--- zde bude zadání práce (originál nebo kopie) ---

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Bakalářská práce :

Řešení problému splnitelnosti booleovské formule (SAT) pomocí binárních rozhodovacích diagramů (BDD)

Vypracoval: Jiří Balcárek

Vedoucí práce: Ing. Petr Fišer

Červenec 2007

Poděkování

Mé poděkování patří Ing. Petru Fišerovi za všechen čas, který mi věnoval, za jeho trpělivost a za připomínky a návrhy, které vedly ke zkvalitnění této práce.

Čestné prohlášení

Prohlašuji, že jsem zadanou bakalářskou práci zpracoval sám s přispěním vedoucího práce a používal jsem pouze literaturu uvedenou v přiloženém seznamu.

Dále prohlašuji, že nemám námitek proti užití tohoto školního díla ve smyslu §60 Zákona č.121/200 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 17.7.2007 ....

podpis

Anotace

Tato bakalářská práce se zabývá implementací nástroje k řešení konstruktivního problému splnitelnosti booleovské formule (SAT), který využívá binární rozhodovací diagramy (BDD). Pro implementaci je použit balík CUDD (Colorado University Decision Diagram package), který obsahuje sadu funkcí pro práci s binárními rozhodovacími diagramy. Práce obsahuje obecný úvod do problematiky BDD diagramů, dále je zde popsán balík CUDD a heuristiky uspořádání uzlů (reorderingu). V další části je popsána implementace mého řešiče splnitelnosti booleovské funkce (SAT solveru) a sat solveru EBDDRES, se kterým je v další části provedeno srovnání a zhodnocení kvality a efektivity řešení.

Obsah

Poděkování I
Čestné prohlášení IV
Anotace VI
Obsah VIII
Seznam obrázků IX
Seznam tabulek X

1. **Úvod** 1
    - **1.1. Výroková logika** 2
    - **1.2. Normální formy** 3
        - **1.2.1. Konjunktivní normální forma – CNF** 3
        - **1.2.2. Disjunktivní normální forma – DNF** 4
        - **1.2.3. Normální forma If-Then-Else** 4
    - **1.3. Shannonův expanzivní teorém** 5
2. **Binární rozhodovací diagramy** 5
    - **2.1. Seznámení s binárními rozhodovacími diagramy (BDD)** 5
    - **2.2. Konstrukce binárních rozhodovacích diagramů (BDD)** 7
        - **2.2.1. Uspořádání proměnných BDD (OBDD)** 7
        - **2.2.2. Redukce počtu proměnných v BDD (RBDD)** 8
        - **2.2.3. Manipulace s ROBDD** 12
            - **2.2.3.1. Operace APPLY** 13
3. **Vlastní implementace** 13
    - **3.1. Popis vstupu** 14
    - **3.2. Popis implementovaných heuristik** 15
        - **3.2.1. Heuristika rovnání termů vstupní funkce** 16
        - **3.2.2. Heuristiky dynamického reorderingu** 20
    - **3.3. Výstupy implementovaného SAT solveru** 22
4. **SAT solver EBDDRES** 25
    - **4.1. Popis vstupu** 25
    - **4.2. Použité heuristiky** 25
5. **Porovnání implementovaného SAT solveru a EBDDRES solveru** 28
6. **Závěr a zhodnocení implementovaného SAT solveru** 31
7. **Seznam použité literatury** 33
8. **Seznam zkratek** 34
9. **Přílohy** 35
    - **A. Seznámení s knihovnou CUDD** 35
        - **A.1 Datové struktury a práce s nimi** 35
            - **A.1.1 Struktura uzlu** 36
            - **A.1.2 Správa paměti** 36
        - **A.2 Příklad vytvoření BDD** 37
        - **A.3 Metody reorderingu** 39
    - **B. Ovládání a výstup SAT solveru EBDDRES** 42
        - **B.1 Ovládání sat solveru** 42
        - **B.2 Výstup SAT solveru** 42
    - **C. Popis funkcí a práce s implementovaným SAT solverem** 44
        - **C.1 Lexikální a syntaktická analýza** 44
        - **C.2 Popis mnou implementovaných funkcí** 46
        - **C.3 Popis použitých funkcí z balíku CUDD** 49
        - **C.4 Práce s implementovaným SAT solverem** 53
    - **D. Obsah CD** 54

Seznam obrázků

Obrázek č.1 : *Příklad binárního rozhodovacího diagramu* **6**
Obrázek č.2 : *Ukázka vlivu uspořádání proměnných na velikost OBDD (Bryant 1986)* **8**
Obrázek č.3 : *Odstranění redundantních terminálů* **9**
Obrázek č.4 : *Odstranění opakujících se vnitřních uzlů* **9**
Obrázek č.5 : *Odstranění nadbytečného uzlu* **10**
Obrázek č.6 : *BDD diagram obsahující redundance* **10**
Obrázek č.7 : *BDD po odstranění redundantních terminálů* **11**
Obrázek č.8 : *BDD po odstranění redundantních vnitřních uzlů* **11**
Obrázek č.9 : *BDD po odstranění nadbytečných uzlů* **12**
Obrázek č.10 : *Graf závislosti velikosti ROBDD(počtu uzlů) na pořadí vstupních termů* . . **18**
Obrázek č.11 : *Průměrný počet uzlů na term pro jednotlivé heuristiky rovnání proměnných* **20**
Obrázek č.12 : *Doba vytvoření ROBDD pro jednotlivé heuristiky rovnání proměnných* . . . **21**
Obrázek č.13 : *Stromová struktura přiloženého CD* **54**

Seznam tabulek

Tabulka č.1 *: Pravdivostní tabulka základních logických spojek* **2**
Tabulka č.2: *Pravdivostní tabulka logické funkce* **6**
Tabulka č.3 : *Obsah polí na počátku heuristiky řazení termů* **17**
Tabulka č.4 : *Obsah polí po načtení pole termů(+četnost výskytů)* **17**
Tabulka č.5 : *Obsah polí po vytvoření pole vah* **17**
Tabulka č.6 : *Obsah polí po seřazení podle vah termů* **18**
Tabulka č.7 : *Výpis pole termů do souboru temp* **18**
Tabulka č.8: *Heuristika rovnání vstupních termů pro nesplnitelné booleovské funkce* . . . **19**
Tabulka č.9: *Heuristika rovnání vstupních termů pro splnitelné booleovské funkce* **19**
Tabulka č.10 : *Srovnání heuristik rovnání proměnných* **22**
Tabulka č.11 : *Srovnání SAT solveru EBDDRES a implementovaného SAT solveru* **29**

1. Úvod

Tématem mojí práce je implementace nástroje pro řešení problému splnitelnosti booleovské funkce. K řešení tohoto problému mám využít knihovnu CUDD, která poskytuje mnoho funkcí pro efektivní práci s binárními rozhodovacími diagramy (BDD), algebraickými rozhodovacími diagramy (ADD) a binárními rozhodovacími diagramy s potlačenou nulou(ZDD–Zero suppressed decision diagrams). Tato knihovna byla implementována skupinou pracovníků kolem Fabia Sommenzi z Coloradské Univerzity a je implementována v prostředí programovacího jazyka C. Původní implementace této knihovny byla určena pro platformu Linux, nicméně pro implementaci SAT solveru jsme po domluvě s Ing. Fišerem zvolili modifikaci knihovny CUDD, kterou v rámci bakalářské práce zpracoval Ondřej Kološ[6], a která je kompilovatelná pod operačním systémem windows.

Dále bylo provedeno srovnání s SAT solverem, který k řešení používá binární rozhodovací diagramy. Ke srovnání jsem zvolil SAT solver EBDDRES [16], který byl vytvořen na universitě Johannese Keplera (Johannes Kepler University) v institutu pro formální modely a ověřování (FMV - Institute for Formal Models and Verification) dvojicí Armin Biere a Toni Jessila.

Jak již bylo dříve zmíněno, je k řešení splnitelnosti booleovské funkce použito binárních rozhodovacích diagramů. Řešiče používající BDD k řešení splnitelnosti booleovské funkce jsou zpravidla pomalejší než řešiče využívající jiných heuristik, ale výhodou je, že ve výsledku získáme nejen odpověď zdali je logická funkce splnitelná či nikoli, ale pomocí backtrackingu můžeme například také vypsat všechny úplně i neúplně určené vektory proměnných, pro které je funkce splněná .

Než začnu popisovat jednotlivé heuristiky a způsob implementace, rád bych nejprve provedl krátký úvod do problematiky týkající se binárních rozhodovacích diagramů a logických funkcí.

1.1. Výroková logika

Výroková logika[6,12] se zabývá studiem výroků, vytvářením výroků a zkoumáním jejich pravdivosti (nepravdivosti). Základním kamenem výrokové logiky jsou výroky. Za výrok považujeme každé tvrzení(větu), u které můžeme rozhodovat jestli je pravdivá či nikoli.

Výroková logika pracuje se základními výroky a zjišťuje pravdivost výroků složených, které jsou vytvářeny z výroků jednoduchých na základě pravdivosti za použití tzv. logických spojek. Těmito základními spojkami jsou negace, konjunkce, disjunkce, implikace a ekvivalence.

Tyto spojky mají následující význam :

1. negace – “není pravda, že“ bývá označována symbolem ¬,
2. konjunkce – “a zároveň“ bývá označována symbolem ^,
3. disjunkce – “nebo“ bývá označována symbolem ∨,
4. implikace – “jestliže … , potom …“ bývá označována symbolem ⇒,
5. ekvivalence – “ … právě tehdy, když …“ a bývá označována symbolem ⇔,

a můžeme je vyjádřit pomocí následující pravdivostní tabulky :

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \Rightarrow \beta$	$\alpha \Leftrightarrow \beta$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

Tabulka č.1 *: Pravdivostní tabulka základních logických spojek*

Formule výrokové logiky je řetězec logických proměnných, logických spojek a závorek, pro který platí, že každá logická proměnná je formule a jsou-li α, ß formule, je (¬α), (α∧ ß), (α∨ ß), (α ⇒ ß), (α ⇔ß) také formule.

Pokud ve formuli α známe pravdivostní hodnoty všech logických proměnných, je pravdivostní hodnota formule α určena jednoznačně. Formule s n logickými proměnnými má 2^n různých pravdivostních ohodnocení.

- Tautologie je formule pravdivá ve všech pravdivostních ohodnoceních (např. $\alpha \Leftrightarrow \alpha$).

- Kontradikce je formule nepravdivá ve všech pravdivostních ohodnoceních (např. $\alpha \wedge \neg\alpha$).

- Splnitelná formule je taková formule, která je pravdivá alespoň v jednom pravdivostním ohodnocení (např. $\alpha \wedge ß$).

1.2. Normální formy

Normální forma (NF) zápisu logické funkce určuje jednotný formát zápisu logických funkcí. Dříve než se podíváme na jednotlivé druhy normálních forem, je důležité abychom se seznámili se základní terminologií, kterou budeme potřebovat při popisu normálních forem.

Literál – je logická proměnná, nebo negace logické proměnné.

Minterm – je logický výraz skládající se pouze s konjunkcí literálů např. $\neg\alpha, \alpha\wedge\beta$… .

Maxterm (klausule) – je logický výraz, který obsahuje literál nebo disjunkci literálů

Mezi normálními formami logických funkcí můžeme provádět převody tzn. Např. převod mezi DNF a CNF můžeme provést pomocí De Morganova pravidla.

1.2.1. Konjunktivní normální forma – CNF

Výroková formule je v konjunktivní normální formě, resp. v klauzulární formě, je-li konjunkcí podformulí, z nichž každá je disjunkcí konečně mnoha literálů výrokových proměnných. Úplná konjunktivní normální forma formule ***A*** je její konjunktivní normální forma, v níž každý konjunkt obsahuje literály všech výrokových proměnných vyskytujících se ve formuli ***A***, přičemž se v žádném konjunktu nevyskytuje současně pozitivní a negativní literál téže výrokové proměnné.

Tuto definici lze zapsat jako obecný výraz takto :

$$(\alpha_{11} \vee \alpha_{12} \vee \ \dots \vee \alpha_{1m}) \ \wedge (\alpha_{21} \vee \alpha_{22} \vee \dots \vee \alpha_{2m}) \ \wedge \dots \wedge (\alpha_{n1} \vee \alpha_{n2} \vee \ \dots \vee \alpha_{nm})$$

kde každé α_{nm} je logická proměnná nebo negace logické proměnné.

Př. $(a \vee b \vee \neg c) \ \wedge (\neg a \vee b \vee c) \ \wedge (\neg a \vee b \vee \neg c)$

1.2.2. Disjunktivní normální forma – DNF

Výroková formule je v disjunktivní normální formě, je-li disjunkcí podformulí , z nichž každá je konjunkcí konečně mnoha literálů výrokových proměnných.

Úplná disjunktivní normální forma formule A je její disjunktivní normální forma, v níž každý disjunkt obsahuje literály všech výrokových proměnných vyskytujících se ve formuli A, přičemž se v žádném disjunktu nevyskytuje současně pozitivní a negativní literál téže výrokové proměnné.

Tuto definici lze zapsat jako obecný výraz takto :

$$(\alpha_{11} \wedge \alpha_{12} \wedge \dots \wedge \alpha_{1m}) \vee (\alpha_{21} \wedge \alpha_{22} \wedge \dots \wedge \alpha_{2m}) \vee \dots \vee (\alpha_{n1} \wedge \alpha_{n2} \wedge \dots \wedge \alpha_{nm})$$

kde každé α_{nm} je logická proměnná nebo negace logické proměnné.

Př. $(a \wedge b \wedge \neg c) \vee (\neg a \wedge b \wedge c) \vee (\neg a \wedge b \wedge \neg c)$

1.2.3.Normální forma If – Then - Else

Logická funkce je v If-Then-Else normálové formě (ITE/INF), jestliže provedeme Shannonův expanzní teorém (viz. kapitola 1.3) přes všechny proměnné ve výrazu. Takto můžeme vyjádřit jakýkoliv booleovský výraz.

Například převedeme pomocí Shannonova expanzního teorému booleovský výraz

$t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ do if-then-else normální formy:

$$\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{00} \\
t_{11} &= x_2 \rightarrow t_{111}, t_{110} \\
t_{000} &= y_2 \rightarrow 0,1 \\
t_{001} &= y_2 \rightarrow 1,0 \\
t_{110} &= y_2 \rightarrow 0,1 \\
t_{111} &= y_2 \rightarrow 1,0
\end{aligned}$$

Kde $x \rightarrow y_0, y_1$ je definováno jako: $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (x \wedge y_1)$.

Na příkladu je vidět, že některé výrazy jsou identické, například t_{001} a t_{111}, a daly by se sloučit do jednoho. Těchto úprav se používá k vytváření binárních rozhodovacích diagramů a budu se jim zabývat v kapitole 2.

1.3. Shannonův expanzivní teorém

Shannonův expanzivní teorém[15] je metoda, kterou můžeme booleovskou funkci rozložit na součet dvou "podfunkcí" původní funkce.

Obecný rozklad funkce může vypadat například takto:

$$f(x_1,x_2,\ldots,x_n)=x_1 f_{x1}(1,x_2,\ldots,x_n)+ \neg x_1 f_{x1}(0,x_2, \ldots, x_n)$$

kde f je jakákoliv booleovská funkce a f_{x1} je funkce f rozložená podle x_1. V první části rozloženého výrazu jsou ve funkci f nahrazeny všechny výskyty proměnné x_1 logickou jedničkou, zatímco v druhé části rozloženého výrazu jsou nahrazeny logickou nulou.

Př. Chceme rozložit funkci F = AB + BC + AC

$$\begin{aligned} F &= AB(C+C') + BC(A+A') + AC(B+B') \\ &= ABC + ABC' + ABC + A'BC + ABC + AB'C \\ &= ABC + A'BC + AB'C + ABC' \\ &= \underline{A (BC+BC') + A'(BC+B'C)} \end{aligned}$$

Využívá se zde jednoduchého faktu, že X + X' = 1.

2. Binární rozhodovací diagramy

Tato kapitola se zabývá seznámením s binárními rozhodovacími diagramy. Popisuje jejich tvorbu a operace, které se nad nimi provádějí za účelem zvýšení efektivity práce s diagramy a zjednodušení jejich reprezentace. Výchozími dokumenty jsou [1,2,3,6].

