SAPS (scaling and probabilistic smoothing) a RSAPS (reactive scaling and probabilistic smoothing), PAWS (pure additive weighing scheme).

Ďalšie úpravy GSAT boli TSAT, ktorý obsahuje tabu zoznam, aby sa prehľadávanie nevracal k predošlým priradeniam a HSAT, ktorý sa snaží riešiť dlhšie zotrvanie algoritmu v lokálnom minime zmenou najdlhšie nenastavenej premennej. Tieto prístupy poskytli urýchlenie, ale nie až také ako náhodné stúpanie pridané do Walksat-u.

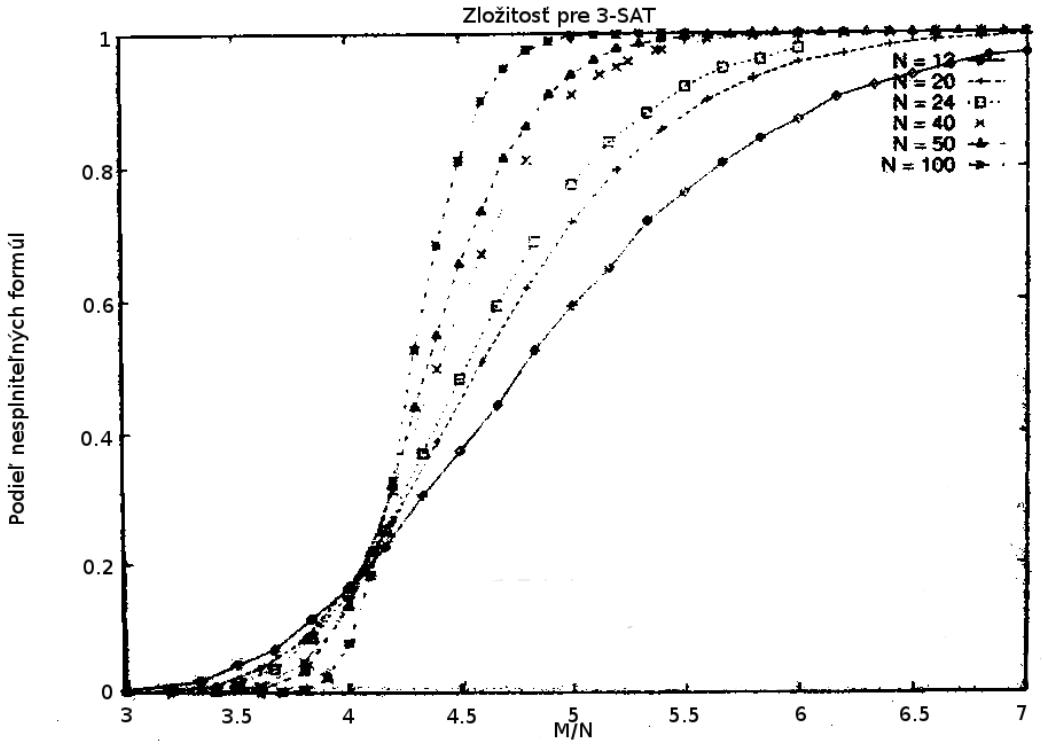
Efektivita algoritmu

V [26] sa autori snažili určiť metriky, pomocou ktorých sa dá určiť efektivita lokálneho prehľadávania. Zaviedli 3 pomerne jednoduché metriky: hĺbka, mobilita, pokrytie. Hĺbka určuje rýchlosť zmeny nesplnených klauzúl počas hľadania. Mobilita určuje rýchlosť presunu z jedného vyhľadávacieho regiónu do druhého. Pokrytie je systematickosť vyhľadávania. Autori predpokladajú, že úspešné prístupy k lokálnemu vyhľadávaniu nefungujú dobre vďaka nejakej zvláštnej vlastnosti, ale proste len preto, lebo klesajú a pri dne prehľadávajú tak rýchlo, obšírne a systematicky ako len dokážu, kým nenarazia na riešenie.

5.2 Fázový prechod v Random-k-SAT

Teraz si ukážeme triedu problémov, vďaka ktorej zažili neúplné SAT solvery rozmach. Je to kvôli vypozorovaniu skutočnosti, že systémy založené na DPLL sú veľmi nevýkonné na istých náhodne vygenerovaných formulách. Ukážeme si na akých.

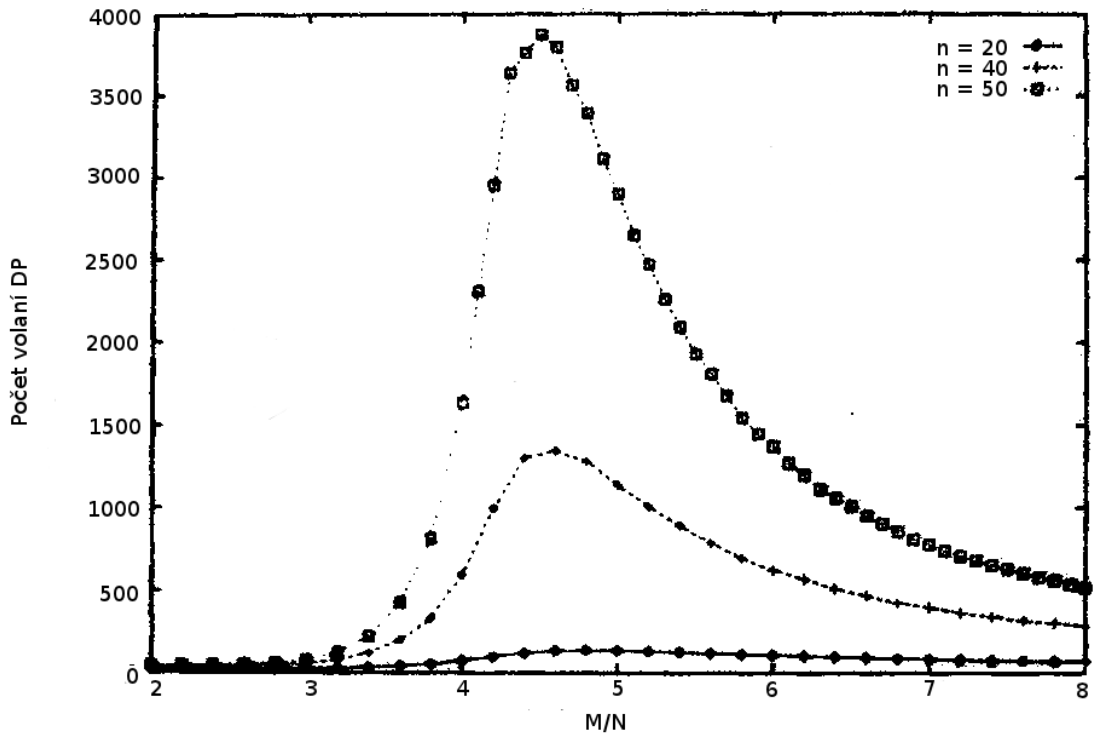
Majme náhodnú k-CNF formulu F na n premenných generovanú nezávislými m klauzulami nasledovne: pre každú klauzulu sa vyberie náhodne k rôznych premenných a každá sa zneguje s pravdepodobnosťou $0,5$. Zistilo sa, že zložitosť takýchto formúl sa dá pomerne jednoducho charakterizovať jedným parametrom: hustotou α, tj. pomer klazúl a premenných. Pre 3-SAT je najvyššia zložitosť v regióne $\alpha \approx 4,26$. Pripomeňme si, že Walksat dokáže pomerne účinne riešiť až do hodnoty $\alpha \approx 4,2$.



Obrázek 5.1: Pomer nesplniteľných inštancií pri daných hustotách (obrázok vytvorený na základe obr. 6.1a z [18])

Formuly s premenným počtom premenných nemajú zrejmú sadu parametrov, ktorá ovplyvňuje ich zložitosť.

Táto kritická oblasť nie je len miestom s vyššími výpočtovými nárokmi, ale takisto je tam nastáva výrazná zmena v splniteľnosti. Čím je α väčšia, tým je menšia pravdepodobnosť, že formula bude riešiteľná, pričom smernica dotyčnice α má najvyššiu hodnotu práve v regióne kritického bodu.



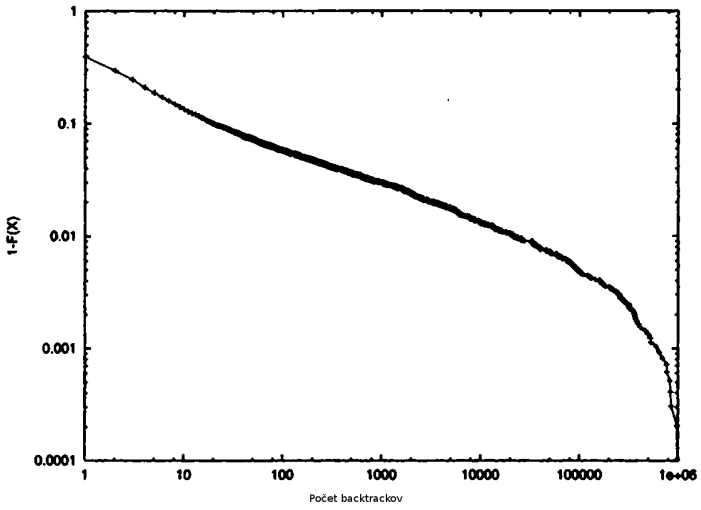
Obrázek 5.2: Priemerná časová zložitosť pri daných hustotách (obrázok vytvorený na základe obr. 6.1b z [18])

Kapitola 6

Reštarty a náhodnosť

Táto podkapitola čerpá z [15]. Výkon randomizovaných backtracking algoritmov sa môže dramaticky meniť každým behom, dokonca aj na rovnakej inštancii. Distribúcia času behu takýchto algoritmov je označovaná ako heavy-tailed. Toto sa netýka fenoménu z kapitoly 5.2, kde v blízkosti nejakej hodnoty parametru sa zložitosť náhle zvýši. Inštancie, na ktorých sa tento jav deje, sú v oblasti, ktorá je málo obmedzená, takže by mala mať veľa riešení. Napriek tomu majú vyššiu časovú náročnosť ako podobné inštancie, dokonca väčšiu ako inštancie v okolí kritického bodu. Tento jav bol spozorovaný najprv pri probléme farbenia grafov a SAT probléme. Dodatočný výskum však ukázal, že tieto problémy nie sú inherentne zložité. Stačilo napríklad premenovať premenné, alebo použiť iné heuristiky a tieto problémy sa zrazu stali jednoduchými. Preto zložitosť v obzvlášť zložitých inštanciách netkvie priamo v inštancii, ale v jej kombinácii s podrobnosťami použitej metódy na riešenie.

Heavy-tailed distribúcie majú niektorý moment nekonečný (priemer, smerodajnú odchýlku atď.). Ukážme si časovú náročnosť inštancie s heavy-tailed rozložením, v grafe s logaritmickými mierkami na osiach, Na obrázku 6.1 vidíme logaritmický pokles až do hodnôt presahujúcich 100 000. To znamená veľký rozptyl zložitostí medzi inštanciami. Metódy



Obrázek 6.1: Príklad heavy tailed funkcie. $F(X)$ je distribučná funkcia (obrázok vytvorený na základe obr. 9.1 z [15])

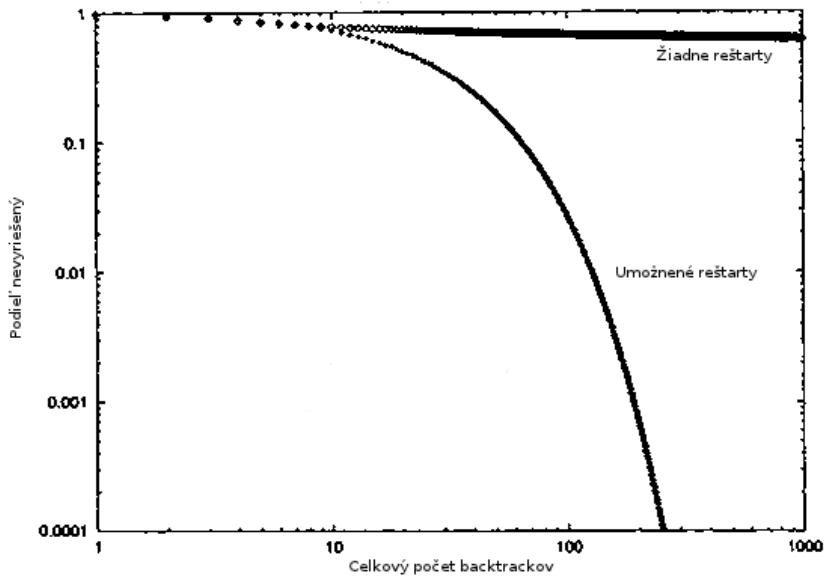
odvodené z DPLL s backtrackingom majú takéto správanie u náhodných, ako aj štruktúro-

vaných inštancií. Ak doba behu metódy odpovedá heavy-tailed rozloženiu, trvajú niektoré behy o niekoľko rádov viac, niekedy však aj o niekoľko rádov menej. Túto vlastnosť sa snažia využiť reštarty a randomizácia.

6.1 Reštarty

Ako už z názvu vyplýva, reštartovanie spočíva v ukončení aktuálneho behu algoritmu a jeho spustení s inou počiatočnou hodnotou. V metódach, ktoré používajú prístup ako je učenie klauzúl, sa naučené dáta nezmažú.

V prítomnosti heavy-tailed rozloženia časovej zložitosti sa sekvencie krátkych behov oplatia viac. Aby sa zaručila úplnosť, môže sa interval medzi reštartami postupne zväčšovať. Na obrázku 6.2 vidno účinok rýchlych reštartov. Konkrétnu inštanciu sa podarí vyriešiť



Obrázek 6.2: Porovnanie zložitosti s umožnenými reštartami a bez možnosti reštartu (obrázok vytvorený na základe obr. 9.5 z [15]

skoro pri každom spustení pri nutnosti použiť iba 150 backtrackingov. Bez reštartov zlyhá pôvodný algoritmus v asi 70% prípadov. Skutočnosť, že krivka v tomto prípade jednoznačne klesá, je ukazovateľom toho, že sa odstránilo heavy-tailed rozloženie časovej zložitosti.

Existuje viac metód ako upravovať počet backtrackov potrebných k reštartu. Ak je distribúcia použitej randomizovanej procedúry plne známa, reštarty stačí opakovať po nejakej konštantnej hodnote. V prípade žiadnych znalostí bola navrhnutá univerzálna stratégia, v ktorej sú dĺžky behov mocniny dvoch nasledujúcim spôsobom: 1, 1, 2, 1, 2, 4, 1, 2, 4, 8... Hoci je tento prístup dokázateľne blízko optimálnej hodnoty, v praxi konverguje príliš pomaly. Preto existuje prístup, kde je hodnota zväčšovaná geometricky. Výhodou takéhoto prístupu je, že je menej citlivý voči detailom distribúcie. Od zChaff-u používajú všetky moderné SAT solvery reštarty. V praxi solvery používajú fixný limit dĺžky behu medzi reštartami. Lineárnym zvyšovaním každých niekoľko reštartov je zaručená úplnosť.

6.2 Ďalšie využitia randomizácie

.Okrem reštartov existuje ešte niekoľko ďalších príležitostí k využitiu randomizácie. Jedna z možností je napríklad využitie v heuristike výberu premennej, heuristika výberu priradenej hodnoty. Využiť sa môže aj pri predvídacích procedúrach napríklad v propagácii, v backtrackujúcich procedúrach pri výbere klauzuly, ktorá sa má naučiť. Aj úplne jednoduché úpravy deterministického algoritmu môžu mať výrazné účinky na jeho správanie. Deterministické heuristiky sú vhodná príležitosť, kde sa dá uplatniť náhodné správanie. Menovite v situáciach, kde heuristika vyhodnotí viac rozhodnutí ako rovnako vhodné, dá sa náhodne vybrať, ktoré sa má použiť. Ďalším využitím je náhodná zmena poradia vstupných dát a aplikácii zvoleného algoritmu. Jedným zo základných predpokladov zavedenia náhodnosti je zaručiť úplnosť algoritmu. Nutno podotknúť, že náhodný výber hodnoty a premennej nezruší úplnosť algoritmu. Treba len zaviesť niekoľko ďalších hodnôt, pomocou ktorých sa zaručí, že sa nebude opakovať voľba. U predvídacích procedúr sa dá použiť náhodnosť v určení, či sa má vôbec predvídacia procedúra zavolať. Toto sa hodí v prípadoch, že je predvídacia procedúra časovo náročná. U náhodného backtrackingu už treba použiť zložitejšie štruktúry, takže náhodnosť v tomto prípade môže spôsobiť značný nárast priestorovej alebo časovej zložitosti.

Kapitola 7

Genetické programovanie

Táto kapitola čerpá z [4]. Posledná podkapitola o CGP, čerpá z [23]. Genetické programovanie (GP) je pomerne mladý druh evolučných algoritmov. Evolučné algoritmy napodobujú aspekty prirodzenej evolúcie pre optimalizáciu riešenia k nejakému dopredu danému cieľu. Vzniklo viacero smerov, napríklad genetické algoritmy, evolučné stratégie, evolučné programovanie. Evolučné algoritmy, spolu s neurónovými sieťami a fuzzy logikou, sú považované za hlavné disciplíny výpočtovej inteligencie.

Základnou jednotkou v genetickom programovaní je jedinec. V ňom je zakódovaná inštancia riešenia. Zakódovanie predstavuje reťazec znakov, ktorých význam závisí od aktuálneho problému. Tento reťazec sa nazýva chromozóm alebo genotyp. Každý jedinec je ohodnotený fitness funkciou. Je to funkcia, ktorá určuje miestu splnenia požiadavkov (počet chýb, vzdialenosť výsledku apod.).

Všeobecný evolučný algoritmus sa dá zhrnúť nasledovne:

1. Vytvor počiatočnú populáciu (množinu chromozómov) a ohodnoť ju.
2. Náhodne vyber indivíduá z populácie, na základe nejakého pravidla.
3. Generuj nové jedince aplikáciou nasledujúcich genetických operátorov s istými pravedpodobnosťami:
   - Reprodukcia: Kopíruj nového jedinca bez zmeny.
   - Rekombinácia: Vymeň štruktúry medzi jedincami.
   - Mutácia: Náhodne nahraď jednu atomickú jednotku v jedincovi.
4. Vypočítaj fitness nových jedincov
5. Ak nie je splnená ukončovacia podmienka, skoč na bod 2.
6. Skonči. Jedinec s najlepším ohodnotením reprezentuje najlepšie nájdené riešenie.

