České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Řešení problému splnitelnosti booleovské formule (SAT)

Jan Lomitzki

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

leden 2008

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 18.1. 2008 .....

Abstrakt

V této práci se podíváme to tajů algoritmů řešících problém splnitelnosti booleovské formule a představíme si některé dnešní SAT solvery. Akčoli se moderní SAT solvery neustále vylepšují a zrychlují, stále poskytují výsledek v podobě prvního nalezeného ohodnocení proměnných, které danou CNF formuli splňuje, popř. ohlásí, že formule je nesplnitelná. Pro praxi je to zcela dostačující. Nabízí se však myšleka implementovat algoritmus, který by generoval vektor ohodnocení obsahující i neúplně určené stavy (tzv. *don't cares*). Navrhnu možnosti modifikace algoritmů a pokusím se jeden z nich implementovat.

Abstract

In this work we uncover the veil from algorithms solving Boolean Satisfiability Problem and present some of the current state-of-the-art SAT solvers. Although current SAT solvers are still being improved, they keep returning one solution which satisfies given CNF furmula, or indicate that the formula is unsatisfiable. It's good enough for a common practice. But there's also an idea to implement an algorithm generating a vektor of assignments including don't cares. I'll propose the modifications of algorithms and I'll try to implement one of them.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Problém rozhodnout, zda daná booleovská formule může být pravdivá, se nazývá problém splnitelnosti (Boolean Satisfiability Problem, zkráceně SAT). SAT je důležitá a velmi studovaná oblast výpočetní techniky. V praxi je SAT základním problémem mnoha aplikací jako automatizace elektronického návrhu (EDA) či umělé inteligence (AI).

Cílem této práce je vypracovat stručnou rešerši některých dostupných open-source SAT řešičů. Popíšu principy, na kterých jsou založeny, jak fungují a jaké heuristiky používají a srovnám jejich výkonnost na testovacích datech. Dále se v této práci budu zabývat možnostmi modifikací algoritmů těchto řešičů. Půjde o to, aby upravené algoritmy generovaly vektory obsahující neúplně určené stavy (don't cares). Takto modifikovaný algoritmus implementuji a otestuji na zkušebních úlohách.

1.1 Problém splnitelnosti (SAT)

Mějme booleovskou formuli obsahující pouze operace **not** ($\neg$), **and** ($\wedge$) a **or** ($\vee$). Řešení splnitelnosti takové formule spočívá v nalezení ohodnocení jednotlivých proměnných, pro něž nabývá formule pravdivostní hodnoty *true*, či dokázat, že takové ohodnocení neexistuje. SAT patří do množiny NP–úplných problémů [12], čili lze jen težko čekat, že existuje algoritmus s polynomiální složitostí. Přesto byly vyvinuty algoritmy, které jsou dostatečně účinné pro řešení mnoha zajímavých SAT instancí. Tyto instance pocházejí z mnoha různých oblastí – mnoho praktických problémů z plánování AI, testování obvodů, ověřování mikroprocesorů, analýzy dosažitelnosti a další mohou být formulovány jako SAT instance.

1.2 Open-Source SAT solvery

Řešení SAT problému je jedním z nejstudovanějších oblastí výpočetní techniky. V současné době existuje několik velmi úspěšných SAT řešičů (SAT solverů), které se utkávají v mezinárodních SAT soutěžích [4]. Drtivá většina těchto SAT solverů je open-source, a tedy volně distribuovaná a modifikovatelná. Jako takový rozcestník do světa SAT bych doporučil [5]. Jelikož téměř všechny dnešní nejlepší solvery pracují na stejných principech a používají velmi podobné heuristiky, uvedu jen několik zástupců[1]. Jmenovitě to budou zChaff, WalkSAT a EBDDRES.

1.3 Členění práce

Rozdělení této práce je následující. V druhé kapitole se seznámíme s teoretickými základy problému splnitelnosti, probereme si základní algoritmy, kterými se dá problém SAT řešit. Podíváme se na současné řešiče (SAT solvery) a na typy heuristik, které používají. V třetí kapitole se budu zabývat možnostmi úprav probraných algoritmů, tak aby generovaly ohodnocení proměnných včetně neurčených stavů (don't care stavů).

[1] Bude-li to potřebné či vhodné, objeví se v textu i odkazy na další SAT solvery.

Kapitola 2

SAT algoritmy

2.1 Teoretický úvod

2.1.1 Booleovská formule

Booleovská formule je řetězec reprezentující booleovskou funkci, která obsahuje booleovské proměnné. Booleovská funkce je zobrazení z n-rozměrného booleovského prostoru (B^n) do booleovského prostoru $B = \{True, False\}$, kde n je počet proměnných funkce. Každá proměnná může nabývat hodnoty z B (říkáme, že proměnná je ohodnocena) nebo může být volná, což znamená, že nemá přiřazenu žádnou hodnotu.

Výroková booleovská formule je taková booleovská formule, která obsahuje pouze logické operace **not** ($\neg$), **and** ($\wedge$) a **or** ($\vee$). Odteď po zbytek práce budu operaci negace značit vodorovnou čárkou nad proměnnou, kterou neguje ($\overline{x}$). Pro názornost uveďme příklady výrokových formulí.

Příklad 2.1.1

$$\varphi_1 = (x_1 \vee \overline{x}_1 \vee x_2) \wedge (\overline{x}_3 \vee x_4) \wedge (x_1 \vee \overline{x}_3 \vee x_4)$$

Příklad 2.1.2

$$\varphi_2 = (x_2 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_2 \vee x_3) \wedge (\overline{x}_1) \wedge (x_1)$$

Ve zbytku práce, pokud nebude uvedeno jinak, předpokládám pouze výrokové formule, takže slovo „výrokové“ bude vypouštěno.

Formule mohou být vyjádřeny v tzv. *konjunktivní normální formě* (CNF), také někdy označované jako *Product of Sums* (POS).

- Formule v CNF sestává z konjunkce (logický **and**) jedné či více *klauzulí*.
- Každá klauzule je disjunkcí (logický **or**) jednoho či více *literálů*.

- Literál je logická proměnná, pak má *kladnou polaritu*[1], nebo její negace, v tom případě má *zápornou polaritu*[2].

Příklady formulí v CNF jsou uvedeny v Příkladech 2.1.1 a 2.1.2.

Klauzule je *redundantní* v CNF formuli právě, když jejím vypuštěním se nezmění booleovská funkce, jejíž reprezentací formule je. V opačném případě je klauzule *neredundantní*. V Příkladu 2.1.2 jsou první tři klauzule funkce φ_2 redundantní. Naopak klauzule (x_1) je neredundantní. Klauzule C_1 je *zahrnuta* klauzulí C_2 právě tehdy, když všechny literály z C_2 jsou zároveň obsaženy v C_1. Jestliže klauzule C je v CNF zahrnuta jinou klauzulí, je klauzule C redundantní. Funkce φ_1 z Příkladu 2.1.1 obsahuje klauzuli $(x_1 \vee \overline{x}_3 \vee x_4)$, která je zahrnuta klauzulí $(\overline{x}_3 \vee x_4)$. Klauzule C se nazývá *tautologickou klauzulí* právě, když obsahuje proměnnou v obou jejích polaritách. V Příkladu 2.1.1 je takovou tautologickou klauzulí $(x_1 \vee \overline{x}_1 \vee x_2)$. Klauzule se nazývá *prázdnou*, pokud neobsahuje žádné literály.

2.1.2 Resoluce

Mějme množinu literálů S, $\sum S$ bude představovat disjunkci všech literálů z S a $\prod S$ konjunkci všech literálů z S. Zákon konsensu říká, že je možné vyprodukovat redundantní klauzuli ze dvou klauzulí, pokud jsou splněny určité podmínky.

Věta 2.1.1 (Zákon konsensu) *Jestliže S_1 a S_2 jsou dvě množiny literálů a x je literál, pak:*

$$(x \vee \sum S_1) \wedge (\overline{x} \vee \sum S_2) \Leftrightarrow (x \vee \sum S_1) \wedge (\overline{x} \vee \sum S_2) \wedge (\sum S_1 \vee \sum S_2) \tag{2.1}$$

Proces vzniku klauzule $(\sum S_1 \vee \sum S_2)$ z klauzulí $(x \vee \sum S_1)$ a $(\overline{x} \vee \sum S_2)$ se nazývá *rezoluce*. Výsledná klauzule $(\sum S_1 \vee \sum S_2)$ se nazývá *rezolventa*.

Vzdálenost mezi dvěma množinami literálů S_1 a S_2 je počet literálů l takových, že $l \in S_1$ a $\overline{l} \in S_2$. Jestliže vdálenost mezi S_1 a S_2 je větší než 1, rezolventa z klauzulí $(x \vee \sum S_1)$ a $(\overline{x} \vee \sum S_2)$ je tautologickou klauzulí. Díky redundanci tautologických klauzulí se omezíme na používání rezoluce pouze dvou klauzulí se vzdáleností 1. Vzdálenost mezi klauzulemi C_1 a C_2 je definována jako vzdálenost mezi množinami literálů S_1 a S_2, kde S_1 (resp. S_2) je množina literálů objevujících se v klauzuli C_1 (resp. C_2).

[1] Také se označuje za literál v *kladné fázi*.
[2] Také se označuje za literál v *záporné fázi*.

2.2 Stručný přehled řešení SAT problému

Ačkoli probíhaly výzkumy již dříve, původní algoritmus pro řešení SAT je připisován pánům Matinu Davisovi a Hilary Putnamovi, kteří v roce 1960 navrhli na rezoluci založený algoritmus (DP algoritmus) [14]. Původně navržený algoritmus však trpěl nedostatkem v podobě paměťové exploze. S tímto problémem se vypořádali pánové Martin Davis, George Logemann a Donald W. Loveland [13]. V roce 1962 přišli s modifikovanou verzí, která namísto rezoluce využívá prohledávání. Nový algoritmus je označován jako DLL (popř. DPLL) algoritmus. Kromě těchto dvou algoritmů se v literatuře objevují i další algoritmy založené na různých principech. Mezi ty nejúspěšnější rozhodně patří algoritmus založený na Binary Decision Diagrams (BDD), Stålmarckův algoritmus, lokální metody a random walk[3]. Tyto techniky mají své silné stránky jako i své slabiny, ale všechny byly úspěšně implementovány do různých aplikačních oborů, aby se v nich potýkaly se SAT problémem.

2.2.1 Na rezoluci založený DP algoritmus

SAT řešící algoritmus navržený Davisem a Putnamem je algoritmus založený na rezoluci. Operace rezoluce je definována v Sekci 2.1.2. Dvě rezolvující klauzule vytvářejí novou klauzuli, je-li jejich *vzdálenost* rovna 1. Výsledná klauzule (tzv. *rezolventa*) obsahuje všechny literály, které se objevují v obou původních klauzulích kromě literálu, který se objevuje v obou s různými fázemi.

Mějme CNF formuli a proměnnou x, klauzule ve formuli mohou být rozděleny do tří množin:

1. klauzule, které neobsahují proměnnou x, označíme jako množinu A;
2. klauzule, které obsahují proměnnou x v její kladné fázi, označíme jako množinu B;
3. klauzule, které obsahují proměnnou x v její záporné fázi, označíme jako množinu C;

Máme-li dvě klauzule, jednu z množiny B a druhou z množiny C, můžeme vytvořit rezolventu, která neobsahuje proměnnou x. Davis a Putnam dokázali, že když vytvoříme všechny možné rezolventy mezi množinami B a C, zmažeme tautologické klauzule a zbytek přidáme ke klauzulím z množiny A, bude výsledná formule mít stejnou splnitelnost jako formule původní, ale o jeden literál míň.

