České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Porovnání dostupných programových balíků pro manipulaci s binárními rozhodovacími diagramy

Martin Felcman

Vedoucí práce : Ing. Petr Fišer

Studijní program : Elektrotechnika a informatika strukturovaný bakalářský
Obor : Informatika a výpočetní technika

leden 2008

Poděkování

Tímto bych chtěl poděkovat především své přítelkyni a synovi, kterým jsem při psaní této práce nevěnoval mnoho času.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Mělníku dne 28.12.2007 ……

Abstrakt

Cílem této práce je vyhledat na internetu dostupné balíčky (knihovny) pro práci s binárními rozhodovacími diagramy tzv. BDD, zdokumentovat, jakým způsobem je BDD reprezentován, uvést příklad vytvoření jednoduchého BDD a práci s ním, a především vyzdvihnout rozdíly mezi jednotlivými balíčky. Vstupními soubory pro zadávání diagramů je formát PLA, jehož načítání bude v některých případech nutné doprogramovat. Závěr práce se bude věnovat přehlednému srovnání jednotlivých balíčků doplněný o testy.

Abstract

The aim of this work is to look for available packages for managing Binary Decision Diagrams on the internet, give account of the representation of BDD, introduce an example of creating and managing a simple BDD, and above all point out the differences between each of the packages. The input file for loading diagrams is in PLA espreso format, whose reading will be necessary to implement. The end of this work is devoted to a lucid comparison of teh packages supplemented by benchmarks.

1 Úvod

Binární rozhodovací diagramy (BDD – Binary Decision Diagram) jsou datové struktury, které se ukázaly jako kompaktní reprezentace booleovských funkcí. Libovolnou konečnou množinu lze reprezentovat jako BDD. Množinové operace jako sjednocení a průnik lze realizovat jako booleovské operace nad BDD, pro které existují poměrně efektivní algoritmy.

V této práci se budu zabývat porovnáním volně dostupných balíčků pro práci s BDD v prostředí MS Windows. V úvodu budu popisovat jednoduchou lineární algebru a teorii zabývající se tvorbou a manipulací s BDD. Dále krátce popíši základní informace o tom, jak je reprezentován uzel (node), jakým způsobem je celý BDD uložen (hash. tabulky,strom…), na příkladu ukážu výstavby jednoduchého BDD použití jednotlivých funkcí. Vzhledem k tomu, že některé balíčky jich obsahují až tisíce, zaměřím se pouze na některé z nich.

Výsledkem mé práce by mělo být poskytnutí základních informací o práci s jednotlivými balíky.

2 Booleovská algebra

V této části se budu zabývat základní lineární algebrou.

2.1 Definice Booleovské algebry a základních operátorů

Booleovská algebra {B, ¬, ∨, 0, 1} je matematickou strukturou složenou z množiny B obsahující alespoň dva rozdílné elementy 0 a 1, na kterých jsou definovány dvě operace. Logická negace (komplement) značená ¬ (v textu je možná i notace pomocí vodorovné čáry nad proměnnou př. $\bar{a}$) a logické „a“ (konjunkce) značené ∨ (nebo také „·“).

	¬
0	1
1	0

∧	0	1
0	0	0
1	0	1

Tabulka1: Základní pravdivostní tabulky

Existují i další logické operátory, všechny jsou zpětně převoditelné na tyto dva. Patří mezi ně logické „Nebo“ (disjunkce) značené „∨“, implikace „⇒“ a logická ekvivalence „⇔“.Jejich definice jsou následující:

$$a \vee b \equiv \neg(\neg a \wedge \neg b)$$

$$a \Rightarrow b \equiv \neg a \vee b$$

$$a \Leftrightarrow b \equiv (a \Rightarrow b) \wedge (b \Rightarrow a) \quad ;$$

přičemž a,b,c jsou termy nabývající pravdivostní hodnoty 0 nebo 1.

Množina B značí libovolnou množinu, která tvoří Booleovskou algebru. Nejjednodušší je množina B = {0,1}. N-tice Booleovských proměnných $B^n = \{0,1\}^n$ také tvoří Booleovskou algebru, jestliže operace nad touto množinou jsou rozumným rozšířením operací z B.

2.2 Booleovská funkce

Booleovská funkce f je zobrazením $f: B^n \rightarrow B$. Její doménou je množina 2^n binárních vektorů {(0,0,…,0), (0,0,…,1), …, (1,1,…,1)}. Vstupní vektor reprezentuje $x_i = (b1,b2,\ldots,b_n)$, kde

$$i = \sum 2^{j-1} * bj \ , bj \in \{0,1\}.$$

Booleovská funkce je tudíž množina uspořádaných dvojic, kde první element je binární vstupní vektor a druhý elementem je výstup ohodnocený log. 0 nebo 1.

Funkce s více výstupy jsou obdobně definovány jako zobrazení

$$f: B^n \rightarrow B^m$$

Algebraické operace s Booleovskými funkcemi vychází z používání proměnných pro reprezentaci prvků z B. Booleovská proměnná je jedním prvkem domény. Značí se malým písmenem, případně doplněné indexem: například x_1.

2.3 Booleovské výrazy

Booleovský výraz je v podstatě reprezentace Booleovských funkcí pomocí proměnných, konstant a operátorů. Je definován následovně:

- Libovolný element $e \in B$ je Booleovský výraz.
- Libovolná proměnná $x_i \in X$ je Booleovský výraz.

- Jestliže F, G, a H jsou Booleovské výrazy, potom (F ∨ G), (F ∧ G) a ¬H jsou také Booleovské výrazy.
- Neexistují žádné jiné Booleovské výrazy než ty vzniklé aplikací výše uvedených pravidel.

Priorita Booleovských operátorů je v tomto pořadí: negace, konjunkce a disjunkce. Pro zpřehlednění zápisu se používají závorky. Operátor konjunkce v případě použití s proměnnými může být vypuštěn. Např. $x \cdot y = xy$.

2.4 Shannonův expanzní teorém

Víceúrovňové logické výrazy jsou rozšířením klasického dvouúrovňového konceptu. Existuje zde konečné množství logických úrovní mezi vstupem a výstupem. Pro derivaci víceúrovňového výrazu z Booleovské funkce můžeme použít dekompozičními techniky. Základem několika těchto metod je Shannonova expanze. Shannonova expanze funkce f rozložena podle proměnné x je definována takto:

$$f(x) = \neg x_i \cdot f(x_1, .., x_i=0,...,x_n) \vee x_i \cdot f(x_1, .., x_i=1,...,x_n)$$

Funkce $f(x_1, .., x_i=0,...,x_n)$ je kofaktorem funkce f rozloženým podle proměnné x_i. Získá se nahrazením všech výskytů proměnné x_i ($\neg x_i$) logickou 1 (resp. 0) v Booleovském výrazu, který určuje funkci.