Genetické programovanie sa môže vo všeobecnosti definovať ako priama evolúcia alebo šľachtenie počítačových programov pre účel interaktívneho učenia sa. Väčšina dnešných reálnych aplikácií GP demonštruje jeho schopnosti v dolovaní dát.

Genetické programy sa môžu považovať za predikčné modely, ktoré aproximujú funkciu $f : I^n \to O^m$, kde I^n je vstup s rozmerom n, O^m je m-dimenzionálny výstup. Iné evolučné algoritmy, ako napríklad genetické algoritmy a evolučné stratégie minimalizujú existujúcu objektívnu funkciu hľadaním optimálnych nastavení svojich premenných.

Samotná objektívna funkcia reprezentuje problém, ktorý sa má vyriešiť pomocou GP. V praxi sa získava pomocou trénovacej množiny. Definujeme ju $T = \{(\overrightarrow{i}, \overrightarrow{o}) | \overrightarrow{i} \in I' \subseteq I^n, \overrightarrow{o} \in O' \subseteq O^m, f(\overrightarrow{i}) = \overrightarrow{o}\}$, kde $\overrightarrow{i}$ je množina vstupných vektorov a $\overrightarrow{o}$ množina výstupných vektorov. . Evolučný algoritmus hľadá program, ktorý reprezentuje najlepšie riešenie daného problému. Trénovacím príkladom sa v GP hovorí aj fitness prípady. Od GP modelov sa neočakáva len schopnosť správne vypočítať výstupy z I' ale aj z mnohých vstupov $I^n \backslash I'$. Schopnosť zovšeobecniť sa skúša na základe testovacej sady z príkladov z rovnakej domény (ale s odlišnými parametrami) ako trénovacia sada.

Genotypový priestor $\mathcal{G}$ v GP zahŕňa programy istej reprezentácie, ktorá sa dá zložiť z prvkov programovacieho jazyka $\mathcal{L}$. Ak budeme predpokladať, že program nezavádza vedľajšie účinky, značí fenotypový priestor $\mathcal{P}$ sadu všetkých matematických funkcií $f_{gp} : I^n \rightarrow O^m$, kde $f_{gp} \in \mathcal{P}$, ktorá sa dá vyjadriť programami $gp \in \mathcal{G}$. Použitý programovací jazyk $\mathcal{L}$ je definovaný užívateľom nad sadou inštrukcií a sadou terminálov (môže obsahovať vstupné hodnoty, premenné, konštanty).

Fitness funkcia $\mathcal{F} : \mathcal{P} \rightarrow V$ meria schopnosť predpovedať kvalitu, tj. fitness fenotypu. V rámci tejto práce predpokladajme, že pre spojité problémy je fitness hodnota $V = \mathbb{R}_0^+$ a pre diskrétne $V = \mathbb{N}_0$. Fitness sa odvodí z mapovania chyby medzi chybou predpokladaného modelu f_{gp} a požadovaným modelom f. Keďže fitness dokáže reprezentovať len časť priestoru dát problému, môže odrážať správanie fenotypu len čiastočne.

Vyhodnotenie fitness funkcie jedincov je časovo najzložitejší krok a treba ho vykonať aspoň raz pre každého jedinca. Predtým treba previesť genotypovú reprezentáciu gp na fenotypovú funkciu f_{gp}. Toto mapovanie je obvykle deterministické a vytvorené interpretom $f_{int} : \mathcal{G} \rightarrow \mathcal{P}$, kde $f_{int}(gp) = f_{gp}$ a $\mathcal{F}(gp) = \mathcal{F}(f_{gp})$. Funkcie nie sú bijektívne, tj. fenotyp môže byť reprezentovaný viacerými genotypmi a rôzne fenotypy môžu mať rovnakú fitness hodnotu.

Zloženie inštrukčnej sady a terminálovej sady určuje vyjadrovaciu schopnosť programovacieho jazyka $\mathcal{L}$. Na jednu stranu musí byť jazyk dostatočne silný na reprezentáciu optimálneho riešenia, alebo aspoň dobrých suboptimálnych riešení. Na druhú stranu sa hľadanie riešení stáva zložitejším každou pridanou komponentou. Teoreticky by sa mohol použiť Turingovsky úplný jazyk na nájdenie akejkoľvek vyčísliteľnej funkcie. V praxi sa však odporúča použiť čo najmenší jazyk. Genetické programovanie vyžaduje istú znalosť o problémovej doméne, aby sa dal určiť vhodný kompromis.

Úspešnosť vyhľadávania aj rast programu závisia nielen na reprezentácii, ale aj na variačných operátoroch. Nech $P(t) \subset \mathcal{G}$ značí veľkosť populácie v čase t. Z náhodnej podpopulácie $P' \subseteq P(t)$ zostávajúcu z $n = |P'|$ jedincov vyberie operátor výberu vyberie μ jedincov na variáciu, pričom platí $\mu < n$. Pre globálne výberové schémy platí $P' = P(t)$. Tento operátor vyberie bod z už navštívených bodov, z ktorých môže pokračovať hľadanie s najväčšou pravdepodobnosťou na úspech. V závislosti od rýchlosti reprodukcie p_{rr} je μ rodičov prevedených do populácie $P(t+1)$ generácie $t+1$.

Genetický operátor alebo variačný operátor $v : \mathcal{G}^\mu \rightarrow \mathcal{G}^\lambda$ vytvorí λ potomkov z μ rodičov z populácie $P(t+1)$. Týchto λ jedincov sa tiež stane súčasťou populácie $P(t)$. Pri rekombinácii sa obvykle vytvoria 2 potomkovia z 2 rodičov. V prípade mutácie je to 1 potomok a 1 rodič. Všetky genetické operátory musia zaručiť, že žiadne syntakticky nekorektné sa programy nevygenerujú počas evolúcie a hodnota každej inštrukcie musí byť vo vymedzenom rozsahu.

7.1 Stromové genetické programovanie

Najskorší a obvykle používaný prístup ku genetickému programovaniu je evolúcia stromových štruktúr reprezentovaných vo výrazoch s premenlivou dĺžkou vo funkcionálnom jazyku ako napríklad S-výrazy v LISP. Tento prístup sa nazýva stromové genetické programovanie (TGP). Vnútorné uzly v takomto programe obsahujú inštrukcie. Uzly sa nazývajú terminály a predstavujú vstupy alebo konštanty. Pri tomto prístupe sa neukladajú premenné do pamäte. Toto treba pridať explicitne pomocou špeciálnych funkcií, ktoré pracujú s externou pamäťou. Ich pridanie ale nezaručuje vyššiu funkcionalitu vo funkcionálnom programe.

Počas interpretácie programu je nutný zásobník na ukladanie prechodných hodnôt z každého podstromu. Pri vyhodnocovaní programu sa prechádza každý uzol v nejakom poradí (preorder alebo postorder). Hodnota každého uzlu sa vypočíta aplikáciou jej funkcie na výsledky jej podstromov. Táto hodnota sa vracia rodičovskému uzlu. Koreňový uzol potom obsahuje výsledok.

7.1.1 Genetické operátory v stromoch

Kríženie slúži na rekombináciu na nové a potenciálne lepšie riešenia. V každom rodičovi vyberie kríženia operátor náhodný uzol.

Operátor mutácie vymení jeden terminál, alebo funkčný identifikátor. Obvykle je každý uzol zvolený s rovnakou pravdepodobnosťou. Ďalšou možnosťou je náhrada podstromu iným náhodným stromom.

V štandardnom TGP prístupe sa kríženie odvolá ak niektorý z potomkov presahuje nejakú hranicu zložitosti.

7.2 Lineárne genetické programovanie

Po vzniku TGP vzniklo viacero nových prístupov. Jedným z nich je lineárne genetické programovanie (LGP).

TGP reprezentujú funkcionálny program v stromovej štruktúre. LGP pracuje so sekvenciami inštrukcií imperatívneho jazyka (prípadne strojového jazyka). Podobne, ako "tree"z TGP značí štruktúru reprezentácie, znamená "linear"z LGP štruktúru, neoznačuje LGP funkcionálne programy obmedzené na lineárne zoznamy, ani to, že dokáže riešiť iba lineárne oddeliteľné úlohy.

LGP prináša dve hlavné zmeny oproti TGP:

1. Lineárne genetické programy poskytujú tok dát založený na grafe, ktorý je dôsledkom viacnásobného používania registrov. V TGP je tok dát určený stromovou štruktúrou programu.

2. Špeciálny neefektívny kód existuje spolu s efektívnym kódom v LGP, ktorý vychádza z imperatívnej programovej štruktúry a dá sa detekovať účinne a úplne. V TGP sú všetky zložky pripojené k uzlu. V dôsledku toho závisí existencia neefektívneho kódu na sémantike programu.

V LGP je koncept vetvení veľmi dôležitý. Umožňuje zmenu toku riadenia (v rámci programu). Vetvenie zvyčajne umožní efektívne riešenie klasifikačného problému. Môže však aj zväčšiť zložitosť programu zavedením špecializácie riešenia, čo môže viesť k menej robustnému alebo všeobecnému riešeniu.

Môže byť implementované ako podmienené vykonanie jednej konkrétnej inštrukcie, alebo môže poskytovať ekvivalent goto z jazyka C, kde môže program skočiť do úplne iného bloku.

7.2.1 Iterácia

Iterácia časti kódu hrá pomerne malú úlohu v genetickom programovaní. Väčšina aplikácií GP, ktoré potrebujú cykly, pracujú s riadiacimi problémami, kde vo všeobecnosti je predmetom evolúcie kombinácia primitívnych akcií. V takýchto programoch nie je nutný tok dát. Miesto toho vykoná každá akcia vedľajší efekt v prostredí a fitness je odvodená z posilovacieho signálu. Iterácia však môže priniesť kompaktnejšie riešenia.

Vo funkcionálnom programovaní je koncept cyklov neznámy. Miesto toho používa rekurziu, ktorá sa pomerne ťažko implementuje v TGP.

7.2.2 Modularizácia

Pre niektoré problémy môže byť modularizácia výhodná. Časté opakované využívanie podprocedúr môže viesť k menším riešeniam. Okrem toho môže byť problém rozložený na jednoduchšie podproblémy, ktoré sa dajú riešiť účinne v lokálnych podmoduloch.

Najčastejší modularizačný koncept v genetickom programovaní sa nazýva ADF (automatically defined functions). Podstatou je, že je genetický program rozdelený na hlavný program a istý počet podprogramov (ADF). Hlavný program potom vypočíta výsledok programu pomocou spoluvyvinutých podprogramov pomocou volaní funkcií. Preto sa ADF pokladajú za súčasť hlavnej inštrukčnej sady. Každý model môže byť zložený z rôznych sád programových komponent. Kríženie musí byť obmedzený tak, aby sa medzi dvoma rodičmi rekombinovali moduly len podobného typu.

Je nepravdepodobné, že vzniknú zložité závislosti modulov, ak nie je modularizácia skutočne potrebná pre lepšie riešenie. Vo všeobecnosti, ak je nejaký koncept prebytočný, výsledný nárast prehľadávaného priestoru môže nepriaznivo ovplyvniť hľadanie. Okrem toho účinnosť tiež závisí od programovacieho konceptu, alebo programovej reprezentácie na variačných operátoroch. To znamená, že hoci je vyjadrovacia schopnosť alebo flexibilita programovacieho konceptu vysoká, v princípe môže byť zložité ju využiť evolúciou.

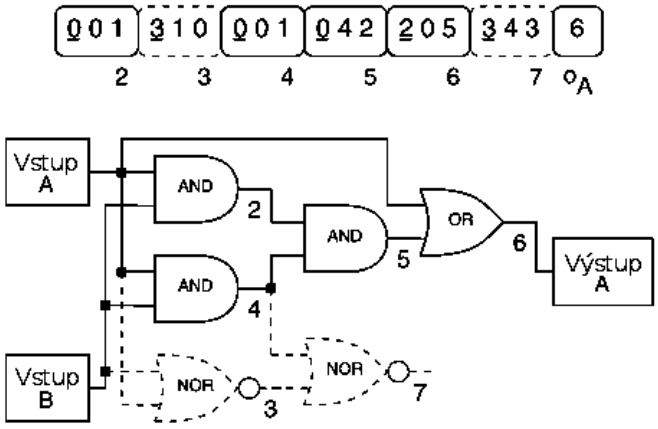
7.3 Kartézske genetické programovanie

Kartézske genetické programovanie (CGP) vzniklo z metódy na vývoj digitálnych obvodov. Pojem "kartézske genetické programovanie"sa objavilo až roku 1999.

Programy v CGP sú programy reprezentované vo forme orientovaného acyklického grafu. Tieto grafy sú reprezentované ako dvojdimenzionálna mriežka. Celočíselné hodnoty uzla predstavujú adresy vstupov, funkciu, ktorú uzol vykonáva, a adresu výstupu.

Genotyp CGP má pevnú dĺžku. Fenotyp môže mať dĺžku od nula uzlov až po počet uzlov ako má genotyp. Je to z dôvodu, že genotyp môže kódovať neefektívne uzly, ktoré nemajú vstup alebo výstup. Na obrázku 7.1 vidíme príklad genotypu aj fenotypu programu v CGP. Vidíme tam aj neaktívne uzly, ktorých výstup sa nikam nenapája (čiarkované uzly)

Uzly berú hodnoty zo svojich vstupov postupne, každý stĺpec po jednom. Každý uzol má maximálne toľko vstupov ako požaduje vstupná funkcia. Uzly v rámci jedného stĺpca sa nemôžu prepájať.



Obrázek 7.1: Príklad CGP (obrázok vytvorený na základe obr. 2.7a z [23])

Pri práci s CGP treba špecifikovať niekoľko parametrov - najmä počet stĺpcov, počet riadkov a počet spätných úrovní. Počet spätných úrovní určuje maximálnu vzdialenosť stĺpca, z ktorého môže uzol brať výstup niektorého uzla ako svoj vstup. Okrem toho treba pri evolúcii brať do úvahy to, že každý uzol musí byť sémanticky správny. Musí sa teda odkazovať do funkcie v rozsahu tabuľky funkcií, a vstupy a výstupy musia odkazovať na existujúce uzly. Pri spojení uzlov ešte musí platiť, že každý vstup odkazuje na uzol s nižším indexom stĺpca ako má aktuálny uzol.

7.3.1 Mutácia

Operátor mutácie používaný v CGP je bodový mutačný operátor. Pri bodovej mutácii sa vyberie náhodný gén a jeho hodnota je zmenená na inú náhodnú platnú hodnotu. To znamená, že kód funkcie sa môže zmeniť len na inú adresu funkcie, adresy vstupov na platné výstupy, atď.

Pre CGP je charakteristické, že drobná mutácia na genotype môže mať drastické účinky na fenotyp.

Aby sa získali dobré výsledky z CGP, treba obvykle experimentovať na danom probléme. Experimentálne sa zistilo, že obvykle závisí podiel génov, ktoré sa môžu mutovať pri vytváraní potomkov závisí od veľkosti genotypu.

Hoci je použitá populácia pre CGP vždy pomerne malá, zistilo sa, že na mnohých problémoch potrebuje v priemere porovnateľne veľa fitness ohodnotení ako iné druhy GP.

7.3.2 Rekombinácia

Kríženie je pomerne málo využívaná vymoženosť v CGP. Pôvodne sa používal bodový operátor kríženia, ale prišlo sa na to, že mal nepriaznivý účinok v rámci CGP.

7.3.3 Evolučný algoritmus

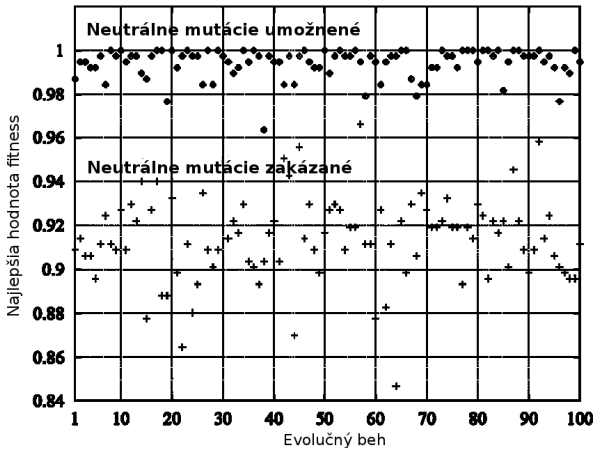
Variant jednoduchého evolučného algoritmu známeho ako $1+\lambda$ algoritmus sa využíva v CGP bežne. Pre λ sa obvykle volí hodnota 4. To znamená, že sa vytvorí 5 genotypov. Z nich sa vyberie najlepšie ohodnotený a vyhlási sa za rodiča. Potom v každom generačom cykle sa

vytvoria 4 potomkovia a následne z množiny týchto potomkov a pôvodného genotypu sa vyberie ten s najlepšou fitness funkciou, ktorý sa vyhlási za nového rodiča.

7.3.4 Genetická nadbytočnosť v CGP genotypoch

Už sme si povedali, že v CGP genotypoch môžu byť úplne neaktívne gény, ktoré nemajú vplyv na fenotyp, teda ani fitness. Prítomnosť týchto génov sa nazýva redundancia. V CGP môžu prevažovať neaktívne gény nad aktívnymi. Vplyv takýchto neaktívnych génov bol detailne preskúmaný a preukázali sa ako veľmi prospešné pre účinnosť evolučného procesu na veľkej škále problémov.

Algoritmus výberu nového novej populácie môže využívať neutralitu. Je to účinok náhrady rodiča potomkom, ak je fitness hodnota potomka a rodiča rovnaká a u žiadneho iného jedinca nie je lepšia. Na obrázku 7.2 ja vidno vplyv neutrality. Sú tam zakreslené najlepšie



Obrázek 7.2: Vplyv neutrality na úspešnosť CGP (obrázok vytvorený na základe obr. 2.8 z [23])

fitness hodnoty 100 nezávislých behov s 10 miliónmi generácií. Cieľom bolo nájsť 3 bitovú paralelnú sčítačku. V prvom prípade algoritme mohol potomok nahradiť rodiča, ak nebol iný jedinec s lepšou fitness hodnotou, v druhom nemohol. V prvom prípade sa našlo 27 správnych riešení a veľmi veľa takmer správnych. V druhom prípade boli fitness hodnoty značne nižšie.

