Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Ťažké inštancie pre SAT solvery

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	prof. RNDr. Rastislav Kráľovič, PhD.

Bratislava, 2020
Erik Kučák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Erik Kučák
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Ťažké inštancie pre SAT solvery.
Hard instances for SAT solvers.

Anotácia: Moderné SAT solvery sú vybavené heuristikami, pomocou ktorých dokážu riešiť prekvapivo veľké inštancie typických problémov. Zároveň platí, že zložitosť v najhoršom prípade je exponenciálna. Je preto zaujímavé skúmať ťažké inštancie. Cieľom práce bude navrhnúť metódu na generovanie ťažkých inštancií špecificky pre niekoľko state-of-art SAT solverov a vyhodnotiť ju.

Vedúci: prof. RNDr. Rastislav Kráľovič, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 12.11.2019

Dátum schválenia: 25.11.2019

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent

....
vedúci práce

Abstrakt

Súčasné SAT solvery disponujú heuristikami, ktoré umožňujú efektívne rátať aj pomerne veľké inštancie. Keďže SAT je NP-úplný problém, dá sa očakávať, že pre každý solver existujú inštancie, na ktorých potrebuje exponenciálny čas. V praxi je ale ťažké nájsť inštancie, na ktorých by daný solver bežal dlho. Takéto inštancie môžu jednak slúžiť ako porovnávacie testy (benchmark) pre rôzne solvery, prípadne ich ďalšia analýza môže priniesť lepšie pochopenie štrukturálnych vlastností, ktoré spôsobujú rýchly beh heuristík na bežných problémoch. Cieľom práce je navrhnúť algoritmus, ktorý je schopný pre daný solver generovať 'ťažké' inštancie. Zároveň nám to umožní porovnať solvery z hľadiska toho, ako náročné je pre ne ťažké inštancie generovať.

Kľúčové slová: SAT solver, SAT, Genetický algoritmus

Abstract

Current SAT solvers use heuristics which provide effective solution also for big instances. When SAT is NP-complete there have to be instances with exponential computing time for all SAT solvers. In practise it is difficult to find instances with long computing time. These instances could be really helpful for comparing different SAT solvers or for another analysis in structural features which cause fast computing in heuristics for standard problems. My goal is to design the algorithm, which will for some SAT solvers say which are the hardest formulas for it. It means they have long computing time. It would help us in the case when we do not know any fast algorithms for some problem. We can transform this problem into the SAT problem and we want to know which SAT solver we should use to solve this problem in the best time.

Keywords: SAT solver, SAT, Genetic algorithm

Obsah

Úvod **1**

1 Problém SAT **2**

1.1 Potrebné definície 2
1.2 Jazyk SAT 3
1.3 Formy SAT 3
1.4 Jazyk k-SAT 4

2 SAT solvery **7**

2.1 SAT solver 7
2.2 Neúplný SAT solver 8
2.2.1 GSAT algoritmus 8
2.3 Úplný SAT solver 9
2.3.1 DP algoritmus 10
2.3.2 DPLL algoritmus 12
2.3.3 CDCL algoritmus 15

3 Rozhodovacie Heuristiky **22**

3.1 Jeroslow-Wang 22
3.2 DLIS 23
3.3 LEFV 23
3.4 VSIDS 24
3.5 BerkMin 24

4 Genetické programovanie **26**

4.1 Generovanie ťažkých inštancií 29
4.1.1 Základné vlastnosti formúl 29
4.1.2 Genetický algoritmus 33
4.1.3 Analýza SAT solvera 35
4.1.4 Inicializácia 35
4.1.5 Vyhodnotenie 35

4.1.6 Selekcia 35
4.1.7 Zmena populácie 36

5 Výskum **37**

Záver **41**

Príloha A **44**

Zoznam obrázkov

2.1 Zdrojový kód algoritmu GSAP 9
2.2 Zdrojový kód algoritmu DP[5] 11
2.3 Príklad DP algoritmu 11
2.4 Zdrojový kód algoritmu DPLL[5] 13
2.5 Príklad DPLL SAT solvera 14
2.6 Príklad implikačného grafu[5] 16
2.7 Príklad implikačného grafu s rezmi[5] 17
2.8 Zdrojový kód algoritmu CDCL[5] 19

4.1 Prehľad metaheuristík 27
4.2 Štatistiky pre k-SAT formuly 30
4.3 Štatistiky pre počet klauzúl 31
4.4 Štatistiky pre náhodné a rovnomerne náhodné formuly 33
4.5 Diagram genetického algoritmu 34

5.1 Výsledky na SAT solveri Cryptominisat bez rovnomerne rozložených premenných 37
5.2 Výsledky na SAT solveri Cryptominisat s veľkými zmenami 38
5.3 Výsledky na SAT solveri Cryptominisat 39
5.4 Výsledky na SAT solveri Glucose 39

Úvod

V roku 1971 páni Cook a Levin v prelomovej práci [7] dokázali, že každý jazyk rozpoznateľný nedeterministickým Turingovým strojom v polynomiálnom čase sa dá redukovať na SAT, čím položili základy skúmania NP-úplných problémov. Následne bola ukázaná NP-úplnosť pre mnohé ďalšie prblémy (viď. prehľad [12]). Napriek tomu, že otázka, či P = NP ostáva stále jedným z najvýznamnejších otvorených problémov v teoretickej informatike, vo všeobecnosti sa verí (viď napr. [14]), že rovnosť neplatí, a teda pre NP-úplné problémy neexistuje polynomiálny algoritmus. Napriek tomu je v praxi častokrát potrebné NP-úplné problémy riešiť a existuje množstvo prístupov, ako sa vysporiadať s nedostupnosťou polynomiálneho algoritmu, napr. randomizované algoritmy [18], aproximačné algoritmy [24], exponenciálne algoritmy [11], a pod. V praxi je najrozšírenejšsi používanie rôznych heuristík, t.j. algoritmov, ktoré nemajú žiadne garancie na čas behu, ale na typických inštanciách pracujú efektívne.

Problém SAT (spolu s niekoľkými ďalšimi, napr. ILP) patrí medzi problémy, pre ktoré bolo vyvinuté veľké množstvo heuristík (DPLL[8], GSAT [22], ...). Heuristiky v moderných SAT solveroch sú natoľko účinné, že pre praktické riešenie ťažkých problémov je častokrát efektívnejšie zapísať daný problém ako inštanciu SAT (z NP-úplnosti SAT vyplýva, že každý NP problém sa takto zapísať dá) a použiť na jej riešenie SAT solver, ako navrhovať ad-hoc heuristiku.

Každoročné porovnávanie SAT solverov [2] ukazuje, že sú schopné v praxi riešiť problémy veľkého rádu. Zároveň sa ukazuje, že nie je ľahké nájsť inštancie, na ktorých by SAT solvery bežali dlho [1], používajú sa buď náhodne generované, alebo formulácie ťažkých problémov (izomorfizmus grafov, farbenia, ...), resp. ručne vyrobené...

Preto je veľmi užitočné vedieť pre každý SAT solver analyzovať ťažké inštancie. Z toho nám potom postupným porovnávaním vyplynie, ktoré SAT solvery sú vhodnejšie na daný typ problému. Táto práca sa preto bude zaoberať analýzou rôznych SAT solverov a vyhodnoteniu, ktoré inštancie sú pre daný typ SAT solveru ťažké.

Kapitola 1

Problém SAT

V tejto kapitole si povieme o čom vlastne je problém SAT [19], a taktiež si ho poriadne zadefinujeme. Najprv ale potrebujeme definovať niektoré veci, ktoré pri probléme SAT budeme používať [9]. Samozrejme nebudeme definovať všetko. Očakáva sa, že čitateľ má aspoň základné znalosti z matematickej logiky.

1.1 Potrebné definície

Definícia 1.1. *Množina P: Nech P je neprázdna množina, jej prvky nazveme prvotné formuly. V našom prípade keď riešime problém SAT sa prvky množiny P myslia booleovské premenné.*

Definícia 1.2. *Výroková formula: Výrokové formuly jazyka L_p definujeme pomocou nasledujúcich syntaktických pravidiel:*

- *1. každá prvotná formula $p \in P$ je výroková formula*
- *2. ak sú výrazy A, B výrokové formuly, potom výrazy $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \longleftrightarrow B)$ sú výrokové formuly*
- *3. každá výroková formula vznikne konečným použitím pravidiel (1) a (2)*

Definícia 1.3. *Výroková podformula: Nech A je formula L_p výrokovej logiky. Jej podformulou je:*

- *1. ona sama; ak A je prvotná formula jazyka L_p*
- *2. ona sama; a každá podformula B, ak $A = \neg B$*
- *3. ona sama; a každá podfomula formúl B a C, ak $A = (A \wedge B)$, $A = (A \vee B)$, $A = (A \rightarrow B)$ alebo $A = (A \longleftrightarrow B)$*
- *4. žiadnych iných podformúl okrem tých, čo sú opísané v bodoch 1—3 niet*

Napríklad: Ak P = $\{x_1, x_2, x_3, x_4\}$, potom x_1, x_2, x_3, x_4 sú prvotné formuly jazyka L_p, výraz $((x_1 \wedge x_2) \longleftrightarrow (x_3 \wedge x_4))$ je formula a napríklad $(x_1 \wedge x_2)$ a $(x_3 \wedge x_4)$ sú jej podfomuly.

Definícia 1.4. *Literál: Prvotná forlmula alebo jej negácia sa nazýva literál. Napríklad:* x, $\neg x$, ...

Definícia 1.5. *Klauzula: Nech* $x_1, x_2, ..., x_n$*, kde* $n \in N$ *sú literály, potom disjunkcie, resp. konujnkcie všetkých literálov nazývame klauzula. Disjunkcie, resp. konujnkcie používame, ak ide o disjunktívnu normálnu formu, resp. konjunktívnu normálnu formu. Napríklad:* $x_1 \vee x_2 \vee ... \vee x_n = \bigvee_{i=1}^{n} x_i$, $x_1 \wedge x_2 \wedge ... \wedge x_n = \bigwedge_{i=1}^{n} x_i$

1.2 Jazyk SAT

Teraz sa pokúsime formálne zadefinovať jazyk SAT[19].

Definícia 1.6. *Jazyk SAT (Satisfiability alebo splniteľnosť) je množina všetkých formúl pre ktoré existuje aspoň 1 ohodnotenie, ktoré je pravdivé.*

$$SAT = \{\phi \ / \ \exists\ f(\phi),\ f(\phi) = true\}$$

1.3 Formy SAT

Formuly v jazyku SAT majú viacero foriem, ktoré používaju SAT solvery a v závislosti od formy potom vie SAT solver robiť veľmi zaujímavé optimalizácie za účelom zrýchlenia výpočtu. Najznámejšie formy, ktoré sa najviac používajú sú konjunktívna normálna forma (KNF) a disjunktívna normálna forma (DNF).

Definícia 1.7. *Konjunktívna normálna forma (KNF): Formula F sa nachádza v konjunktívnej normálnej forme práve vtedy, keď je v tvare:*

$$A_1 \wedge A_2 \wedge ... \wedge A_n$$

kde každá klauzula A_i *je disjunkciou aspoň 1 literálu.*

Definícia 1.8. *Disjunktívna normálna forma (DNF): Formula F sa nachádza v disjunktívnej normálnej forme práve vtedy, keď je v tvare:*

$$A_1 \vee A_2 \vee ... \vee A_n$$

kde každá klauzula A_i *je konjunkciou aspoň 1 literálu.*

My budeme používať najviac KNF formu, lebo existujú 3 prípady, ktoré sa nám môžu vyskytnúť počas výpočtu a pomocou nich môžeme robiť optimalizácie, ktoré urýchlia výpočet. Podrobne si ich vysvetlíme v časti, keď budeme opisovať DPLL algoritmus.

1.4 Jazyk k-SAT

Teraz si formálne zadefinujeme čo je k-SAT jazyk a k-SAT forma.

Definícia 1.9. *Hovoríme, že formula F je v k-SAT forme práve vtedy, keď je v KNF forme a zároveň každá klauzula obsahuje najviac k literálov.*

Definícia 1.10. *Jazyk L splniteľných formúl nazveme k-SAT, ak pre každú formulu z L platí, že je v k-SAT forme.*

Používanie formúl len v KNF forme je bez ujmy na všeobecnosti, keďže každú formulu je možné pretransformovať na ekvivalentnú KNF alebo DNF formulu. My sa budeme zaoberať len 3-SAT formulami, keďže každú formulu vieme pretransformovať na ekvivalentnú KNF formulu a každú KNF formulu vieme pretransformovať na ekvivalentnú 3-SAT formulu. Poďme si najprv formálne dokázať, že akúkoľvek formulu vieme pretransformovať na ekvivalentnú formulu v KNF forme. Potom si dokážeme, že akúkoľvek formulu v KNF vieme pretransformovať na ekvivalentnú 3-SAT formulu. V kapitole genetické programovanie si ukážeme prečo sú práve 3-SAT formuly lepšie.