[3]Nebudu překládat, ponechám v anglickém tvaru.

Kromě rezoluce DP algoritmus představuje také pravidlo pro eliminaci proměnné zvané *pravidlo jednotkového literálu*[4]. Toto pravidlo říká, že nachází-li se v CNF formuli klauzule, která obsahuje pouze jeden literál (tzv. jednotková klauzule), a nahradí-li se tento literál hodnotou *true*, pak má výsledná formule stejnou splnitelnost jako ta předchozí.

Davis a Putnam zavádějí další pravidlo pro eliminaci proměnné zvané *pravidlo unipolárního literálu*[5]. Literál nazveme *unipolární*, pokud se vyskytuje v CNF formuli právě v jedné fázi. Pravidlo unipolárního literálu říká, že odstraníme-li z CNF formule všechny klauzule obsahující nějaké unipolární literály, bude mít výsledná formule stejnou splnitelnost jako formule původní.

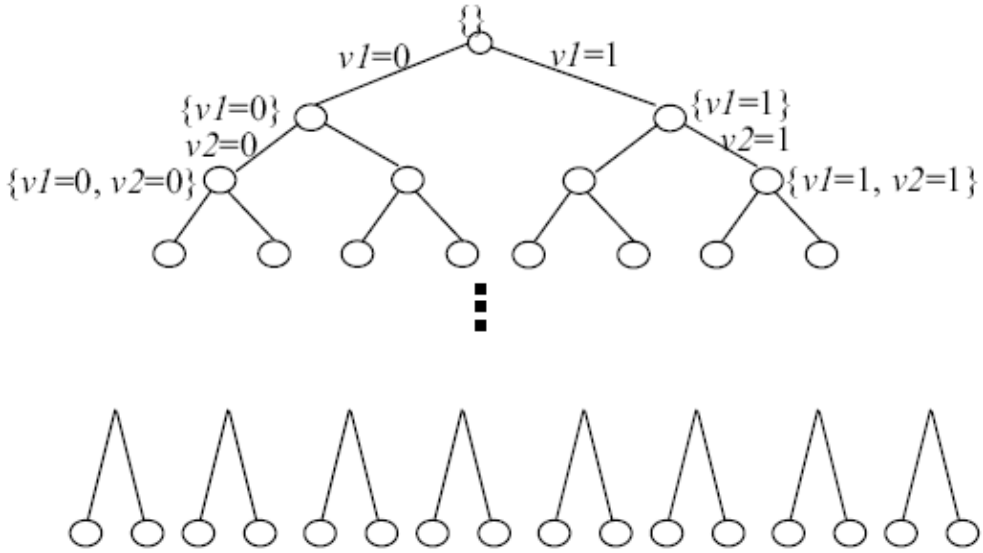
```
Davis-Putnam(φ) {
    if φ is empty return satisfiable
    if ε ∈ φ return not satisfiable
    if ∃c_j ∈ φ with s(c_j) = 1 (c_j = {l})
        then
            /* unit propagation */
            satisfy l and simplify φ to φ_l
            return Davis-Putnam(φ_l)
        else
            /* branching rule */
            select unassigned variable x_i and an assignment v(x_i) = a
            simplify φ to φ_{x_i}
            if Davis-Putnam(φ_{x_i}) == satisfiable
                then return satisfiable
                else
                    change assignment of x_i to v(x_i) = ¬a
                    simplify φ to φ_{¬x_i}
                    return Davis-Putnam(φ_{¬x_i})
}
```

Obrázek 2.1: Rekurzivní DP algoritmus

DP algoritmus si představíme pseudokódem podle [3] na Obrázku 2.1. Jestliže formule φ obsahuje jednotkovou klauzuli c_j ohsahující jediný literál l, pak je tento uspokojen a formule φ zjednodušena pomocí pravidel jednotkového a unipolárního literálu. Neobsahuje-li formule již žádné jednotkové klauzule, vybere se dosud neohodnocená proměnná x_i a ohodnotí se. Formule se opět zjednoduší použitím obou zmiňovaných pravidel. Nevede-li takové ohodnocení k zdárnému konci, daná proměnná x_i se překlopí na opačnou hodnotu. Rekurzivním voláním algoritmu na formuli se z ní „ukrajují“ proměnné

[4]angl. *unit literal rule*

[5]angl. *pure literal rule*



Obrázek 2.2: Stavový prostor SAT problému

jedna po druhé, až zůstane bez jediné proměnné. Jestliže výsledná formule je prázdnou formulí (neobsahuje žádnou klauzuli), pak je původní formule splnitelná. Jestliže výsledná formule obsahuje prázdnou klauzuli ε (klauzule neobsahující žádné literály), pak původní formule je nesplnitelná.

Pokaždé, když je v DP algoritmu rezolucí odstraněna proměnná, počet klauzulí ve formuli se může zvýšit v nejhorším případě kvadraticky. Proto jsou paměťové nároky DP algoritmu v nejhorším případě exponenciální. V praxi se dá použít pro řešení SAT instancí maximálně s desítkami proměnných.

2.2.2 DDL prohledávací algoritmus

Prohledávací algoritmus navržený Davisem, Logemannem a Lovelandem je zatím nejvíce studovaným algoritmem pro řešení SAT problému. Na rozdíl od tradičního DP algoritmu a některých dalších algoritmů probíraných v následujících sekcích, paměťové nároky DLL[6] algoritmu jsou většinou předvídatelné. Proto také SAT řešitelé založené na DLL zvládají velmi mohutné formule bez přetečení paměti.

Stavový prostor se často znázorňuje jako binární strom, viz Obrázek 2.2. Každý uzel stromu představuje booleovskou formuli pod určitým ohodnocením proměnných. Listy stromu představují úplná ohodnocení (tj. všechny proměnné jsou ohodnoceny), zatímco vnitřní uzly částečná ohodnocení (tj. některé proměnné jsou ohodnoceny, jiné jsou volné). Jestliže některý z listů je vyhodnocen jako *true*, pak je formule splnitelná. DLL algoritmus

[6] V některých publikacích označován také jako DPLL algoritmus.

se pokusí prohledat celý strom a určí, zda takový *true* list existuje. Pakliže ho najde, vytyčí cestu od kořene k tomuto listu, což nám dává konkrétní ohodnocení proměnných, pro která je formule splnitelná. V opačném případě je formule nesplnitelná.

Je zřejmé, že jsou potřeba některé ořezávací techniky, abychom se vyhli exponenciálním výpočtům všech listů při DLL prohledávání. Téměř u všech na prohledávání založených DLL algoritmech se vyžaduje, aby formule byly v CNF. CNF formule se splnitelná právě tehdy, když je splnitelná každá její klauzule. Každá klauzule je splnitelná právě tehdy, když je alespoň jeden její literál vyhodnocen jako pravdivý (*true*). Z toho plynou dvě důležitá pravidla DLL algoritmu:

- *Pravidlo jednotkového literálu*: Má-li klauzule všechny své literály kromě jednoho ohodnoceny *false*, pak zbývající volný literál musí nabýt hodnoty *true*, aby byla klauzule splnitelná. Takové klauzule se nazývají *jednotkové klauzule* a takový volný literál se nazývá *jednotkový literál*.

- *Pravidlo konfliktu*: Jestliže nějaké částečné ohodnocení zapříčiní nesplnitelnost klauzule (všechny její literály jsou nepravdivé (*false*)), pak v podstromu pod tímto částečným ohodnocením neexistuje uspokojující list. Řešitel se tedy musí vrátit a pokusit se najít uspokojující listy v jiných větvích stromu. Tomuto postupu se říká *zpětné prohledávání* neboli *backtracking*. Klauzule, která má všechny literály nepravdivé, se nazývá *konfliktující klauzule*[7].

Když je jednotkovému literálu přiřazena hodnota na základě pravidla jednotkového literálu, říkáme, že literál je *implikován*. Jednotková klauzule je *předchůdcem* proměnné odpovídající implikovanému literálu. Iterativní proces implikování všech jednotkových literálů po dobu existence jednotkových klauzulí, se nazývá *Boolean Constraint Propagation* (*BCP*).

BCP a backtracking spoluutvářejí základní operace DLL solveru. Tato dvě pravidla společně s několika dalšími technikami činí z DLL metody velmi silný nástroj. Většina dnešních SAT solverů je právě založena na DLL algoritmu.

Tradičně je DLL algoritmus prezentován jako rekurzivní funkce, avšak současné SAT solvery používají iterativní podobu DLL algoritmu, který je popsán pseudokódem na Obrázku 2.3.

Na začátku prohledávání jsou všechny proměnné *volné* (neohodnocené). Funkce `preprocess()` se pokusí zjistit, jestli formule nemá triviální řešení, popřípadě určit, dají-li se některé proměnné ohodnotit bez nutností větvení.

[7] Neplést s pojmem *konfliktní klauzule*

Pokud `preprocess()` nerozhodne o splnitelnosti formule, začíná hlavní cyklus. Větvení se provádí ve funkci `decide_next_branch()` tak, že se vybere volná proměnná (na základě nějaké heuristiky) a určí se jí logická hodnota. Této proměnné se říká *určená proměnná* a uroveň, na které byla určena se nazývá *úroveň určení*. Funkce `deduce()` je vlastně implementací samotné BCP. Všechny proměnné implikované během propagace mají stejnou úroveň určení jako určená proměnná.

Pokud jsou po `deduce()` všechny klauzule splněné, formule je spnitelná. Pokud existuje *konfliktující klauzule* (tj. klauzule se všemi literály nepravdivými), pak nemůže daná větev vést k uspokojujícímu ohodnocení, takže solver bude muset provést backtracking (neboli zpětné prohledávání). A pokud daná instance není pod aktuálním ohodnocením ani splnitelná, ani v konfliktu, pak se provede další větvení a proces se opakuje.

Když se objeví konflikt, solver se vrátí a anuluje některé větve (funkce `backtrack()`). Úroveň, na kterou se má vrátit určí funkce `analyze_conflict()`. Backtracking na úroveň menší jak 0 značí, že i bez jakéhokoli větvení je instance stále nesplnitelná. V takovém případě solver prohlásí formuli za nesplnitelnou. V téže funkci se provádí analýza a zaznamenávají se informace o konfliktu, které v budoucnu pomohou ořezat stavový prostor.