7.3.5 Cyklické CGP

CGP sa prevažne používa v acyklickej podobe, kde nie je spätná väzba v grafe. Toto však nemá žiadny fundamentálny dôvod. V rámci implementácie stačí prakticky len odstrániť podmienku, že vstupy uzlov sa môžu napájať na výstupy uzlov s nižším číslom stĺpca. Je predpoklad, že by toto rozšírenie mohlo zvýšiť expresivitu programov (keďže by bola umožnená iterácia alebo rekurzia). Toto by umožňovalo vyvinúť synchrónne aj asynchrónne obvody. Nejaký výskum už v tomto smere prebehol, napriek tomu zostáva táto téma značne nepreskúmaná.

Kapitola 8

Implementácia

Úlohou práce bude nájsť pomocou lineárneho genetického programovania heuristiky, ktoré riešia zadané inštancie rýchlejšie ako naivný prístup (viď kapitola 3) pri rôzne nastavených podmienkach. Základným cieľom genetického programovania bude usporiadať predvídací postup a konfliktom riadený prístup, pokiaľ možno, optimálnym spôsobom. Okrem toho bude možné použiť niekoľko pomocných funkcií.

Výstup genetického programovania bude spracovaný implementáciou SAT solveru vytvorenú v rámci diplomovej práce. Nutnosť vlastného SAT solveru pramení zo skutočnosti, že existujúce SAT solvery sú lepšie optimalizované na podmnožinu z možných prístupov. Následkom toho by boli iné prístupy ťažko postihované. Aby sa tomu zabránilo, vytvoril som SAT solver, ktorý sa pokúsi čo najviac zamedziť jednostrannej optimalizácii.

Po skončení evolučného procesu sa vytvorí kód v C++, ktorý sa vloží do existujúceho SAT solveru, ktorý obsahuje všetky definície funkcií a potrebné pomocné funkcie. Kód vytvorený genetickým programovaním teda obsahuje len volania funkcií (prípadne skoky, prácu s pomocnými premennými, atď.). Pomocné premenné sú zavedené kvôli možnosti počítať javy a vykonávať podmienený skoky na základe nich.

SAT solver podporuje 2 hlavné prístupy k riešeniu inštancií: konfliktom riadené procedúry a predvídacie procedúry. Tieto procedúry existujú vo viacerých variantoch. Rozdiel medzi variantami spočíva v použitých heuristikách na výber premennej a jej hodnoty.

V nasledujúcich podkapitolách si nadefinujeme funkcie, ktoré sa používajú a ukážeme prácu genetického programovania na tvorbu kódu. Zároveň si ukážeme implementáciu SAT solveru a jeho spôsob práce.

8.1 Funkcie podporované SAT solverom

Výstupom genetického programovania je program preložiteľný kompilátorom C++. Na začiatku tohoto programu je niekoľko riadkov, ktoré načítajú vstup a vhodne ho spracujú.

Nasleduje heuristika vytvorená pomocou genetického programovania. Táto heuristika sa skladá z volaní funkcií, podmienok a úprav pomocných premenných. Každú jednu z týchto inštrukcií predchádza návestie, ktoré sú potrebné pre skoky. Iný spôsob skokov nie je možný. Program vždy začína prácu na nultom návestí.

Za heuristikou sa nachádza skok na nulté návestie, aby bola zaručená úplnosť programu.

8.1.1 Riešiace inštrukcie

Ako sme si povedali, kombinujú sa predvídacie a konfliktom riadené prístupy (konkrétne DPLL+). Je vytvorených niekoľko variantov funkcií, ktoré používajú jeden, alebo druhý z týchto prístupov. Neskôr si ich ukážeme.

Aby SAT solver bol schopný nájsť riešenie danej inštancie, musí volať predvídacie alebo konfliktom riadené procedúry (tieto procedúry sa budú ďalej nazývať riešiace inštrukcie) až kým nenájde nastavenie premenných, pri ktorých je problém riešiteľný, resp. neusúdi nesplniteľnosť. Každé volanie týchto funkcií je nasledované testom na to, či sa našiel výsledok. Ak áno, program končí a vracia hodnotu 0 pri splniteľnej formule. Pri pri nesplniteľnej formule vracia -1.

Keďže sa počíta s podporou oboch riešiacich inštrukcií, treba pri ich vykonávaní robiť pomocné výpočty pre obe. V tomto prípade to konkrétne znamená nutnosť aktualizovať implikačný graf aj po volaní predvídacich procedúr.

Rozdiel medzi procedúrami spočíva v rozdielnom správaní pri narazení na konflikt. Ak sa narazí na konflikt počas volania predvídacej procedúry, začne sa odvolávať nastavenie hodnôt premenných v opačnom poradí ako boli priradené. Toto prebieha, kým sa nenájde premenná, u ktorej ešte neboli vyskúšané obidve možné hodnoty. Hodnota tejto premennej sa invertuje a volanie procedúry končí. Ak sa na konflikt narazí počas konfliktom riadenej procedúry, vytvorí sa nová klauzula a odvolajú sa premenné podobne ako sme si ukázali v podkapitole 4.4.2. Podrobnejší popis je v podkapitole 8.4.2

8.1.2 Použiteľné heuristiky

Obidva prístupy budú môcť používať niekoľko heuristík na výber premennej a hodnoty, ktorá sa jej priradí (viď podkapitola 4.5.1).

Veľkosť preselect množiny bude nastavená pevne na 10% všetkých nenastavených premenných.

Ako rozdielové heuristiky sú použiteľné CRH, WBH a BSH.

Nastavená hodnota môže byť zvolená týmito spôsobmi:

- Hodnota, ktorá sa vyskytuje vo formule častejšie
- Hodnota, ktorá sa vyskytuje častejšie v najkratších klauzulách
- Kladná hodnota (bez ďalších analýz)

Toto znamená 9 funkcií pre obe riešiace inštrukcie (3 rozdielové heuristiky a 3 heuristiky na výber hodnoty premennej). Okrem toho je ešte možné nastaviť náhodnej premennej náhodnú hodnotu, čím sa dostávame na 10 funkcií.

Aby bolo možné počítať výskyty niektorých javov, môžu sa zaviesť pomocné premenné, nijak nesúvisiace so vstupnou klauzulou. Na základe nich sa môžu vykonávať podmienené skoky. Tieto premenné podporujú 5 operácií: resetovanie (nastavenie na 1), inkrementácia o jedna, zdvojnásobenie hodnoty, inkrementácia o nejakú hodnotu, násobenie nejakou hodnotou. Hodnota pomocných premenných je obmedzená na maximálnu hodnotu 65535. V prípade pretečenia sa hodnota pomocnej premennej delí modulo maximálnou hodnotou. Hodnoty 1 ako resetovacia hodnota je zvolená, aby hodnota pomocnej premennej neuviazla na nule, ak by sa len násobilo. Napriek tomuto je síce možné, že sa po niekoľkých násobeniach priradí hodnota 0 kvôli modulo operácii, ale pravdepodobnosť je pomerne malá. Všetky pomocné premenné majú počiatočnú hodnotu 1.

8.1.3 Skoky

Základom LGP je možnosť použitia skokov. Okrem nepodmieneného skoku je možné vykonávať podmienené skoky na základe niekoľkých parametrov: na základe hodnoty pomocnej premennej, na základe hustoty (viď podkapitola 5.2), s náhodou pravdepodobnosťou. Tieto skoky majú tvar

```
if (podmienka)
{
  skok na návestie
}
```

V ich tele sa teda nevykonáva žiadna iná inštrukcia. Skok na základe hustoty predstavuje snahu podmieniť vykonanie nejakej činnosti štatistickým údajom z aktuálne spracovávaného vstupu.

Podmienené skoky obsahujú porovnanie s nejakými hodnotami. Pri porovnávaní hodnôt premenných, zoberie sa len hodnota uložená v géne a prevedie sa do fenotypu (tj. maximálna hodnota na porovnanie je 65535). Pri porovnaní hustoty sa zoberie hodnota z génu a vydelí sa 65535, čím sa získa desatinne číslo v rozsahu 0 až 100.

8.1.4 Ďalšie pomocné funkcie

Ďalšie dostupné pomocné funkcie sú

- Reštartovanie vyhľadávania
- Zrušenie odvodených klauzúl podľa doby neaktivity
- Prázdna inštrukcia

Reštartovanie sme si vysvetlili v predošlých kapitolách (viď podkapitola 6). Možnosť zrušenia odvodených klauzúl vychádza z podkapitoly 4.4.10. Je možné zrušiť len tie klauzuly, ktoré nie sú vstupné, alebo vytvorené ich zjednodušením. Minimálna hodnota neaktivity, ktorá sa vkladá ako parameter, sa berie hodnota z génu. Rozsah tohto parametru je teda 0 až 65535.

8.2 Genetické programovanie

Používa sa variant LGP. Mutácia je jediný použitý genetický operátor. Používa sa $1 + \lambda$ prístup, pričom sa berie do úvahy neutralita. Kandidátnym programom sa bude označovať výstup genetického programovania, tj. súbor obsahujúci zdrojový kód v C++.

Nastaviteľné parametre sú: počet použitých pomocných premenných, počet inštrukcií v chromozóme, počet zmenených inštrukcií (tj. nie len počet zmenených génov, ale koľkokrát sa reálny výstup zmení) v procesu mutácie a maximálne použiteľný čas na riešenie. Fitness funkciou je súčet časov strávených pri riešení jednotlivých testovacích inštancií.

8.2.1 Fitness

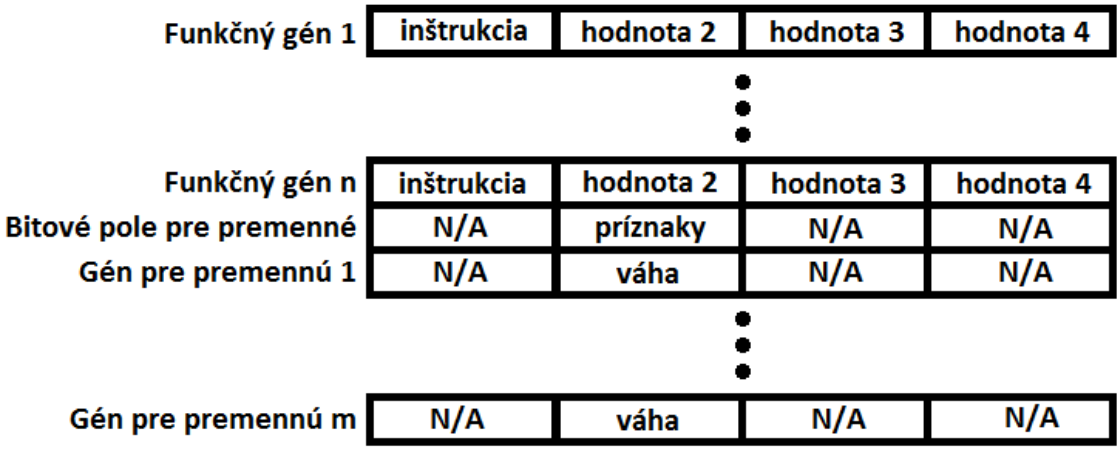
Ak sa nenájde riešenie do uplynutia maximálneho povoleného časového intervalu, ukončí sa daný beh s penalizáciou za neúspech. Hodnotenie prebieha nasledovným spôsobom: prelož aktuálny kandidátny program a postupne spúšťaj preložený program tak, že na jeho

vstupoch budú jednotlivé testovacie inštancie. Sleduj či nebol prekročený časový limit. Ak bol, ukonči aktuálne testovanú inštanciu. Pre kontrolu vypisuje každý dokončený beh návratovú hodnotu, ktorou je výsledok riešenia. Riešiaci program vracia 0 pri riešiteľnej inštancii, hodnotu -1 pri neriešiteľnej inštancii. Ak vyprší časový limit, návratová hodnota sa ignoruje.

Ukážeme si štruktúru genotypu, spôsob mutácie a jej prevod na genotyp.

8.2.2 Genotyp

Genotyp je uložený vo vektore, v ktorom je každý gén je reprezentovaný poľom štyroch hodnôt (viď obrázok 8.1). Tento vektor sa skladá z dvoch typov génov.



Obrázek 8.1: Štruktúra genotypu

Pre jednoduchšiu predstavu ako bude vypadať výstupný kandidátny program – fenotyp, viď tabulka 8.1

Gény označené ako funkčné gény, sú gény, ktoré sa prevádzajú na výstupný kód. Zvyšné slúžia ako pomocné gény pre pomocné premenné, ktoré sa používajú len počas premeny genotypu na fenotyp. Prvý gén pre pomocné premenné je bitové pole príznakov. Ostatné slúžia jednotlivým pomocným premenným. Ich využitie je v podkapitole 8.2.4. Funkčné gény môžu využívať od jedného do štyroch hodnôt, pričom prvá je vždy kód funkcie. U pomocných génov sa využíva vždy len jedna hodnota (nepoužité sú označené N/A).

Na začiatku evolučného procesu sa genotyp, ktorého veľkosť je určená vstupnými parametrami (maximálny počet riadkov výstupu a maximálny počet pomocných premenných), naplní náhodnými hodnotami v rozsahu 0-65535. Tieto hodnoty treba pred prevodom na fenotyp spracovať tak, aby dávali validný výstup. Aby sa mohlo vždy bezpečne mutovať, obsahuje každý jedinec dva genotypy. Jeden nespracovaný, ktorý sa mutuje, a druhý, ktorý obsahuje už spracovaný genotyp. Ukážeme si ako sa spracováva surový genotyp na kandidátny program. Všetky tieto úpravy sa dejú na druhom spomínanom genotype, do ktorého sa prekopíruje celý prvý genotyp.

8.2.3 Obmedzenie rozsahov do platných hodnôt

Prvým krokom je obmedziť kódy funkcií do platného rozsahu, keďže funkcií je menej ako 65535. To sa dosiahne delením modulo prvej hodnoty počtom funkcií v každom géne určenom pre funkcie (tj. gény pre pomocné premenné sú nedotknuté).

```
label_0:
        RestartSearch();
label_1:
        goto label_16;
label_15:
        var_1 = (var_1 * 8474) % 65536;
label_16:
        RemoveInferredInactivity( 29136);
label_17:
        if (var_1 >21458)
        {
        goto label_15;
        }
label_18:
        Lookahead_LitProduct_ShortestClauseFreq();
label_19:
        ;
goto label_0;
```

Tabulka 8.1: Príklad fenotypu

Nasleduje obmedzenie cieľových adries skokov delením cieľovej adresy modulo maximálny počet riadkov výstupu. Keď vieme adresy skokov, vieme určiť, ktoré gény sú aktívne v danom genotype. Táto informácia slúži na určenie génov, ktoré má zmysel mutovať, aby bola nenulová pravdepodobnosť zmeny fenotypu, tj. zamedzí sa tomu, aby sa upravovali gény, ktoré po prevedení na fenotyp nikdy nebudú dosiahnuteľné. Okrem toho sa týmto zjednoduší výstup, keďže bude kratší.

8.2.4 Prevod génov pomocných premenných

Pred ďalším spracovaním je nutné previesť hodnoty pomocných premenných zo svojej surovej hodnoty na reálne. Prvý z génov pre pomocné premenné je bitové pole príznakov. V ňom platí, že nastavený n-tý bit znamená možnosť výskytu n-tej pomocnej premennej na výstupe. Pre každý n-tý nastavený bit sa zoberie hodnota u génu pre n-tú pomocnú premennú (ďalej váha pomocnej premennej) a tieto hodnoty sa sčítajú. Na základe tohto sa vytvoria intervaly pre jednotlivé pomocné premenné nasledujúcim spôsobom: U každej pomocnej premennej, ktorej hodnota sa pridala do súčtu, sa jeho váha vydelí súčtom váh a vynásobí maximom (tj. 65535). Takto sa rozdelí rozsah hodnôt na niekoľko častí, kde veľkosť jednotlivých častí je určený pomerom váh pomocných premenných. Potom sa celý interval hodnôt rozdelí na neprekrývajúce sa úseky, ktorých veľkosti sa zhodujú s veľkosťami častí získaných v predošlom kroku. Týmto sa dosiahne toho, že sa celý rozsah hodnôt dá previesť na pomocnú premennú.

8.2.5 Nedostupné gény

Genotyp môže obsahovať redundantné gény, ktoré sú pri ďalšej práci nepodstatné. Preto sa vytvorí graf dostupných návestí z nultého návestia. Pri nepodmienených skokoch sa pridáva

len adresa cieľu skoku, pri podmienených skokoch sa k tomu ešte pridáva adresa nasledujúceho návestia. U ostatných funkcií sa pridáva len nasledujúce návestie. Po vytvorení grafu sa skontroluje, ktoré adresy sa v ňom nenachádzajú. Adresa, ktorá sa v grafe nenachádza, je označená príznakom. Tento znamená, že sa gén na príslušnej adrese nemá používať.

8.2.6 Zjednodušenie výstupu

Ak by sa takto spracovaný genotyp previedol na výstupný zdrojový kód, obsahoval by veľa zbytočných akcií, ktoré by nemali žiadny vplyv na výsledok. Toto má tri nevýhody. Prvou je skutočnosť, že by výsledný solver vykonával niektoré činnosti zbytočne, čím by sa znižoval výkon. Druhým, vážnejším nedostatkom by bola možnosť existencie cyklu, ktorý by nevykonával žiadnu činnosť, ktorá by viedla bližšie k nájdeniu výsledku inštancie - cyklus by mohol byť prázdny, alebo len neustále inkrementovať hodnotu pomocnej premennej, a podobne. Tretí nedostatok sa vzťahuje na mutáciu. Keďže je ohodnocovanie fitnessu inštancie SAT solveru časovo náročné, treba sa snažiť o to, aby sa evolúciou získavali vhodné jedince. Pri terajšej práci je však možné, že sa vyvinie jedinec, ktorý sa od rodiča líši len rozdielnou úpravou hodnoty pomocnej premennej, ktorá sa ale nikde nečíta a podobne. Z tohto dôvodu treba vykonať ešte posledné úpravy pred prevedením na výstupný zdrojový kód.

Zbytočné skoky

Ako prvé sa odstránia skoky, ktoré medzi sebou a cieľom neobsahujú žiadne neprázdne inštrukcie. Toto zahŕňa aj tie, ktorých cieľová adresa obsahuje samé seba, prípadne okamžite nasledujúce návestie. Je zrejmé, že tieto skoky nemajú význam a pri nepodmienených skokoch by sa program mohol dostať do nekonečného cyklu, v rámci ktorého by sa nevykonávala žiadna riešiaca inštrukcia.