3 Binární rozhodovací diagramy (BDD – Binary Decision Diagrams)

Existuje mnoho vyjádření, jak zapsat logickou funkci, jako například Reed-Mullerovy výrazy, které získaly pozornost díky svým výhodám v určitých aplikacích, tabulkou, konjunktivní nebo disjunktivní formou, Karnaughovými mapami apod. Jde o to, že každá reprezentace může být výhodná pro jiné funkce a použití. Obecně, vhodnou volbu datové reprezentace pro danou aplikaci můžeme měřit rychlostí, s jakou jsou prováděny jednotlivé logické operace nad danou strukturou, a kolik paměti potřebují. Lineární funkce jsou například jednoduše reprezentovatelné pomocí Reed-Mullerových výrazů. S rostoucím počtem vstupů je ale jeho zápis pomocí součtu čtverců paměťově exponenciálně závislý. Rostoucí výzkumné úspěchy v oblasti datových struktur a algoritmů daly vniknout novým reprezentacím logických funkcí, jakými jsou například Binární Rozhodovací Diagramy (BDD) resp. ROBDD nebo If-Then-Else přímé acyklické grafy (DAG), které umí tyto operace provádět efektivně jak po časové, tak po paměťové stránce.

3.1 Historie BDD

Binární rozhodovací diagramy představil v roce 1959 C. Y. Lee. O další studii a rozvoj se postarali Boute (1976) a především S. B. Akers (1978). Nicméně efektivní využití rozhodovacích diagramů přinesla až studie R. E. Bryanta z univerzity Carnegie Mellon. Rozšířil stávající teorie o použití změny pořadí proměnných (variable ordering) pro kanonickou reprezentaci Booleovských funkcí a o sdílené podgrafy pro možnosti porovnávání funkcí. Aplikace těchto dvou poznatků přinesla základ pro vytvoření efektivních datových struktur (ROBDD – Reduced Ordered Binary Decision Diagram) a algoritmů nad touto strukturou pracujících.

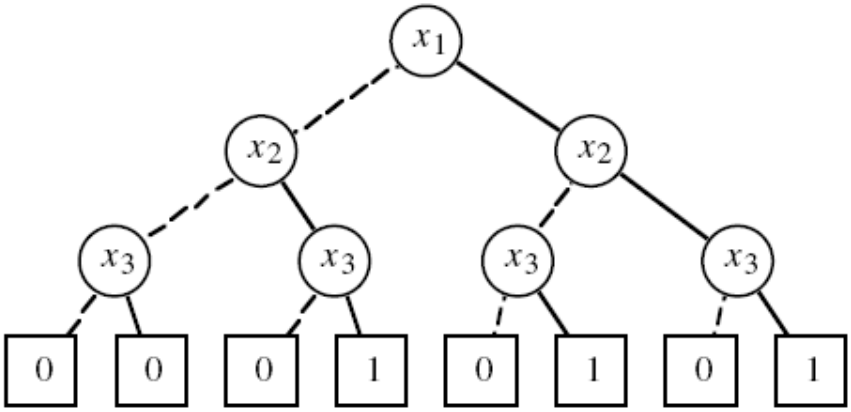
3.2 BDT

Jedna z dalších možností, jak reprezentovat logickou funkci, je Binární rozhodovací strom (BDT – Binary Decision Tree). Nejprve si jej definujme.

Binární rozhodovací strom (BDT) je přímí acyklický graf, kde každý uzel má nejvýše dva „potomky“ a právě jednoho „rodiče“. Existují dva typy uzlů: neterminální (vnitřní) a terminální (listy). Terminální uzly nemají žádného potomka a reprezentují konstantní Booleovskou funkci 1 nebo 0. Každý neterminální uzel n_i je přiřazen vstupní proměnné x_i. Nechť n_i má dva potomky a označme je low(n_i) a high(n_i). Booleovská funkce může být podle Shannonova teorému zapsána ve tvaru:

$$f^{ni}(X) = \neg x_i \cdot f^{low(ni)}(X) \vee x_i \cdot f^{high(ni)}(X).$$

Pro vyčíslení hodnoty funkce pro daný vstupní vektor musíme projít strom od jeho kořene až k listu. Cesta je definována vstupními proměnnými. Pokud proměnná odpovídající uzlu má hodnotu 1, použijeme hranu high(n_i) a přidáme ji do výsledné cesty. V případě, že proměnná je ohodnocena 0, použijeme hranu low. Cena BDT samozřejmě odpovídá 2^n.



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

Obrázek 1: BDT a jeho pravdivostní tabulka

3.3 BDD

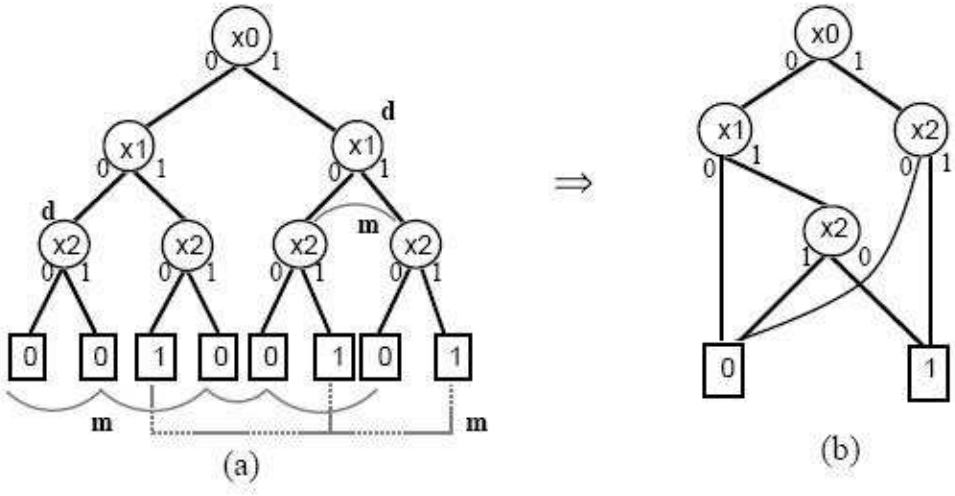
Významný posun v reprezentaci Booleovských funkcí zaznamenal S. B. Akers. Navrhnul sadu zjednodušujících pravidel k redukci složitosti BDT a dal vznik přímému acyklickému diagramu (BDD):

- Dva uzly jsou v BDT ekvivalentní, pokud jsou stejným terminálem nebo jsou svázány se stejnou vstupní proměnnou a oba jejich potomci (low i high) jsou ekvivalentní.

- Uzel n je v BDT redundantní, pokud low(n) je ekvivalentní high(n).

- Binární Rozhodovací Diagram (Binary Decision Diagram – BDD) je přímý acyklický graf derivovaný z BDT, kde redundantní uzly jsou smazány a ekvivalentní uzly jsou spojeny.