Veta 1.1. *Pre každú formulu F platí, že ju vieme pretransformovať na ekvivalentnú formulu F' v KNF forme.*

Dôkaz:
Každá formula má pravdivostnú tabuľku, kde máme pre každé ohodnotenie premenných jej pravdivostnú hodnotu. Ak si vezmeme formulu F_2, ktorá sa skladá z disjukncií klauzúl, kde každá klauzula obsahuje konjukcie literálov (čiže reprezentuje konkrétne ohodnotenie premenných), pre ktoré je formula F nesplnená. Potom F_2 je v DNF a je splnená práve vtedy, keď je F nesplnená. Teraz ak F_2 znegujeme, tak dostaneme formulu F', ktorá je v KNF (negácia DNF nám vráti KNF a naopak) a zároveň je splnená práve vtedy, keď je F splnená. Z toho vyplýva, že F' je ekvivalentná formula v KNF k formule F. Jediný špeciálny prípad môže nastať keď je F tautológia, čo by znamenalo, že F_2 je prázdna formula, čo môžeme vyriešiť tak, že pre tautológie vrátime automatický formulu $F' = (x_1 \vee \neg x_1 \vee ... \vee x_n \vee \neg x_n)$, kde $x_1, x_2, ..., x_n$ sú všetky premenné formuly F

Veta 1.2. *Pre každú formulu F v k-SAT forme platí, že ju vieme pretransformovať na ekvivalentnú formulu F' v 3-SAT forme.*

Dôkaz:
Tým, že je F v k-SAT forme znamená, že $\exists\ n \in N, F = A_1 \wedge A_2 \wedge ... \wedge A_n$, kde pre každú klauzulu A_i platí, že má najviac k literálov. Chceme každú klauzulu pretransformovať na ekvivalentné klauzuly, ktoré budú mať práve 3 literály. Nech máme klauzulu A_i, ktorá má najviac k literálov, ukážeme si ako ju pretransformovať na ekvivalentné 3-SAT klauzuly (ostatné už pretransformujeme analogický).
Rozdeľme si to na prípady:

Ak je k = 1, k = 2 alebo k = 3, tak ich nemusíme vôbec meniť, keďže definíciu 3-SAT formuly spĺňajú.

k > 3 $\rightarrow A_i = (x_1 \vee x_2 \vee ... \vee x_k)$, podformula v 3-SAT forme je rovná týmto klauzulam
$A_i^{'} = (x_1 \vee x_2 \vee p_3) \wedge (\neg p_3 \vee x_3 \vee p_4) \wedge ... \wedge (\neg p_{k-3} \vee x_{k-3} \vee p_{k-2}) \wedge (\neg p_{k-2} \vee x_{k-1} \vee x_k)$

kde $p_3, p_4, ..., p_{k-2}$ sú nové pomocné premenné, ktoré sú v nových klauzulach definované tak, aby zaručili jednu dôležitú vec:
Pre každé ohodnotenie literálov x_j platí, že A_i je splnená práve vtedy, keď sú splnené všetky nové klauzuly.
"$\leftarrow$":
Ak sú splnené všetky nové klauzuly, tak to znamená, že musí byť aspoň jeden literál x_j pravdivý, lebo iba pomocou pomocných premenných p_j (všetky literály x_j sú nepravdivé) všetky klauzuly nesplníme. Keďže vždy máme pomocnú premennú p_j a v ďalšej klauzule jej negáciu. Čiže pri každej permutácií ohodnotení premenných p_j by nám aspoň 1 klauzula ostala nesplnená. Ale tým, že je aspoň jeden literál x_j pravdivý, tak je automatický splnená aj klauzula A_i.
"$\rightarrow$":
Ak je splnená klauzula A_i, t.z. aspoň jeden literál x_j je pravdivý, tak potom by sme vedeli už pomocou pomocných premenných p_j vytvoriť takú permutáciu ohodnotení, aby boli splnené všetky klauzuly.

Výsledná formula $F^{'} = A_1^{'} \wedge A_2^{'} \wedge ... \wedge A_n^{'}$, keďže konjunkcia podformúl v 3-SAT forme je taktiež v 3-SAT (počet literálov v každej klauzule nemeníme).

Na koniec tejto kapitoly si ešte zadefinujeme niektoré pojmy, ktoré budeme využívať v niektorých ďalších kapitolach.

Definícia 1.11. *Heuristická technika alebo skrátene heuristika je akékoľvek približné riešenie problému, ktoré nezaručuje, že je optimálne, ale je oveľa rýchlejšie ako riešenie klasickým vyhľadavacím algoritmom.*

Riešenie heuristikou používame hlavne pri riešení optimalizačných problémoch, kde nie je známy žiadny časovo realizovateľný vyhľadávací algoritmus.

Definícia 1.12. *Metaheuristika je procedúra vyššieho levelu na nájdenie heuristiky, ktorá poskytne dostatočne dobré riešenie na nejaky optimalizačný problém.[20]*

Kapitola 2

SAT solvery

V minulej kapitole sme si presne a formálne zadefinovali čo je vlastne problém SAT a prečo je ho dobre skúmať. V tejto kapitole by sme sa zamerali hlavne na SAT solvery. A to hlavne na to, že aké SAT solvery poznáme, aké algoritmy používajú a na aké formuly sú užitočné.

2.1 SAT solver

V neformálnej reči by sme mohli povedať, že SAT solver je nejaký program (napísaný v nejakom programovacom jazyku), ktorý na vstupe berie zakódovanú formulu. Na výstupe vracia buď ohodnotenie, pre ktoré je formula splnená alebo správu o tom, že formula je nesplniteľná. Pokúsime sa náš SAT solver zadefinovať formálne pomocou Turingovho stroja.

V minulej kapitole sme si zadefinovali čo je presne jazyk SAT. Túto definíciu využijeme pri definícií SAT solvera.

Keďže budeme SAT solver definovať pomocou Turingovho stroja, tak si najprv vysvetlime ako budeme kódovať formuly v našom Turingovom stroji. Všetky výrokové formuly nad abecedou $\Sigma = \{\neg, \wedge, \vee, \rightarrow, \longleftrightarrow, 0, 1, (,)\}$ budeme kódovať tak, že jednotlivé premenné očíslujeme číslami 1, 2, . . . a tieto premenné v booleovskom výraze nahradíme ich číslami v binárnom tvare.

Napríklad:
Booleovský výraz $(p \vee q) \rightarrow (\neg p \vee q \wedge r)$ má kód $(1 \vee 10) \rightarrow (\neg 1 \vee 10 \wedge 11)$.

Definícia 2.1. *Nech T je deterministický Turingov stroj nad abecedou $\Sigma = \{\neg, \wedge, \vee, \rightarrow, \longleftrightarrow, 0, 1, (,)\}$, ktorý pre každú formulu ϕ povie, či $\phi \in SAT$. Potom T nazveme SAT solver práve vtedy, keď $L(T) = SAT$.*

2.2 Neúplný SAT solver

Poznáme rôzne typy SAT solverov, každý sa líši rýchlosťou výpočtu, použitej pamäti, jeho heuristikou, ... Každý SAT solver rieši lepšie iné typy formúl. Medzi základné typy však patria tieto:

Definícia 2.2. *Neúplný SAT solver: Hovoríme, že SAT solver je neúplný, ak pre svoj výpočet využíva nejaký pravdepodobnostný algoritmus. Neúplné SAT solvery môžeme rozdeliť do 2 skupín. Tie, ktoré v konečnom čase skončia, ale nemusia nájsť požadovaný výsledok. Ďalšia skupina sú tie, ktoré vždy nájdu požadovaný výsledok, ale nemusia skončiť v konečnom čase.*

Algoritmy, ktoré využívajú dané SAT solvery: GSAT, MaxWalkSAT, ...
V tejto práci sa budeme zaoberať hlavne úplnými SAT solvermi, keďže sa v praxi ukazujú ako viac použiteľné.

Poďme si teraz vysvetliť nejaký základný algoritmus pre neúplný SAT solver.

2.2.1 GSAT algoritmus

Túto si vysvetlíme jeden zo základných algoritmov pre neúplné SAT solvery. Vysvetlíme si ich základný algoritmus a viac sa k neúplným SAT solverom vracať nebudeme, keďže nie sú až také zaujímavé pre prax.

Algoritmus funguje na jednoduchom princípe, algoritmus má hodnotu MAX_TRIES, ktorá nám hovorí koľkokrát máme skúšať nájsť nejaké ohodnotenie. Každé vyskúšanie funguje tak, že náhodné vygenerujem nejaké ohodnotenie a potom skúšam MAX_FLIPS krát či dané ohodnotenie nám vracia true. Ak áno, tak vrátime ohodnotenie, ak nie, tak vyberieme premennú, ktorá sa nachádza v najviac nesplniteľných klauzúl a znegujeme ju. A znova skúsime, či ohodnotenie je pravdivé. Ako vidíme, toto riešenie skončí v konečnom čase, no nemusí nájsť žiadne riešenie.

Algoritmus 2.1. *GSAP*

```
a = mnozina klauzul
for i = 1 to MAX_TRIES
    T = nahodne vygeneruj ohodnotenie
    for j = 1 to MAX_FLIPS
        if T je splnitelne pre a then return T
        p = premenna, ktora sa vyskytuje v
        najviac nesplnitelnych klauzulách
        T = T z negovanym p
    end for
end for
return „nenajdene ohodnotenie“
```

Obr. 2.1: Zdrojový kód algoritmu GSAP

2.3 Úplný SAT solver

Definícia 2.3. *Úplný SAT solver: Hovoríme, že SAT solver je úplný, ak pre každú vstupnú formulu v konečnom čase nájde pravdivé ohodnotenie, ak je formula splniteľná, alebo povie, že formula je nesplniteľná.*

Úplné SAT solvery väčšinou využívajú algoritmy DPLL, CDCL, ktoré si vysvetlíme neskôr. Medzi najznámejšie SAT solvery patria Cryptominisat a Glucose. Tieto sa budeme hlavne snažiť analyzovať.

Práve sa dostávame do sekcie, kde sa budeme venovať algoritmom, ktoré sa dnes používajú v najviac úplných SAT solverov, a vďaka ktorým je taktiež ovplyvnený náš genetický algoritmus, ktorý generuje ťažké inštancie. Ten si vysvetlíme až v kapitole genetické algoritmy.

Túto si vysvetlíme jeden zo základných algoritmov pre úplné SAT solvery. Na rozdiel od predchádzajúceho algoritmu, tento algoritmus nepoužíva žiadnu pravdepodobnosť.

2.3.1 DP algoritmus

Začneme starším a jednoduchším algoritmom, z ktorého základ sa neskôr použil pre vývoj CDCL algoritmu. DP alebo Davis–Putnam algoritmus pre každú formulu povie, či je splniteľná alebo nie. Menšia nevýhoda tohto algoritmu je, že samotný algoritmus nezisťuje konkrétne ohodnotenie, ak je formula splniteľná. Zistí len samotnú splniteľnosť. Samotný algoritmus funguje na princípe, že sa snažíme eliminovať konfliktné dvojice a postupne zisťujeme či sa dostaneme k prázdnej formule alebo nie.

Definícia 2.4. *Konfliktná dvojica: Nech F je formula v KNF forme obsahujúca premenné $x_1, x_2, ..., x_n$ a klauzuly $C_1, C_2, ..., C_m$. Potom literály l_1 a l_2 nazveme konfliktné dvojice, ak $l_1 = x_i$ a $l_2 = \neg x_i$ a zároveň l_1, l_2 sa nachádzajú v rôznych klauzulách.*

Algoritmus je rozdelený do cyklov. Kde v každom cykle si vyberieme jednu premennú x, ktorú sme ešte neeliminovali a spravíme nasledovné:

- 1. Pre premennú x nájdeme všetky konfliktné dvojice. Čiže nájdeme všetky dvojice x a ¬x, resp. ¬x a x, ktoré nie sú v rovnakej klauzule.
- 2. Pre každú dvojicu literálov l_1 a l_2 premennej x, kde l_1 je z klauzuly C_n a l_2 je z klauzuly C_m. Spravíme to, že l_1 a l_2 z klauzúl odstránime a klauzuly spojíme do jednej klauzuly.
  Napríklad: Nech $C_n = x_1 \vee x_2 \vee ... \vee x_n \vee l_1$ a $C_m = x_1^{'} \vee x_2^{'} \vee ... \vee x_n^{'} \vee l_2$, potom klauzula C po eliminácií konfliktnej dvojice l_1 a l_2 a zjednotení klauzúl bude rovná C $= x_1 \vee x_2 \vee ... \vee x_n \vee x_1^{'} \vee x_2^{'} \vee ... \vee x_n^{'}$
- 3. Toto spravíme pre všetky konfliktné dvojice premennej x a ďalej pokračujeme elimináciou konfliktných dvojíc pre ďalšiu premennú (ktorú sme ešte neeliminovali), ale už na upravenej formule, ktorá neobsahuje konfliktné dvojice premennej x. Tieto 3 kroky vykonávame pokiaľ ešte pre nejakú premennú existuje Konfliktná dvojica.

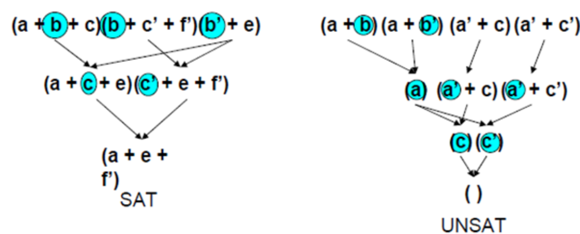
Ak sa vo výpočte dostaneme do stavu, kedy už pre žiadnu premennú konfliktné dvojice neexistujú, tak formula je splniteľná práve vtedy, keď momentálna formula po eliminácií nie je prázdna (obsahuje aspoň jednu neprázdnu klauzulu). Ak je prázdna, tak pôvodná formula je nesplniteľná.

