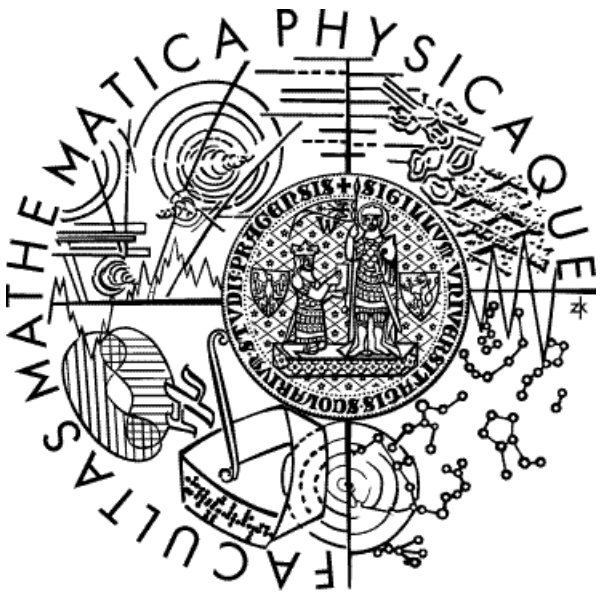
Univerzita Karlova v Praze
Matematicko – fyzikální fakulta

BAKALÁRSKA PRÁCA



Tomáš Balyo

Efektívny riešič problémov boolovskej splniteľnosti

Katedra teoretickej informatiky a matematickej logiky

Vedúci bakalárskej práce: RNDr. Pavel Surynek

Študíjny program: Informatika, obecná informatika

2008

Na tomto mieste by som sa chcel poďakovať vedúcemu tejto práce, pánovi RNDr. Pavlovi Surynkovi za odbornú pomoc a čas, ktorý mi s ochotou venoval.

Prehlasujem, že som túto bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 22.5.2008 Tomáš Balyo

Obsah

1 Úvod....5
2 Algoritmy a návrh riešiča....6
2.1 Problém boolovskej splniteľnosti....6
2.2 Konjunktívny normálny tvar....7
2.3 Algoritmus DPLL....8
2.4 Princíp dvoch sledovaných literálov....10
2.5 Algoritmus riešiča Satsuki....12
2.6 Algoritmus riešiča Mei....16
2.7 Komponenty súvislosti vo formule....18
2.8 Rozhodovacie heuristiky....19
2.9 Heuristika použitá v našich riešičoch....20
3 Implementácia riešiča....23
3.1 Formát DIMACS pre CNF SAT....23
3.2 Reprezentácia formule a načítanie vstupného súboru....23
3.3 Reprezentácia stavu v Satsuki....24
3.4 Reprezentácia stavu v Mei....25
3.5 Implementácia funkcií....26
3.6 Programovanie s ohľadom na cache....26
4 Typy formúl a výkon riešičov na nich....28
4.1 Fázový prechod....28
4.2 Ťažké náhodne generované problémy....29
4.3 Holubníkový princíp....30
4.3 Pole logických členov programovateľné užívateľom....31
4.4 Formule vyriešené nízkym počtom rozhodnutí....31
4.5 Záver k experimentálnej časti....33
5 Záver....34
Literatúra....35

Názov práce: Efektívny riešič problémov boolovskej splniteľnosti
Autor: Tomáš Balyo
Katedra: Katedra teoretickej informatiky a matematickej logiky
Vedúci bakalárskej práce: RNDr. Pavel Surynek
e-mail vedúceho: surynek@ktiml.mff.cuni.cz

Abstrakt: Problém boolovskej splniteľnosti je jedným z najdôležitejších a najviac študovaných problémov umelej inteligencie. Mnoho teoretických a priemyselných problémov sa prevádza na SAT a rieši sa pomocou riešiča SAT. V rámci tejto práce boli vyvinuté dva riešiče. Budú popísane a porovnané s dnešnými najznámejšími riešičmi. Predstavené riešiče sú vo viacerých prípadoch rýchlejšie než víťaz súťaže SAT competition z roku 2007 – R-sat a ďalšie známe riešiče. Za svoju rýchlosť tieto dva riešiče vďačia použitým implementačným technikám a svojej heuristike výberu premennej, ktorá je veľmi výkonná a pri tom má nulovú réžiu.

Kľúčové slová: splniteľnosť, riešič, heuristika

Title: Effective SAT solver
Author: Tomáš Balyo
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Pavel Surynek
Supervisor's e-mail: surynek@ktiml.mff.cuni.cz

Abstract: The boolean satisfaction problem (SAT) is one of the most important and most studied problems of artificial intelligence. Many theoretical and real life problems are transformed to SAT and solved using a SAT solver. During the research for this thesis two SAT solvers were developed. They are described and compared with several of today's state-of-art SAT solvers. The presented solvers in many cases outperform the 2007's SAT competition winner R-sat and other famous SAT solvers. These two new solvers thank their speed to the implementation techniques used and to their decision heuristic, which is very efficient while absolutely costless.

Keywords: satisfiability, solver, heuristic

1 Úvod

Problém boolovskej splniteľnosti (SAT[1]) je jedným z najuniverzálnejších a najznámejších predstaviteľov triedy NP-úplných problémov [1,2]. V teoretickej informatike často dokazujeme príslušnosť úlohy do triedy NP-úplných práve prevodom SAT na danú úlohu. Mnohé úlohy z oblasti umelej inteligencie sa v súčasnosti najrýchlejšie riešia prevodom na SAT a následným použitím SAT solveru. Sú to napríklad plánovanie [4] a automatické dokazovanie viet [5]. SAT solvery sa používajú tiež na formálnu verifikáciu software a hardware [6]. Konkrétne napríklad v elektrotechnickom priemysle sa SAT používa pri návrhu mikročipov na dokazovanie správnosti obvodov [11]. SAT je veľmi úzko spätý s ďalšou významnou oblasťou umelej inteligencie a to s programovaním s obmedzujúcimi podmienkami (CSP[2])[3]. Mnohokrát technika alebo heuristika objavená v CSP je aplikovateľná a veľmi účinná a efektívna v SAT a naopak.

Práve vďaka veľkej užitočnosti tohoto problému vznikalo a dodnes vzniká mnoho riešičov. Vznikajú nové techniky, metódy a implementačné obraty pre ich urýchlenie. Existujú súťaže SAT Competition [8] (koná sa každý nepárny rok) a SAT Race [7] (koná sa každý párny rok), v ktorých si riešiče merajú sily v rôznych kategóriach problémov. Tie kategórie sú napríklad formule náhodne generované a formule z priemyselnej praxe, podkategórie splniteľné a nesplniteľné. Dnešné riešiče dokážu vyriešiť obrovské formule s počtom premenných v ráde stotisícok a počtom klauzúl[3] v ráde miliónov.

Cieľom tejto bakalárskej práce bolo preštudovať možnosti návrhu efektívneho riešiča a využiť získané poznatky pri implementácií konkrétneho riešiča, ktorý zodpovedá požiadavkám súťaží SAT competition a SAT Race. Vznikli dva riešiče, sú pomenované Satsuki a Mei[4]. Solver Satsuki vznikol prvý a je zložitejší a väčší ako Mei. Obe sú ale veľmi podobné a používajú rovnakú heuristiku. Táto heuristika má vynikajúce vlastnosti a je pravdepodobne najväčším prínosom tejto práce.

Organizácia textu je nasledujúca. V druhej kapitole sú popísané obecné metódy a algoritmy, ktoré sa používajú pri programovaní riešičov. Sú popísane algoritmy riešičov Satsuki a Mei a tiež heuristika ktorú používajú. Tretia kapitola sa zaoberá konkrétnou implementáciou riešičov v jazyku C++ a C. Sú tu popísane implementačné metódy, ktoré boli použité a tiež úvaha o tom, ako sa riešiče správajú voči vyrovnávacej pamäti procesoru – procesorovej cache. Vo štvrtej kapitole budú preberané rôzne typy formúl a hneď na nich testované a porovnávané riešiče Satsuki a Mei navzájom a aj s inými riešičmi ako sú R-Sat [12, 23], zChaff [13, 21], Picosat [14] a MiniSat [15, 28]. Posledná kapitola je venovaná záveru a naznačeniu smeru, ktorým by sa ďalšia práca pri vytváraní lepších a rýchlejších riešičov mala uberať.

1 SAT – SATisfiability problem. Je to hľadanie splňujúceho ohodnotenia boolovskej formule.
2 CSP – Constraint Satisfaction Problem
3 Formule sú v konjunktívnom normálnom tvare (CNF)
4 Sú to mená dvoch sestier, hlavných postáv z animovaného filmu My neighbor Totoro [9]. Obe slová znamenajú Máj.

2 Algoritmy a návrh riešiča

2.1 Problém boolovskej splniteľnosti

V úvode tejto kapitoly si formálne definujeme problém boolovskej splniteľnosti. Na to potrebujeme zaviesť niekoľko pojmov z oblasti výrokovej logiky. Jazyk výrokovej logiky obsahuje prvotné formule (napríklad a, b, x_1, x_2, x_3), logické spojky $\neg$ (negácia), $\wedge$ (konjunkcia), $\vee$ (disjunkcia), $\Rightarrow$ (implikácia) a $\Leftrightarrow$ (ekvivalencia) a pomocné symboly (zátvorky). Prvotné formule sa tiež označujú ako výrokové premenné.

Formula výrokovej logiky je definovaná pomocou nasledujúcich syntaktických pravidiel:

1. Každá prvotná formula (výroková premenná) je formula.
2. Ak A a B sú formule, tak $\neg A, (A \wedge B), (A \vee B), (A \Rightarrow B), (A \Leftrightarrow B)$ sú tiež formule.
3. Každá formula vzniká konečným použitím pravidiel (1.) a (2.).

Nech P je množina prvotných formúl. Množina pravdivostných hodnôt pozostáva z dvoch prvkov: True (1) a False (0). Pravdivostným ohodnotením nazveme zobrazenie $f : P \rightarrow \{0,1\}$ z množiny prvotných formúl do množiny pravdivostných hodnôt. Pravdivostné ohodnotenie tiež označujeme slovom model. Čiastočným modelom nazývame zobrazenie z podmnožiny P do {0, 1}, ohodnotíme teda len niektoré výrokové premenné, pravdivostné hodnoty ostatných nedefinujeme.

Pravdivostné ohodnotenie f prvotných formúl sa dá rozšíriť na pravdivostné ohodnotenie f^* všetkých formúl v danom jazyku. Indukciou podľa zložitosti formule definujeme rozšírenie zobrazenia f na f^* nasledovne:

$f^*(A)$	$= f(A)$	Ak je A prvotná formula
$f^*(\neg A)$	$= 1$	Ak $f^*(A) = 0$
	$= 0$	inak
$f^*(A \wedge B)$	$= 1$	Ak $f^*(A) = 1$ a $f^*(B) = 1$
	$= 0$	inak
$f^*(A \vee B)$	$= 0$	Ak $f^*(A) = 0$ a $f^*(B) = 0$
	$= 1$	inak
$f^*(A \Rightarrow B)$	$= 0$	Ak $f^*(A) = 1$ a $f^*(B) = 0$
	$= 1$	inak
$f^*(A \Leftrightarrow B)$	$= 1$	Ak $f^*(A) = f^*(B)$
	$= 0$	inak

Problém boolovskej splniteľnosti (SAT) je rozhodovací problém. Našou úlohou je pre danú výrokovú formulu F rozhodnúť, či existuje také ohodnotenie f výrokových premenných, že $f^*(F)$ má hodnotu 1, teda True. Ak existuje také

ohodnotenie, tak hovoríme, že formula je splniteľná. V opačnom prípade je formula nesplniteľná. Problém boolovskej splniteľnosti je NP-úplný [1] a zatiaľ nebol objavený deterministický algoritmus, ktorý ho rieši v polynomiálnom čase.

2.2 Konjunktívny normálny tvar

Logické spojky konjunkcia a disjunkcia majú niekoľko pekných vlastností. Sú komutatívne, asociatívne a distributívne [16]. Implikácia a ekvivalencia tieto vlastnosti nemajú, preto by určite bolo výhodne ich vynechať. Máme to šťastie, že to môžeme urobiť. Formule ktoré sú tvaru $(A \Rightarrow B)$ a $(A \Leftrightarrow B)$ môžeme prepísať na $\neg A \vee B$ a $(\neg A \vee B) \wedge (\neg B \vee A)$, teda do tvaru, kde máme len konjunkcie, disjunkcie a negácie. Môžeme ísť ešte ďalej. Každú formulu je možné previesť na takzvaný konjunktívny normálny tvar. Návod na prevod a dôkaz v [16].

Formula je v konjunktívnom normálnom tvare ak je konjunkciou klauzúl. Klauzula je disjunkcia literálov. Literál je výroková premenná alebo jej negácia. Príklad je môžné vidieť na obrázku 1. Konjunktívny normálny tvar sa zvyčajne označuje anglickou skratkou CNF[1]. V celej práci budeme predpokladať, že vstupná formula je daná vo formáte CNF. Preto by sa správne táto úloha mala označovať CNF SAT. Keďže však SAT je kratšie, bude sa uvádzať len SAT a myslí sa vždy CNF SAT.

x_1, x_2	výrokové premenné
$\neg x_1, x_2, x_1, \neg x_2$	literály
$(x_1 \vee \neg x_2), (\neg x_2), (x_1 \vee \neg x_2 \vee x_3)$	klauzule
$(x_1 \vee \neg x_2) \wedge (\neg x_2) \wedge (x_1 \vee \neg x_2 \vee x_3)$	formula v tvare CNF

Obrázok 1: Formula v konjunktívnom normálnom tvare.