Hlavní výhodou BDD je ta, že jeho velikost je daleko menší než velikost BDT pro velké funkce. Aplikace redukčních pravidel ušetří značnou část místa. Například zatímco BDT má 2^n terminálních uzlů, BDD potřebuje pouze dva listy k reprezentaci logických hodnot 0 a 1. Názorná ukázka derivace BDD z BDT je na obrázku 3.1. Písmeno „d“ (z angl. delete) značí uzly, které jsou redundantní, a tudíž budou odstraněny, a písmeno „m“ (z ang. merge) značí ekvivalentní uzly, které budou spojeny. Uvnitř uzlů jsou znázorněny vstupní proměnné.



Obrázek 2: Ukázka použití redukčních pravidel

3.4 Vlastnosti BDD

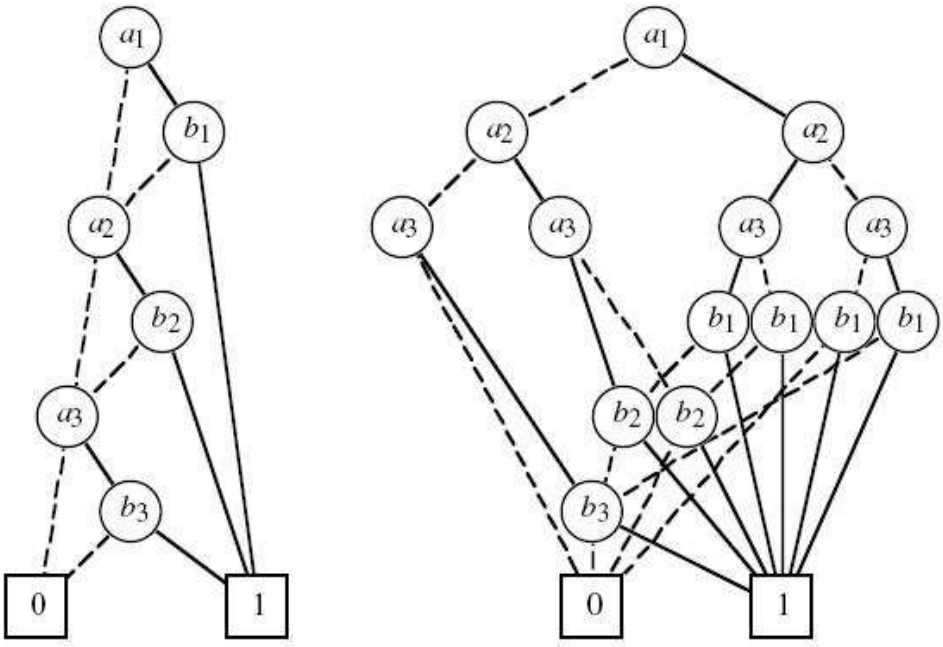
BDD se ukázaly jako efektivní struktura pro analýzu, simulaci a testování digitálních obvodů. Akers zmiňuje několik příkladů takových aplikací pro známé logické a sekvenční obvody. Tvrdí, že přestože v nejhorším případě velikost BDD je exponenciální v závislosti na počtu vstupů, tak pro většinu běžných digitálních zařízení toto číslo roste lineárně. Bohužel příklady a navrhované metody nejsou určeny pro počítačovou implementaci, což zužuje jejich aplikaci pouze na malé problémy. Dalším problémem je složitost logických operací prováděných nad BDD. Jak například spojit dva BDD pomocí operátoru AND? Určitým řešením je spojit tyto dva grafy do série, ale to by bylo plýtvání místem, protože by neexistovalo sdílení podfunkcí. Testy logické ekvivalence a tautologie jsou příklady dalších logických operací prováděných v digitální analýze, které mohou být poměrně složité při implementaci pomocí BDD.

Tyto nedostatky byly odstraněny představením metody řazení vstupní proměnných (variable ordering), které daly vzniknout struktuře Ordered Binary Decision Diagram (OBDD).

3.5 OBDD

Další významnou technikou představenou R. E. Bryantem je řazení proměnných (Variable ordering). Její aplikací získáme Ordered Binary Decisoin Diagram. Uspořádaný binární rozhodovací diagram je binární rozhodovací diagram, ve kterém jsou vstupní proměnné seřazeny takovým způsobem, že v každé cestě OBDD se každá proměnná objevuje pouze jednou a v tom samém pořadí. Obecně lze řazení proměnných vybrat libovolné, algoritmus bude fungovat pro libovolné pořadí proměnných. V praxi má ovšem vhodná volba řadící techniky důležitý vliv na konečnou velikost BDD.

Na obrázku níže můžeme vidět dva OBDD reprezentující funkci $a_1b_1 + a_2b_2 + a_3b_3$. Vlevo vidíme pořadí proměnných $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, zatímco napravo jsou proměnné seřazeny takto $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$.



Obrázek 3: Reprezentace jedné funkce z různým uspořádáním proměnných

Zobecněním prvního případu na n proměnných, tedy na $a_1 < b_1 < \ldots < a_n < b_n$ získáváme OBDD o 2n neterminálních uzlech. V druhém případě $a_1 < \ldots < a_n < b_1 < \ldots < b_n$ vzniká OBDD s počtem neterminálních uzlů roven $2(2^n - 1)$. Pro velké hodnoty n rozdíl mezi

lineárním růstem prvního řazení oproti exponenciálnímu růstu druhého řazení má značný dopad na paměťové požadavky a efektivnost jednotlivých algoritmů.

Při hlubším prohlédnutí grafů na obrázku X můžeme vidět, že v prvním případě jsou proměnné seřazeny podle jejich výskytů v Booleovském výrazu $a_1b_1 + a_2b_2 + a_3b_3$. Tudíž v každém sudém patře jsou zapotřebí pouze dvě cesty. Jedna vede do terminálního uzlu 1 pro případy, kdy je logický součet roven 1, druhá vede do dalšího patra v případě, že je předešlý logický součin roven 0. Naproti tomu v druhém grafu první 3 patra vytváří kompletní Binární rozhodovací strom pro proměnné a. Obecně vzato, pro každou proměnnou a závisí výsledná funkce na určité kombinaci proměnných b. Zobecníme-li toto opět na n, vzniká struktura , kde prvních n pater OBDD tvoří neefektivní kompletní binární strom.

Většina aplikací používající OBDD vybírají pořadí proměnných na začátku a vytvářejí grafy podle tohoto zvoleného pořadí. Toto pořadí je zvoleno buďto ručně, nebo na základě heuristických analýz. Heuristiky nám nezaručují, zda získáme nejlepší pořadí proměnných.

Důležité je poznamenat, že zvolené řazení proměnných nemá žádný dopad na správnost výsledku. Jakmile je možné najít vhodné řazení, abychom se vyhnuli exponenciálnímu růstu, zůstávají operace nad BDD velmi efektivní.