Obr. 2.2: Zdrojový kód algoritmu DP[5]

Algoritmus 2.2. *DP*

```
function resolution-SAT(F): Boolean
    clauses = clausesOf(F)
    do
        new = ∅
        foreach C, D ∈ clauses, C_i = ¬D_j do
            R = resolve(C, D)
            if R is an empty clause then
                return false
            simplify(R) //remove repeated literals
            if R not tautology ∧ R ∈ clauses then
                new = new ∪ {R}
        endfor
        clauses = clauses ∪ new
    while new ≠ ∅
    return true
```

Obr. 2.3: Príklad DP algoritmu



Ako vidíme, v prvom prípade algoritmus DP najprv v jednom cykle eliminoval konfliktné dvojice pre premennú a. V ďalšom cykle eliminoval konfliktné dvojice pre premennú c. Teraz je vidno, že už žiadne konfliktné dvojice neexistujú a formula je neprázdna, čiže počiatočná formula je splniteľná. V ďalšom prípade po všetkých elimináciách zostane formula prázdna, a preto je počiatočná formula nesplniteľná.

Algoritmus vždy skončí, keďže každá formula má konečný počet klauzúl a premenných. A taktiež každá formula má len konečný počet konfliktných dvojíc a po každej iterácií

sa 1 premenná z celej formuly vymaže. Dôkaz správnosti môžete nájsť v knihe[21].

Tento algoritmus je síce správny, ale má hlavné nevýhody, že nenájde ohodnotenie, ale zistí len splniteľnosť. Taktiež časová a pamäťová zložitosť je exponenciálna od počtu premenných a oveľa menej formúl v porovnaní s novými algoritmami vypočíta rýchlo.

2.3.2 DPLL algoritmus

Teraz si predstavíme oveľa praktickejší algoritmus DPLL, ktorý ak je formula splniteľná, nájde aj ohodnotenie. Prejdime k samotnému algoritmu. Algoritmus funguje na princípe, že si vezme všetky premenné v nejakom usporiadanom poradí a potom rekurzívne začne skúšať od prvej premennej možné priradenia. Ale na rozdiel od klasického backtrackingu sa snaží zistiť kedy sa už vnárať ďalej neoplatí. A to aplikovaním pravidiel early termination, pure literal elimination, unit propagation, ktoré si vysvetlíme teraz[5].

- (Early termination) Ak sa počas výpočtu stane, že v nejakej klauzule sú všetky literály nepravdivé, tak ďalšie neohodnotené premenné už nie je potrebné skúšať, lebo vieme, že celá formula už nikdy splnená nebude.

- (Pure literal elimination) V prípade, že sa nám v každých zatiaľ nesplnených klauzulách vyskytne premenná jednej polarity (čiže iba x alebo iba $\neg x$), takýto literál nazveme pure (v klauzulách, ktoré sú v momentálnom ohodnotení splnené môže mať premenná aj opačnú polaritu, ale to nás nezaujíma). Ak sa nám počas výpočtu objaví pure literal, tak mu priradíme hodnotu, aby bol v zatiaľ nesplnených klauzulách pravdivý a tým ho eliminujeme. Správny výsledok to nemôže ovplyvniť, keďže ak existuje pravdivé ohodnotenie formuly s nepravdivým pure literálom, tak musí existovať aj pravdivé ohodnotenie formuly s pravdivým pure literálom. Vďaka vlastnosti KNF formy.

- (Unit propagation) Ak nám počas výpočtu nastane prípad kedy v nejakej klauzule sú už všetky literály nepravdivé a ostáva už len jeden neohodnotený literál x, tak vieme, že x musí byť pravdivý (inak by nastal early termination a k žiadnemu výsledku by to neviedlo).

Celý algoritmus začína na formule, ktorá nemá žiadnu premennú ohodnotenú. Počas algoritmu môže byť priradená hodnota premennej buď novým volaním rekurzívnej funkcie DPLL, aplikovaním unit propagation alebo eliminovaním pure literals. Keď je rekurzívna funkcia volaná prvýkrát, tak žiadna premenná nie je ohodnotená a hĺbka rekurzie je rovná 0. Pri každom ďalšom vnorení funkcie DPLL sa postupne premenným začínajú priraďovať hodnoty. Všeobecne platí, že ak sa funkcia vráti z rekurzie, tak všetky premenné, ktoré nadobudli hodnotu v danej hĺbke budú anulované.

Ak sa na formulu počas výpočtu aplikuje pravidlo unit propagation, tak v nejakej klauzule musí byť každý literál nepravdivý a len jeden ešte neohodnotený literál x. Všetky nepravdivé literály pre literál x budú jeho predchodcovia a označme si ich ako ante(x). Počas výpočtu nám môže dôjsť k nejakému sporu, kde konkrétna premenná kvôli aplikovaniu unit propagation bude musieť byť pravdivá aj nepravdivá. Aby sme takéto prípady vedeli analyzovať, tak si vytvoríme takzvaný implikačný graf, ktorý pre každý unit propagation zaznačí akú musí mať neohodnotený literál hodnotu, a aj kto sú jeho predchodcovia.

Definícia 2.5. *: Implikačný graf je acyklický orientovaný graf $G = (V, E)$, kde V reprezentujú priradenia a $(x, y) \in E \longleftrightarrow x \in ante(y)$.*

Obr. 2.4: Zdrojový kód algoritmu DPLL[5]

Algoritmus 2.3. *DPLL*

```
function DPLL-SAT(F): Boolean
    clauses = clausesOf(F)
    vars = variablesOf(F)
    e = ∅ // partial truth assignment
    return DPLL (clauses, vars, e)

function DPLL(clauses, vars, e): Boolean
    if ∀c ∈ clauses, e*(c) = true then return true
    if ∃c ∈ clauses, e*(c) = false then return false // early termination
    e = e ∪ unitPropagation(clauses, e)
    e = e ∪ pureLiteralElimination(clauses, e)
    x ∈ vars ∧ x ∉ e // x is an unassigned variable
    return DPLL(clauses, vars, e ∪ {e(x) = true}) or
    DPLL(clauses, vars, e ∪ {e(x) = false})
```

Algoritmus začne funkciou DPLL-SAT, ktorá vezme na vstupe formulu F a následne si uloží do pamäti klauzuly a premenné. Ďalej si vytvorí prázdnu množinu pre priradené premenné a následne zavolá funkciu DPLL. Funkcia DPLL je rekurzívna a pri každom priradení novej premennej zavolá samu seba a priradí hodnoty ďalšej premennej. Najprv sa zavolá pre hodnotu true a neskôr pre hodnotu false. Takto pokračuje pokiaľ

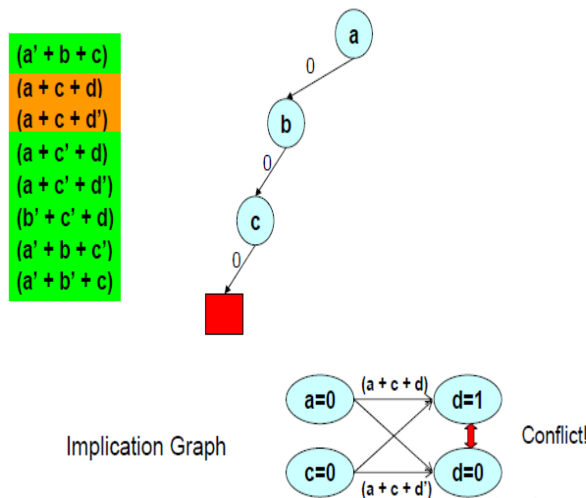
nepriradí hodnotu každej premennej (takto by fungoval klasický backtracking) alebo pokiaľ nenarazí na prípady KNF formy.

Všimnime si, že na začiatku funkcie DPLL ešte pred priradením hodnoty novej premennej vždy najprv zistí či už formula nie je splnená (koniec reukrzie). Následne zistí, či momentálne ohodnotenie nie je v stave early termination (aspoň 1 klauzula má všetky premenné nepravdivé). Ak áno, tak sa môžeme vynoriť z reukrzie. Ďalej na ohodnotenie premenných aplikuje unit propagation a pure literal elimination čo značne urýchli rekurziu. Ak by už nastal stav kedy je už formula pravdivá, resp. nepravdivá, tak v nasledujúcom volaní už funkcia vráti true, resp. false. Takto pokračuje až pokiaľ nenarazí na ohodnotenie, ktoré je pravdivé.

Príklad:

Obr. 2.5:

obr:dpll



Vidíme, že pre priradenie a = 0, b = 0 a c = 0 v našom implikačnom grafe vznikol spor, keďže po aplikovaní unit propagation v našej formule by premenná d musela byť pravdivá a zároveň nepravdivá.

Veta 2.1. *Algoritmus DPLL pre každú formulu F v KNF forme vždy nájde v konečnom čase správny výsledok.*

Dôkaz:
Ako sme videli, DPLL je len optimalizácia klasického backtrackingu. Čiže DPLL vždy spraví najviac 2^n krokov, a preto vždy skončí v konečnom čase. Na optimalizáciu využíva prípady definované pri KNF forme (early termination, pure literal elimination,

unit propagation). Tie sme si vysvetlili prečo ich použitie nájde výsledok práve vtedy, keď klasický backtracking.

2.3.3 CDCL algoritmus

Algoritmus CDCL je veľmi podobný algoritmu DPLL, ale využíva k tomu ešte strojové učenie a pri rekurzívnej funkcií DPLL sa nevynára chronologický, ale môže sa vynoriť aj o pár stupňov vyššie.

Čiže predstavme si to takto, že všetko čo sme sa naučili pri algoritme DPLL je rovnaké, máme nejakú rekurziu, ktorá postupne priraďuje premenným hodnoty true, false. Taktiež máme hĺbku rekurzie, ktorá začína na 0, čo znamená, že žiadnej premennej nie je priradená žiadna hodnota. Ak máme hĺbku n, tak vieme, že aspoň n premenným už je nejaká hodnota priradená. A postupne sa v každej hĺbke rekurzie snažíme na našu formulu aplikovať pravidlá early termination, pure literal elimination, unit propagation a vytvárame si implikačný graf. Avšak veľký rozdiel bude pokiaľ nám vo výpočte nastane v implikačnom grafe spor. Teda existuje premenná, ktorá aplikovaním pravidla unit propagation musí obsahovať obidve hodnoty naraz.

V klasickom DPLL algoritme by sme pri spore v implikačnom grafe len pokračovali zmenou hodnoty poslednej premennej alebo vynorením z rekurzie o 1 stupeň. Túto sa pomocou strojového učenia budeme snažiť z daného sporu vyťažiť čo najviac informácií, aby k podobnému sporu nedochádzalo tak často.

Na zapamätanie daného sporu nám budú slúžiť naučené klauzuly (learned clauses). Teda pokiaľ nám vznikne v implikačnom grafe spor, tak si vytvoríme novú klauzulu, ktorá v sebe bude niesť informáciu o danom spore a pridáme si ju do našej formuly. To nám značne urýchli náš výpočet.

Predstavme si takúto formulu:

$$c_1 = (\neg x_1 \vee \neg x_2 \vee x_8)$$

$$c_2 = (x_5 \vee \neg x_8 \vee \neg x_9)$$

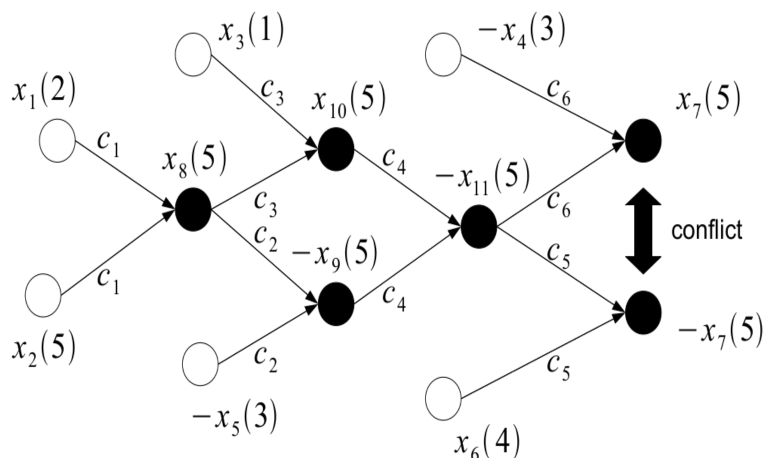
$$c_3 = (\neg x_8 \vee \neg x_3 \vee x_{10})$$

$$c_4 = (\neg x_{10} \vee x_9 \vee \neg x_{11})$$

$$c_5 = (x_{11} \vee \neg x_6 \vee \neg x_7)$$

$$c_6 = (x_4 \vee x_{11} \vee x_7)$$

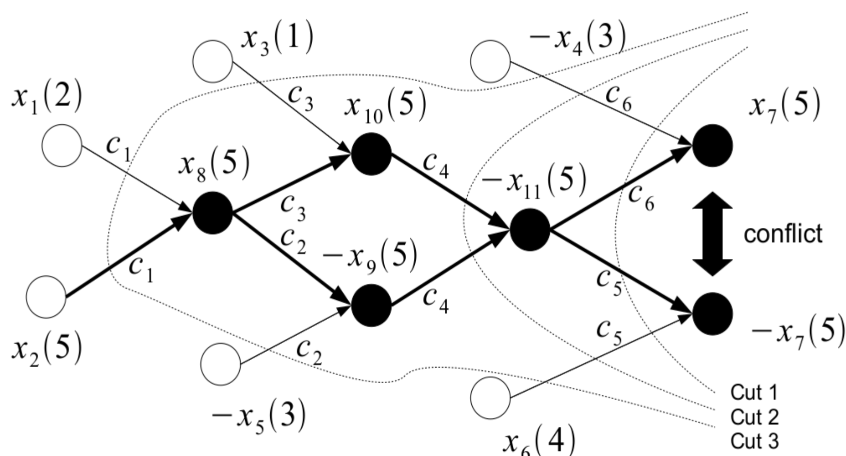
Obr. 2.6: Príklad implikačného grafu[5]



Teraz si k tejto formule predstavme takýto implikačný graf v nejakom konkrétnom stave algoritmu. Ak je nejaký vrchol čierny znamená to, že priradenie premennej bolo vytvorené pravidlom unit propagation. Ak je biely, tak priradenie premennej bolo vytvorené funkciou decide. Číslo na pravo od premennej označuje hĺbku rekurzie kedy dané priradenie nastalo. Vidíme, že daný graf má spor, lebo premenná x_7 musí byť true a zároveň false.