Formát CNF je veľmi užitočný lebo je jednoduchý a dobre sa reprezentuje v pamäti počítača. Dovoľuje nám jednoduchšie formulovať úlohu splniteľnosti. Úlohou je nájsť ohodnotenie, pri ktorom je splnená každá klauzula z nejakej množiny klauzúl. Klauzula je splnená, ak má aspoň jeden jej literál pri danom ohodnotení premenných hodnotu true. Ak takéto ohodnotenie neexistuje, formula je nesplniteľná.

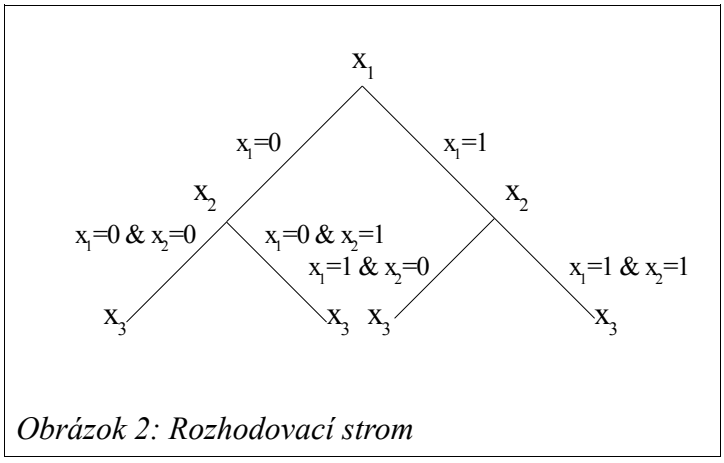
Formát CNF je všeobecné známy a používaný. Všetky známe riešiče vyžadujú vstup vo formáte CNF. Formule prístupne na internete [17] a tiež formule v súťažiach [7,8] sú vždy v tomto tvare.

1 CNF – Conjunctive normal form

2.3 Algoritmus DPLL

Väčšina moderných riešičov používa ako svoj základ algoritmus DPLL[1]. Algoritmus bol prvýkrát popísaný v [18] a [19]. Vstupom je formula v tvare CNF a výstupom je model formule alebo odpoveď nesplniteľná. Základom je prehľadávanie do hĺbky, kde postupne skúšame všetky možné hodnoty premenných a testujeme či už je daná formula splnená. Možné hodnoty premenných sú len dve, preto priestor, ktorý prehľadávame je binárny strom, viď obrázok 2. DPLL pridáva navyše tri vylepšenia:

- **Včasné zastavenie prehľadávania**. Algoritmus môže prehľadávanie nejakej vetvy zastaviť oveľa skôr než sú ohodnotené všetky premenné ak už je známe, že formula je pri súčasnom čiastočnom modeli nesplnená alebo je celá splnená. Akonáhle sa objaví nesplnená klauzula okamžite sa môžeme vrátiť o vrstvu vyššie a tento podstrom zahodiť.
- **Heuristika čistej premennej** (pure symbol). Ak sa nejaká premenná vyskytuje vždy len kladne alebo vždy len záporne, môžeme túto premennú okamžite ohodnotiť hodnotou true respektíve false. Určite nič nestratíme, pretože toto priradenie nemôže spôsobiť to, že sa niektorá klauzula stane nesplnenou. Ak je teda niektorá premenná od začiatku čistá, alebo sa ňou stane počas výpočtu, môžeme ju ohodnotiť a vyhľadávací strom sa zmenší.
- **Heuristika jednotkovej klauzule** (unit clause). Ak máme klauzulu, ktorá má len jeden literál, tak tento literál musíme ohodnotiť na hodnotu true, musíme teda príslušnej premennej priradiť správnu hodnotu. Je to nutné, pretože chceme, aby všetky klauzule boli splnené. Dosadením hodnoty premennej môžu vzniknúť ďalšie jednotkové klauzule, ktoré implikujú ďalšie dosadenia do premenných. Takto sa to môže kaskádovito šíriť ďalej. Tento jav sa nazýva propagácia jednotky (unit propagation).



Obrázok 2: Rozhodovací strom

Algoritmus 1 je pseudokód DPLL, ktorý je prebraný z [10]. Ak sa pozrieme na funkciu DPLL, prvé dva riadky zodpovedajú vylepšeniu s včasným zastavením rekurzie. Funkcie FIND-PURE-SYMBOL a FIND-UNIT-CLAUSE vracajú jednu premennú a hodnotu, ktorá ma byť do nej dosadená. Samozrejme môže sa stať, aj sa

1 DPLL – Davis, Putnam, Logemann, Loveland

stáva, že existuje viac čistých symbolov a jednotkových klauzúl, ale v tomto pseudokóde nám stačí naraz jedna premenná. Ak neexistuje čistý symbol alebo jednotková klauzula, tieto funkcie vracajú hodnotu null. Funkcia EXTEND rozšíri čiastočný model priradením hodnoty value premennej P a vráti nový čiastočný model. Môžeme si všimnúť, že keď neexistuje ani čistá premenná ani jednotková klauzula, tak sa vyberie prvá zatiaľ neohodnotená premenná a rekurzívne skúšame hľadať riešenie s hodnotou true a s hodnotou false tejto premennej.

Algoritmus 1: DPLL

```
function DPLL-Satisfiable?(s) returns true or false
        inputs: s, a sentence in propositional logic
        clauses := the set of clauses int the CNF representation of s
        symbols := a list of proposition symbols in s
        return DPLL(clauses, symbols, [])

function DPLL(clauses, symbols, model) returns true or false
        if every clause in clauses is true in model then return true
        if some clause in clauses is false in model then return false
        P, value := FIND-PURE-SYMBOL(clauses, symbols, model)
        if P is non-null then return DPLL(clauses, symbols – P,
                                          EXTEND(P, value, model))
        P, value := FIND-UNIT-CLAUSE(clauses, model)
        if P is non-null then return DPLL(clauses, symbols – P,
                                          EXTEND(P, value, model))
        P := FIRST(symbols); rest := REST(symbols)
        return DPLL(clauses, rest, EXTEND(P, true, model)) or
                DPLL(clauses, rest, EXTEND(P, false, model))
```

Skutočné moderné riešiče rozširujú tento základný algoritmus o mnohé vylepšenia ako napríklad učenie sa klauzúl, reštarty, rezolúcia a nechronologický backtracking. Používajú sa špeciálne dátové štruktúry a techniky ako je aj princíp dvoch sledovaných literálov, ktorý je popísaný nižšie. Tieto vylepšenia nie sú špecifické pre jednotlivé riešiče, všetky dobré riešiče používajú všetky zmienené vylepšenia.

Moderné riešiče DPLL nie len rozširujú, ale aj zjednodušujú. Heuristiku čistej premennej dnes používa máloktorý riešič. Z dvoch predstavených riešičov v tejto práci ho používa len jeden.

V pseudokóde sme si vyberali prvú vhodnú (zatiaľ neohodnotenú) premennú, skutočné riešiče ale používajú veľmi zložité heuristiky pre výber. Tieto heuristiky potom tiež vyžadujú špeciálne metódy a dátové štruktúry aby sa mohli rýchlo spočítať. Napríklad riešič Picosat [14] implementuje vlastný modul pre počítanie s číslami s plávajúcou desatinnou čiarkou, ktoré používa pre výber premennej, aby bol výsledok na každej platforme rovnaký. Výberu premennej je venovaná podkapitola 2.8.

2.4 Princíp dvoch sledovaných literálov

Riešič musí po každom ohodnotení nejakej premennej zisťovať, či nevznikne jednotková klauzula – klauzula s jediným literálom l. Ak vznikne, tak tento literál l ohodnotíme na hodnotu true. Dôsledkom ohodnotenia literálu l môže byť, že niektorá klauzula, ktorá obsahuje opačný literál, teda $\neg l$ sa stane jednotkovou a vyvolá ďalšie ohodnotenie. Takto môžu kaskádovito vznikať jednotkové klauzule a ohodnotenia. Tento proces je propagácia jednotky.

Väčšinu behového času trávi SAT solver práve propagáciou jednotky. Je preto veľmi dôležité optimalizovať túto časť programu.

Najjednoduchší spôsob ako implementovať zisťovanie jednotkových klauzúl je počítanie literálov. Pre každú klauzulu máme číslo, v ktorom si pamätáme počet tých literálov, ktoré ešte nie sú ohodnotené. Vždy keď sa nejaký literál ohodnotí záporne, tak znížime počítadlo o jedna. Ak sa niektorý literál ohodnotí kladne, tak táto klauzula je splnená a nemusíme sa žiadnym počítaním zaoberať. Vždy po znížení sa pozrieme, či hodnota počítadla nie je jedna, ak je to tak, našli sme jednotkovú klauzulu.

Počítanie literálov je jednoduché implementovať a na prvý pohľad vyzerá ako efektívna metóda. Riešiče do roku 2001 používali tento princíp, bol to napríklad GRASP [20]. Tento prístup má ale niekoľko nevýhod. Vezmime príklad, že sme ohodnotili premennú p kladne, musíme sa pozrieť do každej klauzule kde sa p vyskytuje ako záporný literál a znížiť čítač o jedna. Takých klauzúl môže byť veľa a mnohé z nich navštevujeme často zbytočne. Druhá a oveľa závažnejšia nevýhoda je, že pri vracaní sa v backtrackingu musíme počítadlá nastavovať späť na pôvodnú hodnotu. To môže byť veľa práce ak pri propagácií vznikal veľký počet jednotkových klauzúl, čo vyvolávalo ohodnotenia a teda ďalšie propagácie.

Pri počítaní literálov sme sledovali všetky literály každej klauzule. V roku 2001 prišli autori riešiča Chaff [21] s myšlienkou, že stačí sledovať 2 literály. Prečo stačí sledovať len 2 literály? Pokiaľ v nejakej klauzule vieme o 2 dvoch literáloch, že ešte nie sú ohodnotené, tak tá klauzula nemôže byť jednotková. Vyberieme si z každej klauzuly 2 literály a keď sa nejaká premenná ohodnotí napríklad kladne, tak sa pozrieme či jej negatívny literál nie je v niektorej klauzule sledovaný. Všade, kde ten negatívny literál bol sledovaný, musíme nájsť za neho náhradníka, nový neohodnotený literál na sledovanie. Ak sa nám toto nepodarí, tak klauzula je jednotková a posledným literálom je práve ten druhý zo sledovaných. Príklad je zobrazený na obrázku 3.

Prvá výhoda dvoch sledovaných literálov oproti počítaniu literálov je, že pri ohodnotení premennej navštevujeme len tie klauzule, kde máme sledovaný výskyt. Oproti návšteve všetkých klauzúl kde máme výskyt to môže znamenať veľkú úsporu času. Druhá a omnoho signifikantnejšia výhoda je výborné správanie sa voči backtrackingu. Pri vracaní sa späť pri počítaných literáloch sme museli počítadlá nastavovať na pôvodnú hodnotu, pri dvoch sledovaných literáloch nemusíme robiť

vôbec nič. Literály, ktoré sú dobré na sledovanie vo väčšej hĺbke prehľadávacieho stromu (obrázok 1) sú dobré aj v nižšej hĺbke. Literál je vhodný na sledovanie ak je neohodnotený. Literál, ktorý je neohodnotený vo väčšej hĺbke je určite neohodnotený aj v nižšej.

// na každom riadku je jedna klauzula, sledované literály sú zobrazené tučne, zmeny sú podčiarknuté.

1: ¬x1, ¬**x2**, x4, ¬**x5**, ¬x6

2: ¬x2, ¬**x5**, ¬**x6**

3: ¬**x4**, ¬**x3**, ¬x1, x2, x5

4: **x4, x3**

Ohodnotili sme x2 = True, ¬x2 je sledovaný v 1. klauzule.

1: ¬x1, <u>¬x2</u>, <u>**x4**</u>, ¬**x5**, ¬x6 // namiesto x2 budeme sledovať x4

2: ¬x2, ¬**x5**, ¬**x6**

3: ¬**x4**, ¬**x3**, ¬x1, x2, x5

4: **x4, x3**

Ohodnotili sme x3 = True, 4. klazula je splnená, môžeme ju vypustiť, v tretej klazule bol ¬x3 sledovaný.

1: ¬x1, ¬x2, **x4**, ¬**x5**, ¬x6

2: ¬x2, ¬**x5**, ¬**x6**

3: ¬**x4**, <u>¬x3</u>, <u>¬**x1**</u>, x2, x5 // namieto ¬x3 sledujeme ¬x1

Ohodnotili sme x5 = True, 3. klauzula je splnená, v 2. klauzule je ¬x5 sledovaný.

1: ¬x1, ¬x2, **x4**, ¬**x5**, ¬x6

2: ¬x2, ¬x5, ¬**x6** // ¬x2 nemôže byť sledovaný namiesto ¬x5, pretože x2 už je ohodnotený. Ina možnosť na náhradníka nie je, takže klazula je jednotková. Druhý sledovaný literál je ¬x6, je posledný, musíme ho splniť.