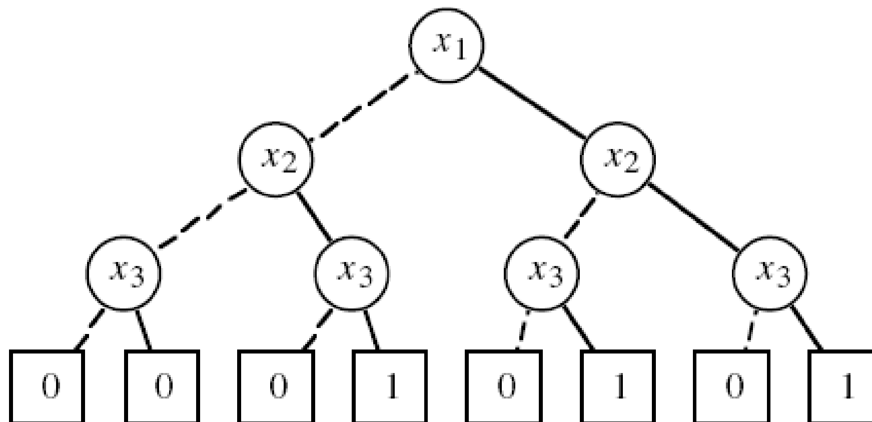
2.1. Seznámení s binárními rozhodovacími diagramy (BDD)

Logické funkce mohou být reprezentovány mnoha kanonickými způsoby. Mezi základní způsoby reprezentace logické funkce může být např. pravdivostní tabulka nebo konjunktivní popř. disjunktivní normální forma. Tyto formy reprezentace logické funkce jsou ale poměrně neefektivní a nevhodné pro reprezentaci logických funkcí s velkým počtem proměnných. Binárních rozhodovacích diagramů (stromů) se využívá právě k efektivní reprezentaci takovýchto funkcí a k usnadnění manipulace s těmito funkcemi. V praxi se pracuje převážně s modifikacemi binárních rozhodovacích diagramů. Těmito modifikacemi jsou například uspořádané binární rozhodovací diagramy (Ordered Binary Decision Diagrams - OBDD) nebo

redukované uspořádané binární rozhodovací diagramy (Reduced Ordered Binary Decision Diagrams - ROBDD).

Binárním rozhodovacím diagramem pro Booleovskou funkci $f(x_0,x_1, \ldots ,x_{n-1})$ je acyklický orientovaný graf s jedním počátečním uzlem a vnitřními uzly, které představují jednotlivé proměnné a jedním popř. dvěma koncovými uzly (terminály), které jsou většinou ohodnoceny logickými hodnotami 0,1. Každý nekoncový uzel má dva následníky nazývané 0-následník a 1-následník, které jsou rozlišené ohodnocením příslušných hran. Do diagramu se většinou nekreslí šipky a hrana, která vede k 0-následníkovi se v grafu většinou značí přerušovanou čárou, označuje se lo(x) a odpovídá případu kdy je do proměnné x přiřazena 0. Hrana která vede k 1-následníkovi se většinou značí plnou čarou, označuje se hi(x) a odpovídá případu kdy je do proměnné x přiřazena 1.

Binární rozhodovací diagram může tedy vypadat například takto:



Obrázek č.1 : *Příklad binárního rozhodovacího diagramu*

Následující pravdivostní tabulka odpovídá BDD z obrázku č.1 :

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabulka č.2: *Pravdivostní tabulka logické funkce*

Jeden binární rozhodovací diagram může reprezentovat několik různých logických funkcí, přičemž ale vytvořením BDD ztrácíme syntaktickou informaci původního výrazu a dále pracujeme pouze se sémantickou informací výrazu. Při zjišťování výsledné hodnoty logické funkce pro určité ohodnocení vstupních proměnných postupujeme od kořenového uzlu směrem k terminálu, přičemž výsledná hodnota logické funkce pro dané ohodnocení je rovna hodnotě terminálu. Pokud se daná proměnná nenachází na cestě od kořene do terminálového uzlu je na ní výsledek funkce nezávislý (don't care). Například pro binární rozhodovací diagram z obrázku č.1 je pro ohodnocení x1=0,x2=0,x3=1 výsledná logická funkce rovna 0.

2.2. Konstrukce binárních rozhodovacích diagramů (BDD)

Binární rozhodovací diagramy vznikají aplikací Shannonova expanzního teorému na logickou funkci tak jak je to popsáno v kapitole 1.3. a v kapitole 2.1, kde je popsán úvod do binárních rozhodovacích diagramů. Z obrázku č.1 z kapitoly 2.1 je také patrné, že BDD obsahuje značné množství redundancí. Tyto redundance vznikají na uzlech BDD a na terminálech. V dalších kapitolách je demonstrováno jak se v praxi provádí odstranění těchto redundancí (redukce BDD) a jak řazení proměnných ovlivňuje velikost BDD (OBDD).

2.2.1. Uspořádání proměnných BDD (OBDD)

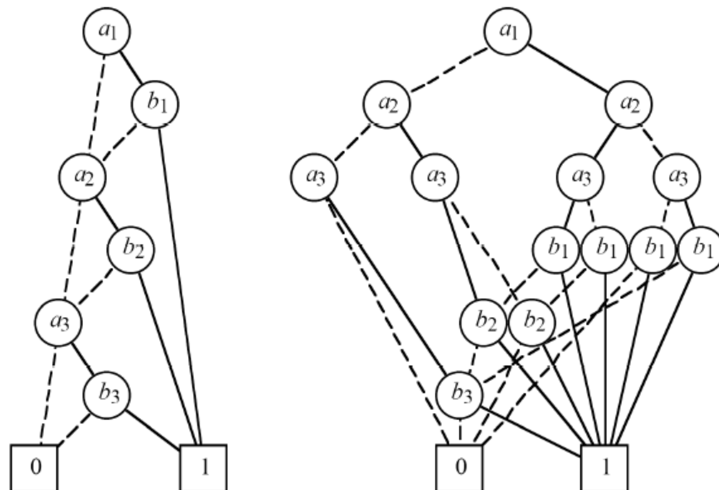
Za léta práce s binárními rozhodovacími diagramy bylo zjištěno, že jejich velikost(počet uzlů) je silně závislá na uspořádání proměnných.

Mějme uzel u a jeho potomka uzel v. Dále mějme binární rozhodovací diagram s určitým uspořádáním proměnných. O tomto BDD tvrdíme že je uspořádaný (OBDD) , jestliže pro všechny cesty od kořene k terminálovému uzlu je pořadí uzlů $u<v$ a zároveň uspořádání zachovává kanonickou vlastnost funkce a její sémantiku.

Obecně platí, že problém nalezení optimálního OBDD (tj. nalezení optimálního uspořádání) patří do množiny NP-úplných problémů což v praxi znamená, že nalezení optimálního OBDD je s využitím současných výpočetních prostředků poměrně značně obtížné.

Na následujícím obrázku č.2 jsou zobrazeny dva binární rozhodovací diagramy stejné funkce $f = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$, přičemž na OBDD vlevo jsou proměnné seřazeny v pořadí

a1<b1<a2<b2<a3<b3 zatímco na OBDD vpravo jsou proměnné v pořadí a1<a2<a3<b1<b2<b3. Z obrázku je na první pohled patrné, že uspořádání proměnných v OBDD vlevo je mnohem výhodnější než uspořádání proměnných v OBDD vpravo.



Obrázek č.2 : *Ukázka vlivu uspořádání proměnných na velikost OBDD (Bryant 1986)*

2.2.2. Redukce počtu proměnných BDD (RBDD)

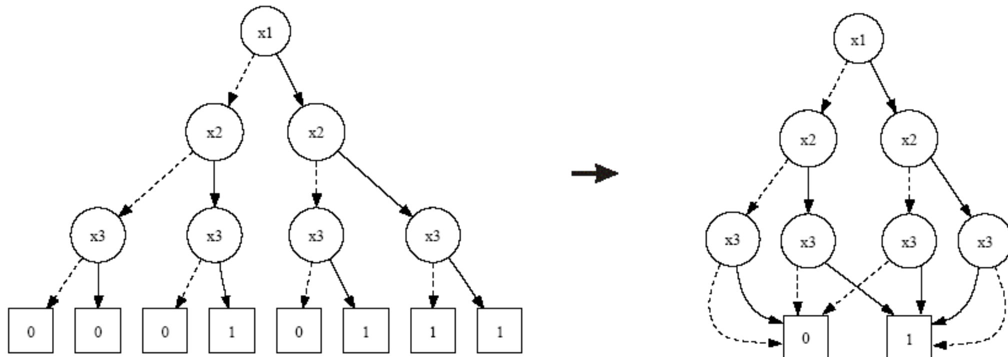
Jak už jsme si dříve řekli, při vytváření BDD dochází ve většině případů ke vzniku redundancí tzn. některé části BDD se opakují. Tyto redundance jsou z hlediska uchovávání informace o struktuře BDD a následně také z hlediska efektivity práce s BDD poměrně nepříjemné a snažíme se je tedy vhodně odstranit abychom zachovali kanonicitu a sémantiku logické funkce reprezentované jako BDD. V praxi dochází k několika typům redundancí :

- ***Redundance na terminálních uzlech*** – tato redundance je patrná například na obrázku č.1 v kapitole 2.1, kde je jasně vidět, že se zde opakují terminální uzly se stejnou hodnotou.

- ***Redundance na neterminálních uzlech*** - může být dvojího druhu :
    - Pokud levý i pravý potomek uzlu *X* je stejný, nemá uzel *X* z hlediska sémantiky žádný význam a tudíž může být vypuštěn.
    - Jestliže se v diagramu vyskytují identické podstromy, nesou tyto stejnou sémantickou informaci a vznikají tak duplicity.

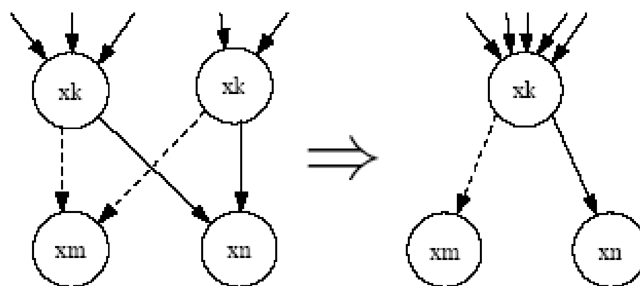
K odstranění redundancí se využívá tří transformačních pravidel, které vedou k redukci BDD aniž by přitom změnily sémantiku funkce, kterou BDD reprezentuje.

1.) Odstranění redundancí vznikajících na opakujících se terminálních uzlech. Této redukce docílíme tak, že odstraníme všechny redundantní terminální uzly až na jeden do kterého poté přesměrujeme všechny hrany, které směřovaly do odstraněných uzlů.



Obrázek č.3 : *Odstranění redundantních terminálů*

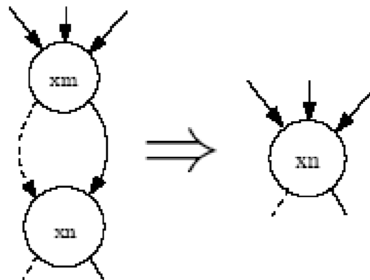
2.) Odstranění opakujících se vnitřních uzlů. Jestliže v BDD existuje uzel *u* a uzel *v*, které jsou stejné přičemž *lo(u)* je stejný jako *lo(v)* a *hi(u)* je stejný jako *hi(v),* pak můžeme odstranit uzel u nebo v a všechny hrany vedoucí do odstraněného uzlu přesměrujeme na zbylý uzel. Tato operace je názorně předvedena na následujícím obrázku č.3.



Obrázek č.4 : *Odstranění opakujících se vnitřních uzlů*

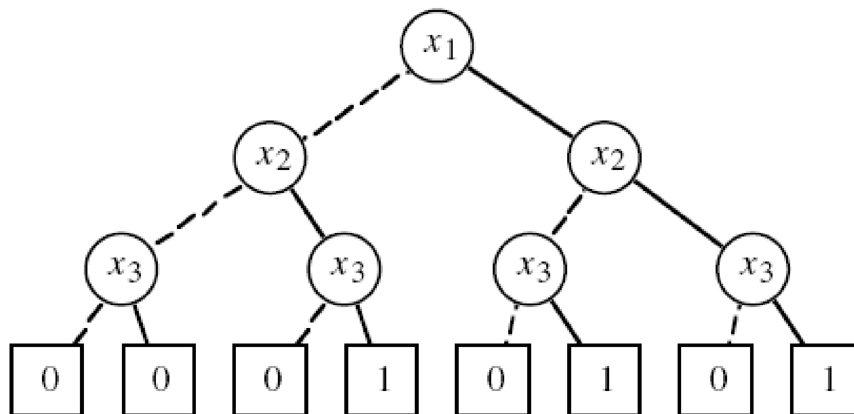
3.) Odstranění nadbytečných vnitřních uzlů. Jestliže uzel *u* má stejného levého i pravého potomka (*lo(u)* je stejný jako *hi(u)*), potom nemá ze sémantického hlediska v BDD žádný

význam a proto ho odstraníme a hrany které do něj vedly přesměrujeme do jeho potomka. Tato operace je znázorněna na následujícím obrázku č.4.



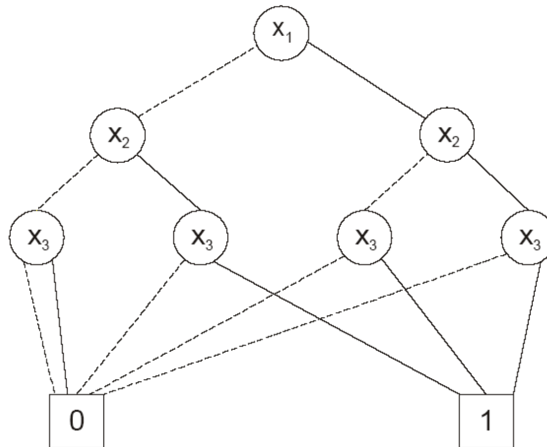
Obrázek č.5 : *Odstranění nadbytečného uzlu*

Opakovanou aplikací těchto tří pravidel snižujeme výslednou velikost BDD a vytváříme RBDD popř. ROBDD. Jak tato redukce vypadá v praxi si ukážeme na následujícím příkladu přičemž jako výchozí BDD použijeme BDD z obrázku č.1 (kapitola 2.1), který je z důvodu přehlednosti umístěn znovu pod tímto textem (obrázek č.5).



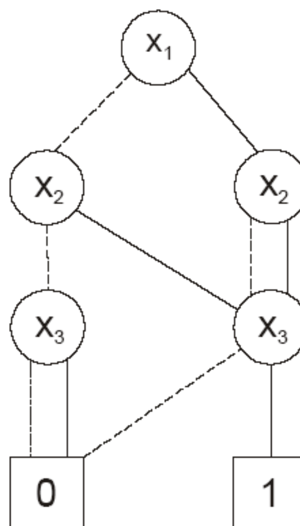
Obrázek č.6 : *BDD diagram obsahující redundance*

Na první pohled je vidět že tento BDD obsahuje značné množství redundancí a proto začneme s aplikací transformačních pravidel. Nejprve použijeme první transformační pravidlo a odstraníme redundantní terminální uzly. Na obrázku č.6 můžeme vidět jak vypadá BDD po aplikaci prvního pravidla.



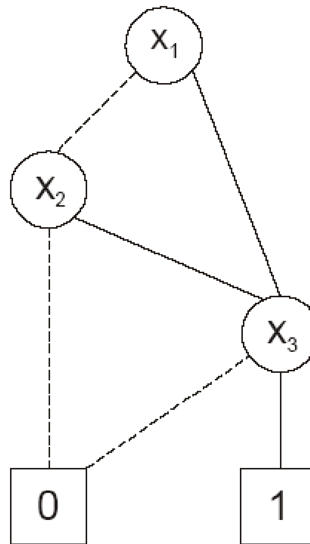
Obrázek č.7 : *BDD po odstranění redundantních terminálů*

Na obrázku č.6 je vidět, že po aplikaci prvního transformačního pravidla(odstranění redundantních terminálů) na BDD z obrázku č.5 jsme snížili velikost BDD o šest uzlů a z původních osmi terminálů nám zbyly pouze dva. Dále si všimněme, že tři uzly x_3 mají stejné potomky (terminály 0 a 1). Aplikujeme tedy transformační pravidlo číslo dvě a odstraníme tyto opakující se vnitřní uzly. Výsledek této operace můžeme vidět na následujícím obrázku č.7.



Obrázek č.8 : *BDD po odstranění redundantních vnitřních uzlů*

Na obrázku č.7 je vidět, že aplikací druhého transformačního pravidla jsme BDD z obrázku č.6 zmenšili o další dva uzly. Dále si všimněme, že jeden uzel x_2 a jeden z uzlů x_3 v BDD na obrázku č.7 mají stejného levého i pravého potomka (lo(x)=hi(x)) což znamená, že ještě můžeme aplikovat třetí transformační pravidlo na BDD z obrázku č.7 a odstranit tyto dva přebytečné uzly. Výsledek aplikace třetího transformačního pravidla můžeme vidět na následujícím obrázku č.8.