Pomocou tohto konfliktného grafu vytvoríme naučenú klauzulu. Najprv si vysvetlíme zopár veci, ktoré v tomto grafe platia. Takýto konfliktný graf sa vždy dá rozdeliť na 2 časti. Časť, kde sa nachádza konflikt (obsahuje vrcholy x_7 a $\neg x_7$), a kde sa nenachádza. Takéto rozdelenie grafu nazývame rez (cut). Ak spravíme takýto rez, tak do naučenej klauzuly budú negácie literálov tých vrcholov, ktoré sú v priamom spojení s konfliktnou časťou.

Obr. 2.7: Príklad implikačného grafu s rezmi[5]



V takomto grafe sa nachádza veľa rezov, ktoré reprezentujú nejakú naučenú klauzulu. Nie každá je ale veľmi užitočná na urýchlenie výpočtu. V našom prípade sme vybrali 3 rezy.

$$\text{cut1} = (x_4 \vee x_{11} \vee \neg x_6)$$

$$\text{cut2} = (x_4 \vee \neg x_{10} \vee x_9 \vee \neg x_6)$$

$$\text{cut3} = (x_4 \vee \neg x_3 \vee \neg x_1 \vee \neg x_2 \vee x_5 \vee \neg x_6)$$

Vidíme, že naučené klauzuly sme získali jednoduchým prechodom rezu. Vyberáme vždy negáciu literálu toho vrcholu, z ktorého smeruje hrana, ktorú prechádzame.

Poznámka: Všimnime si, že každý rez vznikol vlastne zlúčením nejakých klauzúl a odobratím ich konfliktnej dvojice podobne ako sme to mali v algoritme DP. Prvý rez vznikol zlúčením klauzúl c_5 a c_6, a následným odobratím ich konfliktnej dvojice. Podobne to platí aj pre rez 2 a 3. Čiže môžeme vidieť, že algoritmus CDCL využíva aj myšlienku DP algoritmu.

Vidíme, že naučených klauzúl môže byť viac a pridávanie každej jednej by nám extrémne zvýšilo pamäťovú zložitosť a taktiež by to spomalilo aj výpočet. Niektoré naučené klauzuly sú nám trochu zbytočné. Najvýhodnejšie sa ukazujú takzvané presadzovacie (asserting) klauzuly. Presadzovacie klauzuly sú tie klauzuly, ktoré majú práve jeden literál z aktuálneho rozhodovacieho levelu. Ako môžeme vidieť, rez1 a rez3 vytvá-

rajú presadzovacie klauzuly, ale rez2 nevytvára, lebo literály $\neg x_9$ a x_{10} sú z rovnakého rozhodovacieho levelu 5.

Na nájdenie presadzovacích klauzúl potrebujeme najprv nájsť unikátny implikačný bod (UIP). Ide o vrchol, cez ktorý musia prechádzať všetky orientované cesty, ktoré smerujú ku konfliktným vrcholom. V našom konfliktnom grafe ide napríklad o vrcholy x_8 a $\neg x_{11}$. Vrchol vytvorený funkciou decide sa považuje taktiež za UIP. Pre každý UIP vrchol vieme nájsť taký rez, aby výsledná klauzula bola presadzovacia. Najčastejšia metóda na vyhľadanie presadzovacej klauzuly funguje tak, že nájde najbližší UIP vrchol ku konfliktným vrcholom a z neho vytvorí rez. V našom prípade to je prvý rez.

Teraz si ukážeme, že pridanie takejto klauzuly do formuly nezmení množinu pravdivých ohodnotení.

Veta 2.2. *: Pridaním naučenej klauzuly do KNF formuly, ktorá vznikla z klauzúl pôvodnej formuly nezmení množinu pravdivých ohodnotení.*

Dôkaz:
Nech $(x_1 \vee ... \vee x_{n-1} \vee y_1 \vee ... \vee y_{m-1})$ je naučená klauzula, ktorá vznikla z pôvodných klauzúl $(x_1 \vee ... \vee x_{n-1} \vee x_n)$ a $(y_1 \vee ... \vee y_{m-1} \vee \neg x_n)$ zlúčením a odobratím konfliktnej dvojice. Chceme ukázať, že ak pre konkrétne ohodnotenie platí, že sú splnené pôvodné klauzuly, tak je splnená aj naučená klauzula.

Rozdelíme si to na prípady podľa hodnoty x_n:
1. Ak je $x_n =$ true, tak potom aspoň jeden literál $y_1 \vee ... \vee y_{m-1}$ musí byť taktiež pravdivý. A preto bude pravdivá aj naučená klauzula.

2. Ak je $x_n =$ false, tak potom aspoň jeden literál $x_1 \vee ... \vee x_{n-1}$ musí byť pravdivý. A preto bude pravdivá aj naučená klauzula.

Týmto sme ukázali, že pridanie naučenej klauzuly do formuly nikdy nespôsobí, že nám ubudne pravdivých ohodnotení. Taktiež nám ani nemôžu pribudnúť nové pravdivé ohodnotenia, keďže ide o KNF formulu a v KNF formule pridávaním nových klauzúl môžu len ubúdať riešenia.

Ukážme si teraz samotný pseudokód algoritmu CDCL:

Obr. 2.8: Zdrojový kód algoritmu CDCL[5]

Algoritmus 2.4. *CDCL*

```
function CDCL-DPLL(F): Boolean
    clauses = clausesOf(F)
    e = ∅ // partial truth assignment
    level = 0 // decision level
    if BCP(clauses , e) = false then return false
    while true do
        lit = decide(F , e) // an unassigned variable
        if lit = null then return true
        level = level + 1
        e = e ∪ e(lit) = true
        while BCP( clauses , e ) = f alse do
            if level = 0 return false
            learned = analyzeConflict( clauses , e )
            clauses = clauses ∪ learned
            btLevel = computeBTLevel(learned)
            e = removeLaterAssignments(e , btLevel)
            level = btLevel
        endwhile
    endwhile
```

Už na začiatku tohto algoritmu sme si povedali, že hlavným rozdielom medzi CDCL a DPLL algoritme je ten, že CDCL nevykonáva rekurziu chronologický, ale môže sa pri nájdenom spore vynoriť aj o pár stupňov viac. Informáciu o tom koľko stupňov sa má vynoriť algoritmus z rekurzie mu poskytuje naučená klauzula. Keďže tá v sebe nosí hlavnú informáciu o spore. V naučenej klauzule vyberieme literál s druhou najväčšou hĺbkou rekurzie a tá bude znamenať, že na takú hĺbku sa má rekurzia vynoriť. Pokiaľ by naučená klauzula mala len 1 literál, tak sa rekurzia vynorí na hĺbku 0.

Teraz si poďme vysvetliť samotný kód. Na začiatku si podobne ako v DPLL algoritme vyberieme klauzuly z formuly, vytvoríme si pole na priradenia premenných, nastavíme level rozhodnutia alebo hĺbku rekurzie na 0. Nakoniec zistíme či náhodou vo formule už bez priradenia premenných nemáme spor. Napríklad formula s klauzulami

$(x) \wedge (\neg x)$.

Následne si vytvoríme nekonečný cyklus a postupne používame funkciu decide na výber novej premennej, ktorej sa priradí nová hodnota. Postupne zväčšujeme rozhodovací level a pokračujeme ďalším priraďovaním. Ak už nemáme akej premennej hodnotu priradiť, tak každá premenná už nejakú hodnotu má a to znamená, že sme našli pravdivé ohodnotenie a máme výsledok. Môže sa nám stať, že sme došli k sporu (funkcia BCP vráti false). V tom prípade je potrebné spor analyzovať, vytvoriť naučenú klauzulu, zistiť rozhodovací level, kde sa vynoríme a odstrániť priradenia premenným, ktoré majú rozhodovací level väčší ako ten, kde sa vynárame. Pokiaľ nastane situácia, že sme došli k sporu a momentálny rozhodovací level je rovný nule, tak vieme, že formula je nesplniteľná, keďže na odstránenie sporu sa musíme vynoriť.

Poďme si teraz dokázať, že tento algoritmus je správny.

Veta 2.3. *Algoritmus CDCL na každom vstupe vždy skončí v konečnom čase a vráti správny výsledok.*

Dôkaz:
Nech F je formula v KNF forme, ktorá obsahuje n premenných. Potom v algoritme CDCL môžeme na tejto formule dosiahnuť maximálne n+1 rozhodovací level. To koľko priradení premenným nastalo vo všetkých leveloch si môžeme uložiť do vektora DPV (decision level population vector). Vektor DPV sa bude skladať zo zložiek $c_0, c_1, ..., c_{n+1}$, kde zložka c_i vyjadruje koľkým premenným bola priradená hodnota pre rozhodovací level i. Ak máme DPV vektory P a Q, tak hovoríme, že P > Q ak vektor P je lexikografický väčší ako vektor Q. Najmenšia možná hodnota vektora je (0, 0, ..., 0) a najväčšia možná hodnota je (n, 0, ..., 0). Kde najväčšia hodnota vektora označuje prípad kedy všetkým premenným sme priradili hodnotu pre rozhodovací level 0. Ukážeme si dôležitý invariant, že pokiaľ nastane zmena z vektora P na vektor Q počas výpočtu algoritmu, tak Q > P. Pridávaním nových priradení premenným sa zložky vektora len zväčšujú. A pokiaľ nájdeme spor a vynoríme sa z rozhodovacieho levelu l do levelu l', tak všetky zložky medzi l' a l vrátane sa anulujú a zložka l' je zvýšená vďaka tomu, že používame len naučené klauzuly, ktoré sú presadzovacie.

Tým sme dokázali, že algoritmus sa nikdy nezacykli. Teraz dokážeme, že nikdy nevráti zlý výsledok. Vieme, že pridávaním naučených klauzúl nám neubúda ani nepribúda pravdivých ohodnotení našej formuly. Ak algoritmus vráti, že formula je splniteľná, tak vieme, že sa tam nenašli žiadne spory a každá premenná má priradenú určitú hodnotu. Ak na druhej strane nám algoritmus vráti, že formula je nesplniteľná, tak potom je možne vytvoriť prázdnu formulu podobne ako pri algoritme DP. Z toho vyplýva, že formula je nesplniteľná.

V ďalšej kapitole si popíšeme najčastejšie používané heuristiky SAT solverov. Potom sa už môžeme pustiť rovno do genetického algoritmu, ktorý generuje ťažké inštancie pre SAT solvery.

Kapitola 3

Rozhodovacie Heuristiky

V tejto kapitole by sme ešte trochu viac preskúmali jednu dôležitú vec, ktorú využíva každý SAT solver. Predstavme si takúto formulu:

$$F = (d \lor b \lor c) \land (d \lor b \lor \neg c) \land (d \lor a \lor \neg b) \land (\neg d \lor c \lor b)$$

Ideme sa pokúsiť teraz túto formulu vyriešiť. Ak by sme na tejto formule pustili klasický DPLL algoritmus, tak by začal priraďovať postupne hodnoty premenným v poradí a, b, c, d. Hneď nám ale môže napadnúť, že asi by bolo najlepšie začať premennou d, keďže sa vyskytuje v najviac klauzulách. Tým by sme určite skôr našli výsledok alebo spor ako keď začneme premennou a, ktorá sa vo formule vyskytuje len raz. Pri najlepšom výbere premenných by sme mohli dostať lineárnu časovú zložitosť SAT solvera, takú rozhodovaciu heuristiku ale zatiaľ nepoznáme. Je aj otázka či vôbec existuje.

Na to nám slúži rozhodovacia heuristika SAT solvera (ďalej v tejto kapitole len heuristika), určuje v akom poradí sa budú premenným postupne priraďovať hodnoty. To nám potom vytvára otázky akú heuristiku použiť, na aké typy formúl bude daná heuristika efektívna a o koľko zrýchli výpočet SAT solvera.