Odvodili sme x6 = False, tým je splnená aj prvá klazula, sme hotoví.

Obrázok 3: Ilustrácia princípu dvoch sledovaných literálov.

V príklade na obrázku 3 sme vypúšťali klauzule z formule. V programe by sme to mohli realizovať tak, že by sme si držali len platné klauzule v nejakej štruktúre a klauzule odtiaľ vyberali a pri backtrackingu vracali. Druhá a lepšia možnosť je klauzule si pamätať vždy všetky a mať vlastnosť, či je tá ktorá klauzula ešte potrebná. Pri počítaných literáloch môže tento príznak niesť počítadlo aktívnych klauzúl, napríklad hodnotou 0 pre neaktívnu klauzulu. Podobné niečo by sme dokázali previesť aj pri dvoch sledovaných literáloch, napríklad nastavením prvého sledovaného literálu na neplatnú hodnotu, no bola by to veľmi zlá chyba. Prišli by sme totiž o najlepšiu vlastnosť dvoch sledovaných literálov – dobré správanie sa pri backtrackingu. Pri vracaní by sme museli zisťovať ktoré klauzule sú opäť aktívne a vyberať si v nich literály pre sledovanie.

2.5 Algoritmus riešiča Satsuki

V prvom odstavci stručne vymenujeme metódy ktoré riešič Satsuki používa a v ďalších sa k nim vrátime podrobnejšie. Satsuki je založený na DPLL. Používa aj heuristiku čistých premenných, i keď to dnes u moderných riešičov nie je obvyklé. Na propagáciu jednotky používa metódu, ktorá vychádza z dvoch sledovaných literálov, ale trošku sa odlišuje. Satsuki sa reštartuje po pevnom počte konfliktov a náhodne zvolí premennú, ktorou pokračuje. Rozhodovacej heuristike sú venované dve posledné podkapitoly tejto kapitoly.

Vo výpise algoritmus 2 je zobrazený pseudokód hlavnej časti riešiča Satsuki. Je vidieť, že je to v skutočnosti DPLL (algoritmus 1), pričom je odstránená rekurzia a nahradená cyklom while. Premenná *var* obsahuje premennú, ktorú budeme ohodnocovať. Ak má hodnotu 0, tak to znamená, že nie je vybraná žiadna premenná. Premenná *current_state* obsahuje stav hľadania, konkrétne obsahuje tieto informácie: počet nesplnených klauzúl, hodnoty premenných (true, false, undefined), informáciu o aktivite klauzúl a počty kladných a záporných výskytov premenných.

Program nastaví hodnotu *var* na nulu a vstupuje do while cyklu, ktorý trvá dovtedy kým existujú nesplnené klauzule. Funkcia DECISION na základe súčasného stavu *current_state* rozhodne, že ktorú, ešte neohodnotenú, premennú formule *f* budeme teraz ohodnocovať. Súčasný stav *current_state* si zálohujeme použitím SAVE-STATE, aby sme sa ku nemu mohli neskôr, ak to bude potrebné, vrátiť. Stav sme si zálohovali, takže teraz si ho môžeme dovoliť „pokaziť“. Vykonáme rozhodnutie a zvolenú premennú ohodnotíme na hodnotu true alebo false, podľa toho čo obsahuje konštanta FIRST. Konštanta SECOND obsahuje opačnú hodnotu. Zmysel týchto konštánt je teda to, že určujú, ktorú hodnotu budeme skúšať ako prvú a ktorú ako druhú. Funkcia VALUATE nastaví v danom stave hodnotu danej premennej na požadovanú hodnotu. Premennú, ktorú sme si zvolili na ohodnotenie uložíme do zásobníka *decision_stack*.

PROCESS-SUBST je kľúčová funkcia algoritmu. Vykonáva akcie, ktoré vyplývajú z ohodnotenia nejakej premennej, vracia false, ak nastal konflikt, inak vracia true. Konflikt znamená, že niektorá klauzula je nesplnená. Táto funkcia bude podrobnejšie popísaná v ďalších odstavcoch. Ak PROCESS-SUBST vráti true tak pokračujeme ďalším rozhodnutím, ak vráti false, tak sa musíme vracať dozadu. Zo zásobníku rozhodnutí *decision_stack* si vyberieme tú premennú, ktorá bola pridaná čo najneskôr a má hodnotu FIRST. Toto vykonáva funkcia LATEST-WITH-FIRST. Ak sa taká nenájde, tak všetky majú hodnotu SECOND a to znamená, že sme vyskúšali všetky možnosti a formula je nesplniteľná. Ak sa nájde, vrátime sa do stavu pred tým ako sme túto premennú ohodnotili na FIRST. Ten stav sme si uložili použitím SAVE-STATE a teraz sa ku nemu vrátime zavolaním RESTORE-STATE a skúsime to s hodnotou SECOND.

Algoritmus 2: Pseudokód Satsuki, funkcia SATISFIABLE?

```
function SATISFIABLE?( f ) returns true or false
        inputs  f a formula in CNF
        var := 0
        while(NUM-UNSAT-CLAUSES(current_state) > 0)
                if var = 0 then
                        var := DECISION(current_state, f )
                        SAVE-STATE(current_state)
                        VALUATE(current_state, var, FIRST)
                        ADD(decision_stack, var)
                if PROCESS-SUBST(f, current_state, var) = true then
                        var := 0
                else
                        var := LATEST-WITH-FIRST(decision_stack)
                        if  var = NULL then return false
                        current_state := RESTORE-STATE(var)
                        VALUATE(current_state, var, SECOND)
        return true
```

Algoritmus 3 je pseudokód už zmienenej funkcie PROCESS-SUBST a ďalších dôležitých funkcií, ktoré používa. Na začiatku deklarujeme zásobník *var_stack*, ktorý obsahuje jedinú premennú *var*, ktorú dostala funkcia ako parameter. Program ďalej vstupuje do while cyklu, ktorý trvá dovtedy, kým sa *var_stack* nevyprázdni. Pri prvom priechode obsahuje jeden prvok a to *var*. Zo zásobníka *var_stack* si následne vyberieme naposledy pridanú premennú a budeme sa venovať akciám, ktoré implikuje jej ohodnotenie.

Ak má daná premenná hodnotu true tak „vypneme“ (TURN-OFF-CLAUSES) všetky klauzule, kde má kladný výskyt (POS-OCC), pretože tieto klauzule sú tým pádom splnené. Funkcia POS-OCC vracia množinu klauzúl formule *f*, v ktorých má premenná *var* kladný výskyt, teda kladný literál. Funkcia NEG-OCC funguje obdobne, vracia množinu klauzúl, kde má premenná záporný výskyt. Funkcia TURN-OFF-CLAUSES dostáva ako parameter aj *var_stack* aby tam mohla pridávať premenné, ktorých hodnota sa odvodí. K tejto funkcií sa ešte v ďalšom vrátime. Funkciou TEST-UNIT testujeme, či sa niektorá klauzula z klauzúl, kde má *var* záporný výskyt nestala jednotkovou. Ak áno tak sa odvodí hodnota nejakej premennej a tá premenná sa vloží do *var_stack*. Táto funkcia tiež detekuje konflikty. Vracia false ak nastal konflikt, inak vracia true. Ak sa vráti hodnota false, tak je potrebné ukončiť PROCESS-SUBST a vrátiť false, pretože už je jasné, že náš čiastočný model nesplňuje formulu. Funkciu TEST-UNIT ešte samozrejme popíšeme podrobnejšie neskôr. Ak má daná premenná hodnotu false, postupujeme opačne, teda vypíname klauzule so záporným výskytom a testujeme jednotkovosť klauzúl s kladným výskytom.

Algoritmus 3: ***Satsuki****, funkcia PROCESS-SUBST*

```
function PROCESS-SUBST( f, st, var) returns true or false
        var_stk := {var}
        while(NOT-EMPTY(var_stk))
                var := PICK-LAST(var_stk)
                if  VALUE(var, st) = true then
                   TURN-OFF-CLAUSES(POS-OCC( f, var), st, var_stk)
                   if  TEST-UNIT(NEG-OCC( f, var), st, var_stk, var) = false
                        then return false
                else
                   TURN-OFF-CLAUSES(NEG-OCC( f, var), st, var_stk)
                   if  TEST-UNIT(POS-OCC( f, var), st, var_stk, var) = false
                        then return false
        return true

function TURN-OFF-CLAUSES(clauses, state, var_stack)
        foreach (cl in clauses) do
                if  INACTIVE(state, cl) then continue
                MAKE-INACTIVE(cl, state)
                DEC-NUM-UNSAT-CLAUSES(state)
                foreach(lit literal in cl) do
                        if  VALUE(lit, state) = undefined then
                            DEC-OCCURENCE-COUNT(state, lit)
                            if  OCCURENCE-COUNT(state, lit) = 0 then
                                VALUATE(state, VAR(lit), ¬SIGN(lit))
                                ADD(var_stk, VAR(lit))

function TEST-UNIT(clauses, state, var_stack, var) returns true or false
        foreach (cl in clauses) do
                if  ACTIVE(state, cl) and WATCHED(var, cl) then
                        if  FIND-OTHER-TO-WATCH(state, cl)
                                then continue
                        MAKE-INACTIVE(state, cl)
                        lit = ONLY-LITERAL(cl, state)
                        if  VALUE(lit, state) = true then continue
                        if  VALUE(lit, state) = false then return false
                        VALUATE(state, VAR(lit), SIGN(lit))
                        ADD(var_stack, VAR(lit))
        return true
```

Pseudokód TURN-OFF-CLAUSES je zobrazený vo výpise algoritmus 3. Funkcia vypína klauzule, čo v podstate znamená, že ich odoberá z formule. V skutočnosti sa ale formula nemení, mení sa len stav *state*. Ako už bolo uvedené, stav obsahuje informácie o aktivite klauzúl a tieto informácie sa menia. Funkcia sa

tiež stará o počítanie výskytov premenných. Literálom, ktoré sa nachádzajú vo vypínanej klauzule a ešte nie sú ohodnotené, znižuje počítadlo výskytov. Ak sa počet výskytov nejakého literálu zníži na nula, tak to znamená, že premenná príslušná tomuto literálu sa vo formule vyskytuje buď len kladne alebo len záporne a môžeme ju ohodnotiť. Na tomto mieste teda požívame heuristiku čistých premenných. Funkcia VAR vráti premennú príslušnú literálu, funkcia SIGN vracia true ak literál je kladný, inak false. Premennú v aktuálnom stave ohodnotíme ale nezačneme vykonávať akcie, ktoré z toho vyplývajú. Vložíme ju do štruktúry *var_stack* a potom vo funkcii PROCESS-SUBST sa k nej vrátime a urobíme potrebné operácie.

Funkcia TEST-UNIT implementuje heuristiku jednotkových klauzúl. Berie ako parameter množinu klauzúl *clauses* a premennú *var*, ktorej ohodnotenie spracovávame. Funkcia prechádza každú zadanú klauzulu a testuje najprv či je ešte aktívna a potom, či je premenná *var* v nej sledovaná. Ak nie je sledovaná môžeme isť ďalej k nasledujúcej klauzule. Ak premenná *var* sledovaná je, tak sa pokúsime nájsť náhradníka. Funkcia FIND-OTHER-TO-WATCH nahradenie sledovaného literálu vykoná, ak je to možné, a vráti true. Ak to možné nie je, tak vráti false. Ak sa vráti true, môžeme pokračovať nasledujúcou klauzulou, ak sa vráti false, narazili sme na jednotkovú klauzulu. Klauzulu deaktivujeme. Do premennej *lit* uložíme jediný literál, ktorý ostal. Otestujeme či tento literál už nemá hodnotu true, ak je to tak, odvodili sme niečo, čo sme už vedeli a môžeme pokračovať. Ak má literál hodnotu false, tak máme konflikt. Túto klauzulu už nie je možne splniť, nezostal v nej žiadny literál, ktorý by mohol byť pravdivý. Vrátime preto hodnotu false. Ak nenastala ani jedna z týchto možnosti, tak posledný literál je ešte neohodnotený a môžeme ho ohodnotiť. Ohodnotíme príslušnú premennú tak, aby bol literál *lit* pravdivý a pridáme ju do *var_stack* aby bolo jej ohodnotenie vo funkcii PROCESS-SUBST neskôr spracované.

Počas popisu funkcií TEST-UNIT a TURN-OFF-CLAUSES sme zdôrazňovali, že keď odvodíme hodnotu nejakej premennej tak nezačneme hneď vykonávať akcie, ktoré z toho plynú, ale len pridáme premennú na zoznam premenných čakajúcich na spracovanie. Ak by sme tie akcie chceli vykonávať okamžite, nevyhli by sme sa rekurzii. Rekurziu je vždy dobré vynechať, programy, ktoré ju nepoužívajú bežia rýchlejšie, i keď sú väčšinou zložitejšie a menej prehľadné.