3.6 ROBDD

Redukované uspořádané binární rozhodovací diagramy (Reduced Ordered Binary Decision Diagram – ROBDD) jsou v podstatě OBDD, na které aplikujeme redukční pravidla.
Z uspořádání víme, že každá vstupní proměnná je označena indexem, jenž označuje její pozici. Stejně tak každý uzel n v ROBDD má atribut index(n), který označuje, se kterou proměnnou je svázán. Index(n) roste směrem od kořene grafu k jeho listu. Z toho plyne, že v každé cestě ROBDD se indexy jednotlivých uzlů musí objevovat ve vzestupném pořadí. Jelikož se proměnné v cestě objevují pouze jednou, může být funkce spojená s uzlem n s indexem(n) = i, $f^{ni}(X)$, popsána pomocí Shannonovy expanze:

$$f^{n_i} = \overline{x}_i \cdot f_{\overline{x}_i}^{n_i}(X) + x_i \cdot f_{x_i}^{n_i}(X)$$

.

Je zajímavé, že ROBDD byly známy už z doby před Bryantem. Příkladem je práce Thayse o P-funkcích. V této práci autor představuje metody pro syntézu binárních rozhodovacích programů. Jedna z nich , „Simple BDD synthesis“, vede k efektivní konstrukci ROBDD. Bryant byl přesto prvním, jenž si uvědomil významu BDD pro reprezentaci logických funkcí.

3.7 Složitost ROBDD

ROBDD poskytují praktický význam pro symbolickou manipulaci s Booleovskými funkcemi pouze, pokud velikost grafu s rostoucím počtem proměnných grafu zůstává menší, než je nejhorší možný případ (exponenciální). Jak jsme mohli vidět v kapitole 3.5, velikost některých funkcí je silně závislá na volbě pořadí proměnných. Při dobré volbě řazení však zůstávají poměrně kompaktní. Praktické nasazení ukazuje, že valná většina běžně používaných funkcí se dá efektivně vyjádřit pomocí ROBDD. Následující tabulka ukazuje složitosti některých funkcí.

Typ funkce	Složitost	
	Nejlepší	Nejhorší
Symetrická	lineární	kvadratická
Sčítání	lineární	exponenciální
Násobení	exponenciální	exponenciální

Tabulka 2: Složitost ROBDD vytvořeného z běžných funkcí

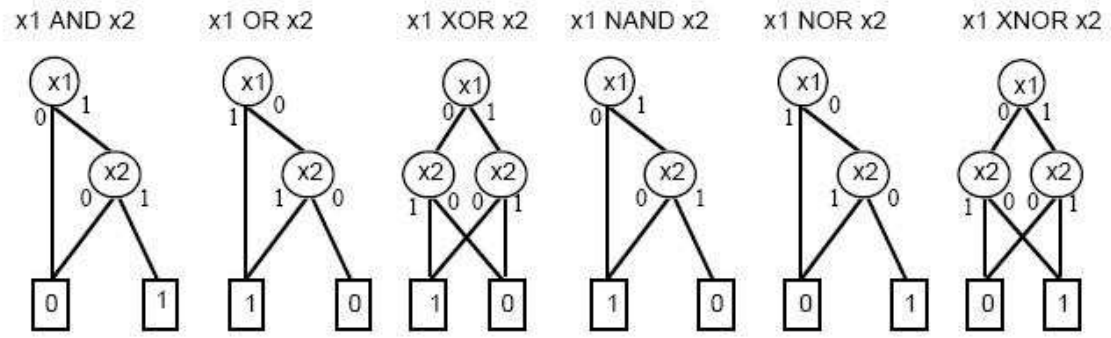
Na symetrických funkcích, kde funkční hodnota závisí pouze na argumentech rovných 1, má změna pořadí proměnných jen velmi malý dopad. Kromě triviálního případu konstantní funkce nabývají velikosti grafů těchto funkcí lineární (např. parita), až kvadratický (např. alespoň polovina vstupů je rovna 1) charakter.

Výstup n-bitové sčítačky můžeme považovat jako Booleovskou funkci dvou proměnných a_0, a_1, … , a_{n-1} a b_0, b_1, …, b_{n-1}. Pro uspořádání $a_0 < b_0 < a_1 < b_1 < \ldots < a_{n-1} < b_{n-1}$ má reprezentace ROBDD lineární složitost, zatímco uspořádání $a_0 < \ldots < a_{n-1} < b_0 < \ldots < b_{n-1}$ má složitost exponenciální. V podstatě se jedná o funkce se stejným uspořádáním jako na obrázku X.

Booleovské funkce reprezentující násobení jsou naopak poměrně složitým případem ROBDD. Nehledě na uspořádání, kterýkoliv ze dvou výstupů (prostřední bity) n-bitové násobičky má exponenciální reprezentaci pomocí ROBDD.

3.8 Kanonická reprezentace ROBDD

Jednou z hlavních vlastností ROBDD je ta, že se jedná o kanonickou reprezentaci. To znamená, že pokud jsou dvě Booleovské funkce ekvivalentní, jejich ROBDD jsou izomorfní. Logická ekvivalence se tedy redukuje pouze na kontrolu, zda-li jsou dva Grafy izomorfní, což odpovídá složitosti O(x), kde x je velikost grafu. Dalším důsledkem kanonicity ROBDD je ten, že kontrola tautologie se stává triviální: grafem musí být terminál 1. Záměna pořadí proměnných pro tvorbu ROBDD je operace s náročností O(x). Obecně operace se dvěma ROBDD mají náročnost $O(x_1 * x_2)$. Následující obrázek ukazuje ROBDD pro základní logické funkce.



Obrázek 4: Základní logické funkce a jejich ROBDD

3.9 Konstrukce ROBDD pomocí Apply/Reduce

Tvorba BDD je rozložena do dvou fází. V první fázi, expanzi shora-dolu, je booleovská funkce rozdělena (dekompována) na jednotlivé pod-problémy pomocí Shannonovy expanze. V další fázi, zdola-nahoru, je poté každý z těchto pod-problémů zapsán v kanonické formě. Jedinečnost této reprezentace nám zajišťují hashovací tabulky (unique tables). Tyto mezi-výsledky jsou poté rekurzivně slučovány algoritmem do hloubky popsané R. E . Bryantem v publikaci „Grafové algoritmy pro manipulaci s Booleovskými funkcemi“. V poslední době se objevují i pokusy o snížení paměťové náročnosti použitím algoritmu do šířky.

Algoritmus je založen na používání dvou operací Apply a Reduce. Apply pomocí binárního operátoru (and, or, xor,…) spojí dvě funkce. Implementace apply operace se opírá o Shanonovu expanzi. Algoritmus je založen na procházení jednotlivých grafů do hloubky. Pro zefektivnění této operace se používají 2 hashovací tabulky (respektive pro urychlení operace se používá 1, druhá k zajištění generování maximálně zredukovaného grafu).

Následně se zavolá funkce Reduce (někdy označována též Restrict) a vytvoří se pomocí pravidel pro redukci (a s využitím druhé hash. tabulky) kanonický ROBDD.