2.2.3 Binary Decision Diagrams

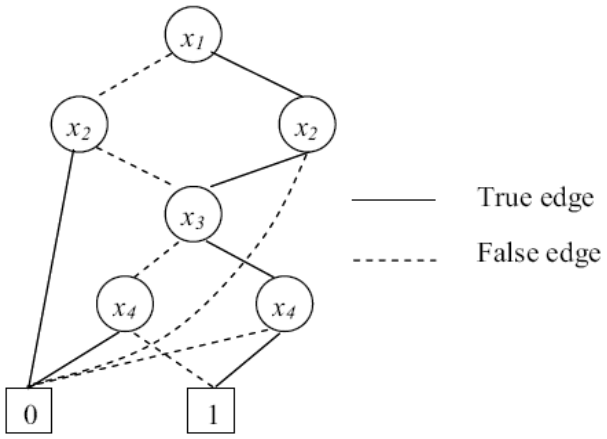
Randal E. Bryant pro zobrazení booleovské funkce navrhnul v roce 1986 Reduced Ordered Binary Decision Diagrams (ROBDD) [11]. ROBDD je orientovaný acyklický graf, jehož uzly představují booleovské funkce. V ROBDD nalezneme dva druhy uzlů: *terminální* a *běžné* uzly. Dva terminální uzly zastupují logické funkce *true* a *false* a nemají žádné vystupující hrany. Každý běžný uzel je spjat s proměnnou a má dvě vystupující hrany. Jestliže běžný uzel reprezentuje funkci f a je spjat s proměnnou x, pak z něj vystupující hrany představují funkce $f(x = 0)$ a $f(x = 1)$. V ROBDD má každá proměnná určité pořadové číslo, ROBDD je seřazený, což znamená, že všechny cesty z běžného uzlu k terminálnímu musí odpovídat řazení proměnných. ROBDD je také redukovaný v tom smyslu, že žádný s uzlů nemá dvě vystupující hrany, které by vstupovaly do stejného uzlu a žádné dva uzly nemají stejný podgraf. Příklad ROBDD je ukázán na Obrázku 2.4.

ROBDD je kanonické zobrazení booleovských funkcí. Dva ROBDD zobrazují stejnou funkci právě tehdy, když jsou oba grafy izomorfní. Abychom mohli zjistit, je-li booleovská formule splnitelná, stačí vyrobit ROBDD reprezentaci formule a vyzkoušet, zda je izomorfní s grafem reprezentujícím konstantní hodnotu *false*. Když není, potom je formule splnitelná, jinak je nesplnitelná.

Naneštěstí vytvoření ROBDD reprezentující danou booleovskou funkci není jednodu-

```
DLL()
{
    status = preprocess();
    if (status!=UNKNOWN)
        return status;
    while(1) {
        decide_next_branch();
        while (true)
        {
            status = deduce();
            if (status == CONFLICT)
            {
                blevel = analyze_conflict();
                if (blevel < 0)
                    return UNSATISFIABLE;
                else
                    backtrack(blevel);
            }
            else if (status == SATISFIABLE)
                return SATISFIABLE;
            else break;
        }
    }
}
```

Obrázek 2.3: Iterativní podoba DLL algoritmu



Obrázek 2.4: Jednoduchý ROBDD představující funkci $(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$

chou záležitostí. Velikost ROBDD pro danou funkci je velmi citlivá na řazení proměnných. Například velikost ROBDD reprezentace sčítačky může být lineární nebo exponenciální v závislosti na řazení proměnných. Ačkoli se pomocí heuristik dá nalézt relativně dobré řazení, výsledek není zaručen. Je též známo, že velikost ROBDD reprezentace určitých booleovských funkcí (např. funkce popisující výstupy celočíselné násobičky) je exponenciální bez ohledu na řazení proměnných [11]. Z těchto důvodů je praktické využití ROBDD omezeno na booleovské funkce o velikosti maximálně několika stovek proměnných.

2.2.4 Stålmarckův algoritmus

Gunnar Stålmarck navrhnul na začátku 80. let 20. stol. algoritmus, jenž pracuje na principu ověřování tautologie. K tomu, aby algoritmus zjistil, je-li booleovská formule splnitelná, potřebuje nejprve prověřit, je-li její komplement tautologií. Následující popis vychází převážně z [23].

Stålmackův algoritmus používá datovou strukturu nazývanou *triplet*. Triplet (p, q, r) reprezentuje logický vztah:

$$p \Leftrightarrow (q \Rightarrow r)$$

kde p, q a r jsou literály. Jakákoli logická funkce může být popsána pouze pomocí implikace ($\Rightarrow$) a logické hodnoty *false*. Na ostatní logické operace (log. součin, součet, negace) se aplikují následující transformace.

$$\begin{aligned} A \wedge B &\Leftrightarrow \neg(A \Rightarrow \neg B) \\ A \vee B &\Leftrightarrow \neg A \Rightarrow B \\ \neg\neg A &\Leftrightarrow A \\ \neg A &\Leftrightarrow A \Rightarrow false \end{aligned}$$

Logická funkce může být popsána množinou tripletů, kde je každá logická vazba zastoupena novou proměnnou. Například formule $p \Rightarrow (q \Rightarrow p)$ může být popsána jako dvojice tripletů (b_1, q, p), (b_2, p, b_1), kde b_2 je sama původní formule.

Pro zjednodušení tripletů a odvození nových výsledků se používá několik jednoduchých pravidel. Některá si uvedeme níže:

$$(0, y, z) : y = 1, z = 0$$

$$(x, y, 1) : x = 1$$

$$(x, 0, z) : x = 1$$

$$(x, 1, z) : x = z$$

$$(x, y, 0) : x = \overline{y}$$

$$(x, x, z) : x = 1, z = 1$$

$$(x, y, y) : x = 1$$

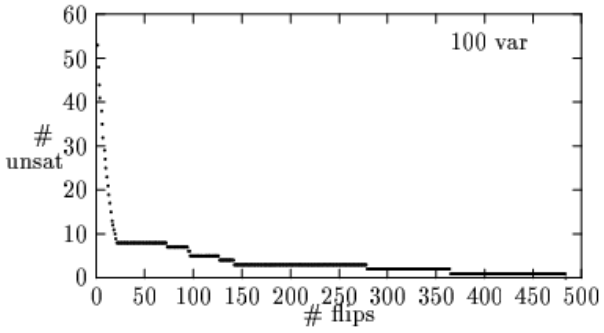
Existují tři triplety, které reprezentují kontradikci, a nazývají se *terminální triplety.* Jsou to triplety $(1, 1, 0)$, $(0, x, 1)$ a $(0, 0, x)$.Pokud se některý z těchto terminálních tripletů ve formuli objeví, můžeme formuli prohlásit za kontradiktorickou. V takovém případě formule nemůže být nepravdivá, čili je tautologickou. Naopak, pokud se předpokládá, že formule je pravdivá, a nalezneme terminální triplet, pak je formule nesplnitelná. Na začátku algoritmus předpokládá, že je formule nepravdivá a snaží se odvodit terminální triplet.

Samotné používání výše zmíněných jednoduchých pravidel nemusí vést k řešení splnitelnosti. Stålmarckův algoritmus proto spoléhá na tzv. *pravidlo dilematu* pro větvení a prohledávání. První úroveň pravidla dilematu vybere z formule jednu proměnnou, ohodnotí ji *true* a pomocí jednoduchých pravidel odvodí množinu výsledků. Poté ji ohodnotí *false* a odvodí jinou množinu výsledků. Jestliže jedna z větví selže (tzn. obsahuje kontradikci), pak proměnná musí nabýt opačné hodnoty. Pokud ani v jednom případě nenarazí na kontradikci, pak průnik těchto dvou množin musí být pro formuli vždy pravdivý, a proto může být přidán k původní formuli. Provede-li se tento postup pro všechny proměnné, dosáhne se *1-saturace.* Pokud 1-saturace nedokáže prokázat tautologii formule, pokračuje se s *2-saturací* tak, že se zároveň vyberou dvě proměnné a ohodnotí se čtyřmi kombinacemi jejich hodnot. Pokud 2-saturace také selže, pokračuje se 3-saturací atd. Máme-li formuli s n proměnnými, Stålmarckův algoritmus potřebuje maximálně n-saturací k tomu, aby zjistil splnitelnost formule.

2.2.5 Stochastické algoritmy

Všechny algoritmy pro řešení problému splnitelnosti (SAT) uvedené v předcházejících sekcích patří do skupiny tzv. *úplných* algoritmů. Dáme-li jim dostatek výpočetního času a paměti, úplný SAT algoritmus nám vždy najde řešení nebo dokáže, že řešení neexistuje. Naproti tomu stochastické SAT algoritmy jsou jinou třídou SAT algoritmů, založenou na technikách matematické optimalizace. Stochastické metody nemohou dokázat nesplnitelnost SAT instance, ale dostanou-li dostatek času, mohou nalést řešení splnitelné instance. Pro některé třídy benchmarků jako je např. hard random SAT, stochastické metody mohou značně předčit stávající úplné algoritmy založené na systematickém prohledávání (např. DLL algoritmus).

Základní idea stochastických algoritmů je taková, že uvažuje počet nesplněných klauzulí jako cílovou funkci, kterou se snaží minimalizovat ohodnocováním proměnných. Po



Obrázek 2.5: Minimalizace cílové funkce (počet nesplněných klauzulí)

léta výzkumníci experimentovali s rozličnými optimalizačními technikami jako simulované ochlazování, genetické algoritmy, neuronové sítě a lokální metody. Ukázalo se, že algoritmy založené na lokálních metodách mají pro SAT nejlepší výsledky.

Stavový prostor je booleovský prostor různých kombinací ohodnocení všech proměnných. Na začátku máme náhodné ohodnocení proměnných, základním úkonem algoritmu lokální metody je pokusit se „pohnout“ z ohodnocení takovým způsobem, aby se snížila cílová funkce (spravidla počet nesplněných klauzulí). Tento „pohyb“ je u většiny algoritmů pouhým překlopením hodnoty proměnné (změnou hodnoty *false* na *true* a naopak). Proto Hammingova vzdálenost ohodnocení před překlopením a po něm je rovna jedné. Obrázek 2.5 ukazuje, že se jedná o „sestupný pohyb“, který ovšem může uvíznout v lokálním minimu. Různé lokální metody se především liší v tom, kterou proměnnou překlopit a jak uniknout z lokálního minima. Z lokálního minima se lze dostat pouhým restartováním algoritmu s novým náhodným inicializačním ohodnocením. Ale existují i jiné metody popsané v [22]:

- *Simulované ochlazování* vnáší do algoritmů lokálních metod „vzestupné pohyby“. Uvažujme δ jako cílovou funkci udávající změnu v počtu nesplněných klauzulí při překlopení vybrané proměnné. Jestliže $\delta \leq 0$ (sestupný pohyb nebo úkroky), pak se proměnná překlopí. Když je $\delta > 0$ (vzestupný pohyb, který zvýší počet nesplněných klauzulí), potom překlopí proměnnou s pravděpodobností $e^{\frac{-\delta}{T}}$, kde T je parametr zvaný *teplota*. Teplota může být konstantní nebo pomalu se snižující k nule podle daného plánu chladnutí.

- *Random walk* je strategie výběru proměnné pro překlopení. S určitou pravděpodobností p se vybere a překlopí náhodná proměnná z některé nesplněné klauzule a s pravděpodobností $1 - p$ se provede nejlepší lokální pohyb (tj. vybere a překlopí se proměnná, která povede k minimalizaci počtu nesplněných klauzulí).

2.3 Současné SAT solvery

V předchozí sekci jsem popsal, jak pracují algoritmy pro nalezení řešení problému splnitelnosti booleovských formulí. Tato sekce se zaměřuje na konkrétní implementace současných SAT solverů. Rodina SAT solverů je obrovská a těžko ji obsáhnout celou. Proto se zaměřím „jen“ na tři vybrané zástupce. Většina dnešních SAT solverů je založena na prohledávacím DLL algoritmu (viz Sekce 2.2.2), který ve spolupráci s technikami jako je konflikty řízené učení, nechronologický backtracking, aj. dosahuje velmi dobrých výsledků. Nejprve si však ukážeme zástupce neúplných algoritmů.