Pri riešení formule je veľmi dôležité, že ako si vyberáme premennú, ktorú chceme ohodnotiť. Inak povedané, je dôležite ako funguje funkcia DECISION (viď algoritmus 2). Napriek našim najlepším snahám sa ale môže stať, že sa vyberie zlá premenná, vykoná sa zlé rozhodnutie. Zlé rozhodnutie je také, že nás navedie na cestu, ktorá k cieľu nevedie v prijateľnom čase. So zlými rozhodnutiami sa môžeme vysporiadať tak, že po určitom čase neúspešného hľadania (po určitom počte konfliktov) výpočet reštartujeme. Vtedy zahodíme celý strom čo sme zatiaľ prehľadali. Znie to ako hrozné riešenie, ale napodiv to funguje veľmi dobre a riešič to urýchli aj o rád. Najlepšie riešiče, ktoré vykonávajú aj učenie sa klauzúl, si po reštarte ponechajú naučené klauzule, teda nezačínajú opäť od nuly po reštarte. Naš riešič sa klauzule neučí, ale predsa nezačína vždy, i keď väčšinou áno, úplne od nuly. Ak sme pre prvých *n* premenných už vyskúšali hodnotu FIRST a neviedlo to

k riešeniu, tak stačí reštartovať od *n+1* -vej premennej. Poradie premenných je poradie, v ktorom sme si ich vyberali pre rozhodnutie. Pre príklad si predstavme, že najprv, na úplnom začiatku programu, sme si vybrali x_{12} a dostal hodnotu FIRST. Výpočet sa vrátil, že to nemôže byť takto. Dali sme teda x_{12} hodnotu SECOND a pokračovali sme v hľadaní. Počas hľadania sme sa ale rozhodli, že budeme reštartovať. Prvú premennú, x_{12} môžeme nechať na hodnote SECOND a reštartovať až od ďalšieho rozhodnutia. Ak by náhodou aj druhá premenná mala hodnotu SECOND tak stačí reštartovať od tretieho a tak ďalej. Pre ďalšie príklady viď obrázok 3.

V každom riadku je jedna situácia, v riadku sú hodnoty premenných F (first) alebo S (second)

S, F, F, S, S, S – reštart od 2. premennej

F, S, S, S, S, S – reštart od začiatku

S, S, S, F, S, S – reštart od 4. premennej

Obrázok 3: Reštartovanie

Keďže pri reštarte strácame oveľa viac ako riešiče s učením, prispôsobíme tomu frekvenciu reštartov. Staršie riešiče, ako napríklad GRASP [20], reštartovali po 1000 konfliktoch. Novšie riešiče, napríklad MiniSat [15, 28], reštartujú po 100 konfliktoch. Môže sa zdať zvláštne, že je to takáto pevná konštanta a nezávisí ani od behu programu ani od formule. Existujú samozrejme riešiče, ktoré toto číslo nemajú konštantné. Riešič RSat2 [22] používa zložité úvahy pre stanovenie tohoto čísla. Riešič Satsuki reštartuje po 1500 konfliktoch. Táto konštanta bola stanovená na základe empirických skúseností a v žiadnom prípade nemožno tvrdiť, že je optimálna.

2.6 Algoritmus riešiča Mei

Riešič Mei je veľmi podobný riešiču Satsuki. Odlišuje sa v tom, že nepoužíva heuristiku čistých premenných a jednotkovú propagáciu vykonáva metódou počítania literálov. Mei sa odlišuje aj v reštartovaní. Po reštarte nevolíme premennú úplne náhodne ale používame heuristiku, ktorá súvisí s komponentmi súvislosti. Tomu sa ale budeme venovať až v ďalšej kapitole.

Základná funkcia Mei je algoritmicky úplne rovnaká ako základná funkcia Satsuki (algoritmus 2). Budeme sa preto zaoberať len funkciou PROCESS-SUBST (algoritmus 4). Táto je tiež veľmi podobná tej z riešiča Satsuki (algoritmus 3), ale je vidieť, že je oveľa jednoduchšia. Rozdiely vyplývajú z toho, že sa nemusíme starať o detekciu čistých premenných a namiesto dvoch sledovaných literálov používame jednoduchší spôsob – počítanie literálov. Neaktívnu klauzulu reprezentujeme ako hodnotu nula počítadla aktívnych literálov. Vypínanie klauzúl trvá konštantný čas, sú to len 2 inštrukcie: SET-COUNTER, ktorá nastaví počet aktívnych literálov na hodnotu nula a zníži čítač nesplnených klauzúl – DEC-NUM-UNSAT-CLAUSES.

*Algoritmus 4: **Mei**, funckia PROCESS-SUBST*

```
function PROCESS-SUBST( f, st, var) returns true or false
        var_stk := {var}
        while (NOT-EMPTY(var_stk))
                var := PICK-LAST(var_stk)
                if  VALUE(var, st) = true then
                        foreach (cl in POS-OCC( f, var))
                                SET-COUNTER(st, var, 0)
                                DEC-NUM-UNSAT-CLAUSES(st)
                        foreach(cl in NEG-OCC( f, var))
                                if  DECREASE(st, cl, var_stk) = false
                                        then return false
                else
                        foreach(cl in NEG-OCC( f, var))
                                SET-COUNTER(st, var, 0)
                                DEC-NUM-UNSAT-CLAUSES(st)
                        foreach(cl in POS-OCC( f, var))
                                if  DECREASE(st, cl, var_stk) = false
                                        then return false
        return true

function DECREASE(state, clause, var_stack) returns true or false
        DEC-COUNTER(state, clause)
        if  COUNTER(state, clause) > 1 then return true
        DEC-NUM-UNSAT-CLAUSES(state)
        lit = ONLY-LITERAL(clause, state)
        if  VALUE(lit, state) = true then return true
        if  VALUE(lit, state) = false then return false
        VALUATE(st, VAR(lit), SIGN(lit))
        ADD(var_stack, VAR(lit))
        return true
```

Detekciu jednotkových klauzúl vykonávame vo funkcií DECREASE, ktorej pseudokód je uvedený vo výpise algoritmus 4. Znižujeme v nej čítač aktívnych literálov pre danú klauzulu. Ak je to číslo nad 1, tak to nie je jednotková klauzula a môžeme vrátiť true. Ak je to jedna, tak ešte musíme vykonať zopár jednoduchých operácií. Znížime čítač nesplnených klauzúl a vyšetríme posledný zostávajúci literál. Ak už je v aktuálnom stave posledný literál splnený, odvodili sme len to, čo sme už vedeli a môžeme sa vrátiť s hodnotou true. Ak je tento literál záporný, narazili sme na konflikt a vraciame false. Ak sme sa dostali cez tieto dva riadky, tak je posledný literál neohodnotený. Príslušnú premennú ohodnotíme tak, aby bol literál splnený a pridáme ho do množiny premenných čakajúcich na spracovanie.

Riešič Mei sa reštartuje presne ako Satsuki až na jeden rozdiel. Po reštarte sa premenná na rozhodnutie nevyberá úplne náhodne. Počas priebehu funkcie

PROCESS-SUBST si zaznamenávame pre každú klauzulu informáciu o tom, že sme ju počas hľadania navštívili. Pri reštarte si vyberáme premennú z takej klauzule, ktorú sme nenavštívili. Ak takú nemáme, tak si vyberáme náhodne ako v Satsuki. Pri reštarte tieto informácie zmažeme, takže budeme v stave, v ktorom nie je navštívená žiadna klauzula. Toto robíme preto, aby sme po reštarte prehľadávali iný komponent formule, ak tam je viac komponentov. O tom, Čo je myslené komponentom formule, pojednáva ďalšia podkapitola.

2.7 Komponenty súvislosti vo formule

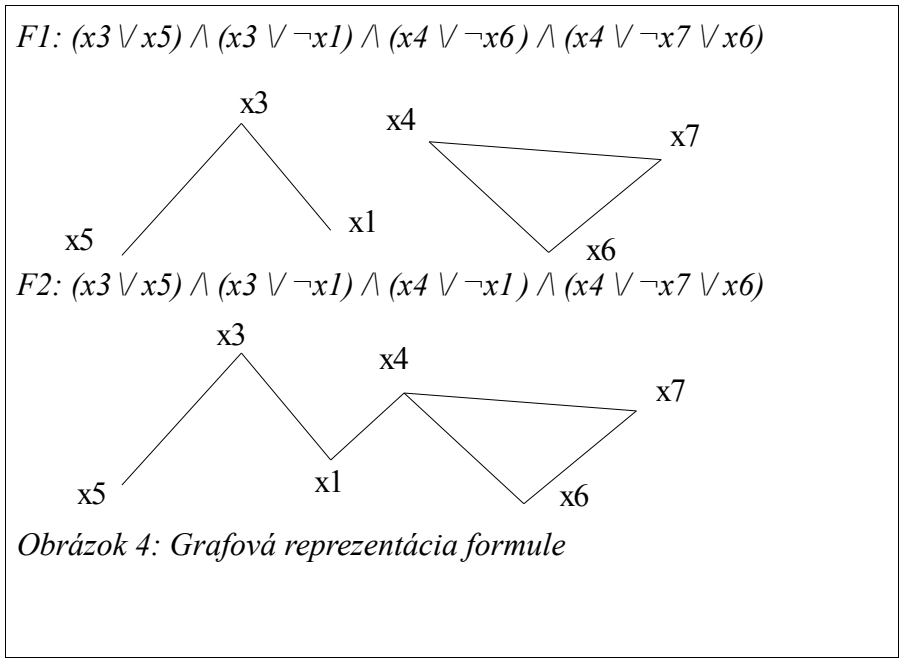
Aby sme ilustrovali čo je komponent súvislosti formule pomôžeme si teóriou grafov. Reprezentovať formulu výrokovej logiky grafom je možné mnohými spôsobmi. Prevodom SAT na grafový problém sa zaoberalo už mnoho ľudí a dosiahli pri tom zaujímavé výsledky. My budeme definovať graf, ktorý nebude jednoznačne zodpovedať formule oboma smermi. Pre danú formulu bude graf definovaný jednoznačne ale naopak nie.

Definujme graf *G(V, E)* pre formulu *f* nasledovne:

V = množina výrokových premenných formule f

$(x, y) \in E \Leftrightarrow$ literál premennej x a literál premennej y sa nachádzajú súčasne v niektorej klauzule formule f.

Náš graf má teda za vrcholy premenné formule a hranou sú spojené tie premenné, ktorých literály sa vyskytujú spolu v niektorej klauzule. Príklady sú na obrázku 4. Graf formule *F1* má dva komponenty súvislosti a graf formule *F2* len jeden. Komponent súvislosti formule definujeme ako množinu premenných, ku ktorým prislúchajúce vrcholy tvoria komponent súvislosti v grafe formuly.



Obrázok 4: Grafová reprezentácia formule

Prečo vôbec hovoríme o komponentoch súvislosti vo formule? Každý komponent má tú krásnu vlastnosť, že ho môžeme riešiť nezávisle od zvyšku formuly. V príklade na obrázku 4 má formula *F1* dva komponenty, môžeme ich riešiť oddelene. Rozložme *F1* na *F1a* a *F1b*:

$F1a = (x3 \vee x5) \wedge (x3 \vee \neg x1)$

$F1b = (x4 \vee \neg x6) \wedge (x4 \vee \neg x7 \vee x6)$

Tým, že hľadáme riešenie *F1a,* teda skúšame rôzne hodnoty pre jej premenné, nijak neovplyvňujeme formulu *F1b*. Ak nájdeme riešenie jednej podformule a aj druhej podformule, tak ohodnotenie, ktoré vznikne zjednotením splňujúcich ohodnotení podformúl, splňuje celú formulu. Ak o niektorej dokážeme, že je nesplniteľná, tak je nesplniteľná celá formula. Druhý komponent už nemusíme vôbec prehľadávať. Ak má formula s *n* premennými dva rovnako veľké komponenty, tak riešiču, ktorý si uvedomuje komponenty, stačí prehľadať priestor veľkosti $2x2^{n/2}$ namiesto 2^n. Formula môže mať samozrejme omnoho viac komponent než len dve. V takom prípade ušetríme ešte omnoho viac prehľadávania.

Komponenty sú vynikajúca vec, ale ich hľadanie je zložité a veľmi často má formula len jeden veľký komponent. I keď komponent je na začiatku len jeden, počas hľadania sa formula na komponenty rozpadáva. Napríklad v príklade *F2* na obrázku 4 sa formula po ohodnotení premennej *x1* alebo *x3* rozpadne na dva približne rovnaké komponenty. Ideálne by bolo po každom rozhodnutí kontrolovať stav grafu formule, zisťovať, či sa rozpadla na komponenty. Toto zisťovanie je drahé, detekcia komponent by sa celkom určite stala úzkym hrdlom riešiča. Mohli by sme to zmierňovať tak, že by sme kontrolu komponent prevádzali len každých *k* rozhodnutí alebo vymyslieť nejaký lepší, iteratívny algoritmus na hľadanie komponent.

Vo svete riešičov sa dynamickou detekciou komponent, teda detekciou komponent počas prehľadávania, väčšina riešičov vôbec nezaoberá. Ako je to uvedené v [24]: „Precízna analýza komponent je cenovo nedosiahnuteľná pre SAT solver.“ V roku 2006 autori riešiča COMPSAT [31] napriek tomuto tvrdeniu zakomponovali do riešiča detekciu komponent a riešič ešte zostal prijateľne rýchly. Statická analýza komponent, teda analýza v čase predspracovania, sa vyskytla aj v niektorých starších riešičoch [25, 26, 27].