Poznáme dva základné typy heuristík. Prvý typ je založený na skóre kedy každý literál má svoje skóre a algoritmus vyberá nasledujúce neohodnotené literály podľa toho, ktorý má najväčšie alebo najmenšie skóre. Druhý typ nie je založený na skóre, čiže vyberá nasledujúce neohodnotené literály nejakým iným algoritmom.

Teraz si predstavíme tie najviac známe a používané heuristiky, ktoré dnešné SAT solvery využívajú[5].

3.1 Jeroslow-Wang

Heuristika SAT solvera Jeroslow-Wang (JW) [15] je založená na skóre a funguje tak, že pre každý literál sa vypočíta jeho skóre. Neskôr počas algoritmu SAT solvera sa neohodnotené literály vyberajú podľa toho aký neohodnotený literál má najväčšie skóre.

Nech F je formula, pre každý literál lit vypočítame jeho skóre podľa tohto vzorca:

$$s(lit) = \sum_{lit \in c, c \in F} 2^{-|c|}$$

Kde c je klauzula a |c| reprezentuje jej veľkosť teda počet literálov v nej.

Vidíme, že heuristika JW uprednostňuje literály, ktoré sa vyskytujú najčastejšie v malých klauzulách (čím menšia klauzula, tým je väčšia pravdepodobnosť, že vznikne spor). Existujú dve typy JW heuristiky:

- (Statická JW) Skóre každého literálu sa vypočíta na začiatku algoritmu SAT solvera a ďalej sa nemení počas výpočtu.
- (Dynamická JW) Ak používame napríklad algoritmus CDCL (resp. iný algoritmus strojového učenia, ktorý pridáva nové klauzuly počas výpočtu), tak skóre literálov novej klauzuly upravujeme dynamický počas výpočtu.

3.2 DLIS

Ďalšiu heuristiku, ktorú si popíšeme je heuristika DLIS (Dynamic Largest Individual Sum) [23]. Heuristika DLIS je rovnako ako JW založená na skóre. Rozdiel je ale v tom, že skóre literálu sa počíta ako počet nesplnených klauzúl, kde sa tento literál nachádza (splnené klauzuly nás už nezaujímajú). Môžeme vidieť, že táto heuristika je dynamická. Keďže vždy musíme prepočítavať skóre keď sa niektoré klauzuly stanú splniteľné. Samozrejme vždy vyberáme literály s najvyšším skóre.

Hneď si môžeme všimnúť aj veľkú nevýhodu tejto heuristiky. Vždy musíme prepočítavať skóre literálov. Mohli by sme to optimalizovať tak, aby sa prepočítavanie vykonávalo len keď po ohodnotení začnú byť niektoré klauzuly splnené. Stále to ale bude dosť pomalé v porovnaní s lepšími heuristikami, ktoré si predstavíme ďalej.

3.3 LEFV

Nasleduje heuristika LEFV (Last Encountered Free Variable)[4]. Na rozdiel od väčšiny heuristík táto heuristika nie je založená na skóre literálov. Funguje na princípe nájdenia nejakého literálu, ktorý spĺňa určitú podmienku a ten literál použije. Túto podmienku si vysvetlíme teraz.

Vždy začneme s nejakým literálom a skontrolujeme klauzuly, ktoré obsahujú negáciu tohto literálu. Tieto klauzuly sú potenciálne unit klauzuly. Niektoré aj budú, ale väčšina nie. Teraz si držíme smerník na tento literál, ktorý sa nachádza v klauzule, ktorá nie je ešte splnená a nie je ešte ani unit. Ak SAT solver požiada priradiť hodnotu ďalšiemu literálu, tak sa vyberie tento.

Táto heuristika je jednoduchá na implementáciu a má tiež minimálnu časovú a pamäťovú zložitosť.

3.4 VSIDS

Teraz si predstavíme jednu z najlepších heuristík, ktoré sa používajú. Heuristika VSIDS (Variable State Independent Decaying Sum) [17] je znova založená na skóre literálov. Vyberá sa literál s najvyšším skóre. Teraz si vysvetlíme výpočet daného literálu.

Každý literál l má skóre s(l) a počet výskytov r(l). Pred vyhľadávaním sa skóre literálov inicializuje ako počet klauzúl, kde sa l vyskytuje. Vždy keď sa pridá nová klauzula počas výpočtu, tak sa inkrementuje hodnota r(l) tých klauzúl, ktoré nová klauzula obsahuje. A každých 255 vyberaní neohodnotených literálov sa hodnoty s(l) a r(l) pre každý literál aktualizujú takto:

$$\mathrm{s(l)} = \mathrm{s(l)}/2 + \mathrm{r(l)},\ \mathrm{r(l)} = 0$$

VSIDS je podobná heuristike DLIS, ale nezaujíma nás či je klauzula splnená alebo nie. Taktiež vždy vydelíme skóre hodnotou 2, aby sme zvýšili vplyv nedávno pridaných klauzúl a ich literálov.

Malá nevýhoda tejto heuristiky spočíva v tom, že výpočet je trochu spomalený, keďže skóre literálov sa aktualizujú len každých 255 vyberaní. Na druhej strane je vďaka tomu oveľa rýchlejší ako heuristika DLIS.

3.5 BerkMin

Poslednú heuristiku, ktorú si vysvetlíme je heuristika BerkMin[13], ktorá je taktiež založená na skóre. Táto heuristika ale nevyberá literál s najvyšším skóre takým štýlom ako v predchádzajúcich heuristikách. Literál je vybraný z nedávnych naučených a ešte nesplnených klauzúl, ktorý má najvyššie skóre. Teraz si vysvetlíme ako sa dané skóre počíta.

Podobne ako v heuristikách VSIDS alebo DLIS skóre je inicializované na počet klauzúl obsahujúci daný literál. Keď sa nám počas algoritmu vyskytne spor, tak všetky klauzuly, ktoré sa podieľajú na spore sú zaznamenané v skóre. Tým sa mysli, že inkrementujeme skóre každému literálu, ktorý sa vyskytuje v danej klauzule. Klauzuly podieľajúce sa na spore sú všetky tie, ktoré sú použité na vytvorenie novej naučenej klauzuly.

Podľa príkladu, kde sme si vysvetľovali algoritmus CDCL by podieľajúce sa klauzuly na spore boli c_5, c_6 a c_4. Čiže by sme inkrementovali skóre literálov $\neg x_{10}$, x_9, $\neg x_{11}$, $\neg x_6$, $\neg x_7$, x_4, x_7. A skóre literálu x_{11} by sme zvýšili až o 2.

Na rozdiel od VSIDS je ten, že v tejto heuristike skóre neklesá. Ďalej v BerkMin heuristike sa skóre aktualizuje hneď a nie každých 255 krokov ako vo VSIDS. Toto rieši problém neskorej reakcie, ktorý sme mali vo VSIDS. Zaregistrovaním všetkých klauzúl, ktoré sa podieľajú na spore získame oveľa viac informácie zo sporu. VSIDS registroval len naučené klauzuly, ale môže sa vyskytnúť literál, ktorý má veľký vplyv v spore a nedostane sa do naučenej klauzuly. VSIDS by ho nezapočítal, ale BerkMin áno. Práve kvôli týmto rozdielom sa BerkMin považuje za lepšiu heuristiku ako VSIDS, a preto sa používa vo väčšine moderných SAT solverov.

Vysvetlili sme si základné rozhodovacie heuristiky pre dnešné SAT solvery, v nasledujúcej kapitole začneme genetickým programovaním.

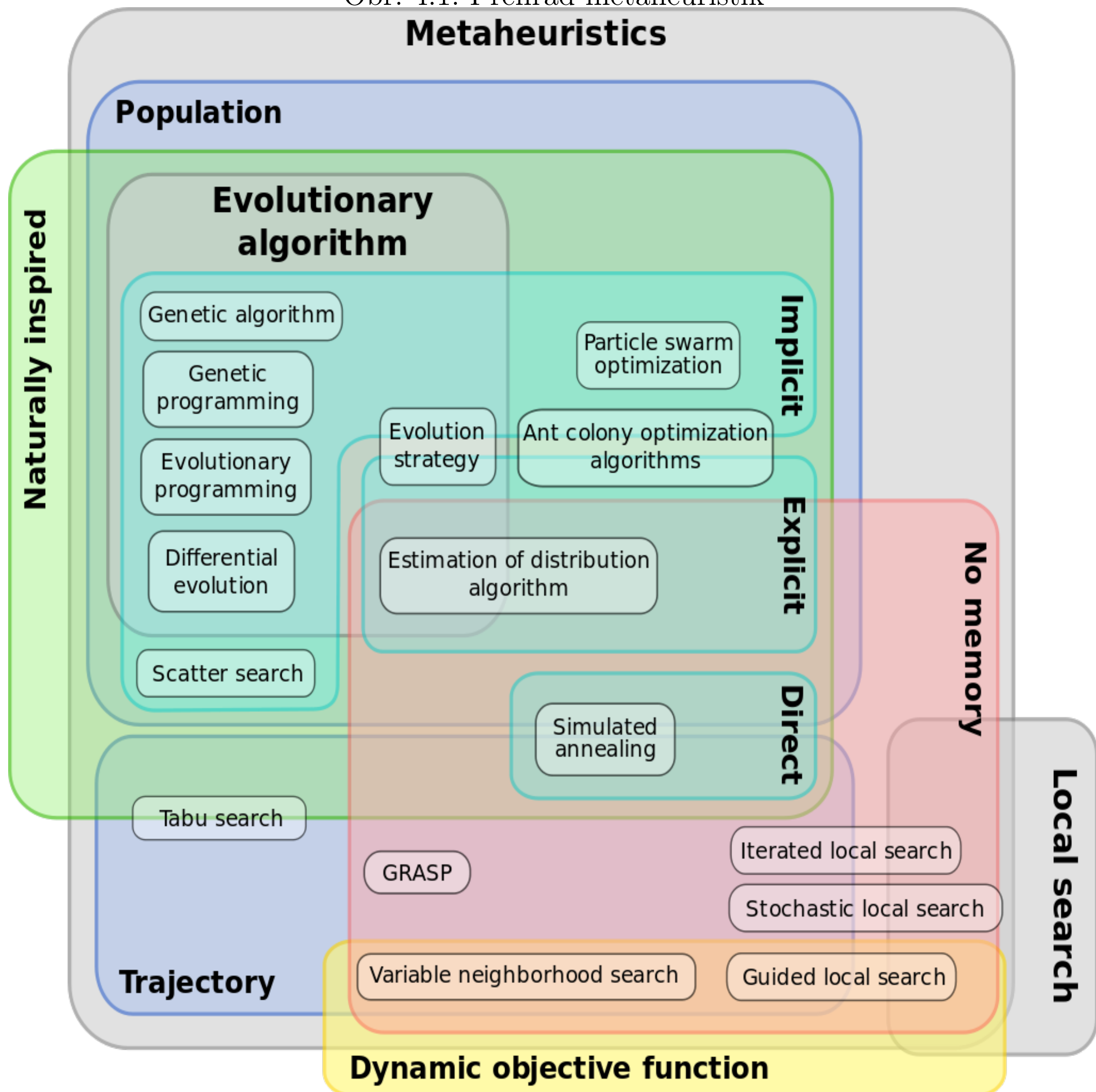
Kapitola 4

Genetické programovanie

V tejto kapitole by sme sa pozreli na to čo je to vlastne genetické programovanie a genetické algoritmy. Pozrieme sa na štatistiky ohľadom 3-SAT formúl, a ktoré dosť súvisia s genetickým algoritmom, ktorý bude generovať ťažké inštancie.

Pred tým než začneme postupne vysvetľovaním čo je genetické programovanie, tak si najprv vysvetlíme na aké typy problémov je daná metaheuristika určená a aké iné podobné metaheuristiky poznáme.

Obr. 4.1: Prehľad metaheuristík



Na tomto diagrame máme znázornené rôzne typy metaheuristík. Medzi hlavné typy patria hlavne metaheuristiky lokálneho a globálneho vyhľadávania. Ďalej máme metaheuristiky založené na populácií (kde patrí aj genetické programovanie) alebo jednotného riešenia (hľadanie len jedného konkrétneho riešenia). Metaheuristiky, ktoré nevyužívajú žiadnu pomocnú pamäť atď.

Vysvetlíme si niektoré najznámejšie:

- (Tabu search) Metaheuristika tabu search používa lokálne prehľadavanie a snaží sa dostať z jedného potenciálneho riešenia x na vylepšené riešenie x", ktoré je v susedstve z x. Prehľadávanie trvá až pokiaľ nedosiahne určitý stanovený limit.
- (Simulated annealing) Je pravdepodobnostná technika na vypočítanie približného globálneho optima pre danú funkciu. Konkrétne, ide o metaheuristiku, ktorá

nájde približne riešenie blízke globálnemu optimu vo veľkom vyhľadávacom priestore pre konkrétny optimalizačný problém.

.

.

.