Nedostupné pomocné premenné

Ďalšiou úpravou je odstránenie práce s nedostupnými pomocnými premennými. Podstatou je rozdelenie inštrukcií pracujúcich s pomocnými premennými na zápis a čítanie. Zápisom sa myslí akákoľvek úprava hodnoty pomocnej premennej. Čítaním sa myslí podmienený skok pracujúci s hodnotou nejakej pomocnej premennej. Pre každú pomocnú premennú sa vytvorí zoznam zápisov, čo je vlastne zoznam návestí, na ktorých sa upravuje dotyčná pomocná premenná. Nasledovne sa sledujú všetky možné cesty v programe z každého zápisu a zaznamenáva sa, či vedie aspoň jedna cesta do čítania hodnoty danej pomocnej premennej. Po vyskúšaní všetkých možných ciest sa nahradia prázdnou inštrukciou čítania také, ku ktorým nie je možné sa dostať zo žiadneho zápisu. Takisto sa nahradia prázdnou inštrukciou zápisy, z ktorých nevedie žiadna možná cesta k čítaniu. Týmto sa odstránia inštrukcie nad pomocnými premennými, ktoré nemajú význam.

Neužitočné cykly

Už spomínaný problém s možnými cyklami sa rieši podobne ako odstránenie nedostupných pomocných premenných. Spraví sa zoznam volaní riešiacich inštrukcií v aktuálne rozpracovanom genotype. Okrem toho sa vytvorí otočený zoznam dostupných adries z každej adresy. Otočený zoznam znamená, že sa ukladajú adresy, z ktorých je dostupná daná adresa. Pomocou týchto dvoch zoznamov sa vytvorí strom dostupnosti, na základe ktorého sa určí, z ktorých inštrukcií je možné sa dostať k aspoň jednému volaniu riešiacej inštrukcie.

Inštrukcie, ktoré sa v tomto strome nenachádzajú, sa nahradia prázdnou inštrukciou. Odstraňovanie neúčinných cyklov a nedostupných pomocných premenných sa cyklicky strieda, až kým nenastane prípad, že ani jedno z dvoch volaní nevedie ku zmene kódu.

Výsledkom týchto úprav je očistený pseudokód, ktorý sa môže preložiť na kandidátny program v C++.

8.3 Mutácia

Mutácia spočíva v inverzii náhodného bitu v jednom z aktívnych génov. V týchto génoch je možné invertovať bit len na hodnotách, ktoré sa pri danej inštrukcii využívajú pre prevode na fenotyp. Ak sa mutáciou zmení niektorý z génov patriaci pomocným premenným, skontroluje sa, či sa niekde v kóde zmenila pomocná premenná, s ktorou sa pracuje. V prípade, že nie, mutuje sa znovu.

Po mutácii sa vykoná spracovanie pseudokódu ako sme si ukázali v predošlej podkapitole. Tento spracovaný pseudokód sa porovná s predošlým výstupným spracovaným pseudokódom. Ak sa vyhodnotia ako rozdielne, mutácia končí. V opačnom prípade sa opakuje celá mutačná procedúra. V tomto prípade síce je možnosť zacyklenia sa, ale pravdepodobnosť je veľmi malá. Na to, aby vznikol nekonečný cyklus by sa museli neustále invertovať hodnoty na rovnakom bite. Keďže je rozloženie pseudonáhodného generátoru C++ lineárne, je prakticky nemožné, aby táto situácia nastala.

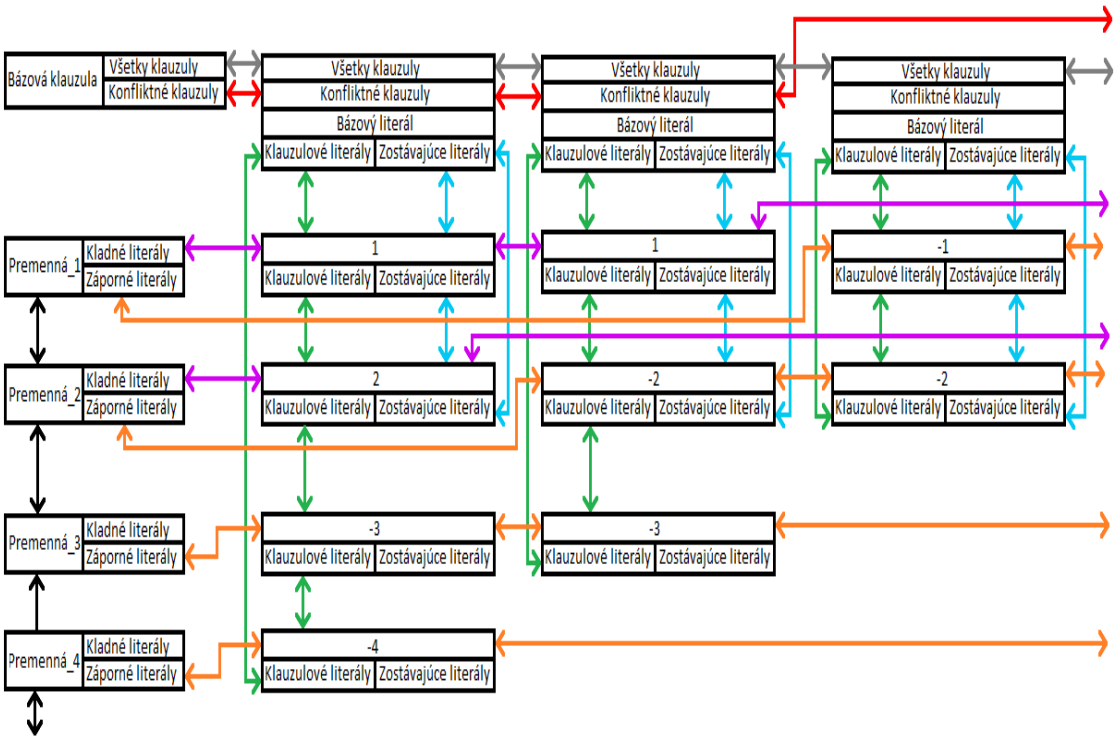
8.4 Implementácia solveru

V tejto podkapitole si ukážeme podrobnejšie implementáciu SAT solveru, ktorý podporuje ako predvídací, tak aj konfliktom riadený prístup.

8.4.1 Všeobecná štruktúra

Základné prvky, s ktorými solver pracuje, sú klauzuly, literály a premenné. Väčšina dát je uchovaná v obojsmernom cyklickom zozname takým spôsobom, že každý prvok obsahuje ukazateľ na nasledujúci, resp. predchádzajúci prvok, takže netreba uchovávať dáta v separátnych zoznamoch. Cieľom takejto architektúry bolo dosiahnuť konštantnú časovú zložitosť väčšiny operácií. U klauzúl sa uchováva zoznam všetkých klauzúl, zoznam konfliktných (tj. vytvorené aby zabránili konfliktu) klauzúl. U premenných je uchovaný zoznam premenných, ktoré ešte nemajú priradenú hodnotu. Okrem toho si uchováva dva separátne zoznamy pre svoje kladné, resp. záporné literály. Kvôli implikačnému grafu vie každá premenná, ktorú premennú implikuje a ktorou je implikovaná. Tieto údaje sa však uchovávajú v zvláštnom zozname. Literály obsahujú zoznamy literálov s rovnakou hodnotou, zoznam všetkých ostatných literálov v klauzule, a zoznam literálov v klauzule bez priradenej hodnoty (grafická schéma viď obrázok 8.2)). Úlohy týchto zoznamov si ukážeme neskôr.

Všetky zoznamy majú bázový prvok, ktorý neobsahuje žiadne užitočné dáta. V prípade prázdneho zoznamu, ukazuje ukazateľ na predošlý prvok, ako aj ukazateľ na nasledujúci prvok, na rovnakú adresu, ktorou je samotný bázový prvok. Vďaka tomuto netreba pri vkladaní na akúkoľvek pozíciu testovať na koniec zoznamu, keďže vždy existuje nasledujúci aj predchádzajúci prvok. Okrem toho je možné pomerne jednoducho obnoviť stav zoznamu, ak z neho niekoľko prvkov vyjmeme. Vyňatie zo zoznamu funguje tak, že sa zoberie nasledujúci prvok a predchádzajúci prvok a u týchto prvkov sa upraví ukazateľ na predchádzajúci prvok, resp nasledujúci prvok tak, že vynímaný prvok obchádzajú. Vo vynímanom prvku



Obrázek 8.2: Zjednodušená schéma štruktúry SAT solveru

sa však ukazatele zachovajú. Ak vyjmeme zo zoznamu niekoľko prvkov a chceme ich vrátiť, stačí ich vložiť v opačnom poradí, ako sa vyňali. Toto prebehne tak, že prvok nastaví u svojho pôvodného nasledovníka a predchodcu, aby naň znova ukazovali. Kvôli zachovaniu konzistencie zoznamu nie je možné vracať vyňaté prvky iným spôsobom.

Keďže sa počíta s častým vytváraním a mazaním klauzúl a ich príslušných literálov, používa sa pre ne štruktúra typu resource pool, do ktorej sa odkladajú nepoužívané prvky a z ktorých sa vyberajú nové prvky. Takto sa zabraňuje zbytočným alokáciám a mazaniam objektov, ktoré sa budú dať ešte využiť.

8.4.2 Práca SAT solveru

Prvým krokom práce solveru je načítanie vstupných dát – klauzúl a ich literálov. Tieto sa ukladajú do už spomínaných zoznamov. Po načítaní nasleduje zjednodušovanie klauzúl.

Zjednodušenie klauzúl

Okrem odstránenia duplicít je možné upraviť vstup ďalšími metódami. Jedna z intuitívnejších je nájdenie subsumovaných klauzúl, tj. klauzúl, ktorých podmnožina sa nachádza vo formule. Ďalšie z možných zjednodušení dvoch klauzúl na jednu je možné za nasledujúcich podmienok:

- Klauzuly majú rovnaký počet literálov
- Ak majú klauzuly n literálov, $n - 1$ z nich je identických
- n-tý literál je v jednej klauzule l, v druhej $\neg l$, kde l je ľubovoľný literál

Za týchto okolností je možné odstrániť tieto dve klauzuly z formuly a nahradiť ich klauzulou, ktorá obsahuje len literály, ktoré sú spoločné. Pracuje to na základe booleovskej susednosti.

Táto skutočnosť vyplýva zo skutočnosti:
Majme klauzuly

$$(x_1 \vee x_2 \vee \cdots \vee x_{n-1} \vee l)$$

a

$$(x_1 \vee x_2 \vee \cdots \vee x_{n-1} \vee \neg l)$$

Tieto sa dajú prepísať na klauzulu

$$(x_1 \vee x_2 \vee \cdots \vee x_{n-1}) \wedge (l \vee \neg l)$$

Druhá klauzula je vždy pravdivá, takže sa z výrazu môže odstrániť a zostane

$$(x_1 \vee x_2 \vee \cdots \vee x_{n-1}) \quad \square$$

Ďalšie zjednodušenie je možné za týchto podmienok:

- Klauzuly majú rozdielny počet literálov
- Ak má menšia klauzula n literálov, musí obsahovať väčšia množinu $n-1$ z nich
- Pre literál menšej klauzuly nezahrnutý vo väčšej musí platiť, že ak je rovný l, musí väčšia klauzula obsahovať $\neg l$, kde l je ľubovoľný literál

V tomto prípade je možné zjednodušiť väčšiu klauzulu odstránením literálu, ktorý má opačnú hodnotu.

Majme klauzuly

$$(l \vee x_2 \vee \cdots \vee x_n \vee y_1 \cdots \vee y_m)$$

a

$$(\neg l \vee x_2 \vee \cdots \vee x_n)$$

Použitím zákona distributivity vzniknú disjunkcie konjunkcií dvoch literálov, kde prvý literál je z prvej klauzuly, druhý z druhej. Dôležité konjunkcie, ktoré vzniknú sú

$$(l \wedge \neg l) \vee (x_2 \wedge x_2) \vee \cdots \vee (x_n \wedge x_n) \vee (y_1 \wedge \neg l) \vee \cdots \vee (y_m \wedge \neg l)$$

Z týchto klauzúl je prvá vždy nepravdivá.
Konjunkcie

$$(x_2 \wedge x_2) \vee \cdots \vee (x_n \wedge x_n)$$

sa dajú zjednodušiť na

$$x_2 \vee \cdots \vee x_n$$

Tieto jednotkové klauzuly môžu úplne nahradiť dvojice, ktoré obsahujú aspoň jeden z týchto literálov. Po zjednodušení zostávajú klauzuly

$$x_2 \vee \cdots \vee x_n \vee (y_1 \wedge \neg l) \vee \cdots \vee (y_m \wedge \neg l)$$

$$x_2 \vee \cdots \vee x_n \vee ((y_1 \vee \cdots \vee y_m) \wedge \neg l)$$

$$(x_2 \vee \cdots \vee x_n \vee y_1 \vee \cdots \vee y_m) \wedge (\neg l \vee x_2 \cdots \vee x_n) \quad \square$$

Jedná sa o zovšeobecnenie situácie keď existuje jednotková klauzula a zároveň existujú klauzuly, v ktorých sa nachádza literál opačnej hodnoty.

Tieto zjednodušenia sa skúšajú na každej dvojici klauzúl, vrátane tých, ktoré týmto zjednodušovaním vzniknú. Toto sa deje po načítaní vstupu ako aj pri vytváraní konfliktnej klauzuly. Z toho vyplýva, že táto činnosť môže byť časovo náročná, ak je prítomných veľa klauzúl.

Práca s premennými

Pri riešení SAT problému sú hlavné operácie nastavovanie hodnoty premennej a odvolanie tohto nastavenia. Pri nastavení premennej sa odstránia všetky literály opačnej hodnoty z klauzúl odstránením literálu zo zoznamu aktuálnych literálov klauzuly. Následne sa prejde celý zoznam literálov s nastavovanou hodnotou, a okrem práve nastavovaného literálu sa odstránia všetky ostatné z celkového zoznamu literálov. Takto bude vo výsledku dostupná klauzula, v ktorej je premenná nastavená, avšak jej literály budú neviditeľné pri nastavovaní ďalších premenných. Aktuálne nastavovaný literál nie je potrebné zrušiť zo zoznamu literálov, keďže sa nikdy už nebude pristupovať k premennej, ktorá aktuálny literál nastavila – premenná môže mať len jednu hodnotu.

Keďže sa pri nastavovaní premennej rušia literály z klauzúl, treba počítať s tým, že môžu vzniknúť nové jednotkové klauzuly. Preto je potrebné pri každom odstránení literálu z klauzuly sledovať, či náhodou nevznikla jednotková klauzula. Takto vzniknuté klauzuly sa ukladajú do zoznamu, ktorý sa spracuje po tom, čo je odstránený každý literál s opačnou hodnotou ako má aktuálne nastavovaná premenná. Po zrušení každého potrebného literálu sa vyberie prvá jednotková klauzula zo zoznamu. Z nej sa vyberie zostávajúci literál a nastaví sa príslušnej premennej hodnota tohto literálu. Tento proces sa opakuje kým nie je zoznam jednotkových klauzúl prázdny. Počas spracovávania týchto klauzúl môžu vznikať neustále nové jednotkové klauzuly.

Implikačný graf

Počas spracovávania jednotkových klauzúl treba aktualizovať implikačný graf. Vždy, keď sa začne spracovávať jednotková klauzula, treba určiť nastavenie ktorých premenných túto klauzulu spôsobilo. Každá klauzula si uchováva zoznam nenastavených literálov, ale aj literálov v pôvodnej klauzule (bez nastavených premenných). Na základe týchto dvoch zoznamov sa dá vytvoriť jednoducho implikácia tak, že sa zoberú premenné z množiny všetkých literálov a následne sa v grafe vytvoria hrany do premennej zostávajúceho literálu z ostatných premenných. Tieto hrany je treba ukladať obojsmerne. Opačná hrana sa využíva pri tvorbe konfliktnej klauzuly a pri odvolávaní nastavenia premennej. Je možné, aby medzi dvoma premennými bolo viac rovnakých hrán, keďže rovnaká implikácia môže vzniknúť z viacerých klauzúl. Nikdy však nemôže byt hrana opačná, lebo to by znamenalo, že implikovaná premenná implikuje implikujúcu premennú. Toto nie je možné preto, lebo implikujúca premenná bola nastavená skôr, takže implikácia musí vychádzať z nej.

Okrem zavádzania samotných implikácií si každá zvolená premenná ukladá zoznam premenných, ktoré boli implikované jednotkovou klauzulou na danej implikačnej úrovni. Je potrebný pri odvolávaní nastavenia premennej, aby sa rýchlo vedelo určiť, ktoré premenné sa môžu odvolať spoločne s danou zvolenou premennou. Vychádza to zo skutočnosti, že pred zavedením danej premennej neexistovala ani jedna z týchto implikovaných premenných, lebo každá jednotková klauzula vzniknutá ako dôsledok nastavenia premennej, obsahovala dovtedy minimálne 2 literály, teda nič neimplikovali. Takto je možné bezpečne odvolávať ľubovoľný počet implikačných úrovní v opačnom poradí, v akom boli vytvorené. Premenné nachádzajúce sa v tomto zozname odstránia posledný prvok z implikačného zoznamu každého prvku zo zoznamu implikujúcich premenných (k tomuto účelu sa použije opačná implikačná hrana). Nezáleží, či aktuálne odstraňovaná implikácia ukazuje na premennú, ktorá ukazateľ zo zoznamu odstraňuje. Odstraňuje sa totiž každá implikácia z danej implikačnej úrovne, takže sa musí odstrániť rovnaký počet implikácií ako bolo vytvorených.

Konfliktné klauzuly

Počas nastavovaní hodnoty premennej môžu vzniknúť aj prázdne klauzuly značiace konflikt (viď podkapitola 4.4.4). Pre tieto klauzuly sa zavedie zvláštny vrchol v implikačnom grafe. Je implikovaný každou pôvodnou premennou tej klauzuly, ktorá sa stala prázdnou. Ak tento konflikt vznikne počas volania konfliktnej procedúry, treba vytvoriť konfliktnú klauzulu. Na jej vytvorenie treba nájsť prvý UIP. Tento sa hľadá na podmnožine implikačného grafu. Tento graf obsahuje každý uzol, ktorý je implikovaný na aktuálne implikačnej úrovni. Z tohto grafu sa vytvorí strom dominátorov pomocou Lengauer-Tarjan algoritmu([22]), kde počiatočným bodom je zvolená premenná aktuálnej implikačnej úrovne. Po nájdení UIP sa vyhľadá každá zvolená premenná, ktorá vedie do konfliktnej klauzuly a nie je z nej je dostupný UIP uzol. K tomuto účelu sa použijú otočené implikácie. Z týchto premenných, spolu s UIP uzlom, vytvorí nová klauzula.