V riešičoch Satsuki a Mei samozrejme tiež nie je použitá precízna analýza komponent. Je použitá heuristika výberu premennej pre rozhodnutie, ktorá má také vlastnosti, že riešič, ktorý sa ňou riadi, rieši formulu s ohľadom na komponenty. Keďže naša heuristika má minimálnu réžiu, dostávame relatívne dobrú dynamickú analýzu komponent úplne zadarmo. Podrobnejšie a presnejšie bude táto problematika popísaná v podkapitole 2.9.

2.8 Rozhodovacie heuristiky

V tejto podkapitole pojednáme o heuristikách, ktoré sa používali a používajú v ostatných riešičoch na výber premennej pre rozhodnutie.

Jednoduchá heuristika, ktorá napadne každého ako prvá, je vybrať takú premennú, ktorá má najviac výskytov. Keď má veľa výskytov, tak mnoho klauzúl bude vďaka jej ohodnoteniu splnených a z mnohých sa môžu stať jednotkové. Ohodnotenie takej premennej tiež môže pomôcť formulu rozbiť na komponenty súvislosti, ale bez detekcie komponent to nijak nepomôže. Jemné modifikácie sú vybrať premennú, ktorá ma najviac kladných alebo najviac záporných výskytov. Táto heuristika neprináša príliš dobré výsledky a navyše je veľmi drahá. Musíme počítať alebo udržovať počty výskytov všetkých premenných a potom z nich ešte vyberať maximum. Druhú časť môžeme efektívne implementovať pomocou binárnej haldy [32], ale s prvou sú problémy.

Podobná heuristika je vyberanie takej premennej, ktorá ma veľa výskytov v krátkych klauzulách. Robí sa to preto, aby vznikalo mnoho jednotkových klauzúl a aby jednotková propagácia čím skôr a čím viac odvodila. Táto heuristika je o poznanie lepšia, ale čo sa týka zložitosti je na tom ešte horšie ako predošlá.

Najjednoduchšia heuristika je vyberanie prvej voľnej (neohodnotenej) premennej. Výhodou je jednoduchá implementácia a malá zložitosť. Pri malých formulách môže dávať lepšie výsledky ako predošlé dve heuristiky. Je to spôsobené relatívne vysokou zložitosťou spomínaných dvoch heuristík. Veľmi podobná metóda výberu prvej voľnej je náhodný výber voľnej premennej. Aké sú výhody? Aké sú nevýhody? O náhode nič nemožno tvrdiť s určitosťou, tým pádom táto heuristika nemá žiadne nevýhody. To by sme mohli považovať za výhodu a navyše má aj veľmi dobrú, nízku réžiu.

Známa a mnohými aj dnešnými riešičmi používaná heuristika je takzvaná VSIDS[1] heuristika, prvýkrát použitá riešičom Chaff [21]. Táto heuristika má zmysel len v riešičoch s učením klauzúl. Čo je to učenie sa klauzúl a ako funguje je popísané napríklad v [29]. Pre nás stačí vedieť toľko, že pri konfliktoch vznikajú nové klauzule a pridávajú sa do databáze klauzúl, teda do formule. VSIDS vychádza z výberu premennej s najväčším počtom výskytov. Pre každú premennú máme počítadlo pre kladné a záporné literály. Na začiatku tieto počítadlá nastavíme na nulu. Vždy keď do databázy pribudne klauzula, teda program sa naučí novú klauzulu, tak sa zväčší čítač všetkých literálov, ktoré sa v nej nachádzajú. Periodicky sa počítadlá vydelia nejakou konštantou. Na ohodnotenie sa vyberá literál, ktorý má najväčšiu hodnotu. Táto heuristika nás teda udržiava pri konfliktoch a najmä pri nedávnych konfliktoch, kvôli periodickému deleniu čítačov. Táto heuristika sa dá efektívne implementovať, takže má nízku réžiu. Riešiče ju používajú s veľkým úspechom.

2.9 Heuristika použitá v našich riešičoch

Naše riešiče používajú heuristiku, ktorá nie je podobná žiadnej v predošlom texte popísanej. Budeme ju označovať LEFV. Táto skratka vychádza z anglických slov Last Encountered Free Variable, teda naposledy stretnutá voľná premenná.

1 VSIDS – Variable State Independent Decaying Sum

Počas jednotkovej propagácie navštevujeme klauzule aby sme zistili, či nie sú jednotkové. Klauzule, ktoré nie sú jednotkové obsahujú voľné premenné. Z poslednej navštívenej nejednotkovej klauzule si vyberieme niektorú voľnú premennú. Ak používame princíp dvoch sledovaných literálov, tak si vyberieme niektorý zo sledovaných literálov. Implementácia je veľmi jednoduchá, stačí mať jednu premennú, do ktorej každá navštívená premenná zapíše svoj názov. Keď dôjde k rozhodovaniu, tak v našej premennej sa nachádza názov tej premennej, ktorá sa tam zapísala ako posledná a teda prepísala predošlú hodnotu. Zložitosť je očividne výborná, nemusíme robiť žiadne výpočty, len prepisujeme hodnotu jednej premennej. Pamäťová zložitosť je konštantná.

Z pohľadu zložitosti teda naša heuristika nemá chybu, ako to ale vyzerá s jej účinnosťou? Prečo by mala byť posledná navštívená premenná dobrým kandidátom na ohodnotenie? Na konci podkapitoly 2.7 o komponentoch sme sľubovali, že naša heuristika má vlastnosti, ktoré vedú riešič k tomu, aby sa správal dobre vzhľadom ku komponentom súvislosti. Heuristika LEFV má nasledujúcu krásnu vlastnosť.

Veta 1: *Nech K je komponent súvislosti formuly F v stave, keď bola pre rozhodnutie vybraná premenná p, $p \in K$. Ak heuristika LEFV pri nasledujúcom rozhodnutí vyberie voľnú premennú q, tak $q \in K$.*

Ešte pred dôkazom si podrobnejšie rozoberieme, čo veta 1 tvrdí. V podmienke implikácie je, že LEFV vyberie voľnú premennú. Čo iné sa ešte môže prihodiť? Môže sa stať, že LEFV nevyberie žiadnu premennú, alebo vyberie už ohodnotenú. Toto sa stane vtedy, ak naposledy prehľadávaná komponenta je celá splnená a teda nemá voľnú premennú. Naposledy prehľadávaná komponenta nemusí byť *K*, môže to byť jej podmnožina, pretože sa *K* mohla rozpadnúť v dôsledku vypínania klauzúl a propagácie jednotky. Ak nám LEFV nevráti voľnú premennú použijeme sekundárnu heuristiku, napríklad výber prvej voľnej premennej. Sekundárnu heuristiku používame aj pri úplne prvom rozhodnutí. Veta 1 hovorí, že ak LEFV vráti voľnú premennú, tak tá určite je z množiny *K*. Zdôraznime, že množinu *K* sme si fixovali v vstave, keď sme si vybrali premennú *p*, teda predtým, než sme vykonali akcie vyplývajúce z ohodnotenia *p*.

Dôkaz: Keďže heuristika LEFV môže vyberať len takú premennú, ktorú stretne počas ohodnocovania premennej *p*, stačí dokázať, že sa algoritmus pri ohodnocovaní nedostane mimo množinu *K*. Vypínanie klauzúl, ktoré sú vďaka ohodnoteniu nejakej premennej z *K* splnené, nevyvedie program z komponenty *K*, pretože sa postupuje po klauzulách a tie vedú len v rámci komponenty. Podobne kontrola jednotkových klauzúl. Premenné ktoré sa odvodia počas jednotkovej propagácie sú určite z *K*, pretože keď sme k tej danej jednotkovej klauzule *c* došli, tak sa to stalo preto, lebo práve ohodnocovaná premenná ma nesplnený literál v *c*. Teda tam má tiež výskyt, čo znamená, že sú v rovnakej komponente. Počas vypínania klauzúl môže klesnúť počet kladných alebo záporných výskytov nejakej premennej na nulu, čo vedie k ohodnoteniu nejakej premennej. Táto premenná ale opäť musí patriť do *K*. Mala totiž výskyt vo vypínanej klauzule, kde mala splnený literál ohodnocovaná premenná. Q.E.D.

Veta 1 nám zaručuje, že ak heuristika vráti voľnú premennú, tak tá je dobrým kandidátom na ohodnotenie. Veta 1 je implikácia jedným smerom. Implikácia druhým smerom by sa nám viac páčila, teda veta v zmysle: Ak existuje v K voľná premenná, tak heuristika LEFV ju vyberie. Táto veta ale žiaľ neplatí, viď protipríklad na obrázku 5. Bolo by pravdepodobne možne zostrojiť heuristiku, pre ktorú by platili obe vety, teda by platila ekvivalencia medzi existenciou voľnej premennej a jej nájdením. V tejto práci sme takú heuristiku nevytvárali, je to však dobrý námet pre budúcu prácu.

$(x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_2 \vee x_1)$ // x_1, x_2, x_3 sú v 1 komponente.

Ohodnotíme x_1= true, vypneme štvrtú klauzulu a koniec, žiadnu premennú sme nestretli, voľné premenné x_2, x_3 zostali nedoknuté.

V tomto príklade by nam pomohlo, ak by sme si povedali, že aj počas vypínania klauzúl môžeme stretávať premenné, stretli by sme x_2 ale napríklad v nasledujúcej formule by ani to nepomohlo

$(x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_2 \vee x_1) \wedge (x_4 \vee x_5)$

ku x_4, x_5 alebo x_2, x_3 by sme sa nedostali. Podľa toho či by sme vo štvrtej klauzule stretli x_4 alebo x_2.

Obrázok 5: Ilustrácia fungovania heuristiky LEFV

Heuristika LEFV nás teda udržiava v jednej komponente, prečo to ale vlastne chceme? Predstavme si, že používame inú heuristiku, ktorá nám dáva premenné z rôznych komponent. Ohodnocujeme teda premenné, ktoré spolu nesúvisia. Ak nastane konflikt tak backtrackujeme cez premenné, ktoré s tým konfliktom nesúvisia. Toto riešiče riešia nechronologickým backtrackingom [20]. Pri našej heuristike také niečo nie je potrebné. S aktuálnym konfliktom vždy súvisí posledných n rozhodnutí. Navyše riešič s našou heuristikou dorazí ku konfliktu oveľa rýchlejšie ako by dorazil s inou a teda nechodí tak hlboko vo vyhľadávacom strome. Pre príklad k pochopeniu, prečo to tak je, viď obrázok 6.

$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_4)$

formula má 2 komponenty: $\{x_1, x_2, x_3\}$ a $\{x_4, x_5, x_6\}$

pre prve rozhodnutie zvoľme x_1 = true, testujeme jednotkovosť prvej klauzule a stretneme pri tom x_2. Ďalej to mohlo pokračovať:

1. LEFV zvolí pre rozhodnutie x_2 a potom čo skúsi obe hodnoty (true, false), vracia sa späť a skúsi x_1 = false. LEFV znovu zvoli x_2 pre rozhodnutie, po návrate k x_1 zistíme, že formula je nesplniteľná po 4 ohodnoteniach.

2. Iná heuristika mohla zvoliť x_5 = true, potom x_6 = true a odvodiť x_4 = true. Potom zvolila x_2 alebo x_3 a po 2 ohonoteniach sa vratila s konfliktom. Backtrackujeme a skúšame x_6 = false atď. Vyskúšame všetky možnosti x_5, x_6, x_2 a x_1 aby sme zistili, že formula je nesplniteľná (16 ohonotení).

Obrázok 6: Užitočnosť zotrvania v komponente

3 Implementácia riešiča

3.1 Formát DIMACS pre CNF SAT

Pre problém CNF SAT sa ujal jednotný formát pre zadávanie formúl. Je to formát DIMACS[1] popísaný v [30].

Formula je zadaná v textovom súbore väčšinou s koncovkou cnf. Môže obsahovať komentáre, komentár je označený písmenom „c“ na začiatku riadku a trvá do konca riadku. Pred zadaním samotnej formule musí byť riadok definujúci problém (problem definition line). Tento riadok sa začína písmenom „p“, potom nasleduje medzera, potom slovo „cnf“, znovu medzera, počet premenných, opäť medzera a nakoniec počet klauzúl a koniec riadku. Názvy premenných musia byť prirodzené čísla od 1 do počet premenných. Po riadku definujúcom problém nasledujú klauzule. Klauzule sú zapísané ako postupnosť literálov oddelené medzerami a ukončené nulou. Kladný literál premennej k zapíšeme ako k a záporný ako $-k$. Príklad je uvedený na obrázku 5.