My sa ale zameriame na metaheuristiku genetické programovanie, ktorá na riešenie problémov využíva metódy biologickej evolúcie, ktoré vymyslel britský prírodovedec Charles Darwin. Genetické programovanie spočíva v tom, že používa konkrétny genetický, resp. evolučný algoritmus, ktorý sa snaží populáciu nejakých jedincov pomocou operátorov a cyklov vylepšiť tak, aby konečná populácia mala čo najlepšie vlastnosti. To aké vlastnosti požadujeme, aby naša populácia mala záleží na konkrétnej úlohe, ktorú našim genetickým algoritmom riešime.

Ešte v 19. storočí Charles Darwin skonštruoval teóriu evolúcie prirodzeným výberom, ktorá v skratke funguje tak, že máme populáciu organizmov (živočíchy, rastliny, ...) a postupom času sa snažia prežiť v prírode tým, že sa postupne prispôsobujú prostrediu, v ktorom žijú. Tie, ktoré prežijú sa radia medzi najschopnejšie a zároveň predajú svoju schopnú vlastnosť na svoje deti a tie pokračujú v tomto cykle. Organizmy, ktoré neprežili radíme medzi menej schopné a ich vlastnosť sa z populácie vymaže, keďže nie je dostatočne dobrá na to, aby spĺňala podmienky prostredia, v ktorom žije.

V roku 1988 sa teoretický informatik John Koza [16] rozhodol pretransformovať túto Darwinovu teóriu na riešenie problémov v informatike. Táto teória sa ukazuje efektívna pri riešení niektorých optimalizačných problémov ako napríklad hľadanie ťažkých inštancií problému SAT.

Poďme si teraz zadefinovať určite pojmy, ktoré pri genetickom programovaní potrebujeme používať. Taktiež si ukážeme ako vyzerá genetický algoritmus.

- (Jedinec) Konkrétny prvok v prostredí, ktorý sa snaží svojimi vlastnosťami prežiť, v našom prípade sa jedinec myslí formula.

- (Zdatnosť) Zdatnosť alebo fitness sa mysli stupeň vlastnosti jedincov, ktoré vyžaduje dané prostredie na to, aby v ňom jedinec dokázal prežiť. Prostredie sa v našom prípade myslí konkrétny SAT solver a vlastnosť prežitia sa myslí rýchlosť výpočtu, ktorou SAT solver vypočíta formulu. Čím pomalší je výpočet, tým je väčšia pravdepodobnosť, že formula prežije.

- (Populácia) Súbor jedincov (formúl), ktorí sa snažia svojimi vlastnosťami prežiť v prostredí.

- (Generácia) Konkrétna populácia.

Poďme si teraz ukázať ako vyzerá všeobecný genetický, resp. evolučný algoritmus, ktorý sa snaží vyevolvovať najschopnejších jedincov v danom prostredí.

Genetický algoritmus sa delí na tieto časti:

- 1. Inicializácia - Náhodne vygenerujeme nultú generáciu.
- 2. Vyhodnotenie populácie - Vyhodnotíme zdatnosť novej populácie.
- 3. Selekcia - Vyberieme niekoľko najzdatnejších jedincov.
- 4. Zmena populácie - Z najzdatnejších jedincov pomocou týchto operátorov vytvoríme novú populáciu, resp. generáciu:
  - Kríženie - Pomocou niektorých jedincov prehoď ich časti a vytvori nového jedinca.
  - Mutácia - Náhodná zmena niektorej časti jedinca.
  - Reprodukcia - Kópia jedinca bezo zmeny.
- 5. Ukončovacia podmienka - Pokiaľ nie je splnená ukončovacia podmienka, tak pokračuj od bodu 2.
- 6. Koniec algoritmu - Jedinec alebo podmnožina jedincov s najlepšou zdatnosťou je výsledok genetického algoritmu.

Predstavili sme si všeobecný genetický algoritmus, ktorý budeme používať. Konkrétne detaily nášho genetického algoritmu (čo je kríženie, mutácia, ...) si vysvetlíme v časti generovania ťažkých inštancií.

4.1 Generovanie ťažkých inštancií

Pred tým než prejdeme na samotný genetický algoritmus by sme si ešte vysvetlili niektoré vlastnosti formúl, ktoré sa štatisticky ukazujú, že sú najvhodnejšie, aby výpočet formuly trval čo najdlhšie. Tieto vlastnosti formúl budú mať počas výpočtu všetky formuly a nebudú sa meniť.

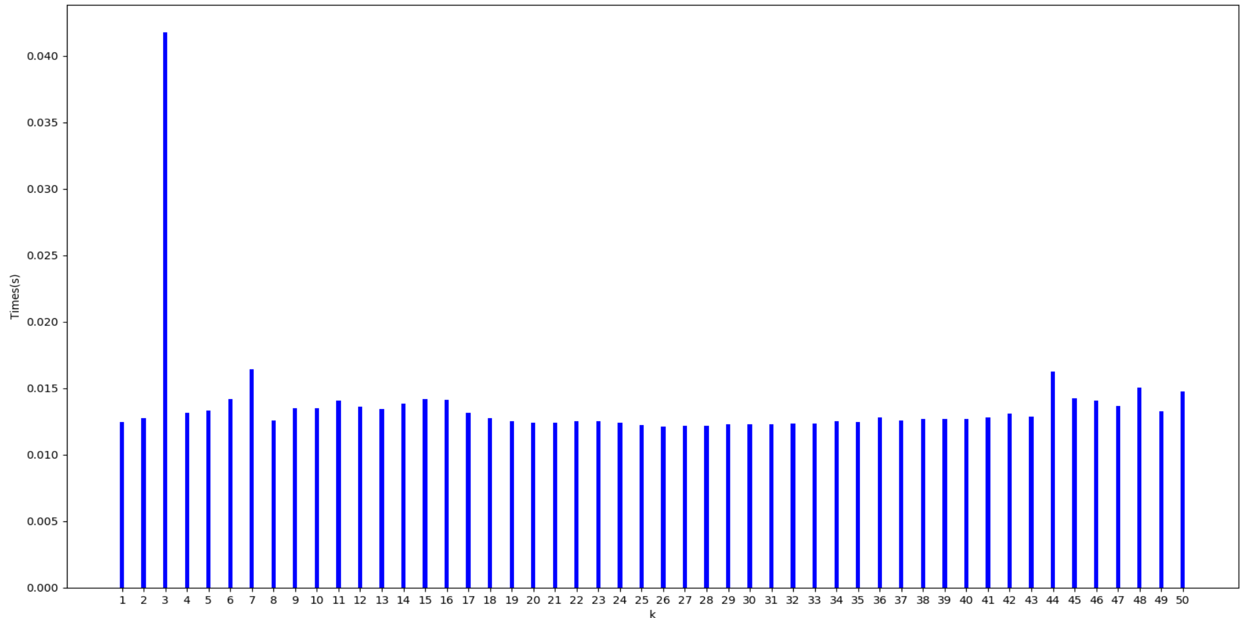
Poznámka: Štatistiky a vlastnosti, ktoré si práve ukážeme platia hlavne pre úplne SAT solvery založené na DPLL alebo CDCL algoritme.

4.1.1 Základné vlastnosti formúl

Vieme, že problém k-SAT je NP-úplný, ak je $k > 2$. Pre problém 2-SAT existuje polynomiálne riešenie. Viac si o tom môžeme prečítať túto [3]. Štatistiky z výskumov o

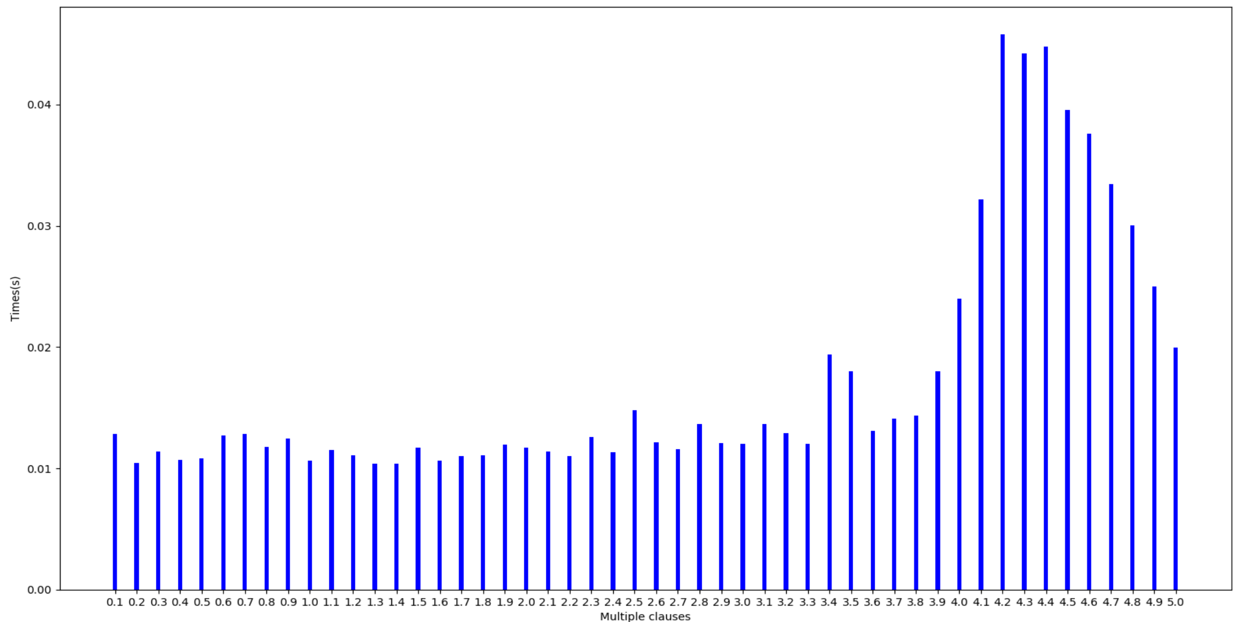
probléme SAT ukazujú, že najťažšie inštancie problémov sa vyskytujú vtedy, keď je k = 3, viď graf 4.2 a počet klauzúl je 4.2 krát viac ako premenných, viď graf 4.3. Podľa [6] platia tieto závislosti medzi pomerom počtu premenných a počtu klauzúl danej formuly:

Obr. 4.2: Štatistiky pre k-SAT formuly



Graf pre každé k od 1 do 50 ukazuje priemerný čas výpočtu formúl, ktoré boli pre dané k vypočítané. Pre každé k sme vygenerovali 1000 náhodných formúl a zobrali sme ich priemerný čas výpočtu. Formuly sme vypočítali na SAT solveri Cryptominisat.

Obr. 4.3: Štatistiky pre počet klauzúl



Podľa tejto štatistiky môžeme jasne vidieť ako formuly, ktoré majú pomer klauzúl k pomeru premenných približne 4.2, tak sa najdlhšie počítajú. Pre každý násobok klauzúl sa vygenerovalo 1000 náhodných formúl a uložil sa priemerný čas ich výpočtu.

- Ak je pomer počtu klauzúl k počtu premenných menej ako 4, tak sa dá pomerne rýchlo nájsť riešenie.
- Ak je pomer počtu klauzúl k počtu premenných približne 4.2, tak štatistiky ukazujú, že práve takéto formuly zaberú SAT solverom najviac času.
- Ak je pomer počtu klauzúl k počtu premenných viac ako 4.5, tak väčšina formúl je už nesplniteľných a SAT solver to vie pomerne rýchlo zistiť.

Ďalej by sme si uviedli štatistiky a vysvetlenia o tom aké musí byť rozmiestnenie premenných vo formule, aby SAT solveru trval výpočet čo najdlhšie.

Vysvetlíme si najprv čo znamená formula s rovnomerne rozmiestnenými premennými a ako takú formulu náhodne vygenerovať. Ide o formulu, v ktorej sa počtom každé 2 premenné líšia najviac o 1. Taktiež pre každú premennú platí, že počet literálov k počtu znegovaných literálov sa líši najviac o 10 (to zabráni častému výskytu pure literal).