Nová klauzula sa pridá do zoznamu existujúcich klauzúl. Túto klauzulu je možné neskôr odstrániť, keďže vznikla odvodením zo vstupu. Takto vytvorené klauzuly však môžu zmenšiť niektoré už z existujúcich. Ak sa vytvára nová klauzula zjednodušením dvoch klauzúl, z ktorých je aspoň jedna vstupná, bude aj nová pokladaná za vstupnú. Takisto, ak sa vytvorí klauzula, ktorá je podmnožinou vstupnej klauzuly, bude aj táto pokladaná za vstupnú. Tieto nové klauzuly sa pridávajú na koniec zoznamu klauzúl a nastavuje sa im príznak. Tento poslúži na určenie klauzúl, v ktorých treba hľadať úroveň, do ktorej treba odnastaviť premenné.

Po redukcii klauzúl sa zoberú všetky klauzuly, ktoré majú nastavený príznak spomenutý v predošlom odseku. Z každej sa vyberie 3. najvyššia implikačná úroveň literálov. Z nich sa vyberie najnižšia. Následne sa zrušia nastavené hodnoty premenným až na túto implikačnú úroveň. Tento rozdiel v porovnaní s literatúrou (viď podkapitola 4.4.2) má ten význam, že zabraňuje vzniku klauzúl, ktoré implikujú premennú, keďže každá z nových klauzúl má aspoň 2 nenastavené literály (s výnimkou klauzúl, ktoré majú len jeden literál).

Po zrušení nastavených premenných sa z klauzúl s nastaveným príznakom vyberú jednotkové a premenné sa nastavia podľa hodnôt ich literálov.

Následne sa na novovytvorené klauzuly zavolá nastavenie všetkých aktuálne nastavených premenných, aby každá klauzula vedela, ktorý literál ešte nemá nastavený.

Kvôli urýchleniu výpočtov váh premenných pre výberové heuristiky sa ich váhy nastavujú pri každej práci s literálmi.

8.4.3 Zoznam funkcií

Keďže pri evolúcii vznikajú programy s rôznym obsahom, uvedieme zoznam funkcií, ktoré sa môžu nachádzať v kandidátnom programe a ich význam.
GetCurrentDensityRatio - vráti hustotu aktuálnej formuly (tj. ignoruje literály s nastavenou premennou)
GetRandom - vracia náhodonú hodnotu od 0.0 do 1.0
RemoveInferredInactivity(int) - odstráni konfliktné klauzuly podľa neaktivity. Vstupný parameter určuje maximálnu dobu neaktivity. Dobou neaktivity sa rozumie počet nastavených premenných odkedy vytvárala klauzula jednotkovú klauzulu. Odstrániť sa môžu len klauzuly, ktoré aktuálne neimplikujú.
RestartSearch - reštartovanie vyhľadávania
DPLL_RandomLitPick - vykonanie konfliktom riadeného kroku s náhodným výberom premennej a jej hodnoty
Lookahead_RandomLitPick - vykonanie predvídacieho kroku s náhodným výberom premen-

nej a jej hodnoty

Ďalšie funkcie majú tvar DPLL_HEUR1_HEUR2, resp. Lookahead_HEUR1_HEUR2, kde HEUR1 značí rozdielovú heuristiku, HEUR2 smerovú heuristiku.

HEUR1 môže mať nasledovné hodnoty:
LitCount - CRH
LitSum - WBH
LitProduct - BSH

HEUR2 môže mať hodnoty:
PickPositive - výber kladnej hodnoty bez heurisitky
ShortestClauseFreq - výber hodnoty, ktorá sa vyskytuje najčastejšie v najkratších klauzulách
TotalFreq - výber najčastejšej hodnoty vo formule

Prázdna inštrukcia je reprezentovaná bodkočiarkou.

Kapitola 9

Experimenty

Táto kapitola zhŕňa experimenty vykonané s vytvorenými programami pre riešenie SAT problému s využitím získaných heuristík pomocou LGP.

9.1 Nastavenie LGP

Veľkosť populácie bola nastavená na $1 + 4$, z dôvodu testovania a trénovania na počítačoch s 4 jadrovými procesormi.

Maximálny počet povolených inštrukcií v kandidátnom programe bol 4 alebo 20.

Počet mutovaných génov bol 1 u kandidátnych programov s maximálne 4 inštrukciami. U kandidátnych programov s maximálne 20 inštrukciami boli vyskúšané počty 1, 3, a 5 mutovaných génov.

Maximálny počet použitých pomocných premenných bol 1. Pri väčšom počte bolo zistené pri rôznych počiatočných pokusoch, že bol veľmi zriedkavý prípad, že sa zapisovalo do pomocnej premennej, s hodnotou ktorej sa niekde v kandidátnom programe aj pracovalo. To isté platí naopak, teda málokedy sa čítala pomocná premenná, do ktorej sa niekde v kandidátnom programe zapisovalo (jej hodnota zostávala teda konštantná).

Evolučný proces prebiehal, kým nenastal prípad, že 20 generácií po sebe nebola vylepšená hodnota fitness funkcie.

Väčšina z týchto hodnôt je založená na prvotných pokusoch s nastaveniami LGP. Preto je možné, že by sa našli vhodnejšie hodnoty pri ďalšom experimentovaní s nimi.

9.2 Trénovanie a testovanie

Pre účely trénovania a testovania bolo vytvorených niekoľko inštancií SAT problému pomocou nástroja uloženého na `www.cs.indiana.edu/cgi-pub/sabry/cnf.html`. Tento nástroj pracuje tak, že prevedie vstupné číslo na problém rozkladu na prvočísla, prevedený na inštanciu SAT problému. Ak je zadané číslo prvočíslom, SAT inštancia je neriešiteľná. V opačnom prípade je riešiteľná. Pomocou takejto inštancie sa dajú nájsť 2 čísla, ktorých násobkom je vstupné číslo, nie všetky prvočísla rozkladu. Počet premenných, literálov a klauzúl v takto získanej inštancii je určený počtom bitov zadaného čísla. Výstupy pre rôzne vstupné čísla s rovnakým počtom bitov sa líšia len literálmi (tj. pozíciou negácií premenných vo formule).

Trénovanie a testovanie prebiehalo na troch rôznych sadách inštancií.

- Sada SA: Inštancie vytvorené z 8 bitových čísiel. Tieto inštancie mali 169 premenných a 643 klauzúl
- Sada SB: Inštancie vytvorené z 12 bitových čísiel. Tieto inštancie mali 413 premenných a 1601 klauzúl
- Sada SC: Inštancie vytvorené z 16 bitových čísiel. Tieto inštancie mali 753 premenných a 2943 klauzúl

Pre každú sadu existovali 3 typy vstupov: neriešiteľné inštancie, inštancie s maximálne dvoma možnými riešeniami, inštancie s až niekoľkými desiatkami riešení. Pre SA bolo vytvorených 5 (tj. spolu 15) inštancií každého typu pre trénovanie. Pre testovanie sa použil rovnaký počet inštancií. Sady SB a SC mali 15 (tj. spolu 45) inštancií každého typu pre trénovanie ako aj pre testovanie. Prieniky trénovacích a testovacích množín boli prázdne.

Jednotlivé sady mali rôzne časové limity počas trénovania na nájdenie riešenia aktuálnej inštancie: SA malo 5 sekúnd, SB malo 10 sekúnd a SC malo 15 sekúnd. Ak sa daná inštancia nevyriešila do stanoveného limitu, ukončil sa riešiaci program násilne. Tieto časy sa určili na základe prvotných pokusov s nastaveniami LGP.

Ako sme si povedali, LGP pracoval s niekoľkými nastaveniami: maximálne 4 inštrukcie kandidátneho programu a 1 mutovaný gén, maximálne 20 inštrukcií a 1, 3, alebo 5 mutovaných génov. To znamená 4 nastavenia. Pomocou genetického programovania sa vytvárali kandidátne programy pre všetky 3 sady inštancií pri všetkých 4 možných nastaveniach. V rámci všetkých 12 kombinácií bolo spustených 20 behov evolučného algoritmu. Z týchto behov sa potom testovala vždy najlepšia získaná heuristika uložená v kandidátnom programe.

Testovanie prebiehalo tak, že sa kandidátny program spustil 5 krát na každú jednu inštanciu testovacej množiny. Kandidátne programy vytrénované na sade SA boli teda testované na 75 spusteniach. Tie, ktoré sa vytrénovali na sadách SB a SC, na 225 spusteniach. Viacnásobné spúšťanie bolo z dôvodu náhodnosti v riešení – každé spustenie malo inú hodnotu semienka pre pseudonáhodný generátor.

Maximálne časové limity boli pre každú testovanú inštanciu nastavené na 5 sekúnd. Ak sa riešenie nenašlo do vypršania tohto časového limitu, ukončilo sa riešenie násilne.

9.3 Výsledky

Grafy v tejto kapitole ukazujú výsledky pre testovaciu sadu. Každý graf popisuje výsledky pri jednom nastavení LGP trénovanú na jednej sade. Každý stĺpec v grafe patrí k výsledkom jedného kandidátneho programu. Čas spotrebovaný na riešenie jednotlivých inštancií je na y-ovej osi. Spotrebovaný čas je meraný v milisekundách. Každé znamienko + v grafe znamená, že sa pri danom kandidátnom programe vyskytla inštancia, ktorej riešenie trvalo daný čas. Každé spustenie na jednom prvku testovacej množiny je vložené do grafu. Ak sa sa vyskytla inštancia, pre ktorú sa nenašlo riešenie v danom časovom limite, zanesená hodnota do grafu bola totožná s časovým limitom.

Okrem toho sú v grafoch vyjadrené priemerné dĺžky spracovania jednotlivých typov testovacích inštancií a celkový priemerný čas riešenia. 1SAT označuje inštancie s maximálne 2 riešeniami, MultSAT inštancie s viacerými riešeniami, UnSAT nesplniteľné inštancie, Avg celkovú priemernú časovú náročnosť všetkých inštancií. Tieto hodnoty sú medzi jednotlivými výsledkami kandidátnych programov spojené len pre názornejšiu ukážku rozdielov medzi výsledkami. Pri testoch vykonávaných na sadách SB a SC, sú vytvorené dva grafy.

Prvý z nich ukazuje celkové výsledky meraní. Druhý graf ukazuje rovnaké hodnoty, ale s menším rozsahom y-ovej osy, aby bolo lepšie vidno rozdiely pri nižších hodnotách času.

Vo všeobecnosti sú relatívne časové náročnosti jednotlivých inštancií podľa očakávaní – jednoduchšie inštancie sa riešia rýchlo, inštancie s malým počtom riešení pomalšie, neriešiteľné najpomalšie. Ak má inštancia viac riešení, má väčšiu šancu nájsť spĺňajúce priradenie hodnôt. U neriešiteľných inštancií treba dokázať neriešiteľnosť vytvorením dostatočného množstva konfliktných klauzúl, alebo vyskúšaní všetkých možných hodnôt.

Z grafov vidno, že niektorým heuristikám sa málokedy, resp. nikdy nepodarilo nájsť riešenie. Skutočnosť, že tieto heuristiky však boli vyhlásené za najlepšie počas trénovania môže byť spôsobená náhodnosťou: v jednom behu mohli dosiahnuť veľmi dobré výsledky, ktoré už žiadna iná heuristika nedokázala opakovať. Keby sa však pustil trénovací proces znovu, zrejme by boli veľmi skoro nahradené inou heuristikou. Keďže sa fitness funkcia ale vyhodnocovala pri každej heuristike len raz, bolo možné aby nastali tieto prípady.

Teoreticky by sa takýmto prípadom dalo zabrániť lepším spracovaním genotypu, pri ktorom by sa komplexnejšie analyzoval možný priebeh kódu.

9.3.1 Sada SA

Pri sade SA vidno, že každá heuristika bola podobne účinná. Rozdiely sú natoľko malé, že sa nedá určiť jednoznačný víťaz. Keďže sú heuristiky testované zakaždým s iným semienkom pre pseudonáhodný generátor, vždy bude medzi jednotlivými behmi nejaká fluktuácia. O trochu vyššie časy pri 20 inštrukciách a 1 resp. 3 mutovaných génoch (obrázky 9.2, 9.3) sú pravdepodobne spôsobené tým, že testovacie výpočty prebiehali na 2 strojoch s podobnou, ale nie rovnakou konfiguráciou.

Zaujímavé je to, že neexistuje jeden prístup, ktorý by jednoznačne bol najlepší. Každá vytvorená heuristika je pestrá zmes všetkých možných inštrukcií. Nutno podotknúť, že tieto inštancie sú pomerne jednoduché, čo je demonštrované skutočnosťou, že heuristika v tabuľke 9.1 je účinná pre tieto inštancie. Na väčšie inštancie by bola nepoužiteľná, lebo vychádza z toho, že dokáže nájsť konflikt po nastavení jedinej premennej, keďže sa hneď volá reštart vyhľadávania.

Zaujímavé sú 2 heuristiky, ktoré boli evolučným algoritmom vyhlásené za najlepšie, ale neboli schopné vyriešiť ani jednu testovaciu inštanciu. Prvá z nich, heuristika v tabuľke 9.2 zlyhala z dôvodu, že po každom vyskúšaní premennej sa reštartovalo hľadanie. Keďže nevytvára klauzule a pri výbere premennej a jej hodnoty vychádza z počtu literálov v formule, zruší každý reštart aktuálny stav a následne sa hneď nastaví tá istá premenná. Výsledkom je v podstate prázdny cyklus.

Druhá neúspešná heuristika obsahovala úsek (viď tabuľka 9.3), na ktorý sa dostala po nastavení niekoľkých premenných. Teda tiež zlyhala kvôli prázdnemu cyklu.

9.3.2 Sada SB

U tejto skupiny vstupov jednoznačne vyhrali heuristiky skladajúce sa z maximálne 4 inštrukcií (obrázok 9.5). Každá z týchto heuristík sa skladala len z konfliktom riadených procedúr. Ani jedna z nich však nevykonávala úplne náhodný výber. Pri dodatočných pokusoch nebola ani jedna heuristika, skladajúca sa čisto z predvídacích procedúr, schopná nájsť riešenie do vypršania časového limitu.

Pri tejto skupine bol väčší počet inštrukcií jednoznačne nevýhodný (obrázky 9.7, 9.9, 9.11) . Jednak kvôli horším časom a jednak kvôli skutočnosti, že všetky heuristiky zložené z 4 inštrukcií boli schopné nájsť riešenie. U väčších heuristík táto záruka nebola.

```
label_0:
        goto label_2;
label_2:
        RestartSearch();
label_3:
        DPLL_RandomLitPick();
goto label_0;
```

Tabulka 9.1: Príklad účinnej heuristiky pre jednoduché vstupy, ktorá je nepoužiteľná pre väčšie vstupy

```
label_0:
        RestartSearch();
label_1:
        goto label_16;
label_15:
        var_1 = (var_1 * 8474) % 65536;
label_16:
        RemoveInferredInactivity( 29136);
label_17:
        if (var_1 >21458)
        {
        goto label_15;
        }
label_18:
        Lookahead_LitProduct_ShortestClauseFreq();
label_19:
        ;
goto label_0;
```

Tabulka 9.2: Jedna z neúspešných heuristík pre sadu SA

```
label_5:
        ;
label_6:
        if (var_1 >10424)
        {
        goto label_5;
        }
```

Tabulka 9.3: Časť heuristiky spôsobujúca nevyriešenie ani jedného prvku testovacej množiny

Najmenej úspešná heuristika v tabuľke 9.4. Táto heuristika má podobný problém ako heuristika v tabuľke 9.2. Po niekoľkých predvídacích krokoch sa vykonáva reštart.

Úspešnejšie heuristiky, s maximálne 20 inštrukciami, sa skladali z rôzne poprekladaných predvídacích a konfliktom riadených procedúr.

```
label_0:
        RestartSearch();
label_1:
        goto label_11;
label_11:
        Lookahead_LitSum_ShortestClauseFreq();
label_12:
        Lookahead_LitCount_TotalFreq();
label_13:
        Lookahead_LitSum_ShortestClauseFreq();
label_14:
        ;
label_15:
        RemoveInferredInactivity( 18447);
label_16:
        if (GetRandom() >0.351959)
        {
        goto label_1;
        }
label_17:
        ;
label_18:
        Lookahead_LitCount_TotalFreq();
label_19:
        DPLL_LitSum_PickPositive();
goto label_0;
```

Tabulka 9.4: Neúspešná heuristika pre sadu SB

9.3.3 Sada SC

Táto sada predstavuje skupinu, ktorá bola časovo najnáročnejšia ako na trénovanie, tak aj na testovanie. Prvým zaujímavým faktom je to, že každá jedna z heuristík bola schopná nájsť riešenie. Toto je spôsobené je tým, že inštancie sú už pomerne náročné. Vďaka tomu bola bola veľmi malá pravdepodobnosť, že by evolučný proces skončil heuristikou, ktorá by bola schopná riešiť akurát niekoľko trénovacích prvkov a nič iné.

Zaujímavosťou je to, že všetky skupiny heuristík z tejto sady majú podobné časové zložitosti. Pri bližšom pohľade je táto skutočnosť jednoducho vysvetlená:

1. Heuristiky s 4 inštrukciami sa znovu skladali výhradne z konfliktom riadených procedúr.

2. Heuristiky s 20 inštrukciami síce obsahovali aj predvídacie procedúry, ale nevyužívali ich tak intenzívne, ako v predošlých sadách. Buď sa po krátkom čase dostali do cyklu, ktorý obsahoval výhradne konfliktom riadené procedúry, alebo aspoň konfliktom riadené procedúry silno preferovali.

Toto demonštruje heuristika v tabuľke 9.5. Táto predstavuje najslabšiu heuristiku pre sadu SC pri 20 inštrukciách a 1 mutovanom géne. Hoci musí len s asi 11% pravdepodobnosťou aplikovať 5 predvídacích procedúr po sebe, už to celkom viditeľne znižuje výkon.