$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee x_1) \wedge (x_4 \wedge \neg x_5 \wedge x_1) \wedge (\neg x_3 \wedge x_5) \wedge x_6$

```
c jednoduchy priklad pre ilustraciu
p cnf 6 5
1 2 -3 0
3 1 0
4 -5 1 0
-3 5 0
6 0
```

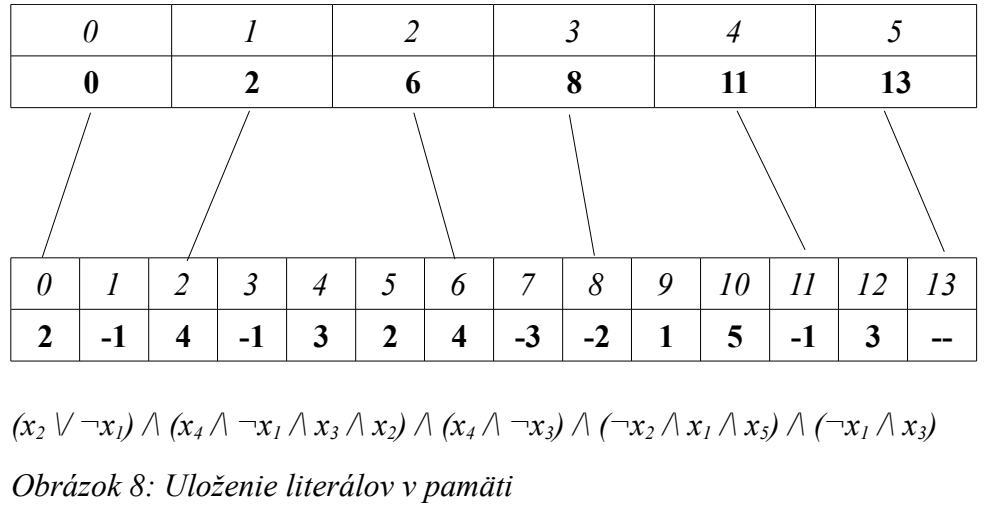
Obrázok 7: Formát DIMACS pre CNF SAT, príklad

3.2 Reprezentácia formule a načítanie vstupného súboru

Reprezentácia formule v pamäti je rovnaká v oboch riešičoch, preto ju popíšeme len raz. Formula je uložená v pamäti redundantne, pretože si pamätáme jednak všetky literály každej klauzule ale tiež všetky výskyty každého literálu. Literály klauzúl sú uložené v jednom súvislom poli P za sebou a v druhom poli I sú uložené indexy, kde sa literály klauzule začínajú. Príklad je na obrázku 8. Klauzule sa počítajú od nuly. Pole indexov I obsahuje jednu položku navyše, ktorá ukazuje za posledný prvok poľa literálov P. Ak chceme zistiť ktoré literály sa nachádzajú v klauzule číslo k, tak sú to prvky poľa P s indexmi $I[k]$ až $I[k+1] - 1$. Rovnaké dve dvojice polí máme aj pre kladné a záporne výskyty premenných. Tieto štruktúry sa vytvoria na začiatku programu a potom, počas hľadania riešenia, sa už vôbec

1 DIMACS – Center for Discrete Mathematics and Theoretical Computer Science

nemenia. Formula vzniká zo vstupného textového súboru formátu DIMACS SAT na začiatku programu.



Obrázok 8: Uloženie literálov v pamäti

Pri načítaní formule zo súboru budujeme pole obsahujúce všetky literály a indexové pole k nemu. Vstupný súbor ďalej nepotrebujeme, preto ho uzavrieme. Pole kladných a záporných výskytov a príslušné indexové polia dopočítame z poľa všetkých literálov v klauzulách.

Funkcia ktorá načíta vstupný súbor a vytvorí formulu je v oboch riešičoch rovnaká, až na spôsob práce so vstupným súborom. Satsuki je v jazyku C++ a práca so súborom je implementovaná použitím objektov std::stream. Riešič Mei je napísaný v jazyku ANSI C a používa štandardné funkcie fopen a fread. Rozdiel v rýchlosti je pri veľkých súboroch očividný. Ako by sa očakávalo, Mei načíta súbor niekoľkonásobne rýchlejšie ako Satsuki.

3.3 Reprezentácia stavu v Satsuki

Stav, ako to už bolo spomenuté pri popise algoritmov, obsahuje tieto informácie: počet nesplnených klauzúl, hodnoty premenných (true, false, undefined), aktivita klauzúl, počet kladných a záporných výskytov pre každú premennú.

Stav je fyzicky uložený do dvoch polí, jedno typu char a druhé typu int. Hodnoty premenných a aktivita klauzúl majú malú doménu a preto sú uložené do poľa typu char s názvom info. Veľkosť poľa info je súčet počtu prvkov a počtu klauzúl zvýšený o jedna. Pozícia 0 v poli info nie je definovaná. Na pozíciach 1 až počet premenných sa nachádzajú aktuálne hodnoty premenných (true, false, undefined). Premenné sa začínajú od jednej a nie od nuly, pretože aj vo formáte DIMACS SAT sú premenné pomenované od 1. Na pozíciach počet premenných + 1 až počet premenných + počet klauzúl + 1 sa nachádzajú informácie o aktivite klauzúl. Klauzule sú číslované od nuly. Ak teda chceme vedieť či je *k*-tá klauzula ešte aktívna, tak sa musíme pozrieť do poľa info na pozíciu počet premenných +1+ *k*.

Aktuálny stav v Satsuki, kvôli heuristike čistých premenných, ešte obsahuje počty výskytov literálov pre každú premennú. Tiež do stavu patrí počet nesplnených klauzúl, ktorý používame na detekciu splnenosti celej formule. Tieto čísla môžu byť vysoké a presiahnuť maximálne číslo reprezentovateľné typom char – 255. Pre tieto premenné používame pole typov int s názvom var_pos_neg_count. Na pozícií 0 je uložený počet nesplnených klauzúl. Na pozíciach 1 až (2x počet premenných) + 1 sú uložene počty výskytov. Počet aktívnych kladných výskytov premennej *n* je uložený na pozícií *n* a počet aktívnych záporných výskytov na pozícií *n* + počet premenných. Aktívnym výskytom sa rozumie výskyt v aktívnej klauzule.

Tieto dve polia sa počas hľadania často kopírujú. Konkrétne pri každom rozhodnutí. Na kopírovanie používame funkciu memcpy, ktorá je veľmi efektívna pri kopírovaní veľkých kusov pamäte. Vždy keď zvolíme premennú pre ohodnotenie, vytvoríme kópiu aktuálneho stavu a program pokračuje s kópiou. Ak sa prehľadávanie vráti s neúspechom, tak navrátenie pôvodného stavu znamená len nastavenie ukazateľov. Stavy sa nealokujú jednotlivo, ale po blokoch. Stavy sú v blokoch uložené súvisle vedľa seba.

Bloky stavov sa nikdy nedealokujú. Bloky stavov sa recyklujú. Máme v programe pole alokovaných blokov. Ak sa predchádzajúci blok zaplní a nasledujúci už je alokovaný, tak ho použijeme. Ak nie je, tak alokujeme nový blok. Pri vracaní sa späť do predošlého bloku, ten nový necháme tak a ak bude znovu potrebný, tak už nemusíme alokovať.

Logicky do stavu patria aj dvojice sledovaných literálov, ale vďaka tomu, že pri návrate z rekurzie ich netreba upravovať, nemusíme si ich zálohovať, teda kopírovať pri rozhodnutí. Sledované literály máme uložené v jednom globálnom poli typov int. Veľkosť tohoto podľa je dvojnásobok počtu klauzúl vstupnej formule. Sledované literály klauzule *c* sú uložené na pozíciach *2*c* a *2*c + 1*. Ako funguje princíp dvoch sledovaných literálov sme popísali v podkapitole 2.4.

3.4 Reprezentácia stavu v Mei

Stav v riešiči Mei obsahuje menej informácií ako v Satsuki. Potrebujeme si pamätať počet nesplnených premenných, hodnoty premenných a počty nesplnených literálov pre každú klauzulu. Všetky tieto informácie máme uložené v jednom poli typov int. Nie je to síce pamäťovo efektívne, že na hodnoty premenných používame tiež int, ale je to jednoduchšie. Na pozícií nula našeho poľa je uložený počet nesplnených klauzúl, na pozíciach 1 až počet premenných sú hodnoty premenných. Za hodnotami premenných na pozíciach počet premenných + 1 až počet premenných + počet klauzúl + 1 sú uložené počty aktívnych literálov v klauzulách.

Zaradenie stavov do blokov, alokácia a recyklácia blokov funguje úplne rovnako ako v Satsuki. Dva sledované literály v Mei nie sú.

3.5 Implementácia funkcií

V tejto podkapitole ukážeme, že ako jednoducho a priamočiaro možno implementovať funkcie z pseudokódov našich riešičov, vďaka spôsobu reprezentácie formule a stavu, ktorý používame. Názvy funkcií sú z výpisov algoritmus 2, 3 a 4.

Funkcia NUM-UNSAT-CLAUSES sa len podíva na nultú pozíciu vhodného poľa. Výsledok máme v konštantnom čase. Funkcia DEC-NUM-UNSAT-CLAUSES pristupuje k tomuto istému miestu v pamäti a dekrementuje hodnotu o jedna.

Funkcie INACTIVE, MAKE-INACTIVE, VALUE, VALUATE pristupujú k stavu do poľa na jednoznačne určené miesto a zisťujú hodnotu alebo ju modifikujú. Podobne v Mei fungujú COUNTER, SET-COUNTER, DEC-COUNTER. Všetky tieto funkcie pracujú v konštantnom čase.

Počas popisu pseudokódov sme pre jednoduchosť definovali, že funkcie POS-OCC a NEG-OCC vracajú množinu klauzúl a funkcie TEST-UNIT, TURN-OFF-CLAUSES a DECREASE berú ako parameter množinu klauzúl. Samozrejme v implementácií so žiadnymi množinami nepracujeme. Funkcie POS-OCC a NEG-OCC vôbec neexistujú a spomínané tri funkcie berú namiesto množín dva ukazatele. Jeden ukazuje na prvú klauzulu, ktorú treba spracovať a druhý za poslednú. Tieto ukazatele smerujú do štruktúry reprezentujúcej formulu. Konkrétne do indexového poľa všetkých literálov klauzúl. Ukazatele získame tiež z formule. Máme ich uložené v poli všetkých výskytov kladných respektíve záporných literálov premenných. Je teda k nim prístup v konštantnom čase.

Funkcia DECISION je trošku odlišne implementovaná v našich dvoch riešičoch. V Satsuki je implementovaná presne tak ako sme to popísali vyššie. Máme jednu globálnu premennú, ktorá nesie názov naposledy navštívenej premennej. Prepisujeme ju vo funkcií TEST-UNIT. V DECISION si ju vyberieme pre rozhodnutie ak je voľná, inak použijeme sekundárnu heuristiku. V Mei nenavštevujeme konkrétne premenné ale len klauzuly. Máme preto globálnu premennú, ktorá nesie názov poslednej stretnutej nesplnenej klauzuly. Vo funkcií DECISION sa z tejto klauzuly vyberie niektorá voľná premenná, ak tam žiadna nie je, tak sa použije sekundárna heuristika.

3.6 Programovanie s ohľadom na cache

V pamäťovej hierarchií počítača zohráva dôležitú úlohu procesorová cache. Vyrovnávacia pamäť cache sa v hierarchií nachádza medzi registrami procesora a hlavnou pamäťou. Prístup do cache je 20 až 100 krát rýchlejší ako do hlavnej pamäte, je to však kompenzované kapacitou ktorá je obvykle 100 krát menšia. Je veľmi dôležité, pre urýchlenie programu, prispôsobiť algoritmy a dátové štruktúry tak, aby údaje, ktoré práve potrebujeme, boli v cachi a nemuseli sa načítavať z hlavnej pamäte. Ďalšia vlastnosť cachu je, že sa tam údaje kopírujú po blokoch. Je preto taktiež výhodne držať si súvisiace informácie fyzicky blízko seba.

Naše dátové štruktúry sme sa snažili optimalizovať pre cache. Všetky sú to veľké polia súvisle naskladané vedľa seba do blokov. Algoritmus sme tiež prispôsobili. Napríklad funkcie PROCESS-SUBST vyberajú pre spracovanie ohodnotenia tú premennú, ktorej hodnotu sme odvodili najneskôr (PICK-LAST). Máme dobrú šancu, že sa budeme pohybovať v takej časti formuly, v ktorej sme sa nedávno pohybovali a teda je ešte v cache.

Napokon aj heuristika LEFV a myšlienka udržiavania sa v jednej komponente napomáha efektívnemu využitiu cache. Malé komponenty sa cele zmestia do cachu a program ich môže veľmi rýchlo celé prehľadať. Aj keď tam komponenty nie sú, tak voľba poslednej stretnutej premennej nás zavedie do častí formuly, ktoré by ešte mali byť v cache.

4 Typy formúl a výkon riešičov na nich

Táto kapitola bude venovaná experimentom s rôznymi riešičmi na rôznych typoch formúl. Budeme merať a porovnávať rýchlosť našich riešičov Satsuki a Mei a najlepších súčasných riešičov, konkrétne MiniSat a Minisat2 [15], Picosat [14], zChaff [13] a RSat2 [12]. Zaradili sme 2 verzie riešiča Satsuki. Satsuki-nr sa líši od Satsuki jedine v tom, že sa nikdy nereštartuje. Experimenty boli prevedené na počítači s procesorom Intel Pentium M 1.7 GHz (2 MB L2 Cache) s operačnou pamäťou 512 MB, pod operačným systémom GNU/Linux Debian 4.0 s jadrom verzie 2.6.18.

4.1 Fázový prechod

Mnoho obrovských formúl, s počtom premenných v ráde miliónov dokážu riešiče vyriešiť vo veľmi rýchlom čase. Ako je to teda s exponenciálnou zložitosťou problému SAT? Exponenciálny čas je samozrejme len horný odhad. Odhad pre najhorší možný prípad. Väčšina formúl sa dá vyriešiť veľmi rýchlo. Vyskytla sa teda otázka, kde sú tie ťažké príklady formúl? Na tuto otázku hľadali odpoveď aj autori článku [34] a zaviedli pojem fázového prechodu.