Obrázek č.9 : *BDD po odstranění nadbytečných uzlů*

Na obrázku č.8 můžeme vidět RBDD (redukovaný BDD) po aplikaci všech tří transformačních pravidel. Všimněme si, že výsledný RBDD je také ROBDD (Redukovaný Uspořádaný BDD) jelikož pořadí proměnných x1<x2<x3 je stejné jako v původním BDD na obrázku č.5. Navíc byl původní BDD z obrázku č.5 redukován z původních patnácti uzlů na pouhých pět uzlů. Je tedy zřejmé, že v praxi je mnohem výhodnější pracovat z RBDD popř. s ROBDD než s poměrně velkými BDD.

2.2.3.Manipulace s ROBDD

Množství operací prováděných nad booleovskými funkcemi lze implementovat jako grafové algoritmy a ty můžeme posléze aplikovat na ROBDD. Důležitou vlastností těchto algoritmů je, že zachovávají pořadí proměnných. V praxi to znamená, že pokud pořadí proměnných v ROBDD před provedením operace a po provedení operace je stejné a tudíž je možné i složitější operaci provést pomocí několika operací jednodušších. Bližší popis je v dokumentu [1,15].

2.2.3.1. Operace APPLY

V praxi je často nutné pracovat s více BDD (chceme provádět booleovské operace mezi několika BDD/ROBDD). To lze realizovat jednoduše tak, že procházíme jednotlivé BDD a na základě dílčích výsledků skládáme výsledek celé operace, kterou chceme provést. V praxi se ovšem ukázalo, že tato metoda je poměrně neefektivní a proto se spíše využívá operace apply, která má jako vstup například funkci *f* a funkci *g*. A dále má k dispozici booleovský operátor, který má na tyto funkce aplikovat (*apply(f,g,operator)*). Operace apply vezme vstupní funkce (ROBDD) *f* a *g* a provede nad nimi operaci danou operátorem a jako výsledek vrací funkci "*f operátor g*" – nejedná se o logickou hodnotu, ale o nový ROBDD.

Algoritmus, kterým je operace apply realizována prochází vstupní funkce do hloubky a potřebné informace si ukládá do dvou hashovacích tabulek. První tabulka obsahuje informace nutné ke zrychlení a zefektivnění výpočtu zatímco druhá tabulka obsahuje informace nutné k vytvoření maximálně redukovaného grafu. Algoritmus se opírá o Shannonův expanzní teorém a to tak, že rozloží obě vstupní funkce (*f,g*) podle stejné proměnné a poté mezi částmi původní funkce provede požadovanou operaci. Následující rovnice je základem rekurzivní procedury pro výpočet ROBDD reprezentace *f<op>g* (*f* operátor *g*).

$$f < op > g = \overline{x} \cdot (f\,|_{x \leftarrow 0} < op > g\,|_{x \leftarrow 0}) + x(f\,|_{x \leftarrow 1} < op > g\,|_{x \leftarrow 1})$$

3. Vlastní implementace

Jak již bylo dříve několikrát zmíněno, využil jsem k implementaci svého sat solveru knihovnu CUDD, ve které je implementována většina funkcí nutných pro práci s binárními rozhodovacími diagramy. Poměrně značnou část práce na solveru tedy zabralo seznámení s knihovnou CUDD . Další důležitou částí implementace byla realizace lexikálního a syntaktického analyzátoru(Příloha B.1), který slouží k načtení a kontrole vstupu. Vstup je poté zpracován pomocí funkcí balíku CUDD a je vytvořen binární rozhodovací diagram. Výstupem řešiče je rozhodnutí o splnitelnosti funkce (splnitelná/nesplnitelná) a zároveň jsou také vypsány všechny vektory proměnných, pro které je daná funkce splněná.

V následujícím textu se blíže seznámíme s mnou implementovaným sat solverem a heuristikami, které používá.

3.1. Popis vstupu

Vstupem mnou implementovaného sat solveru je funkce v CNF (konjunktivní normální forma) uložená v souboru ve formátu DIMACS[4]. Formát DIMACS je obecně uznávaným standardem pro reprezentaci funkce v knojunktivní normální formě. Soubor ve formátu DIMACS obsahuje několik lexikálních elementů o kterých se nyní podrobněji zmíníme:

c - na začátku řádku uvozuje komentář.

p - označuje začátek řádku, který není komentářem a obsahuje informace o formě zadané funkce a její velikosti.

cnf – označuje formát vstupu – konjunktivní normální forma.

Číslo – může označovat počet vstupních proměnných, počet termů, nebo reprezentuje literál.

Řetězec – řetězcem v tomto případě myslíme libovolný komentář.

Na začátku každého souboru ve formátu DIMACS je hlavička, která obsahuje případné komentáře dále příkaz ve tvaru “P CNF počet proměnných počet termů“, který udává velikost reprezentované funkce a dále potom obsahuje jednotlivé termy, přičemž tyto jsou vždy zadávány na řádek. Každý řádek je zakončen nulou a negované proměnné jsou označeny znaménkem mínus. Na následujícím příkladu si ukážeme jak takovýto soubor vypadá v praxi.

Př.: Mějme funkci $f = (a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee b \vee \neg c)$. Tato funkce obsahuje tři proměnné a,b,c namapujeme tedy tyto proměnné na čísla, kterými je posléze budeme reprezentovat :

$$A \rightarrow 1$$

$$B \rightarrow 2$$

$$C \rightarrow 3$$

Dále nahradíme symboly ve funkci f jejich numerickou reprezentací :

$$f = (1 \vee 2 \vee \neg 3) \wedge (\neg 1 \vee 2 \vee 3) \wedge (\neg 1 \vee 2 \vee \neg 3)$$

Vstupní soubor ve formátu DIMACS bude poté vypadat následovně:

```
c ukázkový cnf vstup – toto je komentář
c vytvořeno 13.7.2007 – toto je komentář
p cnf 3 3
1 2 –3 0
-1 2 3  0
-1 2 –3 0
```

Reprezentovaná funkce obsahuje tři proměnné a tři termy a je v CNF což je deklarováno ve třetím řádku uvozeném lexikálním elementem p. Po této deklaraci už následuje zápis termů logické funkce.

3.2. Popis implementovaných heuristik

Tato kapitola popisuje implementované heuristiky a demonstruje také jejich přínos k efektivnějšímu řešení splnitelnosti booleovské funkce.

Konstrukce ROBDD je řešena stejně, jako ve většině ostatních řešičů tzn. pro každý term se vytvoří BDD a ten se poté pomocí operace APPLY(kapitola 2.2.3.1) přidává k výslednému ROBDD. Jinou možností by bylo vytvořit pro každý term BDD, ověřit jeli takto vytvořený BDD splnitelný a mezi dílčími výsledky provést operaci logického součinu. Tato metoda se ovšem v praxi ukázala mnohem pomalejší než vytvoření jednoho velkého ROBDD a tudíž jsem ji také ve své práci nepoužil.

Další důležitou částí implementace bylo nalezení optimálního pořadí proměnných pro ROBDD. Jak již bylo dříve řečeno, pořadí proměnných má zásadní vliv na velikost konstruovaného ROBDD a tudíž také na dobu řešení a množství alokované paměti. V praxi je možné použít statické nebo dynamické heuristiky rovnání proměnných(reorderingu). Statické metody rovnání proměnných sestaví na počátku konstrukce ROBDD pořadí proměnných, které se dále při práci s ROBDD nemění. Mezi tyto statické heuristiky mohou patřit například heuristiky implementované v SAT solveru EBDDRES a popsané v kapitole 4.2. Dynamické heuristiky rovnání proměnných(reorderingu) na rozdíl od statických metod pracují na nalezení lepšího pořadí proměnných v průběhu konstrukce ROBDD. Znamená to tedy, že tyto heuristiky jsou spouštěny v průběhu tvorby ROBDD(např. když přírůstek uzlů dosáhne určité hodnoty) a prohazováním proměnných v pořadí (v ROBDD) se snaží nalézt lepší pořadí což je takové, kterému odpovídá ROBDD s nejmenším počtem uzlů. Heuristiky dynamického řazení

proměnných byly popsány v příloze A.3 a byly taktéž implementovány v mém řešiči. Z hlediska implementace by zřejmě bylo nejlepší použít kombinaci heuristik statického a dynamického řazení proměnných, nicméně knihovna CUDD, která byla použita k implementaci řešiče bohužel nepodporuje vytvoření statického pořadí na začátku tvorby ROBDD a tudíž se statické metody řazení proměnných nepodařilo implementovat. Na druhou stranu knihovna CUDD obsahuje širokou paletu dynamických heuristik reorderingu, které lze využít. Tyto metody jsou v řešiči implementovány a jsou popsány v příloze A.3 tudíž se jimi nyní nebudu blíže zabývat.

3.2.1. Heuristika rovnání termů vstupní funkce

Mimo heuristik řazení proměnných byla také implementována heuristika řazení termů, která měla původně sloužit k tomu aby termy srovnala do pořadí, ve kterém se dříve projeví případná nesplnitelnost logické funkce a tím se sníží doba řešení nesplnitelných funkcí, nicméně měřením bylo zjištěno, že pořadí vstupních termů má také zásadní vliv na počet uzlů vytvářeného ROBDD a může tedy vést k značnému zrychlení řešiče. Tento algoritmus pracuje s dvourozměrným polem termů, jednorozměrným polem výskytů literálů a vah termů a zjednodušeně ho můžeme popsat pomocí několika kroků následovně:

1. Pro každý term vstupní funkce provedeme načtení termu do pole termů a v poli proměnných inkrementujeme čítač výskytu proměnné.
2. Pro každý term v poli termů spočítáme jeho váhu, která je dána součtem četností výskytů literálu v termu obsažených a hodnotu uložíme do pole vah termů.
3. Nyní je provedeno seřazení pole termů podle pole vah termů algoritmem quick sort sestupně.
4. Výsledné pole termů je vypsáno do souboru temp, který se poté používá k dalšímu zpracování v pořadí řazení termů sestupně, nebo vzestupně podle zadaných vstupních parametrů.

Příklad práce heuristiky řazení termů:

Vstupní funkce:

$f = (x_1 \lor \neg x_4 \lor x_6) \land (x_1 \lor x_4 \lor \neg x_6) \land (x_4 \lor x_5 \lor x_6) \land (x_1 \lor x_5 \lor \neg x_6) \land (x_2 \lor x_5 \lor x_6) \land (\neg x_3 \lor \neg x_5 \lor \neg x_6) \land$
$(x_3 \lor \neg x_4 \lor x_5)$

je zadána ve formátu DIMACS a vypadá následovně:

```
p cnf 6 7
6 –4 1 0
-6 4 1 0
6 5 4 0
-6 5 1 0
6 5 2 0
-6 –5 –3 0
5 –4 3 0
```

a v příkladu je označována jako *vstupní funkce*(0 označuje pouze konec termu a v poli *vstupní funkce* se tedy neuvádí).

Krok 0. - inicializace polí

Vstupní funkce

6	-4	1
-6	4	1
6	5	4
-6	5	1
6	5	2
-6	-5	-3
5	-4	3

Pole termů

Index i/j	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0

Pole vah termů

Index	Váhy
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Pole četnosti proměnných

Index	Četnost
0	0
1	0
2	0
3	0
4	0
5	0

Tabulka č.3 : *Obsah polí na počátku heuristiky řazení termů*

Krok 1. – načtení termů do pole a vytvoření pole četnosti výskytů proměnných

Vstupní funkce

6	-4	1
-6	4	1
6	5	4
-6	5	1
6	5	2
-6	-5	-3
5	-4	3

Pole termů

Index i/j	0	1	2
0	6	-4	1
1	-6	4	1
2	6	5	4
3	-6	5	1
4	6	5	2
5	-6	-5	-3
6	5	-4	3

Pole vah termů

Index	Váhy
0	0
1	0
2	0
3	0
4	0
5	0
6	0

Pole četnosti proměnných

Index	Četnost	
0	3	(1)
1	1	(2)
2	2	(3)
3	4	(4)
4	5	(5)
5	6	(6)

Tabulka č.4 : *Obsah polí po načtení pole termů(+četnost výskytů)*

Krok 2. – vytvoříme pole vah termů – term 0: 6(6) + 4(4) + 3(1) = <u>13</u> ⇐ váha termu 0

Vstupní funkce

6	-4	1
-6	4	1
6	5	4
-6	5	1
6	5	2
-6	-5	-3
5	-4	3

Pole termů

Index i/j	0	1	2
0	6	-4	1
1	-6	4	1
2	6	5	4
3	-6	5	1
4	6	5	2
5	-6	-5	-3
6	5	-4	3

Pole vah termů

Index	Váhy
0	13
1	13
2	15
3	14
4	13
5	13
6	12

Pole četnosti proměnných

Index	Četnost	
0	3	(1)
1	1	(2)
2	2	(3)
3	4	(4)
4	5	(5)
5	6	(6)

Tabulka č.5 : Obsah polí po vytvoření pole vah

Krok 3. seřadíme pole termů podle pole vah termů – algoritmus quick sort

Vstupní funkce

6	-4	1
-6	4	1
6	5	4
-6	5	1
6	5	2
-6	-5	-3
5	-4	3

Pole termů

Index i/j	0	1	2
0	6	5	4
1	-6	5	1
2	6	-4	1
3	-6	4	1
4	6	5	2
5	-6	-5	-3
6	5	-4	3

Pole vah termů

Index	Váhy
0	15
1	14
2	13
3	13
4	13
5	13
6	12

Pole četnosti proměnných

Index	Četnost	
0	3	(1)
1	1	(2)
2	2	(3)
3	4	(4)
4	5	(5)
5	6	(6)

Tabulka č.6 : Obsah polí po seřazení podle vah termů

Krok 4. Seřazené pole termů je vypsáno do souboru temp

Vstupní funkce Temp

6	5	4
-6	5	1
6	-4	1
-6	4	1
6	5	2
-6	-5	-3
5	-4	3

Pole termů

Index i/j	0	1	2
0	6	5	4
1	-6	5	1
2	6	-4	1
3	-6	4	1
4	6	5	2
5	-6	-5	-3
6	5	-4	3

Pole vah termů

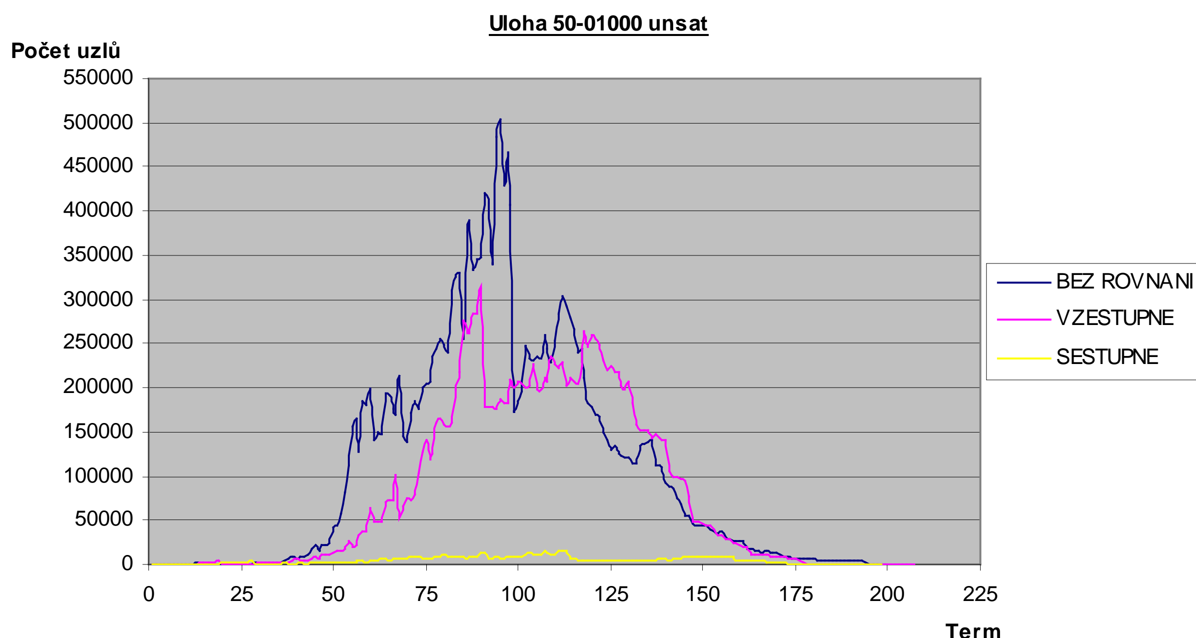
Index	Váhy
0	15
1	14
2	13
3	13
4	13
5	13
6	12

Pole četnosti proměnných

Index	Četnost	
0	3	(1)
1	1	(2)
2	2	(3)
3	4	(4)
4	5	(5)
5	6	(6)