2.3.1 WalkSAT

WalkSAT byl navržen McAllesterem, Selmanem a Kautzem v roce 1997 [19]. Jedná se o stochastický algoritmus používající hill-climbingu na cílovou funkci. Cílová funkce je vždy definována jako počet klauzulí, které jsou nesplněné pod aktuálním ohodnocením proměnných. WalkSAT je popsán pseudokódem na Obrázku 2.6. Na začátku se náhodně vygeneruje ohodnocení všech proměnných. V každém kroku, pokud formule není splněná pod aktuálním ohodnocením, algoritmus vybere proměnnou z náhodné nesplněné klauzule na základě nějaké heuristiky. Vybranou proměnnou poté překlopí (změní pravdivostní hodnotu). WalkSAT uživateli nabízí několi heuristik pro výběr proměnné [16]:

Náhodný výběr Vybere se náhodná proměnná.

Nejlepší výběr Vybere se taková proměnná, která minimalizuje počet klauzulí, které by se po překlopení znesplnily[8] (tzv. *breakcount*).

Tabu Vybere se proměnná, která má nulový breakcount, jinak s pravděpodobností p vybere náhodnou proměnnou a s pravnědodobností $1-p$ vybere takovou proměnnou, která minimalizuje počet klauzulí, které by se po překlopení znesplnily. Heuristika však používá tabu seznam délky tl. Poté, co byla proměnná překlopena, nesmí být překlopena v následujících tl krocích. Pokud jsou v klauzuli všechny proměnné tabu, nepřeklopí se žádná proměnná (tzv. *nulové překlopení*).

Novelty Mějme proměnné v klauzuli seřazeny podle jejich skóre, tj. rozdílu mezi celkovým počtem splněných klauzulí po a před překlopením. Pokud není proměnná s nejvyšším skóre tou, která byla překlopena naposledy, pak se vybere, jinak se vy-

[8] Toto podle všeho nové české slovo má následující význam: sloveso *znesplnit* znamená změnu stavu ze stavu *splněného* do stavu *nesplněného*

bere pouze s pravděpodobností $1-p$. Ve zbývajících případech se vybere proměnná s druhým nejvyšším skóre.

R-Novelty Podobné jako Novelty, ale v případě, kdy proměnná s nejvyšším skóre byla naposledy překlopenou proměnnou, výběr mezi nejlepší a druhou nejlepší proměnnou pravděpodobnostně závisí na jejich rozdílu skóre. Navíc každý 100. krok se místo této heuristiky použije Náhodný výběr.

```
WalkSAT(F, maxTries, maxSteps, Select)
{
  for try = 1 to maxTries {
    A = randomly generated assignment of the variables in F
    for step = 1 to maxSteps {
      if A satisfies F
        return A;
      C = randomly selected clause which is unsatisfied under A;
      V = variable form A selected according to a heuristic Select;
      A = A with V flipped;
    }
  }
  return No solution found
}
```

Obrázek 2.6: Local search algoritmus WalkSATu

Po uplynutí stanoveného počtu interací (*maxTries*), algoritmus ohlásí, že nenašel žádné řešení. To ovšem ještě nutně neznamená, že daná formule je nesplnitelná. Dáme-li WalkSATu dostatek výpočetního času, může nám najít řešení splnitelné formule, avšak za žádných okolností nám neoznačí nesplnitelnou formuli za nesplnitelnou[9]. WalkSAT tedy nemůže rozhodnout o nesplnitelnosti formule, pouze o její splnitelnosti tím, že nalezne řešení.

2.3.2 zChaff

SAT solver zChaff implementuje Chaff algoritmus [20], který je založen na prohledávacím DLL algoritmu (viz Obrázek 2.3).

Strategie určení proměnné

Stejně jako Chaff algoritmus i zChaff používá pro určení proměnné heuristiku zvanou *Variable State Independent Decaying Sum* (VSIDS). VSIDS uchovává skóre každého

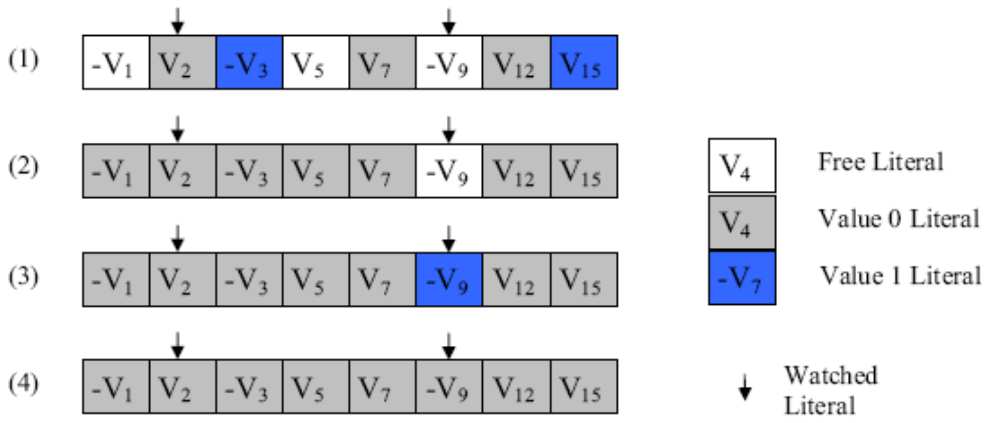
[9]Je to evidentní i z pseudokódu 2.6, v němž se nikde nenachází `return UNSAT`, jak je tomu například u DLL algoritmu.

literálu a přikládá větší váhu naposledy přidaným klauzulím. Heuristika VSIDS pracuje následujícím způsobem:

1. Každá proměnná v obou polaritách má čítač inicializovaný na 0.
2. Když je klauzule přidána do databáze[10], čítač spjatý s každým literálem v této klauzuli je inkrementován.
3. Pro každé určení proměnné je vybrána volná proměnná a polarita s nejvyšším čítačem.
4. Pokud je takových proměnných víc, vybere se náhodně jedna z nich.
5. Čítače jsou pravidelně děleny konstantou.

BCP engine

V praxi, pro většinu SAT problémů, ztráví solver v BCP drtivou většinu výpočetního času (kolem 90%), tudíž implementace efektivního BCP enginu je klíčem k úspěchu. Připomeňme, že BCP je tranzitivní proces, při kterém se na základě současného ohodnocení proměnných odvozují (implikují) další volné proměnné. Proces končí, pokud již nemůže odvodit žádné další proměnné nebo narazí-li na konflikt.



Obrázek 2.7: Čtyři stavy BCP schématu Two Literal Watching

Pro efektivní propagaci je třeba projít všechny jednotkové klauzule, ve kterých se nachází literál, který je pod aktuálním ohodnocením nepravdivý. Má-li klauzule n literálů, stává se jednotkovou klauzulí, pokud má $n-1$ literálů nepravdivých. Není třeba tedy procházet klauzule, které mají $n-2$ a méně nepravdivých literálů. zChaff k tomuto

[10]Výraz *clause database* je datový prostor, kde jsou uložený všechny klauzule právě řešené formule.

účelu implementuje techniku zvanou *Two Literal Watching* (Obrázek 2.7). V každé klauzuli vybereme dva literály různé od *false*. Dokud se jeden z nich nestane nepravdivým, máme zaručeno, že v klauzuli není víc jak $n - 2$ nepravdivých literálů (tj. klauzule není jednotková). Každou klauzuli navštívíme pouze tehdy, pokud se jeden z jejich *hlídaných literálů* stane nepravdivým. V takovém případě se snažíme v klauzuli nalézt jiný literál l, který není nepravdivý. Mohou nastat čtyři možnosti:

1. Pokud takový literál l exitstuje a je různý od druhého hlídaného literálu, potom jím stávající nepravdivý hlídaný literál nahradíme.

2. Pokud takovým literálem l je pouze druhý hlídaný literál a ten je volný, potom je klauzule jednotková.

3. Pokud takovým literálem l je pouze druhý hlídaný literál a ten je pravdivý, potom je klauzule splněná a nemusíme nic provádět.

4. Pokud takový literál l neexistuje, potom je klauzule konfliktující (tj. všechny literály jsou nepravdivé).

Obrovská výhoda Two Literal Watching tkví v tom, že na rozdíl od jiných schémat (např. Head/Tail) není během backtrackingu třeba upravovat hlídané literály v databázi klauzulí.

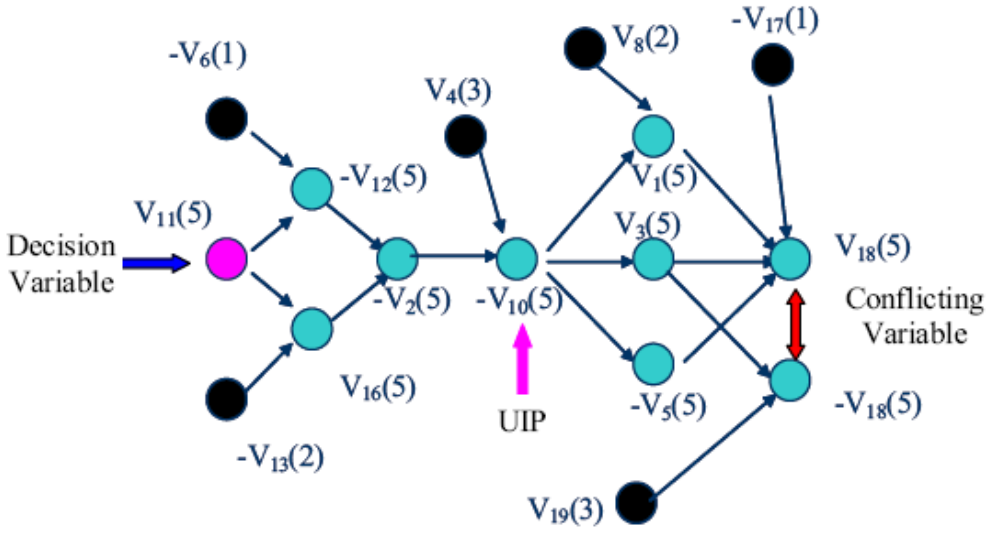
Analýza konfliktu a učení

Pro analýzu konfliktu a konfliktem řízené učení klauzulí používá zChaff (podobně jako převážná většina jiných SAT solverů) techniky navržené pro solver GRASP [18].

Pro popis analýzy konfliktu se často používá *implikační graf*, viz Obrázek 2.8. Jedná se o orientovaný acyklický graf, kde jednotlivé uzly představují ohodnocení proměnné (kladná proměnná pro ohodnocení *true*, záporná proměnná pro ohodnocení *false*). Hrany do uzlu vstupující představují důvod, který vede k ohodnocení proměnné, např. na Obrázku 2.8 vstupující hrany do uzlu $-V_{10}$ vedou z uzlů V_4 a $-V_2$, což znamená, pokud je V_4 *true* a V_2 *false*, potom V_{10} musí být *false*. Určené proměnné nemají vstupující hrany. Úroveň určení je na obrázku značena v závorkách.

Uzel a **dominuje** uzlu b právě tehdy, když každá cesta od určené proměnné na úrovni určení uzlu a k uzlu b vede přes uzel a. Uzel, který dominuje uzlům vztahujícím se ke konfliktní proměnné, se jmenuje *jedinečný implikačí bod* (*UIP*). V našem případě jsou proměnné V_{11}, V_2 a V_{10} všechny UIP.