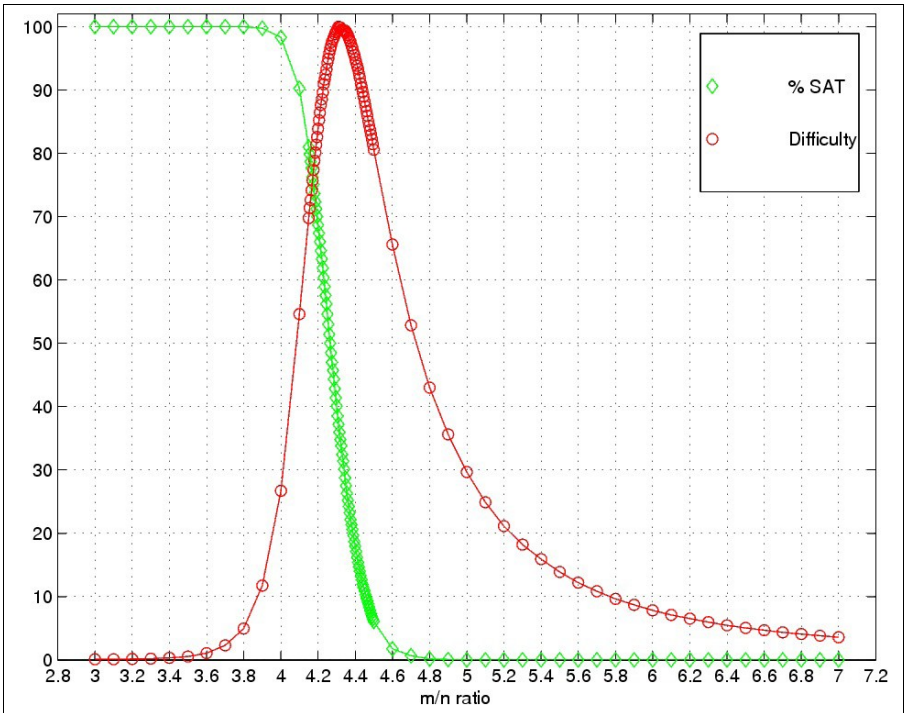
Niektoré formule sú ľahko vyriešiteľné, pretože majú veľa premenných a relatívne málo klauzúl. Majú teda mnoho riešení a riešič ľahko nájde jedno a môže skončiť. Tiež ľahko vyriešiteľné sú formule, ktoré majú málo premenných a relatívne veľa klauzúl. V takej formule sú premenné veľmi obmedzené a existujú malé nesplniteľné jadrá. Nesplniteľné jadro je množina premenných, pre ktorú platí, že žiadne ohodnotenie prvkov množiny formulu nesplňuje. Riešiče rýchlo nájdu nejaké nesplniteľné jadro a môžu skončiť.

Ťažké sú také formule, ktoré sú na tom s pomerom počtu premenných a klauzúl niekde medzi. Tento pomer súvisí so splniteľnosťou. Ak nakreslíme graf pravdepodobnosti splniteľnosti formúl v závislosti od pomeru premenných a klauzúl, tak dostaneme graf podobný obrázku 9. Do grafu je tiež zakreslený čas, za ktorý typický riešič vyrieši formulu s daným pomerom.

Fázový prechod je hodnota pomeru, pri ktorom sa preklápa pravdepodobnosť výskytu splniteľných formúl na nesplniteľné. Pre 3-CNF SAT je to, ako vidíme z obrázku 9, hodnota niekde medzi 4 a 5. Práve formule, ktoré majú pomer počtu premenných a klauzúl blízky tejto hodnote sú pre riešiče obťažné. Presnejšiu matematickú definíciu fázového prechodu pre náhodné *k*-CNF formule máme z [33]:

$R_k(N, L)$ označme množinu problémov *k*-CNF s N premennými a L klauzulami. Prob(sat, X) nech je pravdepodobnosť, že náhodne vybraný problém z množiny X je splniteľný. Potom fázový prechod c_k je hodnota, pre ktorú platí:

$$\lim_{N\to\infty} Prob(sat, R_k(N, c.N)) = \begin{cases} 0\ pre\ c > c_k \\ 1\ pre\ c < c_k \end{cases}$$



Obrázok 9: Fázový prechod, obrázok je prebraný z [36]

4.2 Ťažké náhodne generované problémy.

V tejto podkapitole uvedieme výsledky experimentov s náhodne generovanými ťažkými 3-CNF formulami. Ťažké v zmysle blízke svojim pomerom premenných a klauzúl fázovému prechodu. Budeme používať sady formúl z knihovne SAT problémov SATLIB [17]. V každej sade je 100 formúl, okrem sád so 100 premennými, tie obsahujú 1000 formúl. Budeme merať celkový čas, za ktorý riešiče vyriešia všetky. V tabuľke 1 sú zobrazené výsledky merania v sekundách. Niekde je uvedený symbol „-“ namiesto času, to znamená, že to riešiču trvalo vyše časový limit 5 minút.

Tabuľka 1: 3-SAT, fázový prechod

Satisfiability	SAT					UNSAT				
Vars	75	100	125	150	175	75	100	125	150	175
MiniSat	*1.17*	*11.59*	*2.05*	*3.32*	*9.18*	*0.95*	*11.85*	*2.45*	*6.07*	*15.80*
MiniSat2	*0.70*	*7.26*	*1.47*	*2.79*	*7.19*	*0.45*	*7.78*	*1.91*	*5.10*	*14.04*
Picosat	*1.02*	*10.92*	*1.98*	*3.89*	*10.96*	*0.84*	*11.95*	*2.93*	*7.27*	*22.39*
zChaff	*1.23*	*12.87*	*2.42*	*4.13*	*11.35*	*1.06*	*19.31*	*5.32*	*17.75*	*64.72*
RSat2	*0.72*	*8.45*	*1.76*	*3.60*	*10.01*	*0.64*	*10.22*	*2.71*	*7.05*	*19.23*
Mei	*1.14*	*30.12*	*11.34*	*36.15*	*165.21*	*1.32*	*66.98*	*18.4*	*52.3*	*172.70*
Satsuki	*1.49*	*40.22*	*32.88*	-	-	*1.75*	*154.00*	-	-	-
Satsuki-nr	*1.49*	*32.28*	*13.71*	-	-	*1.78*	*77.99*	*39.86*	*233.61*	-

Ako vidieť z tabuľky 1, naše riešiče sú najpomalšie a to hlavne riešič Satsuki. Vidno taktiež to, že reštarty riešiču Satsuki neprospeli. Ukazuje sa, že pri reštarte stráca príliš veľa. Mei je na tom oveľa lepšie ale tiež sa pomer v časoch postupne zhoršuje. Prečo je veľký rozdiel medzi našimi riešičmi a konkurenčnými je jasné, je to kvôli absencií veľmi dôležitého vylepšenia v našich riešičoch – učenia sa klauzúl. Rozdiel medzi Mei a Satsuki je už zložitejšie vysvetliť. Najpravdepodobnejší dôvod je spôsob reštartovania. Mei sa snaží reštartovať do komponentu, v ktorom v predošlom kole nebol. Satsuki reštartuje úplne náhodne. Ostatné rozdiely Satsuki a Mei totiž predpovedajú, že rýchlejší by mal byť Satsuki. Tieto merania svedčia o tom, že na komponenty treba brať ohľad všade, kde sa to len dá.

4.3 Holubníkový princíp

Dirichletov princíp, tiež známy ako holubníkový princíp hovorí, že ak je umiestených *n* holubov do *n+1* dier, tak aspoň v jednej diere sú aspoň dva holuby. V tejto kapitole budeme riešiť formule, ktoré sú zakódovaním otázky, či je možné umiestniť *n+1* holubov do *n* dier tak, aby v žiadnej neboli dva holuby. Samozrejme odpoveď je záporná a teda aj takáto formula je nesplniteľná.

Tento typ formúl má aj teoretický význam. Je dokázané [35], že neexistuje polynomiálny rezolučný dôkaz. Teda dolný odhad dĺžky dôkazu použitím rezolúcie je exponenciálny. Holubníkové problémy sú ale relatívne dobre štruktúrované. Existujú techniky [37], ktoré ich dokážu vyriešiť bez prehľadávania. Niektoré riešiče používajú rezolúciu, ako bude vidno z meraní, dopadnú takéto riešiče veľmi zle. Testovacie formule opäť pochádzajú zo zbierky SATLIB [17].

Tabuľka 2: Holubníkový princíp

filename	hole6.cnf	hole7.cnf	hole8.cnf	hole9.cnf	hole10.cnf
Vars/Clauses	42/133	56/204	72/297	90/415	110/561
MiniSat	*0.040*	*0.212*	*0.672*	*5.444*	*43.183*
MiniSat2	*0.032*	*0.192*	*0.644*	*3.388*	*31.238*
Picosat	*0.024*	*0.416*	*1.372*	*15.397*	*313.792*
zChaff	*0.032*	*0.120*	*0.320*	*1.468*	*6.932*
RSat2	*0.012*	*0.088*	*0.348*	*37.718*	*326.228*
Mei	*0.012*	*0.102*	*0.268*	*0.552*	*1.080*
Satsuki	*0.012*	*0.096*	-	-	-
Satsuki-nr	*0.012*	*0.096*	*0.480*	*2.844*	*24.482*

Z tabuľky 2 vidieť, že naše riešiče víťazia nad riešičmi v tomto type formúl. Možné vysvetlenie je už spomínaná štruktúrovanosť týchto formúl. Ďalej môžeme pozorovať to, čo v tabuľke 1, Satsuki funguje lepšie bez reštartovania.

4.3 Pole logických členov programovateľné užívateľom

Pole logických členov programovateľné užívateľom (FPGA[1]) je druhom programovateľnej digitálnej logiky. V podstate ide o programovateľný logický obvod, teda skupinu digitálnych integrovaných obvodov, ktorých funkciu definuje užívateľ programovo. Podrobnejšie informácie o FPGA môžete nájsť na stránkach Wikipedie [38].

My budeme riešiť problém smerovania v FPGA. Konkrétne otázku, či je možné nasmerovať *m* spojení cez *n* prepojení. Pre $m > n$ to možné nie je. Tento problém je veľmi podobný holubníkovým problémom. Táto podobnosť určite má súvis s tým, že naše riešiče prekonávajú na nich ostatné riešiče, podobne ako pri holubníkových problémoch. Je treba poznamenať, že tieto problémy sú tiež vyriešiteľné bez prehľadávania technikami prezentovanými v [37].

Budeme určovať, či je to možné pre $m > n$. Formule budú teda nesplniteľné. Testované formule sme získali z [39]. Formule sú pomenované chnln_m, napríklad chnl10_11 je otázka, či je možne nasmerovať 11 spojení cez 10 prepojení. Výsledky meraní sú prezentované v tabuľke 3. Hodnoty sú uvedené v sekundách. Časový limit na riešenie bol stanovený na 3 minúty. Ak to riešič nestihol, má v tabuľke „-“.

Tabuľka 3: FPGA

Title	Vars / Clauses	MiniSat	Minisat 2	Picosat	zChaff	RSat2	Mei	Satsuki	Satsuki -nr
chnl10_11	220/1122	*56.73*	*25.75*	-	*6.88*	-	*4.24*	-	*46.23*
chnl10_12	240/1344	*35.66*	*75.31*	-	*8.27*	-	*4.82*	-	*53.05*
chnl10_13	260/1586	*25.77*	*161.28*	-	*9.73*	-	*5.62*	-	*60.28*
chnl11_12	264/1476	-	-	-	*30.58*	-	*8.60*	-	-
chnl11_13	286/1472	-	-	-	*156.80*	-	*10.11*	-	-
chnl11_20	440/4220	-	-	-	-	-	*23.59*	-	-

Ako vidieť z výsledkov náš riešič Mei v týchto problémoch exceluje. Satsuki bez reštartovania je tiež relatívne dobrý. Riešiče Picosat a RSat2 úplne zlyhali. Zaujímavé je, že riešič MiniSat pri formulách chnl10_*x* s rastúcim *x* zrýchľoval, pričom všetky ostatne riešiče sa spomaľovali.

4.4 Formule vyriešené nízkym počtom rozhodnutí

V tejto podkapitole sa podívame na niekoľko formúl, pri ktorých sa do popredia dostane riešič Satsuki. Tieto formule budú väčšie ako doteraz testované. Formule sme opäť získali zo zbierky SATLIB [17], sú zo sady problémov označených Beijing. Boli to súťažné formule na SAT súťaži v Pekingu [40] v roku 1996.

1 FPGA – Field Programmable Gate Array

Pri týchto experimentoch nebudeme merať čas, ale počet rozhodnutí a reštartov, ktoré riešiče vykonajú pri riešení. Tieto čísla by mali byť priamo úmerné rýchlosti riešičov. V tabuľke 4 je v prvom riadku uvedený počet rozhodnutí a v druhom počet reštartov. Otáznik znamená, že nie je známy počet. Je to pri riešičoch RSat2 a zChaff, ktoré neposkytujú informáciu o počte reštartov. Ak sa v tabuľke nachádza „-“, tak to znamená, že riešič nestihol vyriešiť formulu v časovom limite 3 minút.