Tabulka č.7 : Výpis pole termů do souboru temp

Vliv tohoto řazení termů na velikost ROBDD(počet uzlů) můžeme vidět v následujícím grafu:



Obrázek č.10 : *Graf závislosti velikosti ROBDD(počtu uzlů) na pořadí vstupních termů*

Měření bylo provedeno na nesplnitelné úloze, která měla 50 proměnných a 218 termů. Na ose x grafu jsou vyneseny jednotlivé termy a na ose y jsou počty uzlů, které měl vytvářený ROBDD v těchto termech. Graf tedy ukazuje, kolik uzlů obsahoval vytvářený ROBDD po přidání termu *x*(tzn. např. po přidání 20 termů do ROBDD měl tento ROBDD velikost 120uzlů). Z grafu je jasně vidět, že nejefektivnější je rovnání termů sestupně od nejvyšší váhy po nejnižší, což se také dalo předpokládat, protože tyto termy s nejvyšší váhou obsahují proměnné s vyšší četností a tyto se vyskytují ve stromové struktuře BDD blíže ke kořenu. Tím že začínáme vytvářet ROBDD od nich si tedy ušetříme práci s případným dalším přerovnáváním a ROBDD roste mnohem pomaleji. Z grafu je dále vidět, že i přerovnávání termů vzestupně má vliv na počet uzlů, nicméně nevede k tak razantnímu zlepšení jako při přerovnávání sestupně. Toto měření bylo provedeno pro několik úloh a výsledky ukazují následující tabulky:

UNSAT									
Úloha	BEZ ROVNANI			ROVNANI VZESTUPNE			ROVNANI SESTUPNE		
	Max uzly	Konečný term	Doba řešení	Max uzly	Konečný term	Doba řešení	Max uzly	Konečný term	Doba řešení
50-0100	561982	215	203,485	382142	217	73,766	5704	215	1,938
50-0500	350652	204	92,875	319339	206	125,53	12284	215	2,266
50-01000	503999	199	195,281	311406	207	86,953	14760	198	2,422
75-01	-	-	-	-	-	-	193594	311	53,75
75-0100	-	-	-	-	-	-	98109	302	19,516

***Tabulka č.8:** Heuristika rovnání vstupních termů pro nesplnitelné booleovské funkce*

SAT						
Úloha	BEZ ROVNANI		ROVNANI VZESTUPNE		ROVNANI SESTUPNE	
	Max uzly	Doba řešení	Max uzly	Doba řešení	Max uzly	Doba řešení
20-0100	1806	0,594	1246	0,625	262	0,562
20-0500	1656	0,531	537	0,5	280	0,516
20-0900	1050	0,531	1037	0,515	104	0,453
50-0100	484729	116,593	297310	59,156	5753	1,812
50-0500	839559	403,312	726400	336,969	19191	2,422
50-01000	357752	98,125	791650	262,672	22008	3,297
75-01	-	-	-	-	253467	105,875
75-095	-	-	-	-	451053	122,625
75-0100	-	-	-	-	189468	43,39

***Tabulka č.9:** Heuristika rovnání vstupních termů pro splnitelné booleovské funkce*

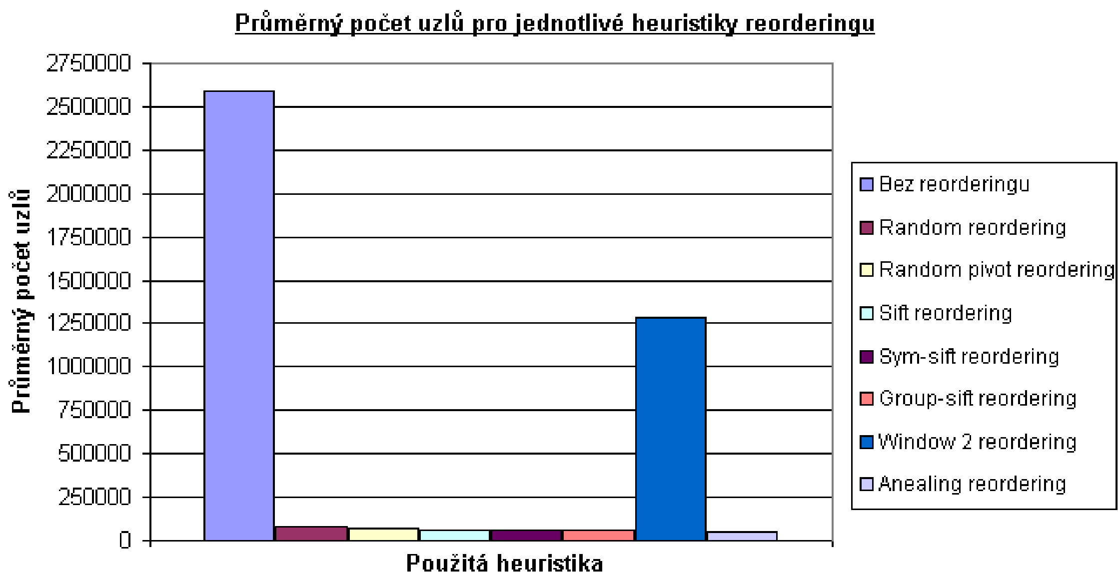
Tabulky č.8 a č.9 obsahují naměřené hodnoty pro splnitelné/nesplnitelné funkce a různé heuristiky řazení vstupních termů(bez řazení/řezení sestupně/řazení vzestupně), přičemž sloupec *úloha* obsahuje označení testované úlohy, sloupec *max uzly* obsahuje maximální počet uzlů, se

kterými byl řešič v průběhu vytváření ROBDD nucen pracovat, sloupec doba řešení obsahuje časový údaj v sekundách odpovídající době potřebné k nalezení výsledku(splnitelná/nesplnitelná) a pro nesplnitelné úlohy je v grafu ještě sloupec *konečný term,* jenž obsahuje číslo termu po jehož přidání se projevila nesplnitelnost a řešič ukončil hledání řešení s verdiktem, že zkoumaná funkce je nesplnitelná.

Z předchozích tabulek taktéž vyplývá, že při použití heuristiky řazení vstupních termů sestupně dochází ke značnému snížení počtu uzlů a tím také dochází ke snížení doby potřebné ke konstrukci ROBDD a nalezení řešení. Na době řešení se snížení počtu uzlů výrazněji projeví až při počtu proměnných ~50 což je také vidět v předchozích tabulkách. K testování byly použity úlohy SAT3 s počtem proměnných 20,50,75 přičemž první část čísla úlohy udává počet proměnných a druhá část udává verzi úlohy. Úlohy které se s použití dané heuristiky nepodařilo otestovat mají v příslušných buňkách znak -. Shodně pro všechny měření bylo také použito dynamického rovnání proměnných heuristikou group sifting(blíže v Příloze A.3).

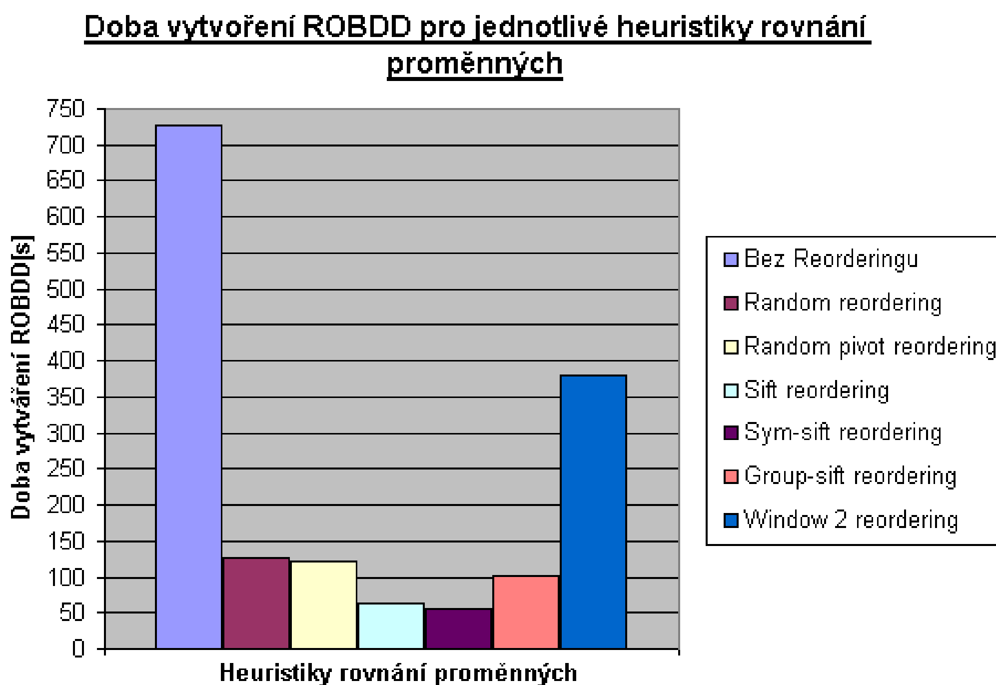
3.2.2. Heuristiky dynamického reorderingu

Jak již bylo dříve řečeno, lze pro nalezení řešení použít množství heuristik dynamického reorderingu, které nabízí knihovna CUDD. Popis těchto heuristik je v příloze A.3 a nyní si tedy spíše ukážeme jaký mají vliv na rychlost řešení(velikost ROBDD). Pro všechny heuristiky bylo provedeno měření na vybrané úloze uf75-01, která obsahuje booleovskou funkce o velikosti 75 proměnných a 325 termů. Výsledky byly pro přehlednost zpracovány do následujících grafů a tabulky.



Obrázek č.11 : *Průměrný počet uzlů na term pro jednotlivé heuristiky rovnání proměnných*

Graf na obrázku č.11 graficky znázorňuje průměrný počet uzlů na jeden term pro jednotlivé heuristiky. Z tohoto grafu je patrné, že s největší počet uzlů musíme zpracovat v případě, že neprovádíme rovnání proměnných. Jako druhá nejhorší heuristika v tomto směru se umístila heuristika window2 reordering. Další heuristiky poté dosahovali přibližně stejných výsledků(co se týče množství zpracovávaných uzlů).



Obrázek č.12 : *Doba vytvoření ROBDD pro jednotlivé heuristiky rovnání proměnných*

Na obrázku č.12 můžeme porovnat heuristiky rovnání proměnných z hlediska doby zpracovávání úlohy. Z heuristik zobrazených v grafu dosáhla nejhorší doby zpracování vstupní funkce heuristika, která neprováděla žádné rovnání proměnných a hned za ní se umístila heuristika Window 2. Ostatní heuristiky dosahovaly srovnatelných výsledků. Toto pořadí by tedy odpovídalo pořadí z grafu č.11. Tudíž bychom mohli tvrdit, že pokud heuristika dokáže udržet přírůstek uzlů v rozumné míře,výrazně tím snižuje dobu nutnou ke zpracování úlohy. V následující tabulce č.10 se ale můžeme přesvědčit, že toto tvrzení nemusí být vždy pravdivé.

Porovnání heuristik reorderingu			
Použitá heuristika	**Průměrný počet uzlů**	**Maximální počet uzlů**	**Doba řešení**
Bez Reorderingu	2589456,742	22121155	727,203
Random reordering	78677,16308	459369	126,735
Random pivot reordering	70788,84923	327891	120,938
Sift reordering	63361,07692	280393	62,469
Sym-sift reordering	59332,64308	287506	56,453
Group-sift reordering	58015,26154	253467	100,985
Window 2 reordering	1286581,705	9315742	379,312
Anealing reordering	45288,21231	207732	61903,1
Genetic reordering	-	-	-

Tabulka č.10 : *Srovnání heuristik rovnání proměnných*

Tabulka obsahuje naměřené hodnoty, ze kterých byly sestaveny grafy na obrázcích č.11 a č.12 a navíc obsahuje také hodnoty měření pro heuristiku simulovaného ochlazování. Výsledky této metody nebyly zaneseny do grafu č.12, protože by zastínily ostatní naměřené hodnoty. Z výsledků měření této heuristiky vyplývá, že ačkoli má nejmenší průměrný počet uzlů na term(dokáže sestavit nejlepší pořadí), doba řešení je mnohem vyšší než u ostatních heuristik. To potvrzuje tvrzení z Přílohy A.3 že tato metoda je velmi pomalá. V praxi je tedy vhodnější najít kompromis mezi velikostí ROBDD a dobou strávenou hledáním optimálního pořadí proměnných, neboť ta tvoří velkou část doby zpracování.

V tabulce nejsou zaneseny hodnoty získané pro heuristiku genetic reordering, neboť tato byla ještě pomalejší než heuristika simulovaného ochlazování a nepodařilo se ji tedy zpracovat.

3.3. Výstup implementovaného SAT solveru

Výstupem standardního sat solveru je rozhodnutí o splnitelnosti funkce (splnitelná/nesplnitelná). Sat solvery řešící problém splnitelnosti booleovské funkce s použitím

BDD (ROBDD) mohou navíc kromě rozhodnutí o splnitelnosti funkce vypsat všechny vektory (i ty neúplně určené), pro které je funkce splnitelná, popř. můžeme pomocí několika funkcí knihovny CUDD vypsat nejkratší cestu(vzhledem k tomu že se vypisují všechny vektory se výpis nejkratší cesty neprovádí) do terminálového uzlu (nejvíce don't care stavů) a mnoho dalších zajímavých údajů. Mnou implementovaný řešič může mít jako výstup tři soubory. Nyní si tyto výstupy popíšeme.

Standardním výstupem mnou implementovaného řešiče je souboru typu *PLA*[14] , který obsahuje úplně i neúplně určené vektory, pro které je daná funkce splněná. Jméno souboru je zadáváno jako parametr při spouštění programu a obsah souboru vypadá následovně :

```
.i 20
.o 1
01101101110011101011  1
0111001110011111001-  1
01111011100111110011  1
.e
```

v hlavičce tohoto souboru se vyskytuje klauzule *.i číslo*, která definuje kolik má daná funkce vstupů (počet vstupních proměnných = číslo), dále obsahuje klauzuli *.o číslo*, která udává počet výstupů funkce a poté následují úplně i neúplně určené vektory obsahující logické hodnoty pro všechny proměnné (logické hodnoty → 0=low,1=high, -=don't care) přičemž za každým vektorem je hodnota výstupu logické funkce (v našem případě je vždy jeden výstup → hodnota výstupu bude vždy 1). Soubor je ukončen značkou konce souboru *.e*. V případě že je funkce nesplnitelná, neobsahuje soubor žádné vektory. Další možností je, že BDD obsahuje velké množství cest do terminálu 1, což by při výpisu vektorů znamenalo uložit značné množství dat. Z tohoto důvodu byla velikost tohoto výstupního souboru omezena na 100MB. Při překročení hranice 100MB je do souboru vypsáno chybové hlášení.

Druhým výstupem řešiče je soubor debug, ve kterém se průběžně ukládají informace o vytvářeném ROBDD. Jméno souboru do kterého se ukládá výstup debug se skládá z klíčového slova *Debug*, celého jména výstupního souboru a čísla, které označuje jaká heuristika byla použita. Název souboru může tedy vypadat následovně "Debug.vystupUf20-0100.txt-5.txt". Tento soubor obsahuje informace o každém přidaném termu přičemž tyto se ukládají v následujícím formátu :

**** term cislo 1 ****
Pridani termu 1 trvalo : 0 sekund
Pocet dead nodes : 0
Pocet vsech uzlu : 4
Pamet zabrana managerem [kB] : 4737
Number of reorderings : 0
Celkova doba tvorby bdd : 0

Dále jsou do něj po vytvoření BDD (ROBDD) uloženy souhrnné informace např. rozhodnutí o splnitelnosti funkce, počet cest do terminálu 1. Formát výsledného souboru tedy může vypadat následovně :

```
****  term cislo 1  ****
Pridani termu 1 trvalo : 0 sekund
Pocet dead nodes : 0
Pocet vsech uzlu : 4
Pamet zabrana managerem [kB] : 4737
Number of reorderings : 0
Celkova doba tvorby bdd : 0

****  term cislo 2  ****
Pridani termu 2 trvalo : 0 sekund
Pocet dead nodes : 3
Pocet vsech uzlu : 7
Pamet zabrana managerem [kB] : 4755
Number of reorderings : 0
Celkova doba tvorby bdd : 0.016

…
…

****  term cislo 91  ****
Pridani termu 91 trvalo : 0 sekund
Pocet dead nodes : 17
Pocet vsech uzlu : 48
Pamet zabrana managerem [kB] : 4787
Number of reorderings : 0
Celkova doba tvorby bdd : 0.594

*******************************************
Vytvoreni BDD trvalo celkem : 0.594 sekund
Cas straveny reorderingem : 0
Pocet cest do terminalu '1' : 3
*******************************************
Funkce JE SPLNITELNA !!!!
```

Třetím a také posledním výstupem řešiče je výstup graph, který lze použít pro statistické účely. Jméno tohoto souboru se skládá z klíčového slova Graph, jména výstupního souboru a čísla označujícího heuristiku použitou pro vytvoření BDD (ROBDD). Toto jméno může vypadat například následovně "Graph.vystupUf20-0100.txt-5.csv". Tento soubor je typu .csv a ukládají se do něj informace o čísle termu, počtu uzlů pro daný term a doba přidání termu do ROBDD.