V roce 1990 publikoval R. E. Bryant efektivnější metodu konstrukce ROBDD, a to pomocí ITE operátoru.

3.10 Využití ITE operátoru

ITE (If-Then-Else) operátor (více viz. 2.5) umožňuje spojit 3 ROBDD do jednoho bez nutnosti následné redukce. Jedná se o stejnou operaci, která se používá pro uzly ROBDD, základní stavební prvky této struktury. Rozdíl je, že ITE má za argumenty 3 funkce, každou reprezentovanou vlastním ROBDD, zatímco u uzlu je první parametr (f) nahrazen proměnnou. Operátor ite je definován takto:

$$ite(a,b,c) \equiv ab \vee \neg ac .$$

Využití operátoru pro definici dvou proměnných funkcí ukazuje následující tabulka. Existuje celkem 16 základních funkcí pro množinu B^2: ((⌐f) (⌐g), (⌐f)g,f(⌐g),fg):

Tabulka	Výraz	Ekvivalentní forma
0000	0	0
0001	fg	ITE(f,g,0)
0010	f ·(⌐g)	ITE(f,⌐g,0)
0011	f	f
0100	(⌐f)·g	ITE(f,0,g)
0101	g	g
0110	f xor g	ITE(f,⌐g,g)
0111	f or g	ITE(f,1,g)
1000	⌐(f or g)	ITE(f,0,⌐g)
1001	⌐(f xor g)	ITE(f,g,⌐g)
1010	⌐g	ITE(g,0,1)
1011	f or (⌐g)	ITE(f,1,⌐g)
1100	⌐f	ITE(f,0,1)
1101	(⌐f)+g	ITE(f,g,1)
1110	⌐(fg)	ITE(f,⌐g,1)
1111	1	1

Tabulka 3: Základní binární operace vyjádřené pomocí ITE

Rekurzivně zapsaný algoritmus ITE s rozkladem podle kořenového uzlu v:

$$z = (v, ite(f_v, g_v, h_v), ite(f_v, g_v, h_v)) .$$

Vidíme, že máme uzel v a dva jeho potomky definované rekurzí. Jedná se o podobný princip jako při operaci Apply. Efektivnost tohoto algoritmu zaručuje používání takzv. Unique tabulky a Computed tabulky (hashovaní tabulky). Složitost je rovna $O(f(s_1) * g(s_2) * h(s_3))$, kde s_1, s_2 a s_3 jsou velikosti jednotlivých ROBBD reprezentující dané funkce.

Programově by algoritmus vypadal následovně:

Terminal case: (0, g, f) = (1, f, g) = f

Ite (f, g, g) = g

ite(f, g, h)

if (terminal case) {

return result;

} else if (existuje záznam v computed table pro (f, g, h)){

return result;

} else {

nechť je v kořenový uzel (f, g, h);

f´← ite(f_v, g_v, h_v);

g´←ite($f_{\neg v}$, $g_{\neg v}$, $h_{\neg v}$);

if (f´ = g´) return g´;

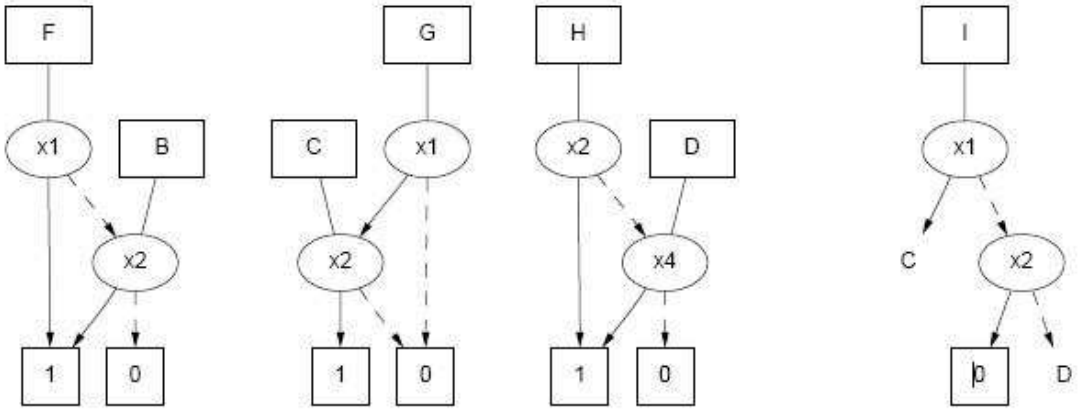
R← najdi_nebo_pridej_do_unique_tabulky(v, f´,g´);

vloz_do_computed_table({f, g, h}, R);

return R; }

}

Uveďme si postup na příkladu:



Obrázek 5: Ověření platnosti ITE vztahu

I = ite(F, G, H)

= (a, ite(F_a, G_a, H_a), ite($F_{\neg a}$, $G_{\neg a}$, $H_{\neg a}$))

= (a, ite(1, C, H), ite(B, 0, H))

= (a, C, (b, ite(B_b, 0_b, H_b), ite($b_{\neg b}$, $0_{\neg b}$, $H_{\neg b}$))

= (a, C, (b, ite(1,0,1), ite(0,0,D)))

= (a, C, (b, 0, D))

= (a, C, J)

Pro kontrolu spočítáme:

F = a + b

G = ac

H = b + d

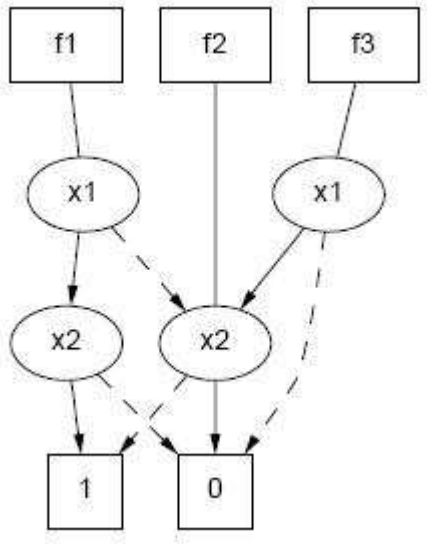
Ite (F, G, H) = (a + b)(ac) + (¬a)(¬b)(b + d)

= ac + (¬a)(¬bd)

3.11 Sdílené ROBDD

Některé funkce mohou být reprezentovány jediným přímým acyklickým grafem s několika kořeny. Tímto způsobem lze podgrafy, které se objevují v několika ROBDD, reprezentovat pouze jednou. Tento způsob representace se nazývá sdílený ROBDD.