Nasleduje stručný popis testovacích formúl:

- *3bitadd_31* a *3bitadd_32* – navrhovanie integrovaných obvodov, 3 bitová sčítačka
- *e0ddr2-10-by-5-1* a *e0ddr2-10-by-5-4* – rozvrhovanie, 10 práci po 5 úlohách, jednotné termíny ukončenia.
- *enddr2-10-by-5-1* a *enddr2-10-by-5-8* – ako predchádzajúce, ale s tesnými termínmi.
- *ewddr2-10-by-5-1* a *ewddr2-10-by-5-8* – ako predchádzajúce, tentoraz s voľnejšími termínmi.

Tabuľka 4

Title	Vars / Clauses	MiniSat	Minisat 2	Picosat	zChaff	RSat2	Mei	Satsuki	Satsuki -nr
3bitadd_31	8432 / 31310	- -	- -	4786536	110720 ?	49751 ?	- -	480	480
3bitadd_32	8704 / 32316	- -	- -	2056610	26521 ?	26787 ?	- -	490	490
e0ddr2-10-by-5-1	19500 / 103887	48040	41361	43491	5668 ?	6751 ?	16151	1060	1060
e0ddr2-10-by-5-4	19500 / 104527	48710	41780	35500	443 ?	5794 ?	113967	17451	- -
enddr2-10-by-5-1	20700 / 111567	46960	38510	36820	5656 ?	5457 ?	1190	890	890
enddr2-10-by-5-8	21000 / 113729	50320	39310	39700	336 ?	5549 ?	112407	1360	1360
ewddr2-10-by-5-1	21800 / 118607	41260	42030	43400	6819 ?	5473 ?	34042	1190	1190
ewddr2-10-by-5-8	22500 / 123329	45740	43020	35070	250 ?	5317 ?	17281	1360	1360

Z tabuľky 4 môžeme vyčítať až neuveriteľné údaje. Napríklad formulu enddr2-10-by-5-1, ktorá má 20700 premenných vyriešil riešič Satsuki z 89 rozhodnutí a zvyšné pravdivostné hodnoty už z toho mohol odvodiť. Skôr než by sme to pripísali faktu, že Satsuki robí aj heuristiku čistých premenných, tak sa podívajme na počet rozhodnutí riešiča Mei – 119. Ostatné hodnoty riešiča Mei sú síce vyššie, ale ak sa pozrieme aj na počet reštartov, tak sa dá spočítať, že po poslednom reštarte boli počty rozhodnutí k riešeniu všade v ráde stoviek. Mei totiž reštartuje po 1500 konfliktoch. Na 1500 konfliktov potrebuje 1500 rozhodnutí. Takže ak od počtu celkových rozhodnutí odpočítame súčin počtu reštartov a 1500, tak

získame počet rozhodnutí od posledného reštartu. Po tomto výpočte sú hodnoty Mei na formulách o rozvrhovaní nasledovné: 115, 896, 119, 740, 404 a 228. Tieto hodnoty potvrdzujú to, že nemá veľký zmysel implementovať heuristiku čistých premenných do riešiča aj keď samozrejme nejaký zmysel to má.

Možno si všimnúť, že formulu e0ddr2-10-by-5-4 riešič Satsuki bez reštartovania nevyriešil. Toto zas dokazuje dôležitosť reštartovania. Napríklad riešič Mei, by nevyriešil bez reštartovania žiadnu formulu okrem enddr2-10-by-5-1 z tejto sady experimentov.

Prvé dve formule zo sady v tabuľke 4 sa ukázali ako veľmi náročné pre riešiče MiniSat, Minisat2 a Mei, napriek tomu, že Picosat, RSat2 a zChaff ich vyriešili za slušný čas a riešič Satsuki za vynikajúci čas. Pre vysvetlenie tohoto obrovského rozdielu medzi Mei a Satsuki nás nenapadá nič iné, než heuristika čistých premenných. Napriek tomu, že väčšinou je to len zbytočné brzdenie riešiča, niekedy skutočne môže pomôcť.

4.5 Záver k experimentálnej časti

Existuje mnoho ďalších druhov formúl, ktoré majú štruktúru a sú inšpirované praktickými problémami. Existujú formuly, ktoré kódujú plánovacie a logistické problémy, overovanie mikročipov, ofarbiteľnosť grafov a ďalšie grafové a iné NP-úplné problémy. Na väčšine týchto problémov sú oba naše riešiče výrazne pomalšie než známe riešiče, napríklad tie, ktoré sme v tejto kapitole testovali.

Snažili sme sa do práce vyberať také formuly, v ktorých sa naše riešiče ukázali ako rýchlejšie. Ak by sme zaradili problémy, ktoré sa používajú v dnešných súťažiach SAT Competition a SAT race, dostali by sme tabuľky, ktoré by boli vhodné len na porovnanie ostatných riešičov medzi sebou. Naše riešiče by boli vo veľkej väčšine prípadov za časovým limitom.

5 Záver

V tejto práci sme pojednali o spôsoboch vytvorenia riešiča problémov boolovskej splniteľnosti založeného na algoritme DPLL. Popísali sme rôzne spôsoby implementácie najdôležitejších častí tohoto algoritmu. Vytvorili sme podrobné pseudokódy pre efektívnu implementáciu DPLL bez rekurzie v dvoch verziách, s použitím heuristiky čistých premenných a aj bez použitia. Popísali sme niekoľko spôsobov implementácie propagácie jednotky.

Venovali sme sa heuristikám výberu premennej pre rozhodnutie. Ukázali sme význam komponent súvislosti formuly. Uviedli sme novú heuristiku a nazvali ju LEFV. Má konštantnú zložitosť a predsa veľmi dobrú a netriviálnu vlastnosť. Heuristika, pokiaľ vráti premennú, nás udrží v aktuálne riešenej komponente súvislosti formuly. Túto vlastnosť sa nám podarilo aj dokázať. Objavenie heuristiky LEFV je najdôležitejším prínosom tejto práce.

Naše myšlienky sme zrealizovali a vytvorili dva rozdielne riešiče Satsuki a Mei, aby sme mohli porovnať ako fungujú rôzne techniky na skutočných formulách. Vykonali sme merania a ukázali, že existujú triedy formúl, na ktorých dokážu naše riešiče prekonať aj najlepšie známe riešiče sveta.

Naše riešiče však ešte zďaleka nie sú dokonalé. Existuje mnoho techník, ktoré dokážu riešič urýchliť a my sme ich neimplementovali. Najdôležitejším z nich je určite učenie sa klauzúl. Naša implementácia propagácie jednotky sa ešte potrebuje zlepšiť. Heuristika LEFV sa dá ešte zdokonaliť a zosilniť. Existuje ešte mnoho ďalších nedostatkov našich riešičov, na ktorých odstránenie poznáme spôsob a v budúcej práci sa tomu budeme venovať.

Cesta k vytvoreniu riešiča, ktorý dokáže vyhrať SAT competition alebo SAT race je ešte dlhá. Nie je to ale nedosiahnuteľný cieľ. V tejto práci sme prešli prvú časť tejto cesty a s optimizmom budeme po nej pokračovať.

Literatúra

1. Stephen A. Cook, *The Complexity of Theorem-Proving Procedures*, STOC, 151-158, 1971
2. Michael Sipser, *Introduction to the Theory of Computation*, Course Technology, 2005
3. Rina Dechter, *Constraint Processing*, Morgan Kaufman, 2003
4. M. Ghallab, D. Nau, P. Traverso, *Automated Planning: Theory and Practice*, Morgan Kauffman, 2004
5. A. Robinson, A. Voronkov a kol., *Automated Reasoning*, The MIT Press, 2001
6. A. Biere, *Formal Systems 2*, http://fmv.jku.at/fs2/, webová stránka prednášky, Johannes Kepler Universität Linz, Rakúsko, (Apríl 2008)
7. C. Sinz, *Sat Race 2008*, http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/ , Eberhard Karls Universität Tübingen, Nemecko, webová stránka súťaže, (Apríl 2008)
8. D. La Berre, O. Roussel, L. Simon, *Sat Competitions*, http://satcompetition.org , webová stránka súťaže, (Apríl 2008)
9. Hayao Miyazaki, *My Neighbor Totoro (Tonari no Totoro)*, animovaný film, Studio Ghibli, 1988
10. S. Russel, P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2003
11. M. N. Velev, R. E. Briant, *Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors*. Journal of Symbolic Computation (JSC), volume 35-2,73-106, Elsevier, 2003
12. K. Pipatsrisawat, A. Darwiche, *RSat Homepage*, http://reasoning.cs.ucla.edu/rsat/, výzkumná webová stránka, University of California Los Angeles, USA, (Apríl 2008)
13. Z. Fu, Y. Marhajan, S. Malik, *zChaf*, http://www.princeton.edu/~chaff/zchaff.html, výzkumná webová stránka, Princeton University, USA, (April 2008)
14. A. Biere, *Picosat*, http://fmv.jku.at/picosat/, výzkumná webová stránka, (Apríl 2008)
15. N. Eén, N. Sörensson, *Minisat Page*, http://minisat.se, výzkumná webová stránka, (Apríl 2008)
16. P. Štepánek, *Predikátova Logika*, Matematicko-fyzikálni Fakulta UK v Prahe, 2000
17. Holger H. Hoos and Thomas Stützle, *SATLIB: An Online Resource for Research on SAT.* In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, p.283-292, IOS Press, 2000. SATLIB is available online at *www.satlib.org*
18. M. Davis, H. Putnam, *A Computing Procedure for Quantification Theory*,

Journal of the ACM 7 (1): 201 – 215, 1960

19. M. Davis, G. Logemann, D. Loveland, *A Machine Program for Therorem Proving*, Communications of the ACM 5 (7): 394 – 397, 1962
20. J. P. Marques-Silva, K. A. Sakallah, *GRASP: A Search Algorithm for Propositional Satisfiability*, IEEE Transactions on Computers, vol. 48, no. 5, 1999
21. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, *Chaff: Engineering an Efficient SAT Solver*, Proceedings of the 38th Design Automation Conference (DAC'01), 2001
22. K. Pipatsrisawat, A. Darwiche, *RSat 2.0: SAT solver description*, Technical Report D153, Automated Reasoning Group, Computer Science Department, University of California, Los Angeles, 2007
23. K. Pipatsrisawat, A. Darwiche, *Rsat 1.03: SAT Solver Description*, Technical Report D152, Automated Reasoning Group, Computer Science Department,University of California, Los Angeles, 2006
24. K. Pipatsrisawat, A. Darwiche, *A Lightweight Component Caching Scheme for Satisfiability Solvers*, Proceedings of European Conference on Artificial Intelligence, pp. 328-332, 2004
25. F. Aloul, I. Markov, K. Sakallah, *Force: a fast and easy-to-implement variable-ordering heuristic,* Proceedings of the 13th ACM Great Lakes Symposium on VLSI 2003, pp. 116-119, 2003
26. V. Durairaj, P. Kalla, *Variable ordering for efficient sat search by analyzing constraint-variable dependencies,* SAT'05, August 2005
27. J. Huang, A. Darwiche, *A structure-based variable ordering heuristic for sat,* IJCAI'03, pp. 1167–1172, 2003
28. N. Eén, N. Sörensson, *MiniSat — A SAT Solver with Conflict-Clause Minimization*, SAT'05, 2005
29. P. Beame, H. Kautz, A. Sabharwal, *Understanding the Power of Clause Learning*, Proceedings of the 18th International Joint Conference on Artificial Intelligence, Acapulco, Mexico, 2003
30. DIMACS, *Satisfiability Suggested Format*, 2003, ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/ , (Apríl 2008)
31. A. Biere, C. Sinz, *Decomposing SAT Problems into Connected Components*, Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) pp. 191-198, 2006
32. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, second edition, MIT Press, 2001
33. I. P. Gent, T. Walsh, *The SAT Phase Transition*, Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94), 2004
34. P. Cheeseman, B. Kanefsky, W. M. Taylor, *Where the Really Hard Problems Are*, Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, {IJCAI}-91, Sidney, Australia, pp. 331-337, 1991

35. A. Haken, *The Intaractability of Resolution*, Theoretical Computer Science, volume 39, pp. 297-308, 1985
36. Anbulagan, *SAT: Algorithms and Applications*, http://users.rsise.anu.edu.au/~anbu/slides/aaai07/SATtutoAAAI07-presented.pdf , poznámky k prednáške, (Máj, 2008)
37. P. Surynek, *Solving Difficult SAT Instances Using Greedy Clique Decomposition*, Proceedings of the 7th Symposium on Abstraction, Reformulation, and Approximation (SARA 2007), Whistler, Canada, 359-374, Lecture Notes in Artificial Intelligence, 4612, Springer, 2007
38. Wikipedia – The Free Encyclopedia, *Field-programmable gate array*, http://en.wikipedia.org/wiki/Field-programmable_gate_array , webová stránka, (Máj 2008)
39. F. A. Aloul, *Fadi Aloul's Home Page - SAT Benchmarks*, http://www.eecs.umich.edu/~faloul/benchmarks.html , osobné webové stránky, University of Michigan, USA, (Máj 2008)
40. J. M. Crawford, *International Competition and Sympoisium on Satisfiability Testing 1996*, http://www.cirl.uoregon.edu/crawford/beijing/ , Beijing, China, (Máj 2008)