4. SAT solver EBDDRES

Tento sat solver byl vytvořen na universitě Johanese Keplera (Johanes Kepler University) v institutu pro formální modely a verifikaci(FMV – Institute for Formal Models and verification) dvojicí Armin Biere a Toni Jessila. K řešení problému splnitelnosti booleovské funkce používá podobně jako mnou implementovaný řešič binární rozhodovací diagramy což bylo jedním z hlavních důvodů proč jsem si ho vybral pro srovnání. Dále také stejně jako můj řešič je výstupem nejen rozhodnutí o splnitelnosti logické funkce, ale také výpis vektorů, pro které je funkce splněná. Řešič také na rozdíl od ostatních umožňuje generovat tzv. rozšířený důkaz řešení (extended resolution proof), nicméně vzhledem k tomu, že tato funkce není pro porovnání s mým řešičem podstatná nebudu se jejím popisem více zabývat. Popis této funkce je spolu s důkazy v dokumentu [9,10]. V následujících několika kapitolách si ve zkratce tento řešič popíšeme.

4.1. Popis vstupu

Vstupem sat solveru EBDDRES je logická funkce zadaná v konjunktivní normální formě. Stejně jako u mnou implementovaného sat solveru je k reprezentaci vstupu použit velmi rozšířený standart DIMACS. Tento standart je popsán v kapitole 3.1 a v dokumentu [4] nebudu se jím tedy dále zabývat.

4.2. Použité heuristiky

Hledání řešení pomocí sat solveru EBDDRES je stejně jako u ostatních sat solverů používající k řešení binární rozhodovací diagramy do značné míry závislé na nalezení optimálního pořadí proměnných. Toto pořadí proměnných má totiž jak již bylo dříve řečeno zásadní vliv na velikost BDD a tudíž je jeho nalezení velmi důležité. Sat solver EBDDRES nepoužívá k nalezení optimálního pořadí dynamické metody řazení proměnných, ale naopak spoléhá na dobrou volbu počátečního statického pořadí. Počáteční pořadí proměnných může být

ponecháno původní popř. obrácené což znamená, že toto pořadí je v podstatě převzato ze vstupního souboru. Další možností je využití algoritmu který řadí proměnné podle četnosti jejich výskytu, podle DFS pořadí popř. použitím algoritmu FORCE. Je zřejmé, že i v případě že zvolíme dobré počáteční pořadí, může docházet k většímu nárustu uzlů a zplnění paměti. Z tohoto důvodu obsahuje sat solver EBDDRES také algoritmus k eliminaci proměnných s jehož pomocí můžeme uvolnit část paměti, kterou zabírají “nepotřebné“ uzly. V další části si tyto použité algoritmy popíšeme.

Řazení podle četnosti výskytu proměnné – tento algoritmus je popsán v dokumentu [13] pod názvem *Variable State Independent Decaying Sum* (VSIDS) pomocí pěti kroků, nicméně v řešiči EBDDRES je implementována jeho zjednodušená verze, která se dá zkráceně popsat ve dvou krocích :

1. Každá proměnná má pro každou polaritu čítač, který obsahuje hodnotu udávající počet výskytů proměnné ve funkci(funkce init_count)

2. Proměnné jsou srovnány do výsledného pořadí podle četnosti výskytů (v našem případě je použito řazení quick sort, řazení a operace s ním související se spouští voláním funkce init_sum_order).

Řazení proměnných algoritmem dfs – Tento algoritmus lze popsat pomocí několika bodů:

1. Všechny proměnné jsou uloženy do zásobníku a poté seřazeny podle počtu výskytů ve funkci v obráceném pořadí (od nejvyššího po nejnižší).

2. Ve všech termech seřadíme literály(proměnné) podle četností výskytu v obráceném pořadí(od nejvyššího po nejnižší).

3. Vybereme proměnnou(literál) ze zásobníku a pokud ještě nemá pozici v pořadí proměnných tak ji přiřadíme aktuální pozici, inkrementujeme proměnnou ukazující na aktuální pozici a ze všech termů, kde se proměnná vyskytuje uložím do zásobníku ostatní proměnné v termech obsažené. Pokud už proměnná má pozici v pořadí proměnných, pokračujeme ve vybírání proměnných ze zásobníku. Sled operací v třetím bodě opakujeme dokud jsou v zásobníku nějaké proměnné.

Mluvíme-li v tomto algoritmu o řazení máme na mysli quick sort.

Algoritmus FORCE – vychází z nějakého počátečního pořadí proměnných a toto pořadí se snaží iterativně zlepšit. Parametrem podle kterého se zlepšení porovnává je celkový span(rozpětí) a je definován jako součet spanů jednotlivých termů, přičemž span termu je největší rozpětí pořadí dvou proměnných v tomto termu. Algoritmus nejprve pro každý term spočítá tzv. center of gravity (COG – střed gravitační síly) a pak pro každou proměnnou spočítá z jednotlivých COG její lepší pořadí. Tento postup se opakuje tak dlouho, dokud dochází k nějakému zlepšení. Detailní popis tohoto algoritmu je v [5,12].

Eliminace proměnných – tento algoritmus lze popsat následujícím pseudokódem :

Algoritmus VE (Variable Elimination) – CNF Δ, pořadí π

```
1.  FOR každou proměnnou v z Δ DO
2.        vytvoř prázdný “kbelík“(bucket) Bv
3.  FOR každý term c z Δ DO
4.        v = první proměnná z c odpovídající pořadí π
5.        Bv = Bv ∪ { BDD(c)}
6.  FOR každou proměnnou v z Δ v pořadí π DO
7.        IF Bv není prázdný THEN
8.          BDDr = konjunkce všech elementů v Bv
9.        IF BDDr = nula THEN
10.         return UNSATISFIABLE
11.       BDDr = ∃vBDDr
12.       u = první proměnná v BDDr odpovídající pořadí π
13.       Bu = Bu ∪  {BDDr}
14.  return SATISFIABLE
```

Na začátku je každý term ci z CNF Δ = c1 ∧ c2 ∧ … ∧ cm převeden na BDD – BDD(ci) (řádek 5 algoritmu VE). Dané celkové pořadí proměnných π z Δ - označené jako eliminační pořadí (elimination order). “Kbelík“ (bucket) Bv je vytvořen pro každou proměnnou v (řádek 1 a 2 algoritmu VE) a každý BDD(ci) je přiřazen “kbelíku“ podle první proměnné podle pořadí proměnných π (řádek 3-5 algoritmu VE). Tyto kbelíky jsou postupně zpracovávány přičemž používají pořadí proměnných

π a přeskakují prázdné “kbelíky“ (řádek 6-13 algoritmu VE). Když je “kbelík“ B_v zpracováván, jsou všechny BDDs v něm sdružené spojené (řádek 8 algoritmu VE) v libovolném pořadí a je spočítán výskyt proměnné v (řádek 11 algoritmu VE) takže je odstraněna z výsledku, který je poté vložen do “kbelíku“ podle své první proměnné (řádek 13 a 13 algoritmu VE). Pokud během tohoto iteračního procesu vznikne BDD, který obsahuje pouze nulovou konstantu, je CNF prohlášena za nesplnitelnou (řádek 10 algoritmu VE), jinak je prohlášena za splnitelnou (řádek 14 algoritmu VE). Detailní popis tohoto algoritmu je v [11].

Předchozí popis heuristik používaných v SAT solveru EBDDRES je vzhledem k minimální dokumentaci tohoto řešiče založen na analýze zdrojového kódu řešiče a také vychází z literatury, kterou tvůrci řešiče použily, tudíž se nepodařilo najít odpovědi na všechny otázky, které by mohly být v souvislostí s touto implementací kladeny např. má algoritmus řazení dfs má něco společného s algoritmem prohledávání do hloubky … .

5. Porovnání implementovaného SAT solveru a EBDDRES solveru

V této kapitole bude provedeno srovnání implementovaného SAT solveru a EBDDRES SAT solveru. Hned na začátku je nutné si uvědomit, že vzhledem k různým přístupům k řešení problému splnitelnosti není možné na základě měření prohlásit, že daný solver je dobrý, nebo špatný, ale můžeme na základě měření nad skupinou úloh prohlásit, pro které problémy je daný solver výhodné použít. V našem případě bylo vybráno 18 úloh na kterých bylo provedeno testování řešičů pro různé heuristiky a výsledky měřená můžeme vidět v následující tabulce č.11:

1			2		3		4		5			6			7		
Úloha			**EBDDRES SAT solver**						**Implementovaný SAT solver**								
			VE		**DFS**		**Sum-order**		**Bez rovnání**			**Random pivot**			**Sym-sift**		
Název	Var	Termy	sec	Size	sec	Size	sec	Size	sec	Size	Vektory	sec	Size	Vektory	sec	Size	Vektory
uf20-0900	20	91	0,01	2730	0	6624	0,01	5595	0,56	104	15	0,55	104	15	0,56	104	15
uf50-0500	50	218	0,81	712321	-	-	-	-	2,38	48045	2	2,52	14014	2	2,30	12363	2
uf75-095	75	325	-	-	-	-	-	-	189	6526432	20	205,66	437069	20	68,05	289806	16
uuf50-0100	50	218	1,24	941985	-	-	-	-	1,89	42669	-	1,66	6504	-	1,66	6279	-
uuf75-0100	75	325	-	-	-	-	-	-	53,3	116542	-	27,18	108004	-	17,54	103274	-
mutcb6	65	172	0,01	3114	0,1	133547	0,05	33468	1,1	2571	-	1,28	2257	-	1,17	2257	-
mutcb7	91	251	0	8127	0,2	242888	0,73	518815	2,06	8133	-	2,66	8699	-	2,62	8148	-
mutcb8	121	344	0	15815	0,6	599983	5,48	4051670	4,19	22489	-	13,34	36078	-	5,86	23547	-
ph8	72	297	0,17	236317	0,3	251154	0,26	290133	2,48	28318	-	3,24	25118	-	3,43	24447	-
ph9	90	415	0,88	864277	1,1	778378	1,03	900015	6,98	127246	-	16,46	135833	-	35,83	140160	-
ph10	110	561	3,72	2973890	3,6	2304338	3,18	2758272	16,1	241912	-	59,59	212321	-	69,57	240290	-
ph11	132	738	14,5	9725225	11	6598919	9,73	8098299	42,1	799322	-	268,78	859427	-	346,91	640189	-
40-10000	40	10000	0,21	58845	5,8	2871724	6,91	3181510	47,2	41	1	48,57	41	1	46,86	41	1
65-10000	65	10000	2,82	1622270	-	-	-	-	46,9	113	1	48,59	113	1	47,37	113	1
add4	60	157	0,01	2161	0	26854	0,02	9548	1,13	6700	-	1,40	4187	-	1,19	2305	-
flat50-192	150	545	-	-	-	-	-	-	33,9	753250	14976	43,14	141567	14976	28	183091	14976
flat75-76	225	840	-	-	-	-	-	-	14,4	168979	82944	44,01	122020	82944	21,31	48373	82944
flat100-73	300	1117	-	-	-	-	-	-	771	7499061	1.548e+06	4050,72	1956086	1.548e+06	2285,49	1077079	1.548e+06

Tabulka č.11 : *Srovnání SAT solveru EBDDRES a implementovaného SAT solveru*

Tabulka se skládá z několika sloupců označených 1-7přičemž sloupec 1 obsahuje údaje o testované úloze(název ulohy, počet proměnných a počet termů vstupní funkce). Úlohy můžeme rozdělit na splnitelné a nesplnitelné. Splnitelné úlohy jsou v tabulce zvýrazněny šedým pozadím buňky. Sloupce 2,3,4 náleží SAT solveru EBDDRES, přičemž každý obsahuje výsledky měření pro jednu z heuristik, které jsou v tomto řešiči použity. Konkrétně se jedná o heuristiku eliminace proměnných(VE) v druhém sloupci, dfs řazení vstupních proměnných ve třetím sloupci a řazení podle četnosti výskytu proměnné ve čtvrtém sloupci. Každý z těchto sloupců se dále skládá ze dvou podsloupců z nichž první je označený *sec* a obsahuje dobu řešení dané úlohy v sekundách a druhý sloupec označený jako *Size* obsahuje maximální počet uzlů, se kterým byl solver nucen pracovat. Dále jsou zde sloupce 5,6 a 7, které obsahují podobně jako sloupce 2,3 a 4 výsledky měření pro jednotlivé heuristiky mnou implementovaného SAT solveru. Oproti SAT solveru EBDDRES obsahují kromě podsloupců *sec* a *Size* také podsloupec Vektory, který udává počet vektorů vygenerovaný implementovaným nástrojem, pro který je daná funkce splněná. U nesplnitelných funkcí je tento sloupec vyplněn znakem -. Z heuristik, které implementovaný řešič může použít byly vybrány heuristiky Random pivot(sloupec 6) a Sym-sift(sloupec 7). Ve sloupci 5 jsou zobrazeny výsledky měření pokud nepoužíváme žádnou metodu rovnání vstupních proměnných. Všechny testování implementovaného SAT solveru používají také mimo dynamického reorderingu heuristiku řazení vstupních termů. Buňky v tabulce které místo hodnoty obsahují znak – nebylo možno daným solverem naměřit.

Z naměřených výsledků, je patrné, že SAT solver EBDDRES dosahuje pro úlohy, které jsme s ním schopni vyřešit lepších výsledků než mnou implementovaný SAT solver, nicméně všimněme si, že téměř pro všechny heuristiky pracuje s mnohem větším počtem uzlů, ale přitom i s relativně velkým množstvím uzlů dokáže pracovat velmi rychle. Mnou implementovaný řešič naopak pracuje ve většině případů s mnohem menším počtem uzlů, ale aby dosáhl tohoto menšího počtu uzlů musí strávit více času dynamickým reorderingem, což ho v některých případech značně brzdí. Dále si všimněme, že pokud není použito žádné heuristiky dynamického řazení, je doba řešení menší i když jsme nuceni zpracovávat větší počet uzlů. Je tedy zřejmé, že pro funkce s malým počtem proměnných(< 40) se nevyplatí používat metody dynamického reorderingu a je vhodné využít statického řazení proměnných. Této vlastnosti také využívá SAT solver EBDDRES, který pracuje výhradně s metodami statického řazení proměnných což mu pro takto malé úlohy poskytuje značnou výhodu(rychlost řešení), ale pro větší funkce může často

docházet k "chybám" – jestliže počet zpracovávaných uzlů(velikost alokované paměti) stoupne nad určitou mez dojde k uzavření programu s chybou díky čemuž se také nepodařilo naměřit poměrně velké množství úloh, které můj řešič vyřešil. Dynamickým řazením proměnných totiž může i v průběhu vytváření ROBDD vylepšovat pořadí proměnných a snižovat jeho velikost, zatímco při použití statických heuristik řazení je velikost ROBDD dána pouze zkoumanou funkcí a kvalitou pořadí proměnných.

Z tabulky č.11 je také vidět, že počet vektorů, pro které je funkce splněná ve většině případů nezávisí na použité metodě řazení proměnných. Z osmi splnitelných úloh, které jsou zaneseny v tabulce č.11 se použití různých heuristik projevilo na počtu výstupních termů, pro které je funkce splněná pouze u úlohy uf75-095, kde pro heuristiku Sym-sift je počet vektorů, pro které je tato funkce splněná 16 narozdíl od ostatních heuristik(bez řazení, random pivot), které shodně nalezli 20 vektorů, pro které je funkce splněná. Z výstupů, které jsou přiloženy na CD je vidět, že vektory získané s použitím heuristiky Sym-sift obsahují větší počet tzv. don´t care stavů.

Kromě zde prezentovaných výsledků bylo provedeno množství dalších měření, na splnitelných i nesplnitelných úlohách s různým poměrem počtu termů ku počtu proměnných od 1 (20/20, 30/30, 40/40, 50/50, 60/60, …) do 250(5000/50, 5000/60, …, 10000/40). Výsledky těchto měření jsou identické s výsledky prezentovanými na předchozích úlohách a jsou obsahem přiloženého CD.

Testování bylo provedeno na sestavě : CPU 1,8GHz Core 2 Duo(Merom), 1GB RAM, HDD Seagate 120GB, Operační systém Windows XP profesional.