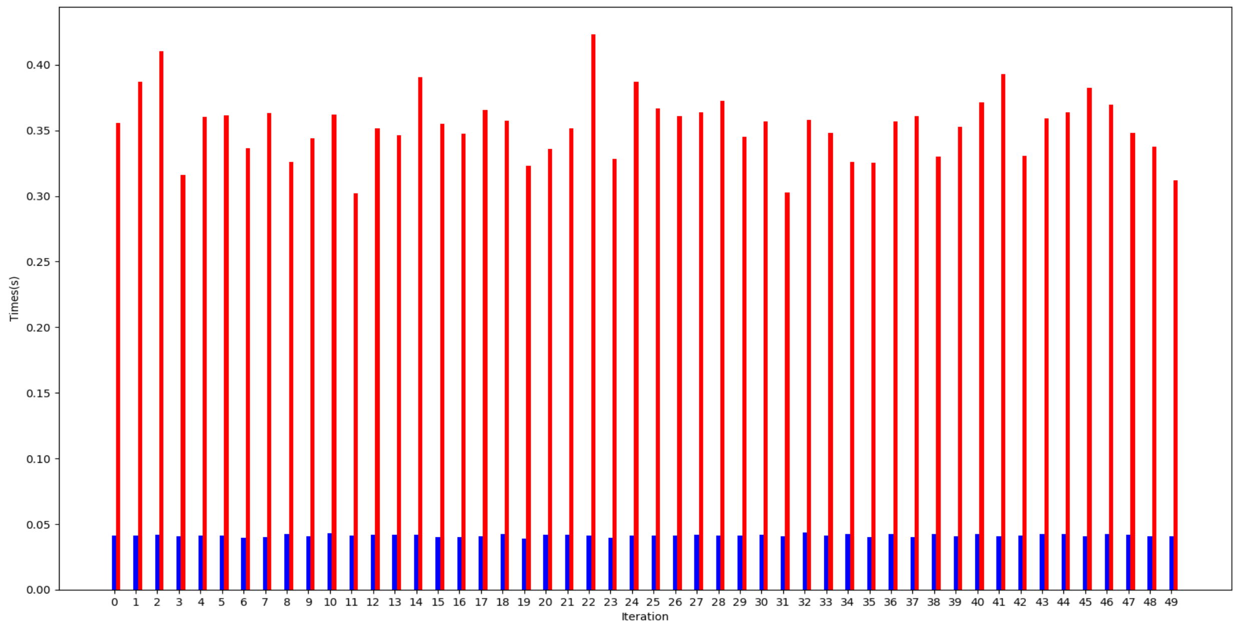
Vieme už koľko máme počet premenných a klauzúl. Začneme najprv tým, že si vytvoríme pole, v ktorom si budeme uchovávať indexy každej premennej, napríklad pre $n = 3$ to bude [1, 2, 3]. Teraz náhodne vyberieme niektorú premennú z daného

poľa a automatický tú premennú dáme na začiatok poľa a v nasledujúcej iterácií už ju vyberať nebudeme. Napríklad vyberieme premennú 2, zmeníme pole na [2, 1, 3] a v nasledujúcej iterácií už budeme vyberať len z premenných 2 a 3. Ak dôjdeme do situácie, že sme už všetky premenné použili, tak opäť budeme vyberať všetky premenné a postupne zmenšovať interval. Týmto nám vo formule budú vznikať postupne permutácie premenných, ktoré sa budú vyskytovať postupne za sebou. K tomu ešte dodáme, že vždy ak nejakú premennú vyberieme z daného poľa, tak sa musíme rozhodnúť či vytvoriť kladný alebo záporný literál. Preto si počítame taktiež koľko už máme celkovo záporných a koľko kladných literálov. Ak už máme viac kladných ako záporných o viac ako 10, tak automatický nový vygenerovaný literál bude záporný a opačne. Ak rozdiel medzi kladnými a zápornými literálmi nie je väčší ako 10, tak sa vyberá náhodne či bude kladný alebo záporný.

Z kapitoly o rozhodovacích heuristikách a SAT solveroch vieme čo urýchľuje výpočet formuly. Sú to tieto veci: early termination, pure literal elimination, unit propagation, dobrá rozhodovacia heuristika a použitie naučených klauzúl pri CDCL algoritme. Tým, že máme rovnomerne rozmiestnené premenné, tak je oveľa menšia pravdepodobnosť, že nastane early termination, pure literal alebo unit propagation. Keďže pri ohodnotení novej premennej sa ohodnotí menej literálov, a preto musí isť algoritmus v rekurzií hlbšie, aby došiel k nejakému sporu. V časti rozhodovacích heuristikách sme mohli vidieť, že väčšina heuristík je taktiež založená na princípe, kde ohodnotíme najprv premennú, ktorá má najvyššie skóre. Tým je taktiež väčšia pravdepodobnosť, že dôjdeme k nejakému sporu. Pri formule rovnomerne rozložených premenných aj heuristiky nebudú vyberať až tak ideálne poradie a výpočet algoritmu to značne spomalí.

Poznámka: Používať len formuly s rovnomerne rozmiestnenými premennými môže dosť obmedziť množinu ťažkých inštancií pre SAT solvery. Preto v našom genetickom algoritme nebudeme používať formuly s úplne striktnou definíciou, ale na nich vykonávať operácie, ktoré budú približne dodržiavať túto vlastnosť.

Obr. 4.4: Štatistiky pre náhodné a rovnomerne náhodné formuly



Z grafu môžeme vidieť, že formuly, ktoré majú rovnomerne rozložené premenné (červené) vo všetkých klauzulách, tak ich výpočet trvá oveľa dlhšie ako formuly, ktoré sú len úplne náhodné (modré). Preto v našom genetickom algoritme budeme v celom výpočte mať formuly, ktoré majú približne rovnaké rozloženie premenných. V grafe sme pre každú iteráciu vygenerovali 1000 náhodných a rovnomerne náhodných formúl a vzali ich priemerný čas výpočtu. Formuly sme počítali na SAT solveri Cryptominisat a každá formula mala 150 premenných.

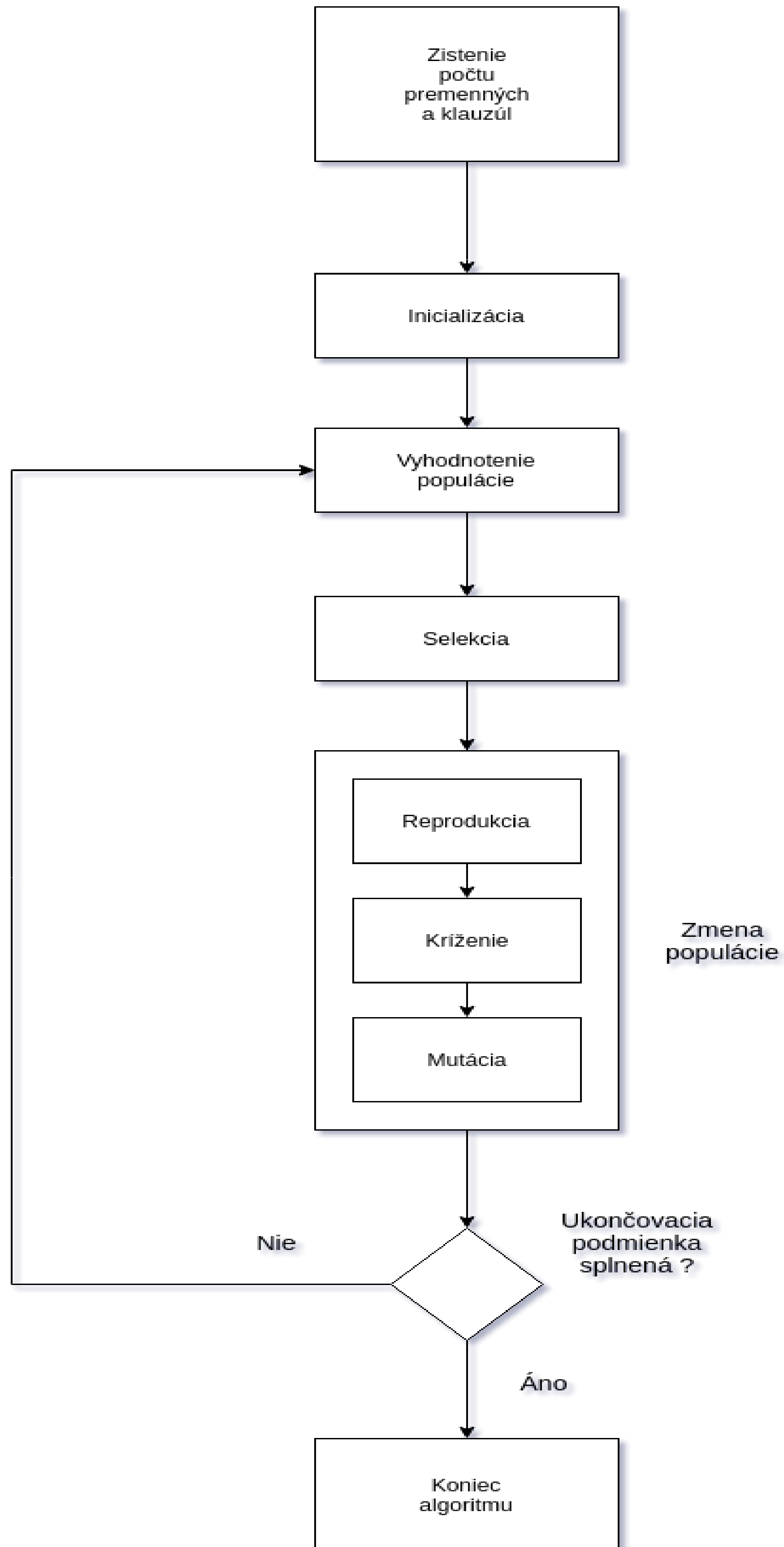
Teraz si môžeme už predstaviť ako funguje náš genetický algoritmus generujúci ťažké inštancie.

4.1.2 Genetický algoritmus

Ako sme na začiatku tejto kapitoly uviedli všeobecný genetický algoritmus, tak teraz si náš algoritmus vysvetlíme postupne ako vyzerá každý krok a každý operátor, ktorý sme si definovali.

Celý genetický algoritmus by sme mohli zakresliť do takéhoto diagramu:

Obr. 4.5: Diagram genetického algoritmu



4.1.3 Analýza SAT solvera

Začneme od prvého kroku, a tým je analýza počtu premenných a klauzúl pre formuly v populácií. Počet klauzúl zistíme jednoducho tak, že vynásobíme počet premenných číslom 4.2 (zistené najideálnejšie číslo). No najprv potrebujeme zistiť počet premenných. Ten zistíme jednoducho binárnym vyhľadávaním. Spravíme si interval od 1 do 200 (viac premenných by už robilo problém takmer každému SAT solveru) čo reprezentuje počet premenných. Ďalej si vždy vypočítame polovicu a vygenerujeme 200 rovnomerne náhodných formúl a vypočítame priemer času výpočtu daných formúl. Ak je daný priemer väčší ako 0.5 sekúnd, tak počet premenných zväčšíme, inak zmenšíme. Pol sekundy sa ukazuje ako ideálny čas na to, aby najlepšie formuly po skončení genetického algoritmu boli pre daný SAT solver ťažké. Chceli by sme priemerný čas výpočtu poslednej generácie približne 8 sekúnd.

4.1.4 Inicializácia

Ďalej nám nasleduje inicializácia populácie. Tu vytvoríme tak, že vygenerujeme 1000 rovnomerne náhodných formúl a z nich vyberieme najlepšie veľkosti populácie.

4.1.5 Vyhodnotenie

Teraz sa presúvame do kroku vyhodnotenia. Prejdeme každú formulu a vyhodnotíme jej čas výpočtu (pokiaľ ide o nultú generáciu, tak tej čas počítať nemusíme, máme ho z inicializácie). Tieto formuly si podľa času výpočtu zoradíme a vyberieme najschopnejších (najpomalšie formuly, v programe máme nastavenú veľkosť elity na 30).

4.1.6 Selekcia

Existuje veľa spôsobov selekcie hlavných rodičov, z ktorých potom vytvoríme novú generáciu. Najznámejšie metódy selekcie sú tieto:

- (Ruleta) Metóda selekcie, kde sa každý jedinec vyberá náhodne na základe hodnoty fitness. Čím vyššia hodnota fitness, tým je väčšia šanca, že daný jedinec bude vybratý.
- (Turnaj) V metóde turnaja vždy náhodne vytvoríme množinu jedincov. Z nej potom vyberieme nejakú podmnožinu najschopnejších.
- (Výber elity) Táto metóda funguje na takom princípe, že z danej generácie jedincov vyberieme len podmnožinu najschopnejších.

My budeme používať metódu výberu elity, v ktorej vždy vyberieme z celej generácie najpomalšie formuly a zreprodukujeme ich do ďalšej generácie. Táto metóda sa ukazuje ako najefektívnejšia.

4.1.7 Zmena populácie

Ostatné formuly vytvoríme tak, že vyberieme 2 náhodné formuly (otec a mama) z tých najpomalších. Chceme tiež, aby novým formulám ostali vlastnosti počtu premenných, klauzúl a rovnomerne rozloženie premenných. V probléme SAT sa ukazuje, že ak sa na formule vykoná príliš veľká zmena, tak je veľká pravdepodobnosť, že formula stratí svoju vlastnosť vďaka ktorej jej výpočet trvá dlho. Preto sa snažíme robiť len také zmeny, aby nám táto vlastnosť pri krížení nezmizla. Novú formulu vytvoríme pomocou rodičov, kde hlavne otec bude dávať novej formule klauzuly. Vieme, že naše formuly majú vlastnosť rovnomerne rozložených premenných, kde každá formula sa skladá z postupných permutácií premenných. Aby nám táto vlastnosť pri krížení veľmi nezmizla, tak novú formulu vytvoríme tak, že všetky klauzuly z otca prejdú na dieťa a v každej permutácií náhodne vygenerujeme jednu klauzulu, ktorá na dieťa pôjde z matky.

Ďalej na každý literál z novej formuly aplikujeme operátor mutácia, ktorý každý literál zmení na ľubovoľný iný s pravdepodobnosťou 0.01.

Po vygenerovaní novej populácie sa vypočíta ukončovacia podmienka, ktorá je nastavená na určitý počet iterácií. Ak sa už vykonal určitý počet, tak sa vyberie podmnožina najpomalších formúl a to je náš výsledok. Ak sa ešte nevykonalo, tak sa pokračuje späť vyhodnotením času výpočtu, selekciou, ...