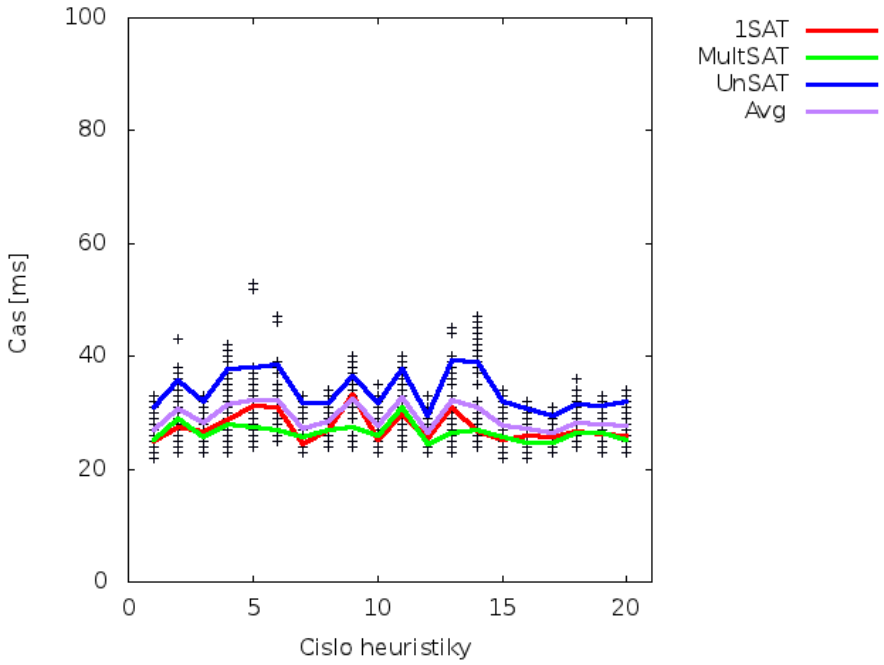
Jedna z najlepších heuristík (viď tabuľka 9.6) obsahuje skoro samé konfliktom riadené procedúry. Evidentne tá jedna predvídacia procedúry nemá negatívny vplyv na účinnosť.

```
label_0:
        RemoveInferredInactivity( 23314);
label_1:
        DPLL_LitProduct_TotalFreq();
label_2:
        if (GetRandom() >0.110703)
        {
        goto label_15;
        }
label_3:
        var_1 = (var_1 + 1) % 65536;
label_4:
        Lookahead_LitProduct_TotalFreq();
label_5:
        Lookahead_LitProduct_TotalFreq();
label_6:
        Lookahead_LitProduct_ShortestClauseFreq();
label_7:
        if (var_1 <15363)
        {
        goto label_16;
        }
label_8:
        Lookahead_LitSum_PickPositive();
label_9:
        Lookahead_LitCount_ShortestClauseFreq();
label_10:
        DPLL_LitSum_ShortestClauseFreq();
label_11:
        DPLL_LitSum_TotalFreq();
label_12:
        DPLL_RandomLitPick();
label_13:
        var_1 = (var_1 * 2) % 65536;
label_14:
        Lookahead_LitProduct_ShortestClauseFreq();
label_15:
        var_1 = (var_1 * 28527) % 65536;
label_16:
        var_1 = (var_1 * 31388) % 65536;
label_17:
        DPLL_LitSum_TotalFreq();
label_18:
        DPLL_LitProduct_PickPositive();
label_19:
        var_1 = (var_1 * 26367) % 65536;
```

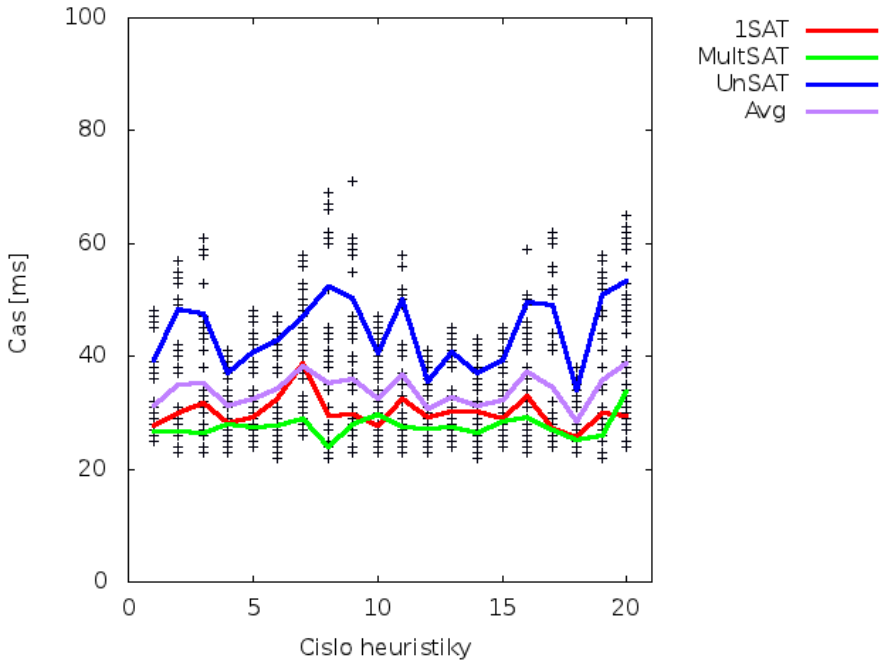
Tabulka 9.5: Najslabšia heuristika pre sadu SC s 20 inštrukciami a 1 mutovaným génom

```
label_0:
        DPLL_LitProduct_TotalFreq();
label_1:
        DPLL_LitCount_PickPositive();
label_2:
        DPLL_LitSum_ShortestClauseFreq();
label_3:
        RestartSearch();
label_4:
        if (GetCurrentDensityMaxDensityRatio() >40.7898)
        {
        goto label_0;
        }
label_5:
        DPLL_LitProduct_PickPositive();
label_6:
        DPLL_LitCount_PickPositive();
label_7:
        Lookahead_LitSum_PickPositive();
label_8:
        DPLL_LitCount_TotalFreq();
label_9:
        DPLL_LitSum_PickPositive();
label_10:
        goto label_19;
label_19:
        DPLL_RandomLitPick();
goto label_0;
```

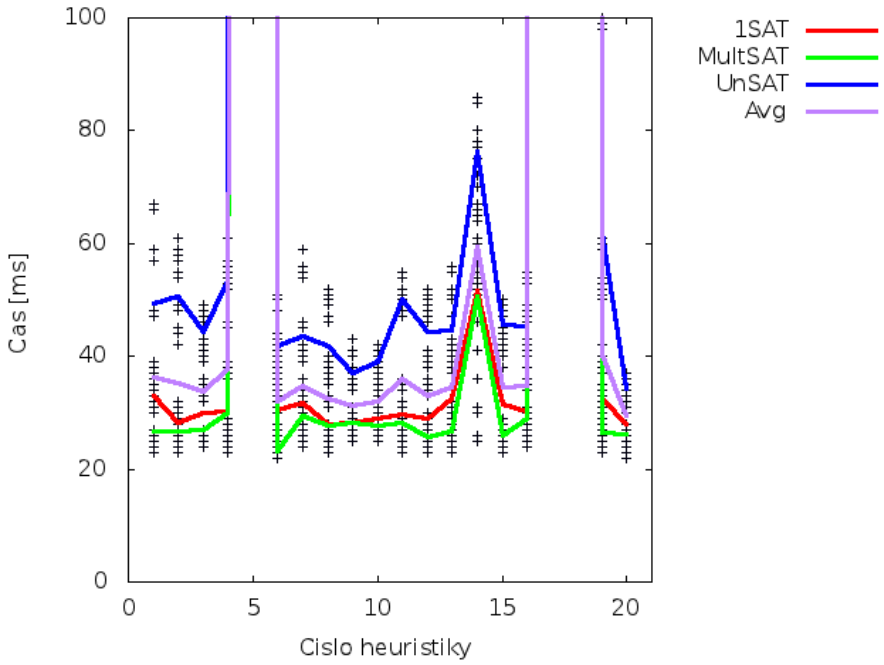
Tabulka 9.6: Jedna z najlepších heuristík pre sadu SC s 20 inštrukciami a 1 mutovaným génom



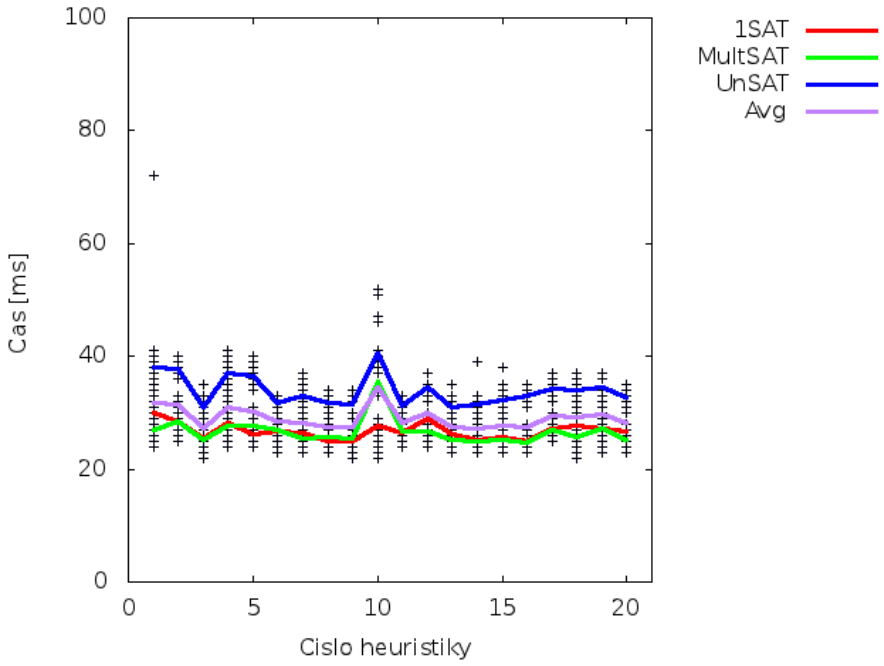
Obrázek 9.1: Sada SA, 4 inštrukcie, 1 mutovaný gén



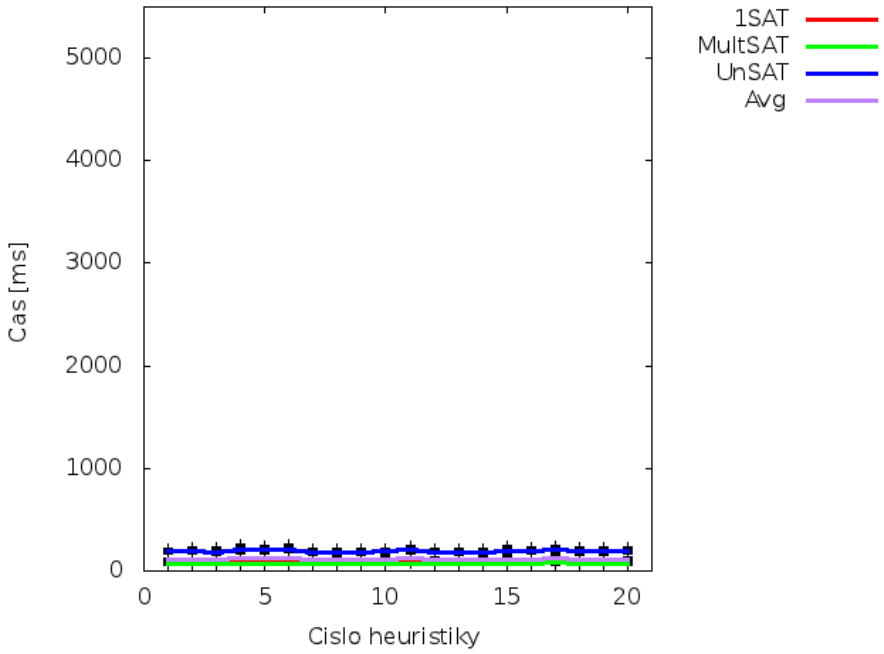
Obrázek 9.2: Sada SA, 20 inštrukcií, 1 mutovaný gén



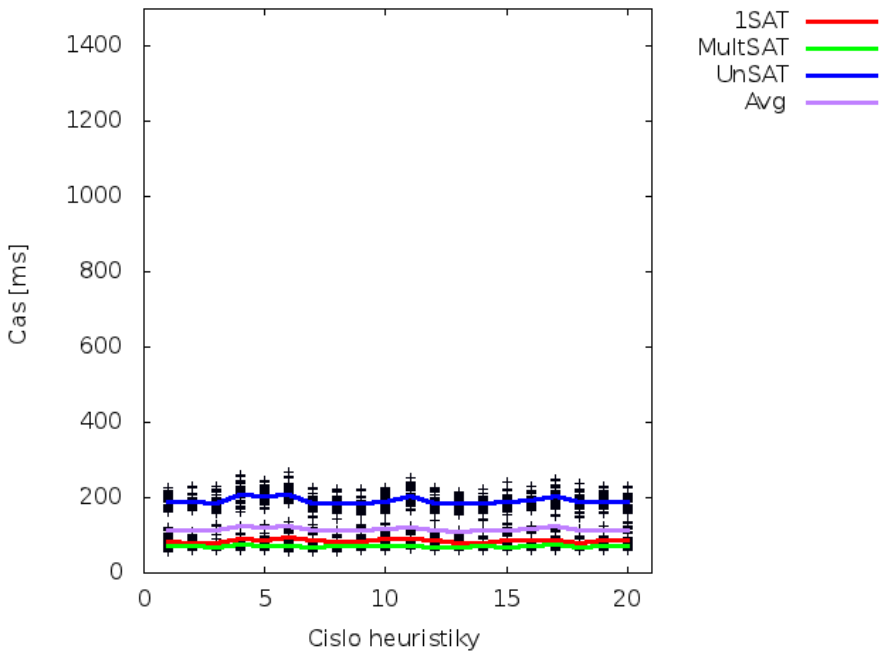
Obrázek 9.3: Sada SA, 20 inštrukcií, 3 mutované gény



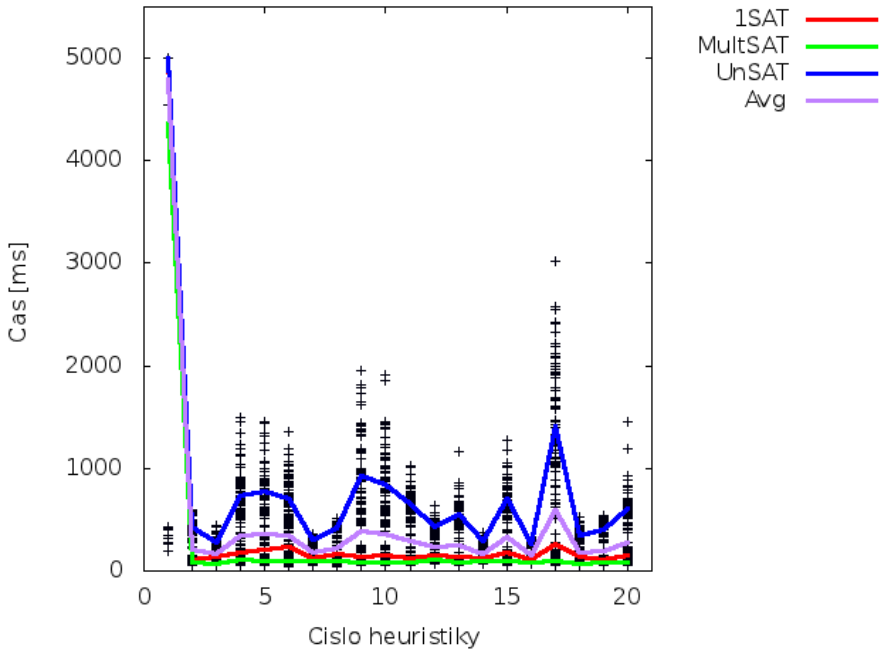
Obrázek 9.4: Sada SA, 20 inštrukcií, 5 mutovaných génov



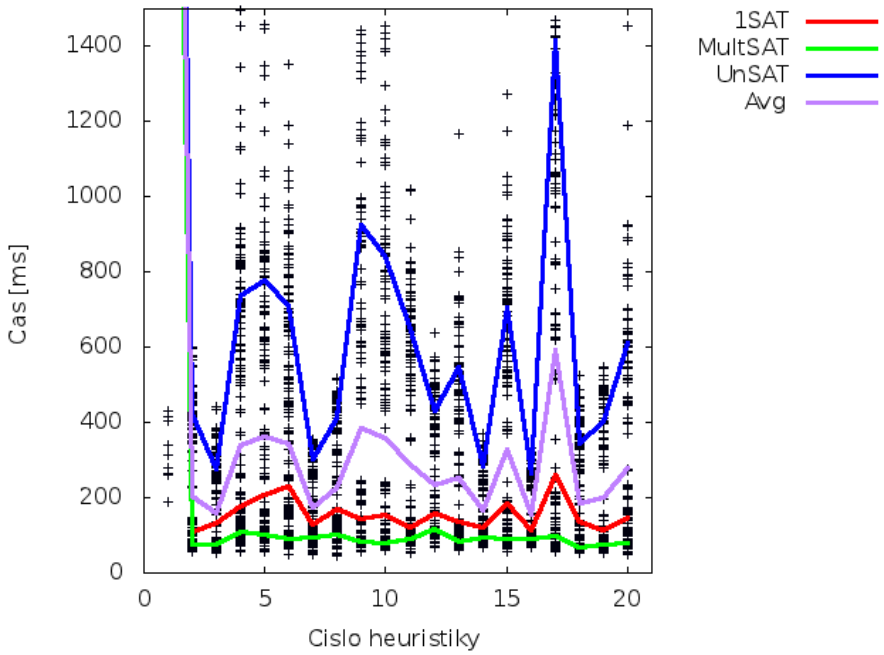
Obrázek 9.5: Sada SB, 4 inštrukcie, 1 mutovaný gén