6. Závěr a zhodnocení implementovaného SAT solveru

V rámci této práce vznikl SAT solver, jehož implementace je založena na knihovně CUDD. Tento řešič splnitelnosti booleovské funkce je ve srovnání s ostatními řešiči, které pracují na odlišných principech mnohem pomalejší, ale umožňuje na rozdíl od ostatních nejen odpovědět na otázku jeli funkce splnitelná či nikoli, ale může navíc vygenerovat úplné i neúplné vektory, pro které je funkce splněná, nalézt nejkratší cestu(vektor, pro který je funkce splněná a který obsahuje nejvíce neurčených stavů), nebo zjistit o vstupní funkci mnoho dalších statistických i praktických informací.

Jak již bylo dříve popsáno, rychlost řešiče je oproti ostatním řešičům, které jsou založeny na odlišných principech mnohem menší, ale ve srovnání s řešičem EBDDRES(kapitola 5.), který k řešení využívá taktéž binárních rozhodovacích diagramů, bylo již dosaženo mnohem lepších výsledků. Vzhledem k tomu, že můj řešič používá pouze heuristiky dynamického řazení, zatímco řešič EBDDRES využívá pouze heuristiky statického řazení bylo dosaženo poměrně dobrých výsledků a zároveň se ukázala také další možnost pro případné budoucí vylepšení. Toto vylepšení by mohlo spočívat v tom, že bychom knihovnu funkcí CUDD upravili tak, aby podporovala statické metody řazení a popř. do ní doplnily několik heuristik statického řazení proměnných. S použitím statického a dynamického řazení proměnných by poté implementovaný řešič pravděpodobně dosahoval mnohem lepších výsledků a byl by použitelný pro širší spektrum úloh.

Závěrem bych ještě dodal, že vzhledem k tomu, že práce obsahuje množství funkcí z balíku CUDD(příloha C.3), který má opravdu rozsáhlou dokumentaci a mnou implementované funkce jsou také poměrně obsáhle popsány v uživatelské příručce(příloha C), neměl by být problém se s řešičem v krátké době seznámit a doimplementovat případné další vylepšení.

7. Seznam použité literatury

[1] R. E. Bryant, “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, Vol. 24, No. 3 (September, 1992), pp. 293-318. Preprint version published as CMU Technical Report CMU-CS-92-160

[2] Rolf Drechsler, Detlef Sieling: Binary decision diagrams in theory and practice. STTT 3(2): 112-136 (2001)

[3] Jan Christiansen: A purely functional implementation of ROBDDs in Haskell. Diploma thesis,6.March.2006

[4] http://www.qbflib.org/qdimacs.html

[5] Fadi A. Aloul, Igor L. Markov, Karem A. Sakallah: FORCE: a fast and easy-to-implement variable-ordering heuristic. ACM Great Lakes Symposium on VLSI 2003: 116-119

[6] Ondřej Kološ : Port balíku CUDD pod Windows, Bakalářská práce, ČVUT v Praze, FEL, 2006

[7] http://vlsi.colorado.edu/~fabio/CUDD/

[8] Přemysl Rucký: Převod víceúrovňové logické sítě na dvouúrovňovou pomocí BDD , ČVUT v Praze, FEL, leden 2007

[9] Toni Jussila, Carsten Sinz, Armin Biere: Extended Resolution Proofs for Symbolic SAT Solving with Quantification. SAT 2006: 54-60 [DBLP:conf/sat/JussilaSB06]

[10] Carsten Sinz, Armin Biere: Extended Resolution Proofs for Conjoining BDDs. CSR 2006: 600-611 [DBLP:conf/csr/SinzB06]

[11] Jinbo Huang, Adnan Darwiche: Toward Good Elimination Orders for Symbolic SAT Solving. ICTAI 2004: 566-573 [DBLP:conf/ictai/HuangD04]

[12] Jan Bílek: Minimalizace neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů, ČVUT v Praze, FEL, leden 2007

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik: Chaff: Engineering an Efficient SAT Solver. DAC 2001: 530-535 [DBLP:conf/dac/MoskewiczMZZM01]

[14] http://service.felk.cvut.cz/vlsi/prj/BOOM/pla_c.html

[15] Randal E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8): 677-691 (1986) [DBLP:journals/tc/Bryant86]

[16] http://fmv.jku.at/ebddres/

8. Seznam zkratek

ADD Algebraic Decision Diagram
algebraický rozhodovací diagram

BDD Binary Decision Diagram
binární rozhodovací diagram

COF Center of gravity
Střed gravitační síly

CNF Conjunctive Normal Form
konjunktivní normálová forma

CUDD Colorado University Decision Diagram (package)
Balík z Coloradské univerzity týkající se rozhodovacích diagramů

DD Decision diagram
Rozhodovací diagram

DNF Disjunctive Normal Form
disjunktivní normálová forma

FMV Institute for formal models and verification
Institut formálního modelování a ověřování

INF If-Then-Else Normal Form
If-Then-Else normálová forma

ITE If-Then-Else (operator)
If-then-else (operátor)

NF Normal Form
normální forma

OBDD Ordered Binary Decision Diagram
uspořádaný binární rozhodovací diagram

RBDD Reduced Binary Decision Diagram
redukovaný binární rozhodovací diagram

ROBDD Reduced Ordered Binary Decision Diagram
redukovaný uspořádaný binární rozhodovací diagram

SAT SATisfiability
Splnitelnost

VE Variable elimination
Eliminace proměnných

VSIDS Variable state independent decaying sum
Statická heuristika vytvoření pořadí proměnných využívající počet výskytu proměnné

ZDD Zero-suppressed binary Decision Diagram
binární rozhodovací diagram s potlačenou nulou

9. Přílohy

A. Seznámení s knihovnou CUDD

Jak již bylo řečeno v první kapitole, knihovna CUDD (Colorado university decision diagram) byla vytvořena na Coloradské univerzitě týmem pod vedením Fabia Someziho. Práce na této knihovně započaly již v roce 1996 a od té doby bylo dosaženo značné efektivity funkcí a také komplexnosti této knihovny. V nynější době je na internetových stránkách projektu k dispozici verze 2.4.1, která jen v základním balíku obsahuje přes 400 funkcí sloužících k manipulaci s BDD (Binary Decision Diagram), ADD (Algebraic Decision Diagram), ZDD (Zero-suppressed decision diagrams) a heuristik soužících k nalezení vhodného pořadí proměnných.

Projekt je realizován v programovacím prostředí jazyka C, nicméně existuje již také C++ rozhraní, které je ale zatím velmi jednoduché.

Tuto knihovnu lze využít jako tzv. "černou skříňku", kdy program aplikace pracuje pouze s funkcemi, které jsou mu přístupné a taktéž se nemusí zabývat detaily uspořádání/přeuspořádání proměnných, které se děje na pozadí. Dále lze tuto knihovnu využít jako tzv. průhlednou skříňku (clear box) což je vhodné při vytváření složitějších aplikací a znamená to v podstatě vytvoření vlastních funkcí pomocí stávajících a začlenění nových funkcí do knihovny. Poslední možnost použití je pomocí rozhraní C++ popř. Pearl umožňují osvobodit programátora od správy paměti.

Knihovna CUDD je detailněji popsána v dokumentech [6,7], ze kterých jsem při psaní této kapitoly vycházel.

A.1 Datové struktury a práce s nimi

Jak již bylo dříve řečeno práce s ROBDD se skládá z množství jednoduchých operací, které nad daným ROBDD provádíme. Takovýto ROBDD může v praxi dosahovat velikosti až několik miliónů uzlů a proto je z hlediska efektivního využití paměti nutné zvolit si vhodnou reprezentaci uzlu a také s těmito uzly efektivně pracovat (reference/dereference, odstranění dočasných uzlů/death uzlů atd.). V knihovně CUDD se k reprezentaci uzlu používá struktura DdNode a k práci s uzly DdManager tyto si nyní popíšeme.

A.1.1Struktura uzlu DdNode

V knihovně CUDD se všechny rozhodovací diagramy (BDD,ADD,ZDD) skládají z uzlů které jsou realizovány strukturou DdNode, která vypadá následovně :

```
struct DdNode {
   DdHalfWord index;
   DdHalfWord ref;
   DdNode *next;
   union {
     CUDD_VALUE_TYPE value;
     DdChildren kids;
   } type;
};
```

Význam jednotlivých proměnných je následující:

Index – obsahuje jméno proměnné, které je zároveň označením uzlu. Index je neměnný atribut, který odráží pořadí vytvoření dané proměnné. První proměnná je tedy označena indexem 0 a poslední vytvořená proměnná má nejvyšší index. Index je typu DdHalfWord a v 32-bitovém prostředí je typu unsigned short integer, přičemž nejvyšší hodnota indexu je rezervována pro konstantní uzel. V 64-bitovém prostředí je typu unsigned integer, přičemž nejvyšší hodnota je taktéž rezervována pro konstantní uzel.

Ref – uchovává informaci o počtu referencí na daný uzel. Tuto informaci používá garbege collector – uzly které mají ref count roven nule lze odstranit.

Next - je proměnná typu DdNode a je ukazatelem na další uzel v hashovaní tabulce (unique table).

Poslední položkou struktury DdNode je unie, která v případě že se jedná o neterminální uzel obsahuje ukazatele na jeho potomky a v případě že jde o terminál, obsahuje hodnotu tohoto terminálu.

A.1.2 Správa paměti (DdManager)

Všechny uzly v BDDs, ADDs a ZDDs jsou uloženy v hashovacích tabulkách označovaných *unique tables.* Jak už ukazuje samotné jméno “unique table“ je účelem této tabulky zaručit že každý uzel v ní uložený je jedinečný tzn. není v ní žádný další uzel se stejným označením stejné

proměnné a se stejnými potomky. Díky této jedinečnosti jsou DDs (Decision Diagrams – rozhodovací diagramy) kanonické. DdManager tedy obsahuje unique table a přes 130 dalších proměnných, které lze využít při práci s rozhodovacími diagramy. Před započetím práce musíme DdManager inicializovat voláním funkce Cudd_Init s příslušnými parametry a po skončení práce DdManager zrušíme voláním funkce Cudd_Quit kde jako parametr použijeme jméno manageru.

S výjimkou několika statistických proměnných obsahuje balík CUDD pouze několik globálních proměnných a většina proměnných je udržována v manageru, je tudíž možné abychom v jedné aplikaci měli aktivních hned několik managerů přičemž ukazatel na manager rozhoduje nad jakými daty bude funkce pracovat.

Při práci s rozhodovacími diagramy pomocí knihovny CUDD je také důležité aktualizovat informaci o počtu referencí (proměnná ref – struktura DdNode) na jednotlivé uzly, kterou využívá garbage collector k určení uzlů které jsou pro další využití nepotřebné a tudíž je může odstranit. Zvýšení počtu ukazatelů na daný uzel se provádí voláním funkce Cudd_Ref a snížení se provádí pomocí funkce Cudd_RecursiveDeref. Špatná práce s referencováním a dereferencováním uzlu má za následek plýtvání s pamětí a u větších úloh vede k rychlému zaplnění paměti.

A.2 Příklad vytvoření ROBDD

Práce s knihovnou CUDD je po obeznámení s dokumentací a osvojení používaných postupů poměrně snadná a tudíž se nebudu zabývat rozborem jednotlivých funkcí, kterých knihovna CUDD obsahuje velkém množství, ale na následujícím výpisu zdrojového kódu si ukážeme použití funkcí nutných pro vytvoření ROBDD.

```
#include <stdio.h>
#include "include/cudd.h"
#include "include/util.h"

int main() {
  DdManager *manager;
  DdNode *f1,*f2,*f,*tmp,*var;
  int i;
  double k;

manager=Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
  //inicializace manažeru - DdManager
  f1=Cudd_ReadOne(manager);  //vytvoření bdd x0*x1*x2
  Cudd_Ref(f1);
```

```
    for(i=2;i>=0;i--){ // cyklus přidávání uzlů do BDD
      var=Cudd_bddIthVar(manager,i);
      tmp=Cudd_bddAnd(manager, var,f1);
      Cudd_Ref(tmp);
      Cudd_RecursiveDeref(manager,f1);
      f1=tmp;
    }
    f2=Cudd_ReadOne(manager); //vytvoření bdd x0'*x1'*x2'
    Cudd_Ref(f2);
    for(i=2;i>=0;i--){ // cyklus přidávání uzlů do BDD
    var=Cudd_bddIthVar(manager,i);
    tmp=Cudd_bddAnd(manager, Cudd_Not(var),f2);
    Cudd_Ref(tmp);
    Cudd_RecursiveDeref(manager,f2);
    f2=tmp;
  }
  f=Cudd_bddOr(manager,f1,f2); // vytvoření celého BDD
  Cudd_Ref(f);
  Cudd_RecursiveDeref(manager,f1);
  Cudd_RecursiveDeref(manager,f2);
  Cudd_Quit(manager);
  return 0;
  }
```

Předchozí kód slouží k vytvoření ROBDD pro funkci $f = (x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$ všimněme si, že jak již bylo dříve řečeno je nutné na začátku celého programu inicializovat manažer voláním funkce Cudd_Init s příslušnými parametry. Dále se v cyklu for vytváří pro každý term ROBDD, přičemž ROBDD se vytvářejí odspodu což znamená že nejprve musíme pomocí funkce Cudd_ReadOne vytvořit (pokud už tento uzel existuje, funkce vrací ukazatel na něj) konstantní uzel 1 a poté v cyklu přidáváme další uzly. To realizujeme tak, že nejprve vytvoříme uzel (proměnnou) v manažeru pomocí funkce Cudd_bddIthVar a tuto posléze přidáme pomocí funkce Cudd_bddAnd do ROBDD. K negování proměnné je použita funkce Cudd_Not, která má jako parametr jméno proměnné, kterou chceme negovat. Mezivýsledky těchto operací jsou průběžně ukládány do pomocné proměnné tmp a je důležité provádět referencování pomocí funkce Cudd_Ref(při vzniku) a dereferencování (když už není obsah proměnné tmp potřeba) pomocí funkce Cudd_RecursiveDeref. Správné referencování a dereferencování je velmi důležité pro správnou funkci garbage collectoru – odstraňování dat, která nejsou dále potřeba (uvolnění paměti). Na závěr jsou ROBDD vytvořené pro jednotlivé termy spojeny do jednoho ROBDD pomocí funkce Cudd_bddOr. Na konci programu je žádoucí ukončit používání manažeru pomocí funkce Cudd_Quit.

A.3 Metody reorderingu

Jak již bylo řečeno dříve, pořadí proměnných výrazně ovlivňuje velikost BDD. Knihovna CUDD disponuje několika heuristikami, které právě k nalezení vhodného pořadí slouží a v této kapitole si o těchto heuristikách a metodách jejich spouštění něco řekneme. Blíže v dokumentech [6,7,8,9].

Řazení proměnných může být vyvoláno buď přímo voláním funkce Cudd_ReduceHeap, nebo je automaticky spouštěno knihovnou CUDD, pokud počet uzlů dosáhne určité velikosti. Aby mohlo být zahájeno řazení proměnných automaticky, musí být tato možnost povolena voláním funkce Cudd_AutodynEnable (deaktivuje se voláním funkce Cudd_AutodynDisable). Jako parametry těchto funkcí se používá název heuristiky, která se má k nalezení pořadí použít a případně další parametry, které vymezují pole působnosti heuristiky. Knihovna CUDD podporuje následující metody reorderingu :

CUDD_REORDER_SAME – jestliže je zadána jako parametr funkci Cudd_AutodynEnable, ponechává metodu pro automatické řazení proměnných beze změn. Jestliže je použita jako parametr volání funkce Cudd_ReduceHeap, spustí se metoda rovnání proměnných používaná pro automatické rovnání proměnných

CUDD_REORDER_RANDOM – dvojice proměnných jsou náhodně vybrány a prohozeny v celkovém pořadí. Výměna je provedena pomocí sady prohození sousedních proměnných přičemž je zachováno nejlepší pořadí proměnných. Počet dvojic, které se budou přehazovat je stejný jako počet proměnných v diagramu.

CUDD_REORDER_RANDOM_PIVOT – metoda stejná jako předchozí (CUDD_REORDER_RANDOM), ale dvojice proměnných je vybrána tak, že první proměnná je nad proměnnou s největším počtem uzlů a druhá je zvolena z proměnných pod proměnnou s největším počtem uzlů. V případě, že maximální počet uzlů má více proměnných stejný, je použita jako pivot ta proměnná, která je blíže kořenu.