Algoritmus zChaffu používá 1-UIP schématu učení. Existují i další schémata (2-UIP, All-UIP, a další), ale experimentálně se zjistilo, že 1-UIP má nejlepší výsledky. Implikační



Obrázek 2.8: Typický implikační graf

graf se směrem od konfliktní proměnné prohledá, dokud se nenajde první UIP (tj. UIP nejblíž konfliktní proměnné). První UIP společně s uzly z nižších úrovní určení, z nichž vystupují hrany vedoucí ke konfliktní proměnné, vedou ke konfliktu. V našem příkladu tedy:

$$(V_8 \wedge \overline{V_{10}} \wedge \overline{V_{17}} \wedge V_{19}) \Rightarrow KONFLIKT$$

Kontrapozice:

$$True \Rightarrow \overline{(V_8 \wedge \overline{V_{10}} \wedge \overline{V_{17}} \wedge V_{19})}$$

Po použití De Morganova zákona:

$$True \Rightarrow (\overline{V_8} \vee V_{10} \vee V_{17} \vee \overline{V_{19}})$$

$(\overline{V_8} \vee V_{10} \vee V_{17} \vee \overline{V_{19}})$ je tedy konfliktní klauzule, která se přidá do databáze klauzulí (naučí se). Po provedení backtrackingu se z ní stane jednotková klauzule (proměnná V_{10} se během backtrackingu stane volnou), takže okamžitě implikuje nové ohodnocení, které se může uplatnit při propagaci.

Restarty

Další zajímavou vlastností zChaffu je schopnost restartů. Proces prohledávání se zastaví, ohodnocení proměnných se smaže a celý proces se restartuje. Avšak výsledky analýz jsou stále přítomny. Naučené klauzule zůstávají v databázi, v důsledku čehož solver po restartu nebude provádět stejné analýzy jako předtím.

zChaff není samozřejmě jediným DLL SAT solverem. Z dalších úspěšných SAT solverů jmenujme např. MiniSat [15], RSat [6], NanoSat [10], či JeruSat [21].

2.3.3 EBDDRES

SAT solver EBDDRES je zástupcem řešičů založených na ROBDD (viz Sekce 2.2.3). Vzhledem k naprosté absenci dokumentace vychází následující představení heuristik z analýzy kódu a z poznatků pana Balcárka [9].

Jak již bylo řečeno dříve, nalezení optimálního řazení proměnných má zásadní vliv na velikost ROBDD. EBDDRES k tomu používá statických metod řazení:

Řazení dané strukturou formule Pořadí proměnných určuje pořadí proměnných, v kterém je daná formule zapsaná. Toto pořadí může uživatel zvolit i obrácené.

Řazení podle četnosti výskytu proměnné Jedná se o zjednodušený algoritmu VSIDS, popsaný v Sekci 2.3.2. Každá proměnná má pro každou svou polaritu čítač nesoucí informaci o počtů výskytů dané proměnné ve formuli. Proměnné sa pak seřadí od proměnné s nejvyšším výskytem po proměnnou s nejnižším výskytem.

DFS řazení Tato metoda řadí proměnné v závislosti na počtu výskytů a také v závislosti na „příbuznosti“ k proměnné v rámci klauzule. Algoritmus iterativně vybírá proměnné ze zásobníku, ve kterém jsou proměnné seřazeny sestupně podle počtu výskytů, a pakliže již není vřazena do výsledného pole seřazení, přidává ji tam. Následně projde všechny klauzule, ve kterých se vybraná proměnná vyskytuje, a vloží na zásobník všechny proměnné různé od té vybrané. Popsaný proces se opakuje, dokud není zásobník prázdný.

Algoritmus FORCE FORCE vychází již z existujícího uspořádání proměnných, které iterativně upravuje tak dlouho, dokud se nepřestane zlepšovat metrika řazení. Touto metrikou je tzv. *total span*, který je definován jako součet spanů jednotlivých klauzulí. Span klauzule je definován jako rozdíl mezi nejmenšími a největšími proměnnými v klauzuli[11]. V každé iteraci algoritmus spočítá tzv. *center of gravity* COG všech klauzulí, dále pro každou proměnnou vypočítá na základě jednotlivých COG zlepšující pořadí. Pro pseudokód a detailnější popis tohoto algoritmu odkazuji na [8].

I přes tyto heuristiky řazení proměnných může velikost ROBDD dosáhnout obrovských rozměrů a zaplnit paměť. Proto EBDDRES zavádí také možnost eliminace proměnných, která uvolní část paměti, a s níž se celý proces hledání splnitelnosti zrychluje. Více viz [17].

[11] Ve smyslu seřazení proměnných, tedy nejmenší proměnné jsou blízko jednomu konci daného seřazení, největší proměnné jsou blízko opačnému konci.

2.4 Srovnání SAT solverů

Na internetu existuje mnoho open-source SAT solverů, které se každoročně (s přestávkou v roce 2006) utkávají v mezinárodní SAT soutěži [4]. K otestování jsem vybral pět z nich: WalkSAT v46, zChaff v3.12, EBDRESS v1.1, MiniSat v1.14 a RSat v2.01. Testováno bylo na počítači s Intel Pentium 4 3GHz a 1024MB RAM s OS Windows XP a přes rozhraní Cygwin. V následujících sekcích provedeme srovnání jak jednotlivých SAT řešičů, tak i porovnáme výkonnost jejich heuristik.

2.4.1 Použité benchmarky

DIMACS SAT BENCHMARKS					
Soubor	Kód	Proměnných	Klauzulí	Klau./Prom.	Splnitelné?
ii16a1.cnf	(Re)	1650	19368	11,74	Ano
ssa6288-047.cnf	(SSA)	10410	34238	3,29	Ne
dubois27.cnf	(Dub)	81	216	2,67	Ne
f1000.cnf	(LRan)	1000	4250	4,25	Ano
f2000.cnf	(LRan)	2000	8500	4,25	Ano
g125.17.cnf	(GC)	2125	66272	31,19	Ano
g250.29.cnf	(GC)	7250	454622	62,71	Ano
hanoi5.cnf	(Han)	1931	14468	7,49	Ano
jnh210.cnf	(JNH)	100	800	8,00	Ano
jnh211.cnf	(JNH)	100	800	8,00	Ne
aim-200-2_0-no-1.cnf	(AIM)	200	400	2,00	Ne
aim-200-6_0-yes1-1.cnf	(AIM)	200	1200	6,00	Ano

Tabulka 2.1: Použité benchmarky

V Tabulce 2.1 je přehled benchmarků, které posloužily k testování. Jedná se o DIMACS benchmarky, které jsou volně ke stažení z [1]. Popišme si, co znamenají kódy u jednotlivých souborů:

Re Od Mauricio Resende `mgcr@research.att.com`. Z „Acontinuous approach to inductive inference“, by Kamath, Karmarkar, Ramakrishanan, and Resende (Mathematical Programming 57: 215–238(1992)).

SSA (Od Allen Van Gelder `avg@cs.ucsd.edu` a Yumi Tsuji `tsuji@cse.ucsc.edu`) Instance analýzy chyb obvodu: hledání “single–stuck–at” chyby.

Dub (Od Olivier Dubois `dubois@laforia.ibp.fr`) Instance z kódu gensathard.c.

LRan (Od Bart Selman `selman@research.att.com`) Velké random satisfiable instance.

GC (Od Bart Selman `selman@research.att.com`) Složitý problém obarvení grafu ve formě booleovské splnitelnosti.

Han (Od Bart Selman `selman@research.att.com`) Zakódování problému Hanoiských věží.

JNH (Od John Hooker `jh38+@andrew.cmu.edu`) Množina random instancí vygenerovaných bez jednotkových klauzulí a s nastavenou hustotou na složité hodnoty.

AIM (Od Eiji Miyano `miyano@csce.kyushu-u.ac.jp`) Uměle generované 3–sat instance. Všechny splnitenlé instance mají právě jedno řešení.

2.4.2 Porovnání heuristik WalkSATu

V Sekci 2.3.1 jsem podrobně popsal heuristiky, které WalkSAT používá. Připomínám, že WalkSAT je zástupcem neúplných algoritmů, a tudíž nemůže rozhodnout, zda je daná formule nesplnitelná. Testování na nesplnitelných benchmarcích tedy nemá smysl a budu se soustředit jen na ty splnitelné. WalkSAT pracuje s náhodným ohodnocováním, takže pokaždé mu může trvat jinou dobu, než nalezne řešení. Proto jsem provedl pět měření pro každou heuristiku a výsledné časy sprůměroval. Vstupní parametry jsem nastavil $restart$[12]$= 50$, $cutoff$[13]$= 500000$, dále chci nalézt jen jedno řešení, takže $numsol = 1$.

Z Tabulky 2.2 je patrno, že neexistuje jasná volba heuristiky. Ani jedna z nich není tou ultimátní volbou, která by řešila všechny SAT problémy v přijatelném čase. Jako nejméně úspěšná heuristika z testu vychází jednoznačně **Náhodný výběr**. Heuristiky **Novelty** a **RNovelty** mají dobré výsledky, ale selhávají při problému obarvení grafu. Zlatou střední cestou se jeví volba heuristiky **Nejlepší výběr** a **Tabu** s vhodně zvoleným argumentem n.

2.4.3 Porovnání heuristik EBDDRESu

SAT solver EBDDRES nabízí několi heuristik řazení proměnných (viz Sekce 2.3.3). Tabulka 2.5 srovnává heuristiky řazení se zapnutým algoritmem **Force** a bez něho. Pro přehlednost tabulky byly vypuštěny benchmarky, ve kterých EBDDRES nebyl schopen nalézt řešení, jako ukázku ponechávám pouze benchmark f1000.cnf.

[12]Odpovídá proměnné *try* z pseudokódu na Obrázku 2.6.
[13]Odpovídá proměnné *step* z pseudokódu na Obrázku 2.6.

Benchmark	Random	Best	Tabu(n)	Novelty	RNovelty
ii16a1.cnf	-	0,003	0(libovolná n)	0,103	7,519
f1000.cnf	-	0,416	4,216(10) 6,856(3)	0,584	0,447
f2000.cnf	-	2,988	-(2) -(1)	9,688	10,456
g125.17.cnf	-	-	28,840(3) 11,575(1)	89,821	-
g250.29.cnf	-	-	32,041(2) 46,09(1)	-	-
hanoi5.cnf	-	-	-(libovolná n)	-	-
jnh210.cnf	1,081	<0,001	0,031(200) <0,001(nízká n)	<0,001	0,09
aim-200-6_0-yes1-1.cnf	-	0,26	4,625(20) 0,031(10)	<0,001	<0,001

Tabulka 2.2: Srovnání heuristik WalkSATu. Údaje jsou zaznamenány v sekundách, parametr u heuristiky Tabu značí délku tabu seznamu. Proškrtaná pole značí, že WalkSAT nebyl schopen nalézt řešení.

EBDDRES používá k určení splnitelnosti formule binární rozhodovací diagramy ROBDD, které jsou velmi náročně na paměť. Není tedy divu, že zvládl vyřešit pouze formule, jejichž počet klauzulí je v řádu stovek. Algoritmus Force nalezení řešení urychluje, avšak u benchmarku aim-200-6_0-yes1-1.cnf došlo k mírnému zpomalení, které je způsobeno nejspíš dlouhým výpočtem samotného Force algoritmu.