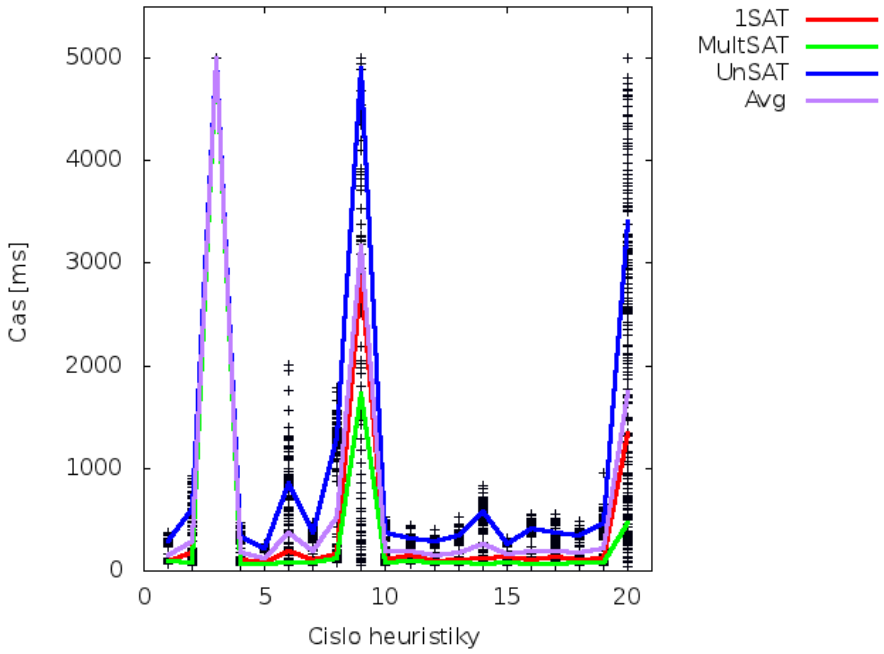
Obrázek 9.6: Sada SB, 4 inštrukcie, 1 mutovaný gén



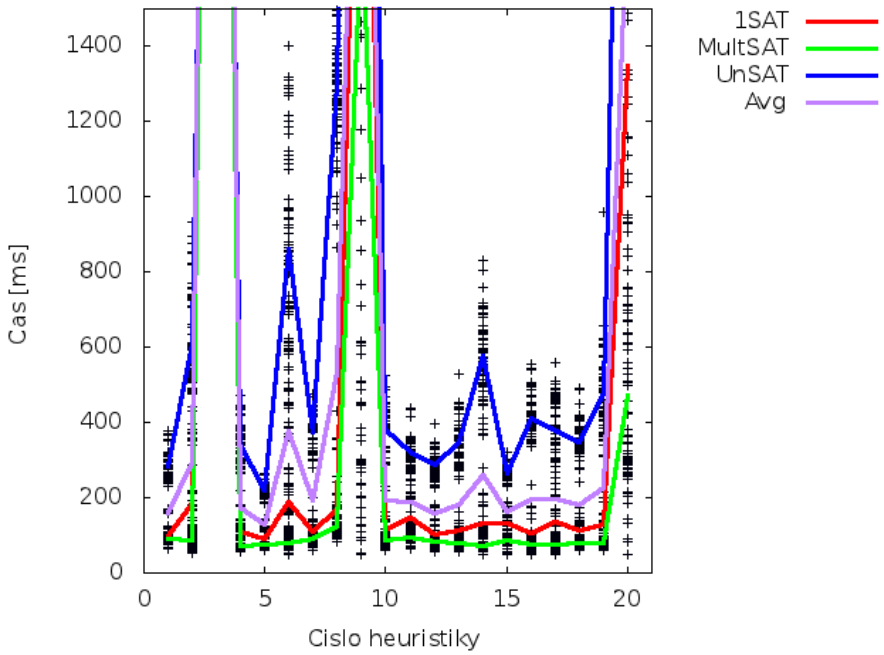
Obrázek 9.7: Sada SB, 20 inštrukcií, 1 mutovaný gén



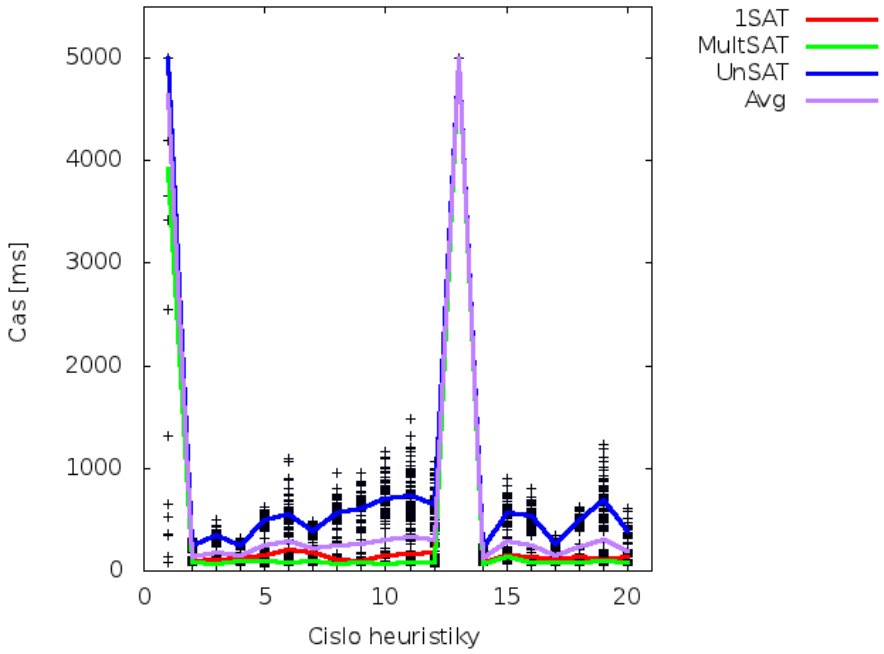
Obrázek 9.8: Sada SB, 20 inštrukcií, 1 mutovaný gén



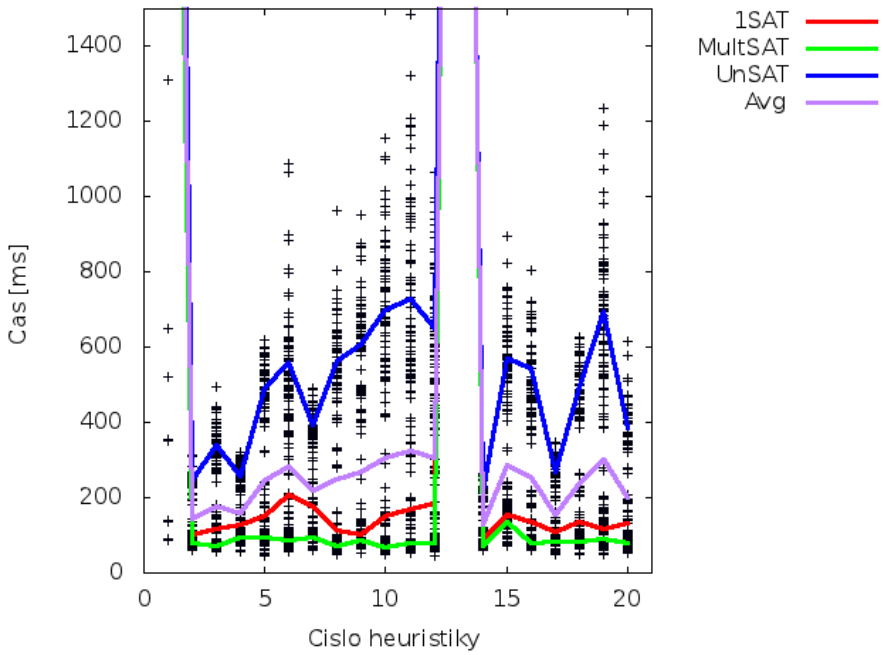
Obrázek 9.9: Sada SB, 20 inštrukcií, 3 mutované gény



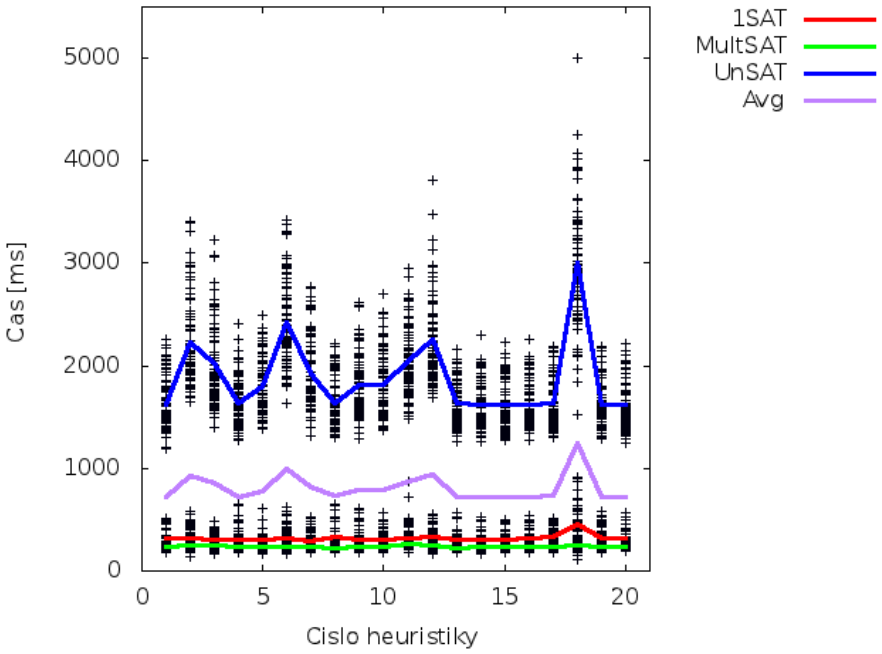
Obrázek 9.10: Sada SB, 20 inštrukcií, 3 mutované gény



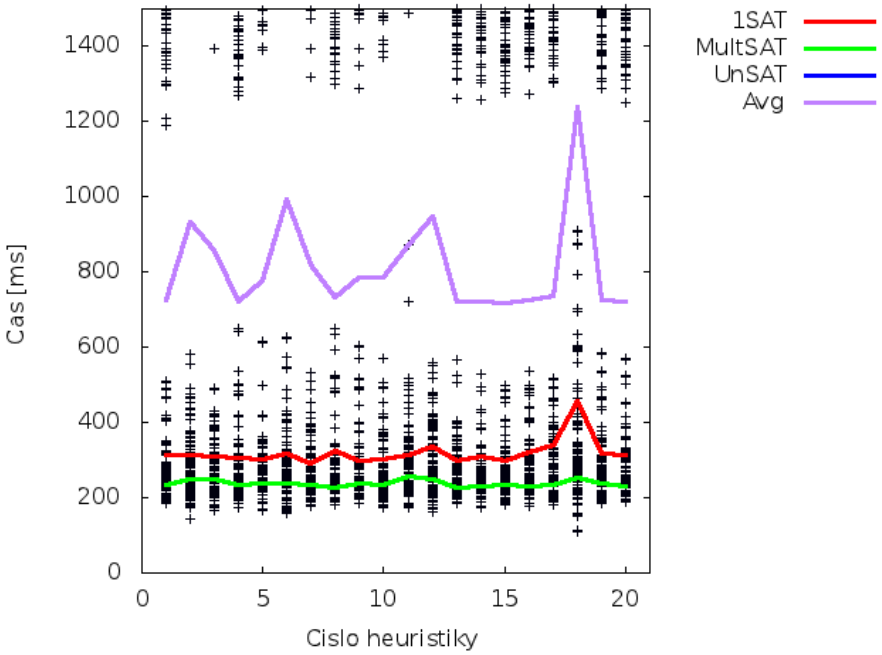
Obrázek 9.11: Sada SB, 20 inštrukcií, 5 mutovaných génov



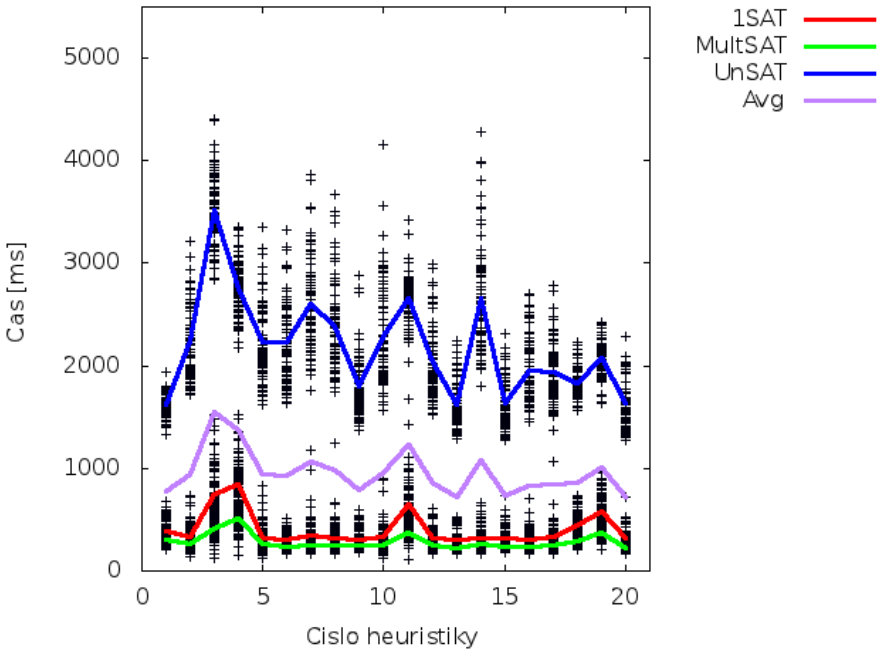
Obrázek 9.12: Sada SB, 20 inštrukcií, 5 mutovaných génov



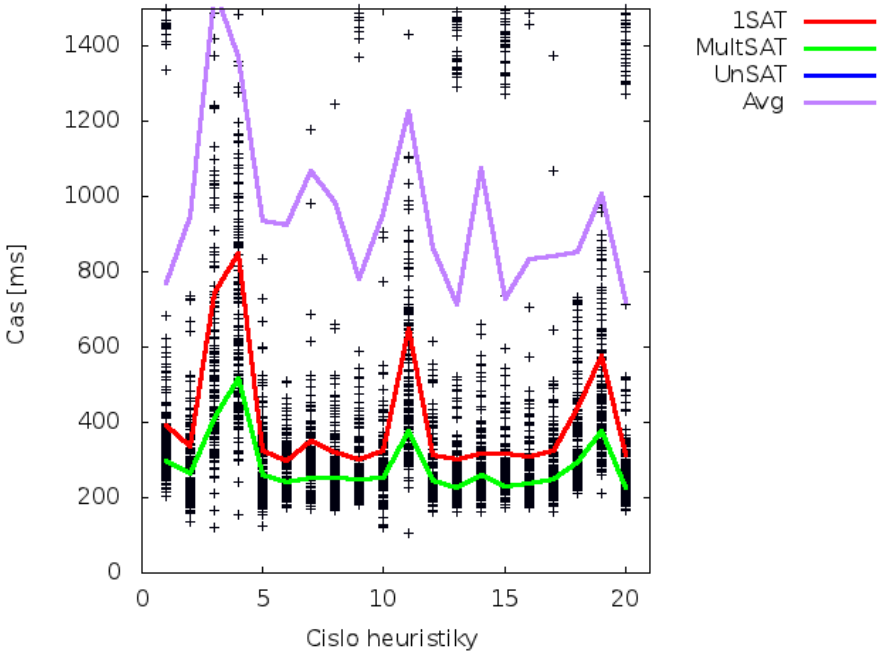
Obrázek 9.13: Sada SC, 4 inštrukcie, 1 mutovaný gén



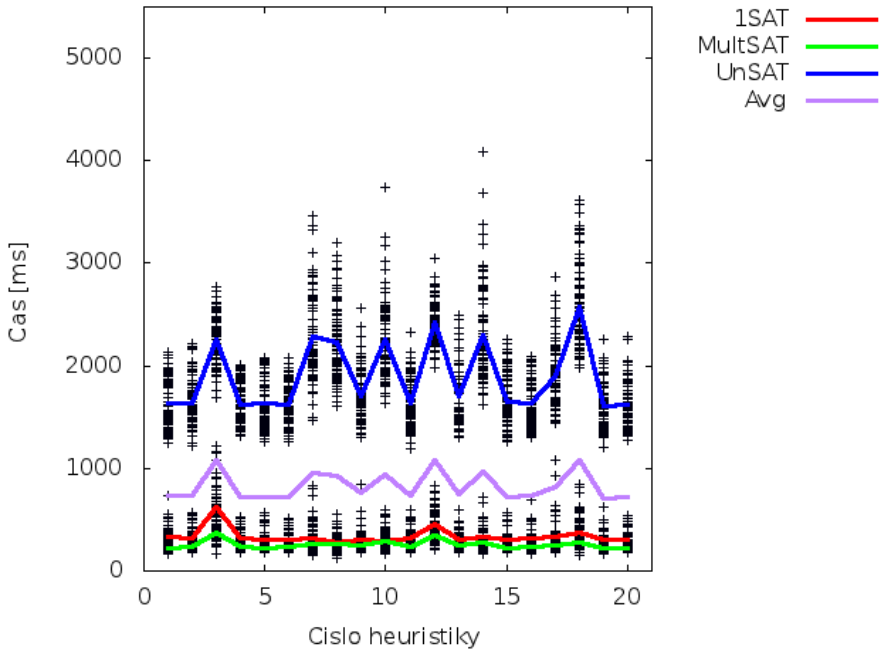
Obrázek 9.14: Sada SC, 4 inštrukcie, 1 mutovaný gén



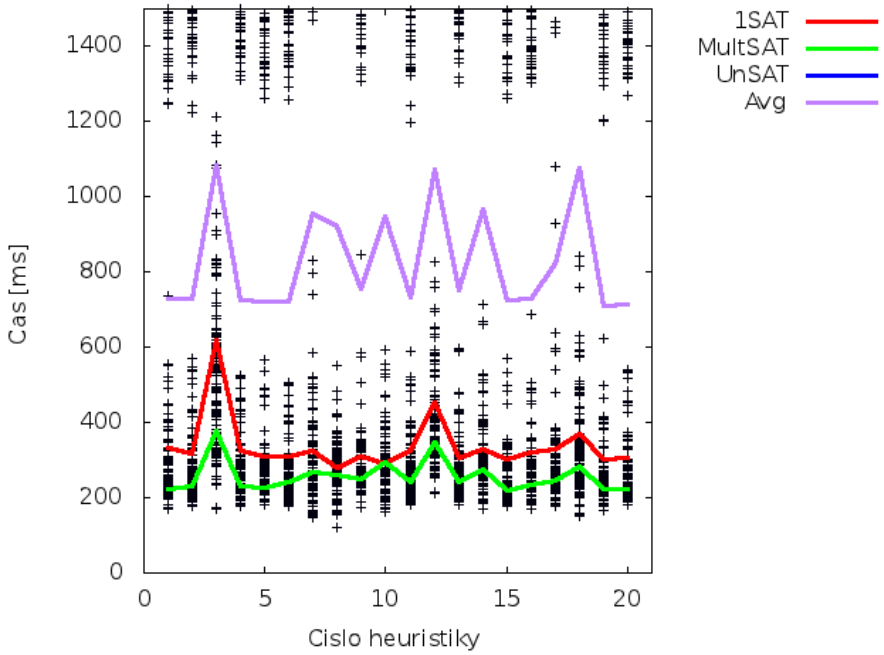
Obrázek 9.15: Sada SC, 20 inštrukcií, 1 mutovaný gén