CUDD_REORDER_SIFT – tato metoda používá "prosévací" algoritmus implementovaný R. Rudellem (Rudell's sifting algorithm). Tento algoritmus v podstatě hledá nejlepší pozici pro každou proměnnou tak, že ji vezme a posunuje v pořadí směrem nahoru popř. dolů až projde všechny možné pozice a postupně si pro jednotlivé pozice zaznamenává počet uzlů

BDD. Nakonec je vybrána pozice s nejmenším počtem uzlů a tato je přiřazena dané proměnné. Tento postup se opakuje pro všechny proměnné v BDD. V praxi se jako parametry této metody používá například maximální počet proměnných, které se budou prosévat (tento limit může být přečten funkcí Cudd_ReadSiftMaxVar a nastaven funkcí CuddSetSiftMaxVar), popř. můžeme nastavit maximální velikost, které může BDD dosáhnout během prosévání. Pokud je tato velikost překročena, je přesouvání proměnné přerušeno před dosažením konce pořadí. Maximální velikost lze přečíst voláním funkce Cudd_ReadMaxGrowth a nastavit voláním funkce Cudd_SetMaxGrowth.

CUDD_REORDER_SIFT_CONVERGE – verze algoritmu CUDD_REORDER_SIFT. Předchozí algoritmus (CUDD_REORDER_SIFT) je spouštěn tak dlouho dokud metoda přináší nějaké zlepšení.

CUDD_REORDER_SYMM_SIFT – tato metoda je další variantou prosévacího algoritmu (siftingu). Na rozdíl od standartního prosévacího algoritmu je rozšířena o testování symetrie mezi sousedními proměnnými, které vznikly během prosévání. Sousední proměnné jsou symetrické, pokud jejich prohození nemá vliv na velikost BDD.Takovéto symetrické proměnné jsou poté sjednoceny do skupin a prosévací algoritmus dále pracuje se skupinami symetrických proměnných a ne s jednotlivými proměnnými.

CUDD_REORDER_SYMM_SIFT_CONV – verze algoritmu Sym-sift. Předchozí algoritmus (CUDD_REORDER_SYMM_SIFT) je spouštěn tak dlouho dokud metoda přináší nějaké zlepšení.

CUDD_REORDER_GROUP_SIFT – tato metoda je rozšířením algoritmu CUDD_REORDER_SYMM_SIFT. Na rozdíl od symetrického prosévání není slučování do skupin omezeno pouze na symetrické proměnné.

CUDD_REORDER_GROUP_SIFT_CONV – verze algoritmu Group-sift. Předchozí algoritmus (CUDD_REORDER_GROUP_SIFT) je spouštěn tak dlouho dokud metoda přináší nějaké zlepšení.

CUDD_REORDER_WINDOW2 – algoritmus založený na permutaci oken, který popisuje Fujita a Ishiura. Velikost okna je 2.

CUDD_REORDER_WINDOW3 – algoritmus stejný jako CUDD_REORDER_WINDOW2, ale velikost okna je 3.

CUDD_REORDER_WINDOW4 – algoritmus stejný jako CUDD_REORDER_WINDOW2, ale velikost okna je 4.

CUDD_REORDER_WINDOW2_CONV – verze algoritmu CUDD_REORDER_WINDOW2. Algoritmus je prováděn dokud je dosahováno nějakého zlepšení.

CUDD_REORDER_WINDOW3_CONV – verze algoritmu CUDD_REORDER_WINDOW3. Algoritmus je prováděn dokud je dosahováno nějakého zlepšení.

CUDD_REORDER_WINDOW4_CONV – verze algoritmu CUDD_REORDER_WINDOW4. Algoritmus je prováděn dokud je dosahováno nějakého zlepšení.

CUDD_REORDER_ANNEALING – tato metoda používá k nalezení optimálního pořadí iterativní metodu simulovaného ochlazování. Tato metoda je potenciálně velmi pomalá.

CUDD_REORDER_GENETIC – tato metoda využívá k nalezení optimálního pořadí genetických algoritmů a je inspirována prací R. Drechslera. Tato metoda je taktéž potenciálně velmi pomalá.

Další možností je přerovnání proměnných do uživatelem určeného pořadí. Toto lze provést voláním funkce Cudd_ShuffleHeap, která má jako parametr pole proměnných, které jsou seřazeny do požadovaného pořadí.

B. Ovládání a výstup SAT solveru EBDDRES

V následujícím textu si popíšeme ovládání SAT solveru EBDDRES a také si ukážeme jeho výstup.

B.1 Ovládání sat solveru

Sat solver EBDDRES se stejně jako ostatní ovládá pomocí parametrů zadaných při spuštění do příkazové řádky. V následující části jsou vypsány parametry, které lze do příkazové řádky zadávat spolu se stručným popisem :

Do příkazové řádky zadáváme : ebddres [<parametry> ...]

Přičemž parametry mohou být následující :

```
-h              vypíše na obrazovku seznam parametrů(tento) -
                nápověda
--dump          vypíše(do souboru .log) pomocnou informaci o
                řádcích a uzlech
--config        vypíše čas, kdy byla provedena kompilace
                konfiguračního souboru
-t <stopa>      vypíše pevnou stopu důkazu
-T <stopa>      vypíše rozšířenou stopu důkazu
--original-order   použije původní pořadí
                   proměnných(nastaveno jako výchozí)
--reverse-order  použije obrácené pořadí proměnných
--dfs-order     použije DFS pořadí proměnných
--sum-order     použije počet výskytů proměnné k určení pořadí
--no-force      zákaz optimalizace pořadí proměnných
                algoritmem FORCE
--no-core       zákaz vyjmutí jádra a vše stopuje
--schedule      pokus o optimalizaci rozvržení termů
--tree          stromový tvar místo přímého výpočtu
--ve            eliminace proměnných
--vetree        eliminace proměnných, stromový výpočet
--vetrial       odstraňuje stejné proměnné
-o <výstup>     nastavuje výstupní soubor
<vstup>         vstupní soubor ve formátu DIMACS
-w <pořadí-výstup-soubor>
```

B.2 Výstup SAT solveru

Sat solver EBDDRES může mít jako výstup tři možné soubory. Prvním z nich je soubor typu .out, který obsahuje rozhodnutí o splnitelnosti/nesplnitelnosti booleovské funkce a v případě že je funkce splnitelná obsahuje také vektory pro které je splnitelná. Tyto vektory jsou zapsány

podobně jako vstupní funkce pomocí kladného popř. záporného čísla proměnné. Takovýto výstupní soubor tedy může vypadat například následovně :

```
s SATISFIABLE
v 1 2 0
```

Druhým výstupem řešiče je soubor typu .log, který obsahuje informace o průběhu řešení a o výsledku (uzly, splnitelná/nesplnitelná, vektory, použité řazení, doba řešení, …). Tento soubor může vypadat například následovně :

```
c parsed header 'p cnf 20 91'
c original variable order
c average cut width 93.100
c average cut width 89.300
c 2 optimizations of variable order
c
c  seconds   MB  progress  ands     exs      nodes       steps       lines
c
c     0.0    0.1    5.00%     20    1/20        513         728         300
c     0.0    0.1  10.00%     25    2/20        808        1442         511
c     0.0    0.1  10.00%     28    2/20       1025        2167         871
c     0.0    0.1  10.00%     29    2/20       1137        2466        1023
c     0.0    0.1  15.00%     34    3/20       1817        4888        2047
c     0.0    0.2  15.00%     40    3/20       2049        5695        2448
c     0.0    0.2  20.00%     55    4/20       2552        7987        3450
c     0.0    0.2  45.00%     83    9/20       2958        9651        4095
c     0.0    0.3  95.00%    109   19/20       2982        9896        4180
c
c solved instance in 0.00 seconds
c
s SATISFIABLE
v -1 2 3 -4 5 6 -7 8 9 10 -11 -12 13 14 15 -16 17 -18 19 20 0
c
c 2980 nodes (0.35 average number collisions)
c 4180 lines (1112 redundant, 0.33 average number collisions)
c 704 orlines (141 redundant, 0.33 average number collisions)
c 109 ands (19 trivial, 19 redundant)
c 19 exs (5 trivial, 4 redundant)
c 9896 steps (5012 trivial, 6265 redundant)
c 19% cache hit ratio
c
c 0.02 seconds (0.00 sat + 0.00 core + 0.00 trace)
c 0.3 MB
```

Třetí a poslední možností výstupu je soubor obsahující rozšířený důkaz řešení. Tento výstup je ale pro nás nepodstatný a tudíž se mu nebudu v této práci věnovat.

C. Popis funkcí a práce s implementovaným SAT solverem

C.1 Lexikální a syntaktická analýza

V kapitole 3.1 byl popsán formát vstupu DIMACS, který využívá mnou implementovaný řešič. Ke správné funkci řešiče je nutné abychom správně interpretovali vstupní hodnoty. K tomu je použit lexikální analyzátor, který jsem implementoval jako samostatnou třídu LexAn. Tato třída obsahuje několik funkcí pro práci se vstupem, přičemž výstupem těchto funkcí jsou lexikální elementy, které se dále zpracovávají syntaktickým analyzátorem, popř. chybová hlášení, které zpravidla ukončují program. Jednotlivé lexikální elementy jsou popsány v předchozí kapitole 3.1 a nebudu se jimi tedy znovu zabývat. V následující části si trochu více přiblížíme třídu LexAn a funkce, které obsahuje :

Error – tato funkce je volána v případě že v průběhu lexikální analýzy došlo k chybě např. nezdařilo se inicializovat vstupní soubor, lexikální elementy obsahují nepovolený znak atd. . Vstupním parametrem funkce je chybové hlášení (řetězec). Funkce vypíše chybovou hlášku a ukončí program.

InitVstup – tato funkce inicializuje vstupní soubor tzn. jako vstupní parametr převezme jméno vstupního souboru a pokusí se ho otevřít. Pokud otevření proběhne bez problémů je ukazatel na soubor uložen do globální proměnné *Vstupf* v opačném případě je volána funkce *Error*, která vypíše chybové hlášení a ukončí program.

CtiZnak – tato funkce nemá žádný vstupní parametr, ale pracuje s globální proměnnou *Vstupf* (ukazatel na otevřený soubor), přičemž přečte znak a zkontroluje jestli se nejedná o znak konce řádku, popř. konce souboru. V případě že se jedná o znak konce řádku, nebo o znak konce souboru vrací funkce hodnotu *EOF*, jinak vrací aktuálně přečtený znak ze souboru.

CtiVstup – tato funkce používá jako vstup návratovou hodnotu funkce *CtiZnak* a "rozhoduje" jestli je vstupním znakem číslice, písmeno popř. speciální znak a podle vstupního znaku nastavuje globální proměnnou *Vstup*.

readIK – tato funkce je používána k přečtení celého identifikátoru popř. klíčového slova. Na základě globální proměnné *Vstup*, kterou nastavuje funkce *CtiVstup* se rozhoduje jestli vstupem může být identifikátor nebo klíčové slovo. Pokud funkce

nenarazí na znak, který není nepřípustný pokračuje v dalším načítáním dokud není načten celý identifikátor/klíčové slovo. Načtený identifikátor/klíčové slovo se ukládá do globální proměnné *id*.

readNUMBER – tato funkce se používá k přečtení čísla. Pracuje s globální proměnnou Znak, která je podle potřeby aktualizována voláním funkce *CtiVstup*. V případě že číslo neobsahuje žádné nepovolené znaky(např. písmena, speciální znaky…) je celé přečteno a uloženo do globální proměnné *Cislo*.

KeyWord – tato funkce nemá žádný vstupní parametr, ale pracuje s globální proměnnou *id*, která je výstupem funkce *readIK*. Tato proměnná obsahuje vstupní řetězec, který může být identifikátorem, popř. klíčovým slovem. Funkce *KeyWord* tedy provede porovnání řetězce uloženého v proměnné *id* s tabulkou klíčových slov a rozhodne, jestli se jedná o identifikátor, nebo klíčové slovo a podle tohoto zjištění vrací jako svou návratovou hodnotu lexikální symbol *IDENT* (pokud vstupem není klíčové slovo) nebo klíčové slovo.

Element – tato funkce je klíčová pro práci lexikálního analyzátoru, neboť v sobě sdružuje volání "všech" předchozích funkcí a jejím voláním přeskakujeme mezi jednotlivými lexikálními elementy. Výstup této funkce je aktuální lexikální element a je uložen v globální proměnné *Cislo* (pokud je elementem číslo), *id*(pokud je elementem identifikátor/klíčové slovo). O tom v které proměnné je uložen aktuální lexikální element rozhoduje globální proměnná *symb*, která je nastavena funkcí Element na *NUMB* (jedná se o číslo, platný je element v proměnné *Cislo*), *IDENT* (je-li platným elementem identifikátor v proměnné *id*), jedno z klíčových slov (je uloženo v proměnné id např. : *kwCNF, kwP, kwC*) .

Při popisu funkcí lexikálního analyzátoru bylo použito termínů identifikátor, klíčové slovo, číslo, i když tyto nebyly dříve definovány. Klíčovým slovem se v našem případě rozumí např. c, p, cnf jejichž význam je blíže popsán v kapitole 3.1. Dále je zde identifikátor, který může v našem případě reprezentovat jakýkoli řetězec a používá se k realizaci komentářů. Nakonec je zde element číslo jehož použití je taktéž popsáno v kapitole 3.1.

Nyní si řekneme něco o syntaktickém analyzátoru. Syntaktický analyzátor jsem vzhledem k poměrně jednoduché syntaxi vstupu neimplementoval jako samostatnou třídu, ale je přímo začleněn do řešiče a využívá tří funkcí, které jsou volány hned po úvodní inicializaci. Tyto funkce si nyní ve zkratce popíšeme :

Start – tato funkce je volána na začátku práce se vstupem. Jako vstup má instanci třídy *LexAn* a rozhoduje, jestli je na začátku vstupu komentář (uvozeno klíčovým slovem *c*) nebo jestli jde o "deklaraci" funkce (uvozeno klíčovým slovem *p*). Na základě předchozího zjištění zavolá funkci *koment* (jde-li o komentář) popř. funkce *declCNF* (jde-li o deklaraci funkce).

koment – tato funkce provede přečtení a výpis komentářů na standardní výstup. Jako vstupní parametr má taktéž instanci třídy LexAn.

declCNF – tato funkce kontroluje správnou syntaxi "deklarace" vstupní funkce a ukládá počet proměnných funkce do globální proměnné *inputN* a počet termů funkce do globální proměnné *termN*. Jako vstupní parametr má taktéž instanci třídy *LexAn*.

Po úspěšném provedení těchto tří funkcí už mohou ve vstupním souboru následovat pouze platná data zadaná v číselné formě. Tyto jsou poté kontrolovány(je testováno jedná-li se o číslo, jeli správný počet termů, počet proměnných…) přímo při vytváření ROBDD.

C.2 Popis mnou implementovaných funkcí

V této kapitole budou popsány funkce , které používá můj řešič k práci s BDD a vstupními termy. Vzhledem k tomu, že vlastní řešič využívá funkce knihovny CUDD, nebylo nutné implementovat mnoho složitých funkcí. Na druhou stranu jsem byl při vývoji poněkud limitován možnostmi této knihovny. Popis implementovaných funkcí:

InitVystup – tato funkce slouží k inicializaci výstupního souboru typu .pla. Vstupem je jméno výstupního souboru. Funkce se pokusí otevřít soubor se jménem které získá jako parametr a v případě že uspěje, je ukazatel na soubor uložen v globální proměnné vystup. Pokud se soubor nepodaří otevřít, je vypsáno chybové hlášení a program je ukončen.

InitDebug – tato funkce slouží k inicializaci výstupního souboru debug přičemž pracuje stejně jako funkce InitVystup. Rozdíl je pouze v tom, že v případě otevření souboru, je ukazatel na něj uložen do globální proměnné debug.

InitGraph – tato funkce slouží taktéž k inicializace výstupního souboru graph a vykonává operaci stejnou jako předchozí funkce InitVystup a InitDebug pouze s tím rozdílem že pracuje s globální proměnnou graph.