Následující obrázek reprezentuje sdílený podgraf funkcí: $f_1=(x_1 \wedge x_2) \vee ((\neg x_1) \wedge (\neg x_2))$, $f_2= \neg x_2$, $f_3= x_1 \wedge (\neg x_2)$



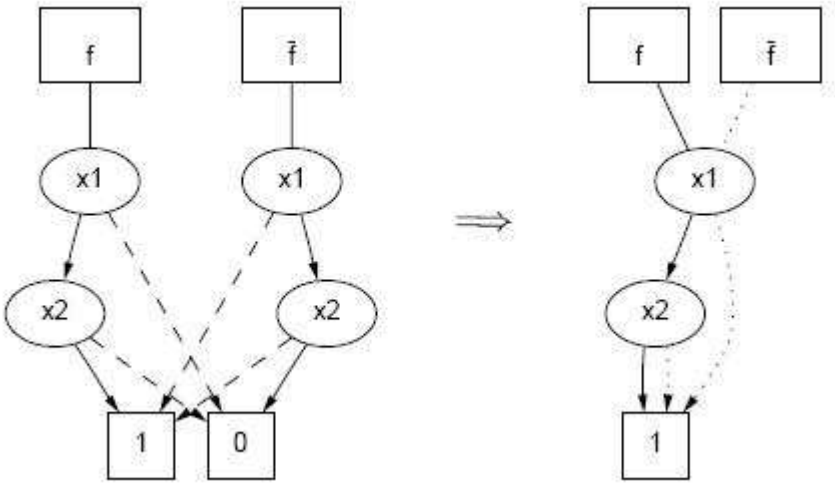
Obrázek 6: Ukázka sdílených funkcí f_1, f_2 a f_3

Takovýto způsob reprezentace ušetří čas a paměť oproti tomu, abychom museli udržovat informace o třech ROBDD. V případě, že se každá podfunkce v ROBDD nachází maximálně jednou, přechází kanonická forma v silně kanonickou formu. Toto označení znamená, že dvě stejné funkce f a g nemají pouze stejnou ROBDD reprezentaci, ale v implementaci mají ukazatele na stejné místo v paměti. Jedním z důsledků je triviální kontrola ekvivalence, které spočívá pouze v tom, že se porovnají ukazatele.

3.12 Negované hrany

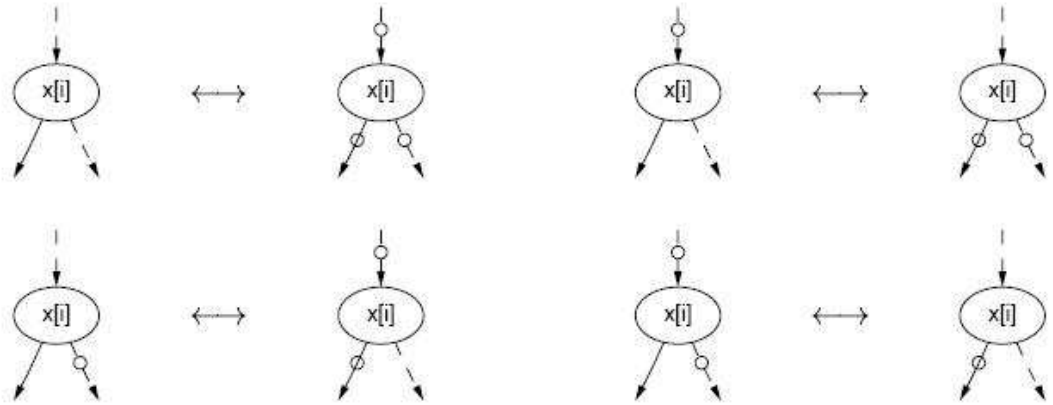
V roce 1978 S. B. Akers představil další techniku používanou v BDD známou jako negované hrany (Complemented edges). Vychází z faktu, že můžeme velice snadno najít

negaci (komplement) BDD přehozením logických hodnot terminálů. Zbytek BDD zůstane nezměněný. Tedy místo toho, abychom reprezentovali dvě funkce f a ¬f pomocí dvou různých grafů, použijeme jediný graf pro f a označíme, že některý uzel používá ¬f přidáním 1 bitového příznaku do hrany, jež je spojuje. Příznak může být viděn jako vložená operace negace aplikovaná na danou funkci určenou daným uzlem. Tato technika byla implementována v některých BDD balíčcích a je známo, že dokáže v průměru snížit velikost grafu o 7% (teoreticky až o 50%). S tím souvisí další náklady přidáním příznaku na každou hranu (většinou 1 bit, někde se používala jako maska poslední (low) bit adresy uzlu). V případě přítomnosti negovaných hran je zapotřebí pouze list 1, protože 0 může být reprezentována jako komplement (negace) listu 1. Na obrázku jsou negované hrany označeny tečkovanou čarou.



Obrázek 7: Demonstrace použití negovaných hran

Hlavním problémem, kterému musíme při použití negovaných hran čelit, je ztráta kanonické reprezentace. Jeden ze způsobů k obnovení kanonicity je zakázaní negovaných hran v určitých pozicích. Existuje 8 možností, jak umístit negované hrany, z nichž 4 páry jsou vždy ekvivalentní. Další obrázek ukazuje tyto 4 páry na uzlech ROBDD. Negovaná hrana je zde označena kolečkem.



Obrázek 8: Znázornění 4 párů ekvivalentních negovaných hran

Obnovení kanonicity dosáhneme tím, že high(v) hranu nebudeme nikdy označovat jako negovanou. Podle obrázku tedy použijeme vždy levou část daného páru.

3.13 Použití BDD

BDD byly v praxi nasazeny především v oblasti návrhu, verifikace a testování digitálních číslicových systémů. Nedávno se jejich použití rozrostlo do dalších odvětví. Například technologie pevné desetinné čárky používaná při analýze stavových automatů může být použita pro nesčetné množství problémů v matematické logice a formálních jazyků, pokud se jedná o konečnou množinu. V oblasti umělé inteligence vědci vymysleli pravdivostní systém založený na ROBDD.

4 Popis jednotlivých BDD balíčků

V této sekci postupně představím mnou nastudované balíčky (packages) pro manipulaci s BDD.

4.1 CUDD

Autorem tohoto balíčku je Fabio Somenzi. Spolu se svými kolegy z Colorádské univerzity tento balík už od roku 1996 stále vyvíjejí. Postupnými úpravami dosáhli vynikajících paměťových a výkonnostních výsledků. Balíček CUDD přináší funkce pro práci s binárními rozhodovacími diagramy (BDD), algebraickými rozhodovacími diagramy (ADD) a binárními rozhodovacími diagramy s potlačenou nulou (ZDD).

Balíček umožňuje širokou škálu operací s BDD, ADD a ZDD, jejich vzájemný převod mezi sebou a navíc implementuje sadu metod a algoritmů pro zlepšení upořádání proměnných.

Tento balíček muže být použit třemi způsoby:

- Tzv. Černá skřínka: V tomto případě aplikace určená pro práci s rozhodovacími diagramy pouze používá exportované funkce tohoto balíčku popsané v nápovědě. Postačující sada těchto funkcí umožňuje většině programátorům psát tímto stylem své aplikace a nemusí se zabývat metodikou uspořádávání proměnných.