Proměnných	Klauzulí	Poměr	EBDDRES
10	80	8,0	<0,001
20	80	4,0	0,02
30	80	2,7	0,25
50	80	1,6	29,02
55	80	1,5	11,02
60	80	1,3	-

Tabulka 2.3: Vliv počtu proměnných na výkon EBDDRESu

Proměnných	Klauzulí	Poměr	EBDDRES
40	60	1,5	0,94
40	80	2,0	3,23
40	100	2,5	6,12
40	200	5,0	6,34
40	2000	50,0	12,97
40	20000	500,0	10,25

Tabulka 2.4: Vliv počtu klauzulí na výkon EBDDRESu

Testování EBDDRESu na náhodně generovaných instancích (algoritmus viz [7]), jehož výsledky shrnují Tabulky 2.3 a 2.4, nám prozrazuje, že kritický vliv na rychlost algoritmu má jednoznačně počet proměnných. Pro nízký počet proměnných je EBDDRES schopen zvládat desetitisíce klauzulí, avšak pro relativně nízký počet klauzulí je schopen řešit v našem případě jen desítky proměnných.

2.4.4 Výsledky srovnání SAT solverů

V Tabulce 2.6 jsou zachyceny výsledky WalkSATu a EBDDRESu v takových konfiguracích, které pro daný benchmark dosahují nejlepších výsledků dle Tabulek 2.2 resp. 2.5. Tabulku doplňují SAT solvery pracující na principech DLL algoritmu.

DLL algoritmy jsou známy pro svou rychlost, což také potvrdily, avšak u složitých náhodně generovaných 3–SAT formulí s velkými počty proměnných a klauzulí „ztrácí

	Orig		Reverse		Dfs		Sum	
Benchmark	bez Force	s Force	bez Force	s Force	bez Force	s Force	bez Force	s Force
dubois27.cnf	-	0,03	-	0,03	-	-	-	0,02
f1000.cnf	-	-	-	-	-	-	-	-
aim-200-2_0-no-1.cnf	2,34	0,03	3,18	0,01	0,87	0,47	2,37	0,03
aim-200-6_0-yes1-1.cnf	0,09	0,12	0,06	0,12	0,08	0,11	0,06	0,09

Tabulka 2.5: Srovnání heuristik EBDDRESu

Benchmark	WalkSAT	MiniSat	RSat	zChaff	EBDDRES
ii16a1.cnf	<0,001	0,062	0,156	0,047	-
ssa6288-047.cnf	X	0,125	0,171	<0,001	-
dubois27.cnf	X	0,031	0,062	<0,001	0,02
f1000.cnf	0,416	>36000	>36000	>36000	-
f2000.cnf	2,988	>36000	>36000	>36000	-
g125.17.cnf	11,575	>36000	>36000	>36000	-
g250.29.cnf	32,041	>36000	>36000	>36000	-
hanoi5.cnf	-	31,64	3,031	98,97	-
jnh210.cnf	<0,001	0,015	0,046	<0,001	-
jnh211.cnf	X	0,031	0,046	<0,001	-
aim-200-2_0-no-1.cnf	X	0,015	0,031	<0,001	0,01
aim-200-6_0-yes1-1.cnf	0,031	0,015	0,046	<0,001	0,06

Tabulka 2.6: Srovnání jednotlivých SAT solverů. Velké X znamená, že se solver v daném benchmarku neúčastní testu. Proškrtaná pole značí, že solver nebyl schopen nalézt řešení.

dech“. U těch benchmarků, se kterými si WalkSAT poradil v řádu desítek sekund, MiniSatu ani jiným DLL solverům nestačilo 10 hodin výpočtů. Tímto mě osobně WalkSAT mile překvapil a ukázal tak, že je velmi platným nástrojem. Avšak jeho nevýhodou stále zůstává fakt, že nedokáže zjistit, je-li zadaná formule nesplnitelná. Jak již bylo řečeno, EBDDRES doplácí na své extrémní paměťové nároky a jeho použití se tak omezuje jen na malé instance.

Proměnných	Poměr	MiniSat	RSat	zChaff	WalkSAT
100	12,0	0,015	0,046	<0,001	<0,001
200	6,0	0,062	0,078	0,062	<0,001
240	5,0	0,156	0,078	1,672	0,372
250	4,8	0,5	2,125	0,375	-
255	4,7	3,859	7,390	99,20	-
267	4,5	3,109	0,328	92,84	-
316	3,8	0,703	0,796	0,188	0,016
800	1,5	0,015	0,046	<0,001	<0,001
1200	1,0	0,015	0,031	0,015	<0,001

Tabulka 2.7: Vliv počtu proměnných na výkon SAT solverů. Měřeno pro 1200 klauzulí.

Zatímco u EBDDRES solveru založeném na ROBDD je kritický pro rychlost výpočtu počet proměnných, u DLL algoritmů i u WalkSATu je touto omezující veličinou poměr

Klauzulí	Poměr	MiniSat	RSat	zChaff	WalkSAT
700	1,4	0,015	0,031	0,015	<0,001
1000	2,0	0,031	0,046	<0,001	<0,001
1500	3,0	0,031	0,046	<0,001	<0,001
2000	4,0	>300	>300	>300	<0,001
2350	4,7	>300	>300	>300	-
2500	5,0	>300	>300	>300	-
3000	6,0	32,551	>300	>300	0,015
6000	12,0	0,187	0,312	0,016	<0,001
120000	240,0	0,39	0,406	0,046	0,172

Tabulka 2.8: Vliv počtu klauzulí na výkon SAT solverů. Měřeno pro 500 proměnných.

počtu klauzulí k počtu proměnných. Jak můžeme vypozorovat z výsledků měření zachycených v Tabulkách 2.7, 2.8 a 2.9[14], nejhorších výsledků dosahují řešiče při poměrech v rozmezí od 4 do 5, v našem případě nejčastěji kolem $4,7$.

Proměnných	Klauzulí	Poměr	MiniSat	RSat	zChaff	WalkSAT
400	800	2,0	0,015	0,031	<0,001	<0,001
2000	4000	2,0	0,031	0,031	<0,001	<0,001
33000	66000	2,0	0,328	0,4376	0,219	0,063
300	900	3,0	0,031	0,046	<0,001	<0,001
2000	6000	3,0	0,062	0,093	<0,001	<0,001
22000	66000	3,0	>300	7,531	0,218	0,125
300	1200	4,0	1,375	26,921	6,578	0,016
2000	8000	4,0	>300	>300	>300	0,182
16500	66000	4,0	>300	>300	>300	-
100	470	4,7	0,031	0,046	0,016	<0,001
200	940	4,7	0,109	0,156	0,609	0,084
300	1410	4,7	12,421	149,89	38,078	0,085

Tabulka 2.9: Vliv poměru počtu klauzulí ku počtu proměnných

[14]Testování probíhalo na náhodně generovaných instancích dle algoritmu G2 [7].

Kapitola 3

Modifikace SAT algoritmů

V Kapitole 2 jsem ukázal, které základní algoritmy pro řešení problému splnitelnosti booleovských formulí se používají. Popsal jsem také heuristiky, které jednotlivým SAT solverům pomáhají, ať už tím, že značně ořezávají prohledávaný stavový prostor, nebo informovaně vybírají proměnné. Všechny SAT solvery, s kterými jsem se setkal, dokázaly najít vektor ohodnocení (*true* nebo *false*) všech proměnných, který danou booleovskou formuli splňuje, popřípadě dokázaly, že takový vektor nelze nalézt. Co když však budeme stát před problémem, k jehož řešení budeme potřebovat vědět také, které proměnné jsou v neurčeném stavu (*Don't Care*, zkráceně DC). A právě v této kapitole se zamyslím nad možnostmi modifikace SAT algoritmů, které by následně generovali vektor ohodnocení $V = (v_1, v_2, ..., v_n)$, kde $v_i \in \{false, true, DC\}$[1], a kde DC bude značit neurčený stav proměnné.

3.1 Modifikace ROBDD

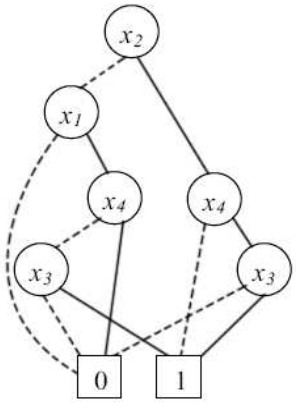
Binární rozhodovací diagramy (stále uvažujeme ROBDD, viz Sekce 2.2.3) mají obrovskou výhodu oproti ostatním algoritmům v tom, že generování neurčených stavů je jejich vlastní přirozeností. Mějme ROBDD reprezentující nějakou booleovskou funkci. Cesta vedoucí od kořene k terminálnímu uzlu představujícímu log. hodnotu *true* je řešením oné funkce, přičemž proměnné (uzly) nabývají hodnot podle hran, po kterých cesta vede. Proměnné, které na této cestě neleží, jsou tím pádem neurčeny, neboli jsou DC. Obecně může být cest vedoucích k řešení dané funkce víc. Chceme-li tedy, aby nám algoritmus označil co nejvíce DC proměnných, budeme přirozeně hledat nejkratší cestu od kořene k terminálu *true*.

Nutno si uvědomit, že booleovská funkce může mít obecně více řešení, jinými slovy

[1] V dalším textu se pro jednoduchost zápisu může objevovat ve formě $v_i \in \{0, 1, X\}$

množina proměnných ohodnocených *true* může být pokaždé jiná. Totéž platí i o množině proměnných ohodnocených *false* a totéž by také platilo o množině proměnných ohodnocených *DC*. U ROBDD (podobně jako u dalších algoritmů, viz dále) má řazení proměnných zásadní vliv na to, kolik proměnných bude označeno za DC. Ukažme si to na Příkladu 3.1.1:

Příklad 3.1.1 *Mějme funkci* $\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_3} \vee \overline{x_4})$. *Je evidentní, že k splnitelnosti formule* φ *postačuje ohodnocení* $x_2 = 1, x_4 = 0$. *Proměnné* x_1 *a* x_3 *jsou pri tomto ohodnocení nepodstatné, jsou DC. Právě tuto situaci představuje Obrázek 3.1, na němž je ROBDD funkce* φ *v řazení* (x_2, x_1, x_4, x_3). *Uvažujeme-li však řazení* (x_1, x_3, x_2, x_4) *(Obrázek 3.2), dostáváme ohodnocení např.* $x_1 = 1, x_3 = 1, x_4 = 0$ *a pouze* x_2 *je označen za DC.*

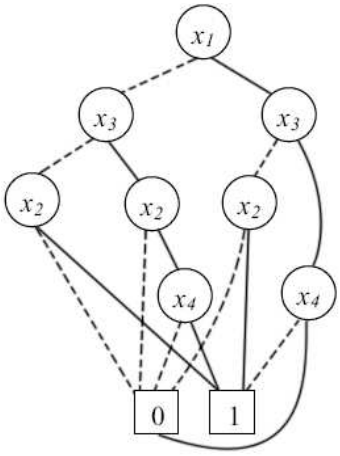


Obrázek 3.1: ROBDD reprezentace funkce φ k Příkladu 3.1.1 v řazení proměnných (x_2, x_1, x_4, x_3)

3.2 Modifikace WalkSAT algoritmu

Zatímco u algoritmů spoléhajících na ROBDD stačilo k detekci neurčených stavů podrobné zkoumání samotného ROBDD, u WalkSAT algoritmu budeme muset rozšířit množinu hodnot, kterých může proměnná nabývat, na $H = \{0, 1, X\}$. Mějme stále algoritmus tak, jak je popsán pseudokódem na Obrázku 2.6. Pro výběr proměnné, která se má překlopit, uvažujme pro jednoduchost pouze strategii **Nejlepšího výběru**. Pro každou proměnnou je veden parametr *breakcount*, který prozrazuje, kolik klauzulí se po překlopení proměnné znesplní[2]. Jelikož nám proměnné již můžou nabývat i hodnot X,

[2]Opět to nové slovo. Připomeňme, že znamená změnu stavu ze splněného do nesplněného



Obrázek 3.2: ROBDD reprezentace funkce φ k Příkladu 3.1.1 v řazení proměnných (x_1, x_3, x_2, x_4)

je třeba zavést namísto jednoho parametru *breakcount* tři nové parametry *breakcoutTo0*, *breakcoutTo1* a *breakcoutToX*. Udávají, kolik klauzulí se znesplní, pokud se proměnná překlopí na hodnotu *false*, resp. *true*, resp. *DC*.