InitBDD – tato funkce provádí "inicializaci BDD". Nemá žádné vstupní ani výstupní hodnoty. Pracuje pouze s několika globálními proměnnými(DdManager, DdNode) a pomocí volání několika funkcí z balíku CUDD(Cudd_Init, Cudd_ReadOne, Cudd_Ref) inicializuje BDD – inicializuje manažer, vytvoří ukazatel na uzel f(k tomuto uzlu budeme postupně přidávat dílčí BDD) a na uzel f1(tento se používá ke konstrukci dílčích BDD) a nakonec vytvoří ukazatele na jedničkový a nulový terminál.

partition, quicksort – tyto dvě funkce realizují řazení quick sort. Funkce partition má jako vstup pole hodnot typu int a horní a dolní hranici intervalu se kterým bude pracovat. Vrací hodnotu, která odpovídá polovině původního intervalu. Funkce quicksort má stejný vstup jako funkce partition a pomocí volání funkce partition a rekurzivního volání sama sebe provádí rovnání vstupního pole hodnot od největší po nejmenší.

varCount – tato funkce zjišťuje počet výskytů proměnných ve vstupní funkci. Vstupem je počet proměnných booleovské funkce. Po volání funkce je vytvořeno pole čítačů proměnných a inicializován lexikální analyzátor. Ze souboru, který obsahuje vstupní funkci jsou přečteny jednotlivé literály a proměnné odpovídající literálu je inkrementován její čítač v poli čítačů proměnných. Po ukončení funkce tedy máme pole obsahující počty výskytů jednotlivých proměnných.

termOrder – tato funkce načte do pole termy vstupní funkce a provede seřazení termů podle jejich vah od největší váhy po nejmenší. Funkce má jako vstup počet

proměnných a počet termů. Na začátku vytvoří pole termů a pole vah termů. Do pole termů postupně načítá termy ze vstupního souboru a pro každý term také počítá jeho váhu(váha je dána součtem výskytů literálů obsažených v termu) a ukládá ji do pole vah. Po načtení všech termů je pole termů seřazeno podle pole vah od nejvyšší váhy po nejnižší(je použito řazení quick sort).

dumpTermDesc – tato funkce provádí výpis pole termů do souboru temp. Vstupem této funkce je počet proměnných a počet termů vstupní booleovské funkce. Výpis termů je proveden od termu s nejvyšší váhou po term s nejnižší váhou.

dumpTermAsc – tato funkce provádí výpis pole termů do souboru temp. Vstupem této funkce je počet proměnných a počet termů vstupní booleovské funkce. Výpis termů je proveden od termu s nejnižší váhou po term s nejvyšší váhou.

AddTerm – tato funkce provádí načtení termu ze souboru a vytvoření dílčího binárního rozhodovacího diagramu. Vstupem této funkce je instance třídy LexAn (lexikální analyzátor) a návratovou hodnotou je ukazatel na vytvořený BDD.

MakeBDD – tato funkce realizuje vytvoření celého ROBDD. V úvodní části se volají funkce varCount, termOrder, dumpTermAsc/dumpTermDesc, které provedou seřazení vstupních termů vzestupně/sestupně podle vah termů. V další části se voláním funkce AddTerm vytvoří dílčí BDD pro aktuálně zpracovávaný term a tento dílčí BDD se posléze začlení(funkce Cudd_bddAnd) do celkového ROBDD. Po přidání dílčího BDD se provádí rovnání proměnných výsledného ROBDD přičemž výběr heuristiky(jsou k dispozici téměř všechny metody reorderingu, které poskytuje knihovna CUDD) použité při rovnání je dání vstupními parametry programu. Rovnání proměnných se spouští automaticky knihovnou CUDD v závislosti na počtu uzlů v ROBDD. V poslední části funkce se provádí kontrola splnitelnosti funkce a pokud je více než 30 "mrtvých uzlů" je spuštěn garbage collector(odstraní mrtvé uzly a uvolní paměť). Kontrola splnitelnosti spočívá v testování vysledného ROBDD. Pokud je po přidání termu do výsledného ROBDD tento ROBDD tvořen pouze konstantním uzlem, znamená to, že zkoumaná funkce je nesplnitelná, je

vypsáno hlášení o nesplnitelnosti a program je ukončen. V opačném případě pokud po přidání BDD do ROBDD není výsledný ROBDD tvořen pouze konstantním uzlem, je funkce ještě stále splnitelná a pokračuje se v přidávání dalších termů(dílčích BDD) do výsledného ROBDD dokud se projeví nesplnitelnost, nebo dokud nejsou přidány všechny termy. Vstupem této funkce je instance třídy LexAn(lexikální analyzátor).

Output_Minterm – tato funkce je upravená funkce Cudd_PrintMinterm z knihovny CUDD. Provádí “přípravu“ k vypsání výstupu do souboru typu .pla, realizuje výpis hlavičky souboru typu .pla a jeho zápatí. Dále volá funkci PrintMinterm, která provede vlastní výpis do souboru – vypisuje vektory pro, které je vstupní funkce splněná. Vstupem funkce je DdManager, ukazatel na výsledný ROBDD, počet vstupních proměnných a počet termů. Funkce vrací hodnotu 1 pokud byl výpis proveden bez potíží popř. 0 v případě výskytu chyby(např. CUDD_MEMORY_OUT).

PrintMinterm – tato funkce je realizuje výpis vektorů pro, které je vstupní funkce splněná do souboru typu .pla. Tato funkce je obměnou funkce ddPrintMintermAux z knihovny CUDD. Vstupem je DdManager, ukazatel na výsledný ROBDD, pole prvků datového typu int o velikosti manageru, počet proměnných a počet termů.

Tyto funkce společně s funkcemi knihovny CUDD, které jsou popsány v následující kapitole tvoří mnou implementovaný řešič.

C.3 Popis použitých funkcí z balíku CUDD

Jak již bylo dříve řečeno je k realizaci mého řešiče použito knihovny CUDD[6,7]. V následující části jsou popsány funkce, které byly v řešiči použity:

Cudd_Init – tato funkce vytváří DD(rozhodovací diagram) manager, inicializuje hashovaní tabulku(unigue table), základní konstanty a pomocné funkce. Vstupním parametrem je počáteční počet BDD proměnných, počáteční počet ZDD proměnných, počáteční velikost hashovaní tabulky, počáteční velikost

cache, a maximální velikost zabrané paměti. Jeli maximální velikost zabrané paměti rovna 0, funkce Cudd_Init určí vhodné hodnoty pro maximální velikost cache a limit pro růst hashovaní tabulky – tyto operace vycházejí z množství dostupně paměti. Návratovou hodnotou je ukazatel na manager pokud se vytvoření zdaří popř. NULL v opačném případě.

Cudd_ReadOne – tato funkce má jako vstupní parametr DdManager a vrací jeho jedničkovou konstantu – BDD se vytváří zdola nahoru tzn. od konstantního uzlu ke kořenovému.

Cudd_Not – tato funkce neguje(vytváří komplement) DD(rozhodovací diagram) tak, že změní nastavení bitu v ukazateli - atributu doplňku (nejméně důležitý bit). Vstupem této funkce je ukazatel na uzel.

Cudd_Ref – tato funkce inkrementuje počet referencí na uzel který je zadán na vstupu v případě, že počet referencí je menší než maximální přípustná hodnota pro daný datový typ.

Cudd_bddIthVar – tato funkce získá proměnnou s požadovaným indexem z binárního rozhodovacího diagramu v případě že tato proměnná existuje. V případě, že ještě neexistuje vytvoří v daném binárním rozhodovacím diagramu novou proměnnou. Jestliže tato operace proběhne v pořádku, vrací ukazatel na proměnnou, v případě že se operace nezdaří je návratovou hodnotou NULL. Vstupem funkce je DdManager a index požadované proměnné.

Cudd_bddAnd – tato funkce provede logický součin(konjunkci) dvou binárních rozhodovacích diagramů f a g. Vrací ukazatel na výsledný binární rozhodovací diagram v případě, že operace proběhla v pořádku a NULL v případě, že se operace nezdařila. Vstupem je DdManager a dva binární rozhodovací diagramy, mezi kterými chceme provést logický součin.

Cudd_RecursiveDeref – tato funkce dekrementuje počet referencí na uzel a v případě že uzel "zemře"(počet referencí je roven nule), rekurzivně dekrementuje počet referencí u jeho potomků. Vstupem funkce je DdManager a uzel, u kterého chceme snížit počet referencí.

Cudd_bddOr - tato funkce provede logický součet(disjunkci) dvou binárních rozhodovacích diagramů f a g. Vrací ukazatel na výsledný binární rozhodovací diagram v případě, že operace proběhla v pořádku a NULL v případě, že se operace nezdařila. Vstupem je DdManager a dva binární rozhodovací diagramy, mezi kterými chceme provést logický součet.

Cudd_AutodynEnable – tato funkce povoluje spouštění dynamického rovnávání proměnných automaticky(pro BDDs a ADDs). Vstupem je DdManager (správce paměti, obsahuje informace o všech uzlech) a parametr, který udává metodu rovnání proměnných. Jestliže je jako parametr označující metodu rovnání proměnných použit CUDD_REORDER_SAME, nemění se metoda přerovnávání.

Cudd_ReduceHeap – tato funkce spouští přerovnávání proměnných, přičemž vstupem je DdManager, parametr který udává jaká heuristika přerovnání proměnných bude použita(viz. Příloha A,Kapitola 3 metody reorderingu) a číslo reprezentující dolní hranici pro kterou se přerovnání neprovádí.

Cudd_IsConstant – tato funkce vrací hodnotu 1 v případě, že uzel zadaný jako vstup je konstantním uzlem(terminálním uzlem). Všechny konstantní uzly mají stejný index (CUDD_CONST_INDEX). Ukazatel předaný funkci Cudd_IsConstant může být normální nebo jím může být jeho doplněk.

Cudd_ReadDead – tato funkce má jako vstup DdManager a vrací počet "mrtvých uzlů"(dead nodes – uzly, na které neexistuje žádná reference) v hashovaní tabulce(unique table).

CuddGarbageCollect – tato funkce spouští operaci grabage collection(sbírání smetí – dealokace mrtvých uzlů,…). Jestliže vstupní parametr clearCache je nastaven a nulu, neprovádí se čištění cache. Funkce vrací počet vymazaných uzlů.

Cudd_DagSize – tato funkce provádí součet uzlů v DD(rozhodovací diagram). Vrací tedy počet uzlů v grafu. Vstupem je ukazatel na "kořenový" uzel.

Cudd_ReadSize – tato funkce vrací počet existujících proměnných přičemž vstupem této funkce je DdManager.

Cudd_ReadMemoryInUse – tato funkce vrací velikost paměti alokované managerem v bajtech přičemž vstupem je DdManager.

Cudd_ReadReorderings – tato funkce vrací počet rovnání proměnných. Tato hodnota udává kolikrát bylo spuštěno rovnání proměnných ať už bylo spuštěno voláním funkce Cudd_ReduceHeap nebo automaticky. Pokud je ale rovnání proměnných spuštěno s parametrem CUDD_REORDER_NONE, nezapočítává se do počtu rovnání proměnných. Do počtu rovnání proměnných také není započítáno volání funkce Cudd_ShuffleHeap(změna pořadí proměnných).

Cudd_ReadReorderingTime – tato funkce vrací čas(v milisekundách), který byl od inicializace manageru strávený rovnáním proměnných. Čas obsahuje kromě doby rovnání proměnných také čas strávený "sbíráním smetí"(odstranění dead uzlů atd.) před rovnáním. Vstupem je DdManager.

Cudd_CountPathsToNonZero – tato funkce projde DD(rozhodovací diagram) a spočítá počet cest do nenulového terminálu. Návratová hodnota(počet cest do nenulového terminálu) je reprezentován jako double, což umožňuje aplikovat tuto funkci i na větší množství proměnných(větší počet cest). Vstupním parametrem je ukazatel na "kořenový" uzel.

Cudd_CountPath – tato funkce spočítá všechny cesty v DD(rozhodovací diagram) vedoucí do terminálového uzlu. Návratová hodnota(celkový počet cest) je reprezentována jako double, což umožňuje pracovat s větším počtem proměnných(větší počet cest). V případě že, počet cest z "kořenového" uzlu je větší než rozsah proměnné typu double funkce vrací konstantu CUDD_OUT_OF_MEM. Vstupním parametrem je ukazatel na "kořenový" uzel.

Cudd_CheckZeroRef – tato funkce kontroluje hashovaní tabulku(unique table), která je vstupem a vrací počet uzlů s nenulovým počtem referencí. Funkce je volána nejčastěji před voláním funkce Cudd_Quit, abychom si ověřili, že je dealokována paměť zabraná uzly a že nevznikly nekonzistence. Nekonzistence mohou vzniknout například zapomenutím dereferencování uzlů(funkce Cudd_RecursiveDeref).

Cudd_Quit – tato funkce uvolní zdroje alokované DD managerem a resetuje globální statistické čítače. Vstupem je DdManager.

C.4 Práce s implementovaným SAT solverem

Mnou implementovaný sat solver má vstup jako většina ostatních sat solverů ve formátu DIMACS, který je popsán v kapitole 3.1, dále je zde několik možných výstupů(tyto jsou popsány v kapitole 3.3) a při zpracování lze použít několik heuristik. Řešič se ovládá zadáním vhodných parametrů do příkazové řádky, přičemž první povinný parametr udává jméno vstupního souboru, druhý povinný parametr udává jméno výstupního souboru(výstup typu .pla), třetí povinný parametr je dvouciferné číslo, které označuje jaké dynamické heuristiky bude použito pro rovnání proměnných a čtvrtý povinný parametr udává kolik vektorů, pro které je funkce splněná se vypíše do výstupního souboru typu .pla. První dva povinné parametry nepotřebují žádné bližší vysvětlení, proto si nyní pouze popíšeme význam třetího a čtvrtého číselného parametru. Již dříve jsme si popsali možné metody reorderingu, které jsou dostupné v balíku CUDD(Příloha A, kapitola 3) a většina z nich je také dostupná v mém řešiči. Nižší cifra třetího parametru udává heuristiku dynamického reorderingu, která bude použita přičemž je 9 možných variant :

0 - reordering se neprovádí
1 - random reordering
2 - random pivot reordering
3 - sift reordering
4 - sym-sift reordering
5 - group-sift reordering
6 - window 2 reordering
7 - anealing reordering
8 - genetic reordering

Druhá cifra třetího parametru udává jak budou seřazeny vstupní termy podle svých vah(kapitola 4.3). Možnosti jsou následující:

0 - řazení vstupních termů se neprovádí(nula se nepíše)
1 - provádí se řazení termů podle vah od termů s nejvyšší váhou po termy s nejnižší váhou.

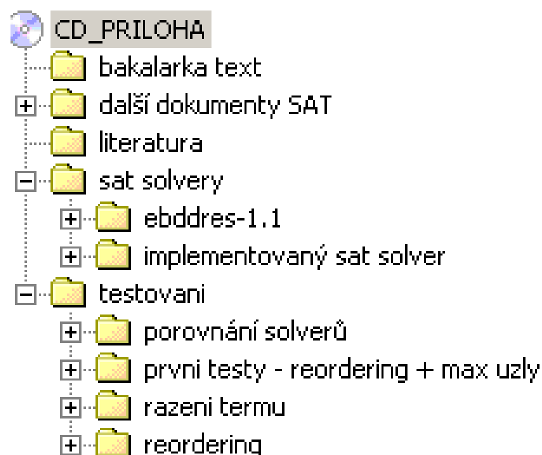
2 - provádí se řazení termů podle vah od termů s nejnižší váhou po termy s nejvyšší váhou.

Jak již bylo dříve řečeno, čtvrtý parametr udává kolik vektorů, pro které je funkce splněná se vypíše do souboru .pla. Vypisuje se vždy počet vektorů daný parametrem v případě, že počet vektorů je větší než velikost parametru. V opačném případě jsou vypsány všechny vektory. V případě, že je jako tento parametr zadána –1, použije se implicitní nastavení, kdy se použije maximální hodnota, která lze uchovat v datovém typu long. V případě zadání jakéhokoli jiného čísla bude vypsán počet vektorů daný číslem(bereme absolutní hodnotu). Výpis vektorů se provádí pouze pokud je výstup menší než 100MB.

Spuštění řešiče může tedy vypadat následovně "BddCudd.exe vstup.cnf vystup.pla 10" v tomto případě bude použit jako vstup soubor vstup.cnf, výstup ve formátu .pla bude uložen do souboru vystup.pla, reordering se provádět nebude(nižší cifra je 0) a bude provedeno řazení vstupních termů od těch nejvyšší váhou po ty s nejnižší váhou(vyšší cifra je 1).

D. Obsah CD

Na přiloženém CD jsou v adresářové struktuře, která je zobrazena na následujícím obrázku uloženy adresáře a soubory obsahující implementovaný sat solver, sat solver EBDDRES, použitou literaturu, další dokumenty použité při vývoji, naměřené hodnoty a samozřejmě také tuto bakalářskou práci.



Obrázek č.13 : *Stromová struktura přiloženého CD*

Nyní si jednotlivé adresáře popíšeme:

bakalarka text – adresář obsahuje bakalářskou práci ve formátu .doc a .pdf.

další dokumenty SAT - adresář obsahuje další dokumenty obsahující informace o problematice řešení problému splnitelnosti booleovské funkce a práce s BDD(OBDD,ROBDD).

literatura – adresář obsahuje většinu literatury ze seznamu literatury použité pro tvorbu této práce.

sat solvery – adresář obsahuje sat solver EBDDRES a mnou implementovaný řešič.

testování – adresář obsahuje výsledky měření(i z průběhu vývoje řešiče) heuristik reorderingu, maximálního počtu uzlů které je nutné zpracovat pro množství úloh atd.