- Průhledná skřínka: Při psaní náročnějších projektů zabývajících se problematikou rozhodovacích diagramů je vzhledem k efektivitě nutné přidat do balíku vlastní funkci místo použití stávajících exportovaných a vnitřních funkcí.

- Rozhraní: Objektově orientované jazyky jako C++ a Perl5 umožňují programátorům nestarat se o správu paměti. Rozhraní C++ je zahrnuto přímo v distribuci. Rozhraní Perl5 je distribuované samostatně. Jako poslední novinka se objevilo i rozhraní pro Javu, které ovšem nepochází z Coloradské univerzity.

Balíček je dispozici ke stažení v poslední verzi 2.4.1 přímo z FTP vlsi.Coloradu.EDU. Je psán v jazyce C/C++ a je určen pro Linuxové operační systémy. Návod pro zprovoznění je k dispozici v manuálu.

Nahlédneme do struktury CUDDu.

4.1.1 Reprezentace uzlu

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

V proměnné index je uložen název proměnné, podle které je pojmenován uzel. Index je neměnný atribut odrážející pořadí vzniku uzlů. Index 0 odpovídá uzlu vytvořenému jako první. V proměnné ref je uložen počet odkazů na tento uzel. Jakmile klesne na 0, je Garbage Collectorem vyhodnocen jako nepoužívaný a je odstraněn. Pod proměnnou next se ukrývá ukazatel na další uzel v unique table. Ve struktuře union je uložen ukazatel na potomky v případě neterminálních uzlu či v jiném případě obsahuje hodnotu uzlu v případě terminalních uzlů (listů).

4.1.2 Manažer

Všechny uzly používané v BDD, ADD a ZDD jsou uloženy ve speciální hashovací tabulce (unique table). BDD a ADD sdílejí stejnou tabulku, zatímco ZDD používají svou vlastní. Tyto tabulky společně s dalšími strukturami tvoří DdManager (zkráceně manažer). Aplikace používající pouze exportované funkce (varianta černá krabička) nemusí znát poměrně rozsáhlou strukturu manažeru. Každá aplikace musí nejprve zavoláním Cudd_init() manažer zinicializovat, funkce vrací ukazatel na všechny funkce pracující s rozhodovacími diagramy.

S výjimkou pár statických čítačů nejsou v CUDDu použity globální proměnné. Je tedy možné mít více aktivních manažerů paralelně.

4.1.3 Cache

K efektivnímu rekurzivnímu zpracovávání rozhodovacích diagramů potřebujeme tabulku pro uchovávání dočasných výpočtů. Zde se tato tabulka nazývá cache, jelikož se s ní pracuje jako s cache proměnné, ale omezené délky. Uživatel může nastavovat správu této tabulky, avšak pro většinu aplikací postačí defaultní nastavení.

4.1.4 Příklad pro vytvoření jednoduchého BDD

Vytvoříme funkci f=x1*x2+x1'*x2' a realizujeme ji například jako f = f1 + f2.

```
#include <stdio.h>
#include "include/cudd.h"
#include "include/util.h"
int main() {
DdManager *manager;
DdNode *f1,*f2,*f,*tmp,*var;
int i;

manager=Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);

f1=Cudd_ReadZero(manager);
Cudd_Ref(f1);

for(i=1;i>=0;i--){
var=Cudd_bddIthVar(manager,i);
tmp=Cudd_bddAnd(manager, var,f1);
Cudd_Ref(tmp);
Cudd_RecursiveDeref(manager,f1);
f1=tmp;
}

f2=Cudd_ReadZero(manager);
Cudd_Ref(f2);

for(i=1;i>=0;i--){
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);
Cudd_Ref(f);
Cudd_RecursiveDeref(manager,f1);
Cudd_RecursiveDeref(manager,f2);
Cudd_Quit(manager);
return 0;
}
```

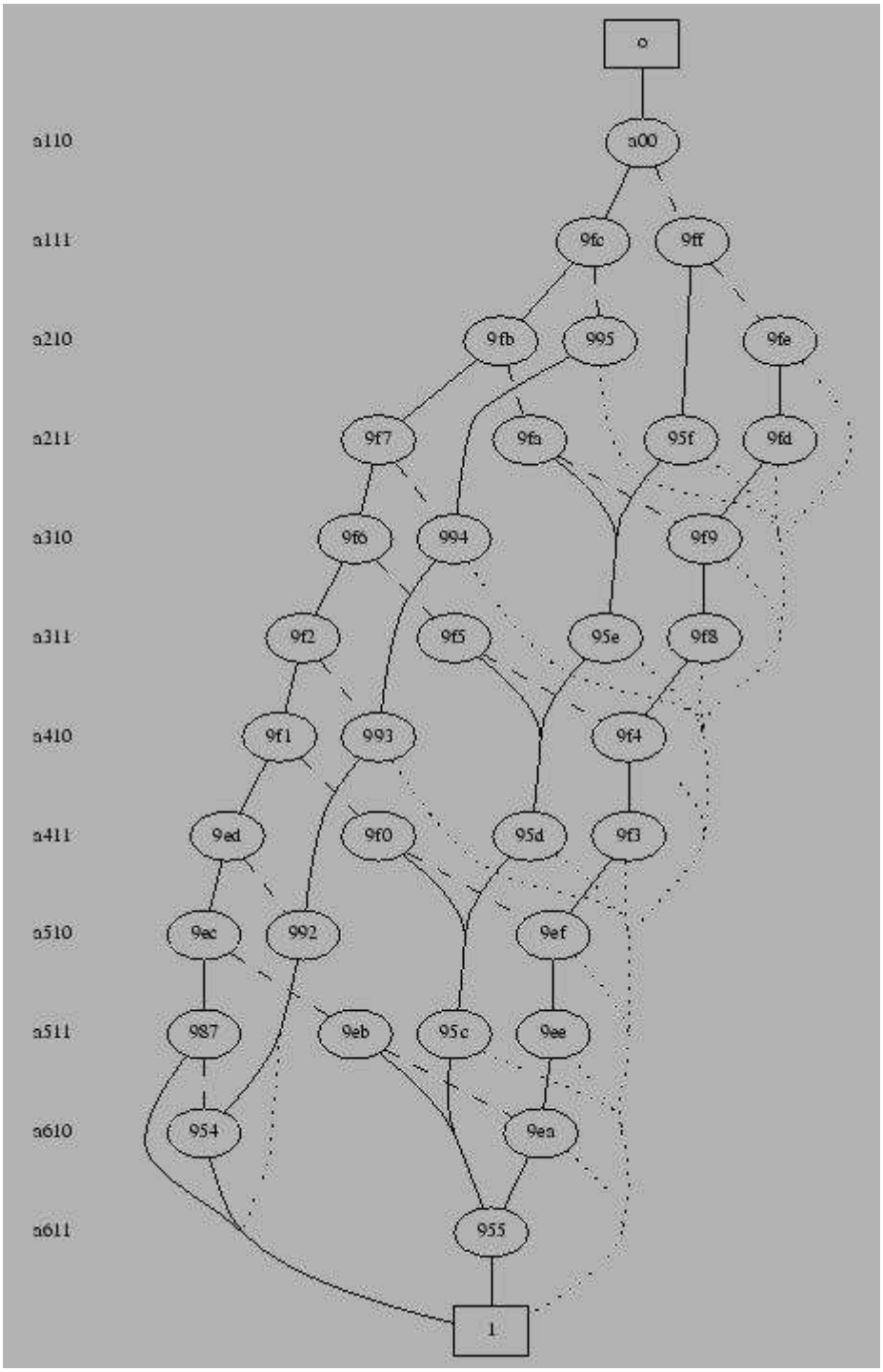
Vytvoříme si manažer a jednotlivé instance DdNode. Poté zinicializujeme manažer voláním funkce Cudd_Init(). Funkci f1 deklarujeme jako konstantní funkci s hodnotou 1 voláním Cudd_ReadOne(). Volání Cudd_Ref() inkrementuje proměnnou reference popsanou ve struktuře DdNode. Dvě proměnné vytvořím ve for cyklu zavoláním funkce Cudd_bddIthVar(). Tato funkce vrátí proměnnou na indexu i, v našem případě, kde proměnná neexistuje, vytvoří proměnnou a uloží ji na pozici i a vrátí ukazatel na ni. Pomocí Cudd_bddAnd() vytvoříme logický součin f1 (tedy log. 1) a námi vytvořené nové proměnné. Voláním Cudd_RecursiveDeref(), které snižuje hodnotu proměnné reference daného uzlu o jedna. (Volá se rekurzivně i na potomky, takže je možné použít ji pro zbavení se nepotřebných uzlů). Na konci for cyklu máme v proměnné f1 uloženou funkci $x_0 \cdot x_1$.
Druhý for cyklus je téměř stejný s rozdílem, že jako parametr funkce pro logický součin používáme další funkci pro logickou negaci, a to Cudd_Not(var). Na konci for cyklu máme v f_2 uloženo $(\neg x_0) \cdot (\neg x_1)$. V dalším kroku jednotlivé BDD spojíme v jeden pomocí operace OR voláním funkce Cudd_bdd_Or(). Zbývá uvolnit nepotřebné f1 a f2 a ukončit manažer.