V závislosti na aktuální hodnotě proměnné se porovnají oba zbývající breakcounty (např. pokud proměnná $x = 0$, porovnají se *breakcountTo1* a *breakcontToX*) a proměnné se přiřadí hodnota podle menšího z nich. Stále lze tedy říct, že se vybere taková proměnná, která minimalizuje počet klauzulí, které by se po překlopení znesplnily. Překlopením se však v tomto případě myslí změna hodnoty z 0 do X nebo z 0 do 1 nebo z X do 0 apod.

Zatím algoritmus stále vycházel z náhodného ohodnocení, i když se v něm mohli objevit hodnoty *DC*. To může vést k tomu, že nám algoritmus ohodnotí všechny proměnné pouze hodnotami *true/false*, i když je zřejmě, že v dané formuli jsou některé proměnné DC. Další modifikací tedy je, aby algoritmus v první iteraci svého hlavního cyklu vycházel z ohodnocení, ve kterém mají všechny proměnné přiřazenu hodnotu X.

Příklad 3.2.1 *Stejně jako v Příkladu 3.1.1 mějme funkci* $\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_3} \vee \overline{x_4})$*. Algoritmus začne se všemi proměnnými ohodnocenými* X*. Náhodně se vybere jedna nesplněná klauzule, např. klauzule* $(\overline{x_1} \vee x_2 \vee x_3)$*. Heuristika nejlepšího výběru nám radí vybrat proměnnou* x_2 *a překlopit ji na hodnotu* 1*. V druhém kroku už máme jen jedinou nesplněnou klauzuli, a sice klauzuli* $(\overline{x_3} \vee \overline{x_4})$*. Heuristika nám dává na výběr buď* x_3 *nebo* x_4*, náhodně se tedy vybere např.* x_4*. Ve třetím kroku je již celá formule splněna.*

1.					2.					3.
2	$\overset{0}{\leftarrow}$	$x_1 = X$	$\overset{1}{\rightarrow}$	1		$x_1 = X$				$x_1 = X$
3	$\overset{0}{\leftarrow}$	$x_2 = X$	$\overset{1}{\rightarrow}$	0		$x_2 = 1$				$x_2 = 1$
2	$\overset{0}{\leftarrow}$	$x_3 = X$	$\overset{1}{\rightarrow}$	2	0	$\overset{0}{\leftarrow}$	$x_3 = X$	$\overset{1}{\rightarrow}$	1	$x_3 = X$
		$x_4 = X$			0	$\overset{0}{\leftarrow}$	$x_4 = X$	$\overset{1}{\rightarrow}$	1	$x_4 = 0$

3.3 Modifikace DLL algoritmu

Jak jsem popsal v Sekcích 2.2.2 a 2.3.2, DLL algoritmus ohodnocuje proměnné a následně implikuje na základě pravidla jednotkového literálu další proměnné, dokud není formule splněná nebo nenarazí na konflikt. V takovém případě konflikt analyzuje a za pomocí backtrackingu upravuje ohodnocení jednotlivých proměnných. Pro následující modifikaci DLL algoritmu uvažujme algoritmus s nechronologickým backtrackingem a konflikty řízeným učením klauzulí se schématem 1-UIP.

Stejně jako u modifikace WalkSATu, ani u DLL algoritmu si nevystačíme s klasickým ohodnocením *true/false*, ale budeme muset k nim přidat také hodnotu *DC*. Celá modifikace je založena na premise, která říká, že proměnná na úrovni určení δ, na které vznikl konflikt, nemůže být *DC* pod aktuálním ohodnocením proměnných z úrovně $\alpha < \delta$.

Kdykoli nastává větvení (funkce `decide_next_branch()` pseudokódu z Obrázku 2.3) volná proměnná se ohodnotí *DC*. Následuje propagace přes všechny nesplněné klauzule (funkce `deduce()`). Dojde-li při propagaci ke konfliktu, funkce `analyze_coflict()` nám vytvoří konfliktní klauzuli. Složení konfliktní klauzule pak rozhoduje o tom, zda se naučí (přidá se do databáze klauzulí) a o tom, do které úrovně se provede backtracking. Existuje-li v konfliktní klauzuli literál v neurčeném stavu, klauzule se nenaučí. Nemusí totiž vést k správnému ořezání stavového prostoru a tím pádem nám může odříznout i část, v níž se nalézá řešení formule. Konfliktní klauzuli můžeme stále použít k implikaci UIP, ze kterého se během backtrackingu stane volná proměnná. To, na které úrovni určení s ní bude dále nakládáno, stanoví její předchozí stav. Mějme úroveň backtrackingu *bl*. Pokud byl UIP před zahájením backtrackingu v neurčeném stavu (*DC*), je určen na stejné úrovni *bl*. Pokud byl v určeném stavu (*true/false*), pokračuje na úrovni $(bl - 1)$.

Řazení proměnných má opět zásadní vliv na množství odhalených DC stavů. Opět mějme formuli φ z Příkladu 3.1.1. Uvážíme-li seřazení (x_1, x_2, x_3, x_4), námi modifikovaný DLL algoritmus ohodnotí proměnné $x_1 = X, x_2 = 1, x_3 = X$ a $x_4 = 0$. Kdežto při seřazení (x_2, x_1, x_3, x_4) se dobereme výsledku $x_1 = 1, x_2 = X, x_3 = 1, x_4 = 0$.

3.4 DCSAT

Po dlouhém zvažování mezi modifikací algoritmu WalkSATu a DLL algoritmu jsem se nakonec rozhodl pro DLL, konktrétně pro úpravu open-source SAT solveru MiniSat. MiniSat jsem zvolil zejména proto, že je k němu snadno dostupná a velmi podrobná dokumentace, což mi usnadnilo pochopení zdrojového kódu. Nově modifikovaný MiniSat jsem pojmenoval DCSAT. Úpravy sledují myšlenky popsané v Sekci 3.3 a nyní tyto úpravy popíšu podrobněji:

Třída **Lit** byla upravena tak, aby odlišovala literály, které nabývají *true*/*false* ohodnocení (takové literály jsou reprezentovány kladnými celými čísli), a které nabydou ohodnocení *DC* (reprezentovány zápornými celými čísli). Nechybí ani pomocné funkce pro převod mezi oběma typy literálů.

Funkce `assume(Lit p)` nyní rozlišuje, zda chceme literál `p` ohodnotit *DC* či určitým ohodnocením. S tím souvisí přidáním nové funkce `enqueueDC(Lit p, GClause from)`, která tyto změny umožňuje.

Je-li literál ohodnocen *DC*, ve funkci `propagate()` se prohledávají všechny klauzule, které obsahují daný literál v kladné i záporné fázi.

Analýza konfliktu prováděná ve funci `analyze(...)` nově vrací logickou hodnotu podle struktury konfliktní klauzule. Nachází-li se v konfliktní klauzuli literál ohodnocený *DC* vrací *true*, jinak vrací *false*. Na základě toho se provede/neprovede učení konfliktní klauzule. Dále funkce `analyze(...)` nastavuje hodnotu přepínače `assumeUIP` podle toho, zda je UIP v konflikní klauzuli v určeném či neurčeném stavu.

Jestliže se nemá konfliktní klauzule naučit, provede se nová procedura `learnDC(const vec<Lit>& ps)`, kde parametr `ps` obsahuje konfliktní klauzuli. V závislosti na struktuře konfliktní klauzule a přepínači `assumeUIP` se UIP literál ohodnotí na stávající úrovni určení nebo na úrovni o jedna vyšší[3].

DCSAT se spouští se dvěma parametry (druhý parametr je nepovinný):

```
dcsat.exe vstupni_sobor [vystupni_soubor]
```

Vstupem je soubor, který obsahuje CNF formuli v DIMACS formátu [2]. Výstupní soubor obsahuje rozhodnutí o splnitelnosti formule SAT/UNSAT a vektor ohodnocení proměnných ($c_1V_1 \quad c_2V_2 \quad ... \quad c_nV_n \quad 0$), kde V_i je proměnná a c_i je znak určující hodnotu proměnné: `'-'` pro *false*, `''` (prázdný řetězec) pro *true* a `'x'` pro *DC*. Například výstup:

[3] learnDC() se provádí po backtrackingu.

```
SAT
-1 2 x3 0
```

říká, že formule je splnitelná při ohodnocení $x_1 = false$, $x_2 = true$, $x_3 = DC$.

3.4.1 Testování DCSATu

Testování DCSATu probíhalo na stejném počítači se stejnou konfigurací, která je popsána v Sekci 2.4, a byly k tomu opět využity DIMACS benchmarky a instance generované algoritmem G2 (viz [7]). Údaje doby výpočtu DCSATu jsou opět v sekundách a dále je uvedeno též procentuální vyjádření počtu proměnných, které DCSAT vyhodnotil jako don't care.

DCSAT je upravený MiniSat, takže je nasnadě srovnání se svým „vzorem". Z Tabulek 3.1, 3.3 – 3.7 je na první pohled patrno, že modifikace algoritmus zpomalily, což ovšem není žádným překvapením. To jsem také očekával.

Benchmark	MiniSat	DCSAT	%DC
ii16a1.cnf	0,062	1,686	12,9
ssa6288-047.cnf	0,125	0,999	UNSAT
dubois27.cnf	0,031	>1000	-
hanoi5.cnf	31,64	>1000	-
jnh210.cnf	0,015	0,062	3
jnh211.cnf	0,031	0,046	UNSAT
aim-200-2_0-no-1.cnf	0,015	0,046	UNSAT
aim-200-6_0-yes1-1.cnf	0,015	0,421	0

Tabulka 3.1: Srovnání MiniSatu a DCSATu na DIMACS benchmarcích

Co se týče vlivu počtu proměnných, klauzulí a jijich poměru, tam se situace příliš nemění. Stále zůstává pravdou, že nejhorších výsledků DLL algoritmus dosahuje při poměrech počtu klauzulí k počtu proměnných kolem $4,7$. Vzhledem k výsledkům zaznamenaných v Tabulkách 3.3 – 3.7 se zdá, že se u DCSATu tato oblast poněkud rozšířila přibližně od $3,8$ po 6.

Na internetu jsem nenašel žádný open-source SAT solver, který by zvládal generovat ohodnocení včetně neurčených stavů, tudíž nelze výkon DCSATu porovnávat s ostatními. Jako jedinou referenci lze tedy použít utilitu p. Tomana **BoolTool** [24], která mimo jiné také zvládá test splnitelnosti i označení proměnných v neurčeném stavu. Rychlost BoolTool je oproti DCSATu nepoměrně pomalejší, zato vyvážena výstupem v podobě všech

možných řešení formule. Pro srovnání jsem volil instance maximálně s desítkami proměnných, protože pro vyšší počty není BoolTool schopen poskytnout výsledek v rozumné době (Tabulka 3.2).