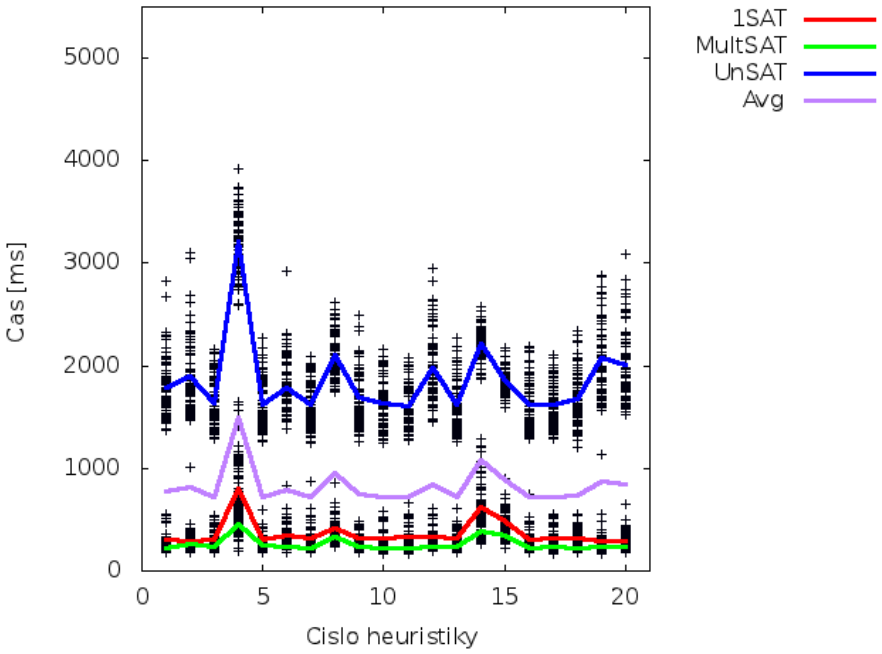
Obrázek 9.16: Sada SC, 20 inštrukcií, 1 mutovaný gén



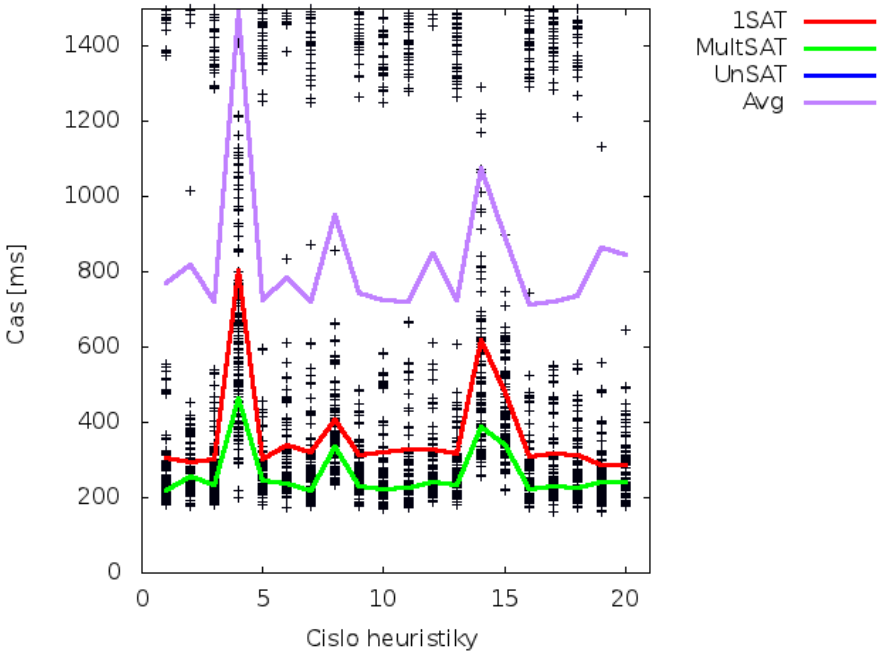
Obrázek 9.17: Sada SC, 20 inštrukcií, 3 mutované gény



Obrázek 9.18: Sada SC, 20 inštrukcií, 3 mutované gény



Obrázek 9.19: Sada SC, 20 inštrukcií, 5 mutovaných génov



Obrázek 9.20: Sada SC, 20 inštrukcií, 5 mutovaných génov

Kapitola 10

Záver

Diplomová práca popisuje princípy konštrukcie SAT solveru, ktorý využíva LGP pre nájdenie vhodnej heuristiky. V práci sú porovnané výkony heuristík pre riešenie SAT problému, ktoré boli vytvorené pomocou lineárneho genetického programovania. Bolo vyskúšaných niekoľko trénovacích a testovacích sád s rôznymi zložitosťami, ako aj rôzne parametre LGP. V rámci testovania sa ukázalo niekoľko nedokonalostí analýz kódu kandidátnych programov.

Vo výsledných heuristikách bola jasne viditeľná preferencia konfliktom riadených procedúr nad predvídacími procedúrami s nárastom zložitosti trénovacej sady. Táto preferencia by sa s najvyššou pravdepodobnosťou len zväčšovala pri ďalšom náraste zložitosti trénovacej sady. To znamená, že možnosť obmedziť prehľadávaný stavový priestor je dôležitejšia ako možnosť rýchlej analýzy jeho menších častí. Pri malých inštanciách bol však stavový priestor tak malý, že predvídacie procedúry boli rovnako úspešné ako konfliktom riadené procedúry.

Logickým možným rozšírením do budúcnosti by mohlo byť zavedenie neúplných algoritmov do heuristík, ako aj ďalšie metódy vytvárania klauzúl. Do úvahy pripadá možnosť vyskúšať iné typy SAT inštancií na testovanie a trénovanie, keďže sa v rámci tejto práce riešia len inštancie vzniknuté z rozkladov čísel na prvočísla.

V terajšom stave nie je možné použiť evolučný algoritmus na väčšie inštancie ako tie, s ktorými sa pracovalo v rámci diplomovej práce. Toto by sa mohlo vylepšiť dôkladnejšími analýzami kandidátneho programu, aby sa zamedzilo niektorým nedostatkom uvedených v analýze výsledkov.

Momentálne sa v kandidátnych programoch vytvorených LGP preferuje konfliktom riadený prístup. Ak by sa urýchlili výpočty váh premenných pre heuristiky, vylepšila by sa pomerná rýchlosť predvídacích algoritmov a spolu s tým, pravdepodobne, aj ich účinnosť na zložitejšie problémy.

Heuristika by mohla byť začlenená aj do existujúcich výkonnejších solverov, po vykonaní časovo náročnej analýzy a prípadnej vhodnej úpravy zdojových kódov na výber vhodnej formy výstupu genetického programovania.

Porovnanie výkonnosti heuristík získaných pomocou LGP s existujúcimi SAT solvermi nebolo vykonané z dôvodu niekoľkorádoveho rozdielu vo výkone – priemerné časy riešenia inštancií zo sady SC (viď podkapitola 9.2) jednotlivými heuristikami získanými evolučne boli 100 až 1000 krát pomalšie ako riešenie pomocou solveru zChaff. Tento solver bol už niekoľkokrát prekonaný vo výkone, takže porovnanie s lepšími solvermi by zrejme ešte viac prehĺbilo rozdiely vo výkone. Vidno jednoznačnú výhodu špecializácie sa na konkrétny spôsob riešenia. Tento rozdiel vo výkone bol spôsobený SAT solverom vytvorenom v rámci diplomovej práce.

Porovnanie s inými ručne vytvorenými heuristikami tiež nebolo vykonané. Dôvodom tohto bola skutočnosť, že 3 najlepšie evolučne získané heuristiky s maximálne 20 inštrukciami obsahovali len jednu, alebo dve konfliktom riadené inštrukcie, ktoré vykonávali. Obsahovali síce viac inštrukcií, ale na tie sa nevykoávali kvôli podmienenému skoku na základe hodnoty premennej. Hodnota tejto premennej sa však nikdy nezmenila, lebo jej hodnota sa zmenila len po návestí, na ktorý ukazovala. Skok sa však nikdy nevykonal, takže hodnota zostával konštantná, teda skok sa nikdy nevykonával Z toho vyplýva, že heuristiky na riešenie SAT problému vytvorené pomocou LGP pri terajších podmienkach parametroch nemajú výhodu nad ručne vytvorenými heuristikami obsahujúcimi len cyklické volanie jednej konfliktom riadenej procedúry. Takéto heuristiky by človek pravdepodobne skúsil ako prvé, keby skúšal spraviť ručne nejakú heuristiku.

Literatura

[1] Arora, S.; Barak, B.: Computational Complexity: A Modern Approach [online]. 2007 [cit. 2013-01-15].
URL www.cs.princeton.edu/theory/index.php/Compbook/Draft

[2] Bansal, N.; Raman, V.: Upper bounds for maxsat: Further improved. In *In 7th International Symposium on Algorithms and Computation*, ročník 1741, Springer, 1999, s. 247–258.

[3] Biere, A.; Heule, M. J. H.; van Maaren, H.; aj. (editoři): *Handbook of Satisfiability*, *Frontiers in Artificial Intelligence and Applications*, ročník 185. IOS Press, February 2009, ISBN 978-1-58603-929-5, 980 s.

[4] Brameier, M.: *On Linear Genetic Programming*. Dizertační práce, Fachbereich Informatik, Universität Dortmund, Germany, Únor 2004.
URL
https://eldorado.uni-dortmund.de/bitstream/2003/20098/2/Brameierunt.pdf

[5] Carlson, J.; Jaffe, A.; Wiles, A.: *The Millennium Prize Problems*. Providence, RI: American Mathematical Society and Clay Mathematics Institute., 2006, iSBN 978-0-8218-3679-8.

[6] Cook, S.: The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, s. 151–158.

[7] Cook, S.; Nguyen, P.: *Logical foundations of proof complexity*. Cambridge University Press, 2010, 335 s., iSBN 978-0-521-51729-4.

[8] Darwiche, A.; Pipatsrisawat, K.: *Complete Algorithms*, kapitola 3. Ročník 185 Biere aj. [3], February 2009, s. 99–130.

[9] Davis, M.; Putnam, H.: A computing procedure for quantification theory. *Journal of the ACM*, 1960: s. 7:201–215.

[10] Dechter, R.; Rish, I.: Directional resolution: The davis putnam procedure, revisited. In *Principles of Knowledge Representation (KR-94)*, 1994, s. 134–145.

[11] Dudka, K.: Rešení problému splnitelnosti výokových formúl [online]. 2008 [cit. 2013-01-07].
URL http://dudka.cz/fss/files/doc/proj_doc.pdf?action=sendPdf

[12] Franco, J.; Martin, J.: *A History of Satisfiability*, kapitola 1. Ročník 185 Biere aj. [3], February 2009, s. 3–74.

[13] Fukunaga, A. S.: Evolving Local Search Heuristics for SAT Using Genetic Programming. In *GECCO (2)*, 2004, s. 483–494.

[14] Garey, M. R.; David; Johnson, S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979, iSBN 0-7167-1045-5.

[15] Gomes, C. P.; Sabharwal, A.: *Exploiting Runtime Variation in Complete Solvers*, kapitola 9. Ročník 185 Biere aj. [3], February 2009, s. 271–288.

[16] Gramm, J.; Hirsch, E. A.; Niedermeier, R.; aj.: New worst-case upper bounds for max-2-sat with application to max-cut. *Discrete Applied Mathematics*, 2003: s. 130(2):139–155.

[17] Hertli, T.; Moser, R. A.; Scheder, D.: Improving PPSZ for 3-SAT using Crtitical Variables. *CoRR*, ročník abs/1009.4830, 2010.

[18] Heule, M. J. H.; van Maaren, H.: *Look-Ahead Based SAT Solvers*, kapitola 5. Ročník 185 Biere aj. [3], February 2009, s. 155–184.

[19] Karp, R. M.: Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, New York: Plenum, 1972, s. 85–103.

[20] Kautz, H. A.; Sabharwal, A.; Selman, B.: *Incomplete Algorithms*, kapitola 6. Ročník 185 Biere aj. [3], February 2009, s. 185–203.

[21] Kutzkov, K.; Scheder, D.: Using CSP To Improve Deterministic 3-SAT. *CoRR*, ročník abs/1007.1166, 2010.

[22] Lengauer, T.; Tarjan, R. E.: A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, ročník 1, č. 1, Leden 1979: s. 121–141, ISSN 0164-0925, doi:10.1145/357062.357071.
URL http://doi.acm.org/10.1145/357062.357071

[23] Miller, J. F.: *Evolution of Electronic Circuits*. Natural Computing Series, Springer Verlag, 2011, ISBN 978-3-642-17309-7, s. 1–33.

[24] Paturi, R.; Pudlák, P.; Zane, F.: Satisfiability coding lemma. In *38nd IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 1197, s. 566–574.

[25] Schöning, U.: A probabilistic algorithm for k-sat and constraint satisfaction problems. In *40nd IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 1999, s. 410–414.

[26] Schuurmans, D.; Southey, F.: Local search characteristics of incomplete SAT procedures. 2001: s. 132(2):121–150.

[27] Szeider, S.: Minimal unsatisfiable formulas with bounded clause variable difference are fixed-parameter tractable. In *9th International Conference on Computing and Combinatorics*, ročník 2697, Springer-Verlag, 1999, s. 548–558.

[28] Češka, M.; Vojnar, T.; Smrčka, A.: Teoretická informatika - Studijní opora[online]. 2011-09-01 [cit. 2013-01-07].
URL
https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/oporaTIN.pdf

Příloha A

Obsah CD

- Adresár *Cppsource* s podadresármi
  - *DIP* so zdrojovými súbormi evolučného algoritmu.
  - *SAT solver* so zdrojovými súbormi SAT solveru.
- Adresár *DIP_exe* so spustiteľným programom pre LGP a podadresármi
  - *SAT solver* so zdrojovými súbormi SAT solveru - pre trénovanie a testovanie.
  - *test* 15 jednoduchých inštancií na trénovanie.
  - *verify* 45 zložitých inštancií na testovanie.
- Adresár *SAT_solver_exe* so spustiteľným SAT solverom s konfliktom riadenou heuristikou.
- Adresár *Texsource* so zdrojovými súbormi tohoto dokumentu pre LaTeX.
- Tento dokument vo formáte pdf.

Příloha B

Manual

Diplomová práca bola implementovaná v prostredí VS 2012. V prípade SAT solveru by nemal byť problém s prekladom verziou gcc, ktorá podporuje range-based for. Iné časti nového štandardu sa nepoužívajú. Program pre LGP však využíva Windows API volania pre prácu s vláknami, takže spustiteľnosť pod Linuxom by bola pracnejšia (zmeny by bolo treba pravdepodobne vykonať len v *Population.cpp*).

B.1 SAT solver

SAT solver očakáva minimálne 1 parameter. Prvý parameter je cesta k súboru obsahujúci inštanciu SAT problému vo formáte DIMACS CNF (tj. bežná forma ukladania inštancií SAT problémov). Druhý parameter je voliteľný. Predstavuje počiatočné semienko pseudonáhodného generátoru. Ak nie je prítomný, nahradí sa hodnotou time(NULL). Poradie týchto parametrov je nastavený natvrdo. Príklad použitia:
SAT solver.exe C:\subor.cnf 10

Výstupom je reťazec UNSTATISFIABLE v prípade neriešiteľnej inštancie, alebo SATISFIABLE v prípade riešiteľnej. V prípade riešiteľnej sa ešte vypíše spĺňajúce priradenie hodnôt.

B.2 Evolučný algoritmus

Spustiteľný súbor vykonávajúci LGP očakáva vo svojom adresári adresár *SAT solver* so zdrojovými súbormi SAT solveru. Ak sa má vykonávať trénovanie, musí byť v adresári binárneho súboru prítomný adresár *test*. V prípade testovania sa očakáva prítomnosť adresára *verify*. Evolučný algoritmus potom čerpá z týchto adresárov všetky inštancie SAT problému, na ktorom sa má aktuálne uložený SAT solver testovať alebo trénovať.

Pred každým testovaním sa prekladá SAT solver uložený v podadresári. Následne sa spúšťa s každým jedným súborom z podadresára *verify* ako prvým parametrom pre solver. Každé skončenie behu, či už prerušené, alebo nie, vypíše názov súboru na vstupe, koľko trvalo riešenie a návratovú hodnotu. V prípade riešiteľnosti je hodnota 0, v prípade neriešiteľnej inštancie 1, v prípade vypršania časového limitu 259.

V prípade trénovania sa vytvorí nový SAT solver, ktorý sa prekladá. Vypočíta sa jeho fitness funkcia. Následne sa mutuje a vytvorí sa o 1 menej binárnych súborov ako je veľkosť populácie (tieto obsahujú rodiča po mutácii). Počíta sa s tým, že každý binárny súbor bude mať vlastné vlákno na vyhodnocovanie svojej fitness funkcie. Po skončení behu sa vytvorí

súbor s názvom *bestXXXX.txt*, kde XXXX je počet milisekúnd, koľko trval aktuálny beh. Obsah tohto súboru môže nahradiť súbbor *SAT solver.cpp* v podadresári *SAT solver*, aby bolo možné vykonať testovanie tejto heuristiky.

Pri preklade sa počíta s prítomnosťou VS 2012 a prekladom na 64 bitovom procesore. Ak nie, treba upraviť reťazec *g_buildCommand* v *Population.cpp*. Veľkosť populácie je napevno v súbore *Population.cpp* v konštante *POPSIZE*. Okrem toho je potrebné mať nastavenú cestu k *msbuild.exe* v PATH premennej.

Cesta, na ktorej sa program pracujúci s LGP nachádza, nesmie obsahovať medzery!

V prípade testovania sú požadované 2 parametre

- -t pre časový limit na riešenie v ms
- -m s hodnotou "test"

V prípade trénovania sú to

- -t pre časový limit na riešenie v ms
- -m s hodnotou "train"
- -n s počtom mutovaných génov
- -i s maximálnym počtom inštrukcií v genotype

Príklady použita:
DIP.exe -m test -t 500
DIP.exe -m train -t 500 -i 20 -n 1