Poznámka: Väčšina parametrov (populácia, počet iterácií, počet formúl pri inicializácií a analýze, ...) majú nastavené konkrétne hodnoty tak, aby algoritmus skončil v rozumnom čase na bežnom počítači. Určite ale platí, že čím by boli tieto parametre vyššie, tým by bol lepší aj výsledok.

Vysvetlili sme si ako funguje náš genetický algoritmus generujúci ťažké inštancie pre SAT solvery. V poslednej kapitole si ukážeme vypočítané dáta, ktoré sme dostali aplikovaním algoritmu na SAT solvery Cryptominisat a Glucose. A porovnáme, na ktorom je zložitejšie vygenerovať ťažké inštancie. A teda, ktorý je podľa nášho algoritmu lepší SAT solver.

Kapitola 5

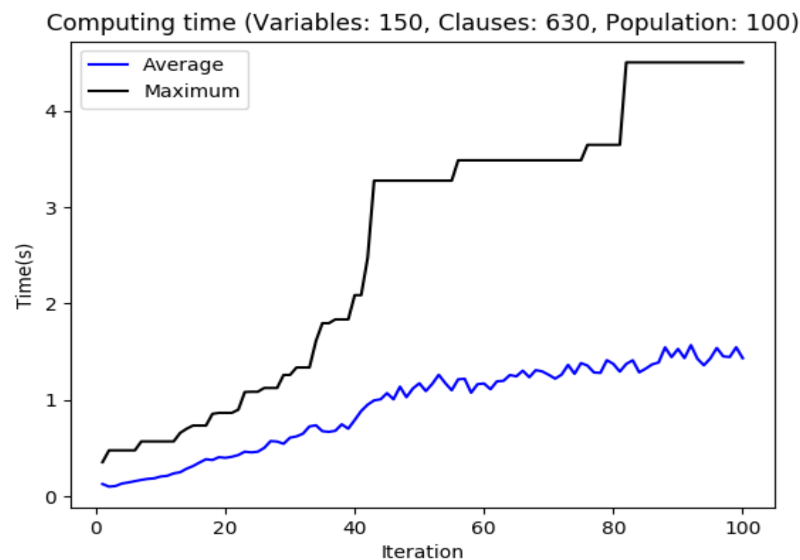
Výskum

V poslednej kapitole si ukážeme aké výsledky nám priniesol náš genetický algoritmus. Všetky výpočty budeme vykonávať na najznámejších SAT solverov Cryptominisat a Glucose. Postupne si budeme ukazovať grafy výpočtu pre náš genetický algoritmus.

Najprv si ukážeme aké by sme mali výsledky, ak by sme niektorú vlastnosť, ktorú používa náš genetický algoritmus odstránili.

Obr. 5.1:

obr:randomCryptominisat

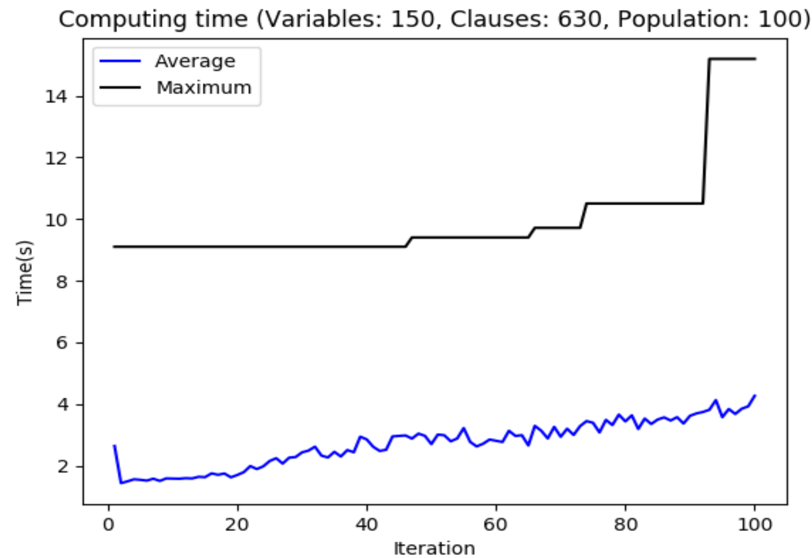


V prvom grafe máme SAT solver Cryptominisat, ktorý v inicializácií vygeneroval úplne náhodné formuly. A ako môžeme vidieť, priemerný čas výpočtu každej generácie trval len približne 1 sekundu. Najpomalšia formula, ktorá sa vygenerovala trvala len približne 4 sekundy.

Z tohto môžeme vidieť, že vlastnosť rovnomerne rozložených premenných je dosť dôležitá pokiaľ chceme, aby výpočet trval dlho.

Obr. 5.2:

obr:changesCryptominisat



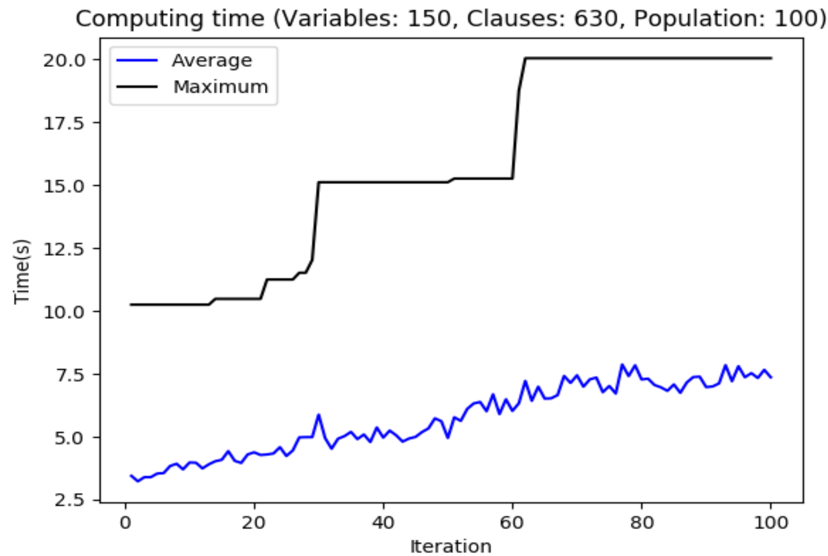
Na tomto grafe je zmena v tom, že SAT solver už vygeneroval formuly s rovnomerne rozloženými premennými a genetický algoritmus mal oproti nášmu, ktorý sme si definovali zmenené kríženie. V tomto prípade je kríženie nastavené tak, že vezme 2 náhodné formuly. Prechádza postupne klauzuly otca a matky a vždy vezme náhodne jednu klauzulu buď od otca alebo od matky s rovnakou pravdepodobnosťou.

Z grafu možno vidieť, že hneď na začiatku nám čas výpočtu rapídne klesol. Keďže sa nám hneď narušila vlastnosť rovnomerne rozložených premenných kvôli kríženiu. Na druhej strane sa výpočet oproti predošlému grafu oveľa zlepšil, ale ako uvidíme ďalej, tak ak robíme kríženie s menšími zmenami, tak výsledky sú ešte lepšie.

Ukážeme si teraz najlepšie výsledky nášho genetického algoritmu na SAT solveri Cryptominisat.

Obr. 5.3:

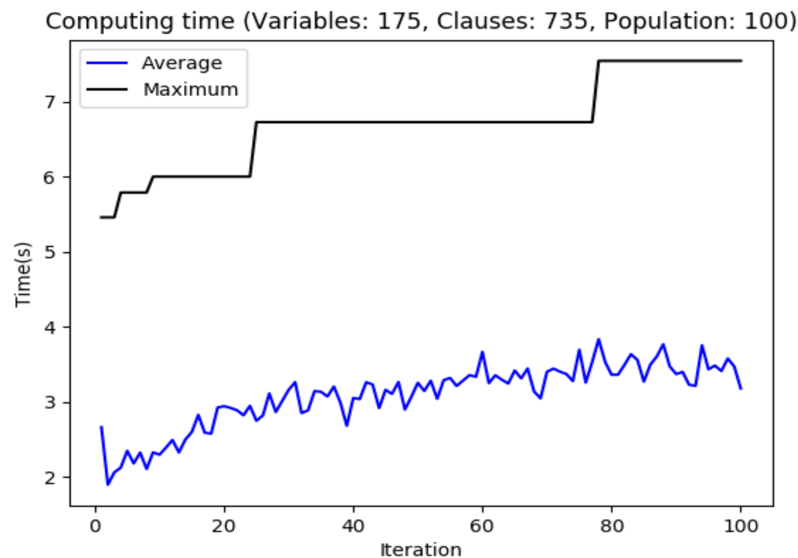
obr:cryptominisat



Ako môžeme vidieť z grafu, Cryptominisat vygeneroval ťažké formuly obsahujúce 150 premenných a 630 klauzúl (čo je 4.2 krát viac ako premenných) pri populácií 100 formúl. Po skončení algoritmu sme dostali formuly, ktoré majú priemerný čas výpočtu približne 7.5 sekúnd. Najťažšiu formulu trvá vypočítať 20 sekúnd.

Obr. 5.4:

obr:glucose



Na rozdiel od Cryptominisatu, Glucose vygeneroval ťažké formuly obsahujúce 175 premenných a 735 klauzúl (čo je 4.2 krát viac ako premenných) pri populácií 100 formúl. Po skončení algoritmu sme dostali formuly, ktoré majú priemerný čas výpočtu približne 4 sekundy. Najťažšiu formulu trvá vypočítať takmer 8 sekúnd. Vidíme, že pre SAT solver Glucose aj náš analyzátor analyzoval primeraný počet premenných až 175.

Podľa nášho genetického algoritmu teda vychádza, že na SAT solveri Glucose je ťažšie vygenerovať ťažké inštancie v porovnaní so SAT solverom Cryptominisat.

Poznámka: Všetky výpočty boli vykonané na procesore Intel Core i3-7100U CPU @ 2.40GHz $\times$ 4 s operačnou pamäťou 3.7 GiB.

Záver

Jazyk SAT sa ukazuje ako veľmi silný nástroj na riešenie rôznych NP problémov vďaka transformácií na SAT a následnom použití vhodného SAT solvera. V tejto práci sme ukázali ako môžeme analyzovať rôzne SAT solvery a vyhodnotiť, ktorý je najvýhodnejšie použiť. Genetickým algoritmom vieme vygenerovať ťažké inštancie a zhodnotiť tak jeho efektivitu. Čím slabšie inštancie pre daný SAT solver vygenerujeme (formuly s viac premennými a rýchlejším časom výpočtu), tým lepšie vieme posúdiť, že daný SAT solver je v mnohých problémoch lepší a efektívnejší.

Existuje ale ešte stále veľa iných algoritmov, ktorými by sme mohli analyzovať SAT solvery. Napríklad vymyslieť iný spôsob selekcie, reprodukcie, kríženia a mutácie. Použiť úplne iný genetický algoritmus alebo dokonca skúsiť použiť úplne inú metaheuristiku. Alebo skúsiť vymyslieť algoritmus, ktorý by bol zameraný len na určitý typ SAT solvera a jeho rozhodovaciu heuristiku. Tým by sme získali možno oveľa lepšie výsledky na danom SAT solveri, ale za cenu ujmy na všeobecnosti.

Literatúra

[1] Sat benchmarks, https://www.cs.ubc.ca/ hoos/satlib/benchm.html.

[2] Sat competition, http://www.satcompetition.org/.

[3] Dimitris Achlioptas. *Random Satisfiability.*

[4] Tomas Balyo and Pavel Surynek. *Efektivni heuristika pro sat zalozena na znalosti komponent souvislosti grafu problemu.* 2009.

[5] Tomáš Balyo. Solving boolean satisfiability problems. Master's thesis, 2010.

[6] David G. Mitchell a Hector J. Levesque Bart Selman. *Generating hard satisfiability problems.* 1993.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[8] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[9] doc. RNDr. Eduard Toman CSc. *Vybrané partie z logiky*, volume 46. 2005.

[10] Jan Fagerberg, David C. Mowery, and Richard R. Nelson. Satisfiability solvers. chapter 2, pages 89–122. 2008.

[11] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010.

[12] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[13] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discret. Appl. Math.*, 155(12):1549–1561, 2007.

[14] O. Goldreich. *P, NP, and NP-Completeness: The Basics of Computational Complexity.* Cambridge University Press, 2010.

[15] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.

[16] John R. Koza. Introduction to genetic programming tutorial: from the basics to human-competitive results. In Martin Pelikan and Jürgen Branke, editors, *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010, Companion Material*, pages 2137–2262. ACM, 2010.

[17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[18] Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 141–161. CRC Press, 1997.

[19] prof. RNDr. Pavol Ďuriš CSc. *Výpočtová zložitosť*, volume 36. 2003.

[20] Balamurugan Rengeswaran, A. M. Natarajan, and K. Premalatha. Stellar-mass black hole optimization for biclustering microarray gene expression data. *Applied Artificial Intelligence*, 29(4):353–381, 2015.

[21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern approach.* 2003.

[22] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In William R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992*, pages 440–446. AAAI Press / The MIT Press, 1992.

[23] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996.

[24] Vijay V. Vazirani. *Approximation algorithms.* Springer, 2001.

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód programu a súbory s výsledkami experimentov. Zdrojový kód je zverejnený aj na stránke `https://github.com/Riko196/SAT-solver-analyzer`.