Prom.	Klauzulí	Poměr	DCSAT	BoolTool
10	80	8,0	0,015	0,109
20	140	7,0	0,015	14,547
20	80	4,0	0,015	22,484
30	80	2,7	0,015	2117,7
30	30	1,0	<0,001	1619,8

Tabulka 3.2: Porovnání DCSATu s BoolToolem

Prom.	Poměr	MiniSat	DCSAT	%DC
10	8,0	0,015	0,015	0
20	4,0	0,015	0,015	5
30	2,7	0,015	<0,001	20
50	1,6	0,015	<0,001	30
60	1,3	0,015	<0,001	30
80	1,0	0,015	<0,001	45

Tabulka 3.3: Vliv počtu proměnných na výkon DCSATu. Měřeno pro 80 klauzulí.

Prom.	Poměr	MiniSat	DCSAT	%DC
100	12,0	0,015	0,093	0
200	6,0	0,062	21,187	0
255	4,7	3,859	550,65	0
316	3,8	0,703	>300	-
400	3,0	0,031	0,125	13,25
500	2,4	0,015	0,078	20,4
800	1,5	0,015	0,093	33,25
1200	1,0	0,015	0,078	48,1

Tabulka 3.4: Vliv počtu proměnných na výkon DCSATu. Měřeno pro 1200 klauzulí.

Klauzulí	Poměr	MiniSAT	DCSAT	%DC
40	1,0	0,015	<0,001	57,5
60	1,5	0,015	0,015	30
80	2,0	0,015	0,015	30
100	2,5	0,031	0,015	27,5
200	5,0	0,015	0,062	5
2000	50,0	0,015	0,078	0
20000	500,0	0,062	0,828	0

Tabulka 3.5: Vliv počtu klauzulí na výkon DCSATu. Měřeno pro 40 proměnných.

Klauzulí	Poměr	MiniSat	DCSAT	%DC
500	1,0	0,015	0,031	49,8
700	1,4	0,015	0,046	39,6
1000	2,0	0,031	0,062	27
2350	4,7	>300	>300	-
3000	6,0	32,531	>300	-
6000	12,0	0,187	76,13	0

Tabulka 3.6: Vliv počtu klauzulí na výkon DCSATu. Měřeno pro 500 proměnných.

Proměnných	Klauzulí	Poměr	MiniSat	DCSAT	%DC
100	200	2,0	0,015	0,015	25
400	800	2,0	0,031	0,062	24,75
2000	4000	2,0	0,046	0,156	25,5
33000	66000	2,0	0,328	2,531	24,75
100	300	3,0	0,015	0,046	12
300	900	3,0	0,031	10,28	8
2000	6000	3,0	0,062	>300	-
100	400	4,0	0,015	2,468	9
300	1200	4,0	1,375	>300	-
2000	8000	4,0	>300	>300	-
100	470	4,7	0,031	0,203	5
200	940	4,7	0,109	51,20	0
300	1410	4,7	12,421	>300	-
100	800	8,0	0,015	0,140	0
300	2400	8,0	0,109	54,28	0
900	7200	8,0	>300	>300	-

Tabulka 3.7: Vliv poměru počtu klauzulí k počtu proměnných na výkon DCSATu

Kapitola 4

Závěr

Prostudoval jsem současné dostupné open-source SAT solvery, jejich algoritmy a heuristiky. Přesvědčil jsem se, že k řešení SAT problému vede několik přístupů – algoritmů. Vzhledem k informacím v literatuře, jež jsem svými testováními vybraných SAT solverů jen potvrdil, se jako nejefektivnější algoritmy jeví Davis Logemann Loveland algoritmus a stochastický algoritmus WalkSATu. Také jsem ověřil, že využitelnost algoritmu založeného na ROBDD je omezena na booleovské formule s nejvýše stovkami proměnných. Bohužel Stålmarckův algoritmus je patentovaný a neexistuje žádný open-source SAT solver, který by tento algoritmus implementoval, takže jsem ho kromě základní charakteristiky zcela ignoroval.

Dále se mi podařilo modifikovat DLL algoritmus takovým způsobem, aby zvládal generovat i neurčené stavy proměnných. Naimplementovanou verzi takto modifikovaného algoritmu jsem pojmenoval DCSAT. Z výsledků testování vyplývá, že DCSAT značně „přibral“ na složitosti zejména u splnitelných formulí s právě jedním možným řešením. Možnost znalosti neurčených stavů si tedy vyžádala daň v podobě prodloužení výpočetní doby, avšak dále lze na tomto poli bádat a zkoušet různé heuristiky, jejichž implementace nebyly zrealizovány jednak z časových důvodů a jednak proto, že to nebylo v zadání bakalářské práce.

Závěrem mi dovolte rozloučit se slovy klasika Járy da Cimrmana, která jsem si ovšem dovolil poupravit do současných poměrů:

> *„Kdokoli se rozhodneš tento můj algoritmický stück po mé smrti upraviti, nikterak mne nešanuj. Nebudu se zlobiti, změníš-li ku své potřebě některé funkce, některou datovou strukturu, či celý algoritmus. Mně o myšlénku se jedná. Ta aby zůstala.“*

Literatura

[1] Dimacs benchmarks.
`ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf`.

[2] Dimacs cnf formát.
`ftp://dimacs.rutgers.edu/pub/challenge/sat/doc`.

[3] Dp algorithm pseudocode.
`ira.informatik.uni-freiburg.de/ ~herbstri/publications/HB_2003/img5.html`.

[4] The international sat competitions. `http://www.satcompetition.org/`.

[5] Links for the satisfiability problem. `http://www.satlive.org/`.

[6] Rsat homepage. `http://reasoning.cs.ucla.edu/rsat/index.html`.

[7] Sat instance generation page.
`http://www.is.titech.ac.jp/~watanabe/gensat/`.

[8] F. Aloul, I. Markov, and K. Sakallah. Force: A fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI 2003*, pages 116–119, 2003.

[9] Jiří Balcárek. Řešení problému splnitelnosti booleovské formule (sat) pomocí binárních rozhodovacích diagramů (bdd), 2007. Bakalářská práce, FEL ČVUT v Praze.

[10] Armin Biere. The evolution from limmat to nanosat, 2004. Technical Report No. 444, Dept. Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

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

[14] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[15] Niklas Eén and Niklas Sörensson. An extensible sat-solver, 2003.

[16] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 661–666, Orlando, Florida, 1999.

[17] Jinbo Huang and Adnan Darwiche. Toward good elimination orders for symbolic sat solving. In *ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 566–573, Washington, DC, USA, 2004. IEEE Computer Society.

[18] João P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.

[19] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence, Rhode Island, 1997.

[20] 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'01)*, 2001.

[21] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability : Review and innovations, 2002. Master Thesis, The Hebrew University.

[22] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 521–532, 1996.

[23] Mary Sheeran and Gunnar Stålmark. A tutorial on stålmark's proof procedure for propositional logic. *Formal Methods in System Design*, 16:23–58, 2000.

[24] David Toman. Nástroj pro manipulaci s logickými funkcemi popsanými algebraickým výrazem, 2007. Bakalářská práce, FEL ČVUT v Praze.

Příloha A

Manuál DCSATu

A.1 Nové funkce

DCSAT je, jak bylo již jednou zmíněno, modifikovanou verzí MiniSatu. Myšlenky modifikace jsou popsány v Sekci 3.4. Pro popis funkcí původního MiniSatu odkazuji na [15] přiložený též na CD v adresáři *SAT solvers→MiniSat*. V této Příloze se soustřeďme pouze na nově vytvořené funkce.

`class Lit`

- `LitDC(Var var)` – vytváří z proměnné *var* její DC literál reprezentovaný záporným celým číslem.
- `DCtoNeg(Lit p)` – převádí literál p ze své DC representace (záporné číslo) na reprezentaci literálu v záporné fázi ($\overline{p}$) (kladné liché číslo).
- `DCtoPos(Lit p)` – převádí literál p ze své DC representace (záporné číslo) na reprezentaci literálu v kladné fázi (p) (kladné sudé číslo).

`class Solver`

- `enqueueDC(Lit p, GClause from)` – jestliže je literál p ve své DC reprezentaci, jemu odpovídající proměnná se ohodnotí *DC*, uloží se úroveň určení a důvod tohoto ohodnocení (parametr *from*), literál p se pak uloží na zásobník. Jestliže literál p je v reprezentaci literálu v kladné/záporné fázi, pak se postupuje stejně jako v původní funkci `enqueue(Lit p, GClause from)`.
- `learnDC(const vec<Lit>& ps)` – na základě hodnoty přepínače `assumeUIP`, který se nastavuje při analýze konfliku, se UIP obsažené v konfliktní klauzuli *ps* ohodnotí a uloží se na zásobník buď na stávající úrovni

určení nebo na úrovni o jedna vyšší. Důvod ohodnocení (konfliktní klauzule bez UIP) se také uloží.

A.2 Instalace a užívání

Instalace je velmi jednoduchá. Stačí rozbalit do vybrané složky .rar archív, který obsahuje spouštěcí sobor `dcsat.exe` společně s knihovnami potřebnými k jeho chodu.

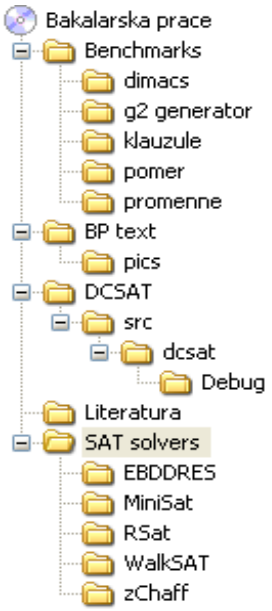
Program se spouští z příkazové řádky příkazem `dcsat.exe` s dvěma parametry. První parametr je vstupní soubor obsahující zkoumanou formuli v CNF DIMACS formátu, druhý, nepovinný, parametr je výstupní soubor, do kterého se uloží výsledek. Příklad takového volání může tedy vypadat následovně:

```
dcsat.exe formule.cnf vystup.txt
```

Projekt DCSATu byl vytvořen v MS Visual Studio 2005 v jazyku C++ pro systémem Windows XP a je k dispozici na přiloženém CD. Verze pro Linux není k dispozici.

Příloha B

Obsah CD



Obrázek B.1: Obsah přiloženého CD

Benchmarks Zde nalezneme testovací data použitá v této BP. Jednak DIMACS benchmarky a jednak náhodně generované instance vygenerované algoritmem G2, jehož zdrojový kód a zkompilovaný .exe soubor jsou taktéž přiloženy. Náhodně generované instance jsou rozděleny do podadresářů podle toho, zda byly použity při testování vlivu počtu proměnných, klauzulí nebo vlivu poměru.

BP text obsahuje zdrovové kódy této bakalářské práce ve formátu .tex včetně přeložených výstupů v .pdf a .ps formátech.

DCSAT Tento adresář obsahuje archív se zkompilovaným programem dcsat.exe a knihovnami potřebnými k jeho spuštění, a také celý projekt DCSATu pro MS Visual

Studio. Více viz readme.txt v podadresáři *src→dcsat.*

Literatura Vetšina použité literatury, ze které jsem čerpal, se nachází právě v tomto adresáři.

SAT solvers Ostatní SAT solvery testované v této práci jsou zde k dispozici v podobě, ve které jsem je stáhnul z internetu. Přiložena je i dostupná dokumentace.