4.1.5 Vstup

Programátor si vstup zajišťuje sám. K dispozici je rozšíření pro načítání BDD ze souboru ve vlastním formátu. PLA není podporován.

4.1.6 Výstup

Jednotlivá BDD je možné zapsat v několika formátech. Jedním z nich je bliff, kde diagramy jsou zapisovány jako soustavy multiplexorů, přičemž každému z nich odpovídá vnitřní uzel diagramu. Dále existuje export pro programy Dot a DaVinci, které jsou určeny pro zobrazování grafů. Specialitou je zápis výstupu do programu DDcal. Baliček má k dispozici funkce pro výstupy v různých faktorizovaných formách.



Obrázek 9: Ukázka výstupu z programu dot.

4.1.7 Závěrem

CUDD je univerzální balíček pro práci s rozhodovacími diagramy. Je součástí mnohých systémů pro logickou syntézu. CUDD má poměrně podrobnou a především dobře zpracovanou dokumentaci, včetně uživatelského a programátorského manuálu. Velkou zajímavostí je bohatá škála funkcí pro řazení proměnných. Programátor může volit mezi žádným řazením, náhodným, použitím metody síta, plavoucího okénka nebo dokonce genetických algoritmů. Pro CUDD je též napsáno spoustu dalších rozšíření. Za zmínku stojí Perl a Java interface, ukládání a načítání BDD diagramů do souboru, DDcal (grafická kalkulačka).

4.2 BuDDy

Buddy je další v řadě balíčků pro manipulaci s BDD. Tento balíček vytvořil J. Lind-Nielsen jako součást své doktorandské práce na univerzitě v Kadani. Balíček prošel velkým vývojem. V prvopočátku se jednalo pouze o jednoduchý nástroj demonstrující principy BDD, ve své poslední verzi 2.2 se však již jedná o plnohodnotný balíček se všemi standardními operacemi, řazením a dokumentací. Balíček je napsán v C / C++ a je určen pro kompilaci v prostředích Unix a Windows NT.

4.2.1 Struktura uzlu

```
typedef struct s_BddNode
{
  unsigned int refcou : 10;
  unsigned int level  : 22;
  int low;
  int high;
  int hash;
  int next;
} BddNode;
```

Každý uzel obsahuje počet referencí, které na něj vedou, proměnná level ukazuje pozici proměnné v současné změně pořadí, high a low reprezentují hrany, hash je ukazatel na kořen stromu a next je ukazatel na další uzel. Takováto struktura zabere celkem 20 bytů.
Jiné balíčky používají jen 16 bytů, ale zase musí obsahovat další hashovaní tabulku, takže se celková velikost poté stejně rovná v průměru 20-ti bytům na uzel.

4.2.2 Implementace

Všechny uzly jsou uloženy podobně jako v CUDDu ve velké poli, které se používá zároveň jako hashovací tabulka při hledání uzlů. I zde existuje hashovaní tabulka, která urychluje provádění operací nad BDD. Každá aplikace musí na začátku inicializovat správu paměti. To se děje zavoláním funkce bdd_init(modenum, cachesize). Funkce má dva parametry, modenum je počáteční počet BDD uzlů, cachesize nastavuje velikost paměti pro provádění operací.

4.2.3 Příklad vytvoření jednoduchého BDD pomocí BuDDy

```
#include <bdd.h>

main(void)
{
        bdd f1,f2,f, tmp1, var;
        bdd_init(1000,100);
        bdd_setvarnum(5);

f1 = bdd_false();
bdd_addref(f1);

for(i=1;i>=0;i--){
var=bdd_ithvar(i);
tmp=bdd_apply(f1,var,bddop_and));
```

```
bdd_addref(tmp);
bdd_delref(f1);
f1=tmp;
}

f2 = bdd_false();
bdd_addref(f2);

for(i=1;i>=0;i--){
var=bdd_ithvar(i);
tmp=bdd_apply(f2,bdd_not(var),bddop_and));
bdd_addref(tmp);
bdd_delref(f2);
f2=tmp;
}

f=bdd_apply(f1,f2,bddop_or));
bdd_addref(f);
bdd_delref(f1);
bdd_delref(f2);
bdd_done;
}
```

Konstrukci jsem záměrně použil podobnou té v CUDDu. Na začátku zinicializujeme Správu paměti pomocí bdd_init(). Ihned po init následuje volání bdd_setvarnum(), které nám říká, kolik proměnných budeme v dané session používat. (Může být později upraveno voláním bdd_setvarnum() nebo bdd_extvarnum()). F1 deklarujeme jako konstantní 0 bdd_false(), přidáme referenci bdd_addref(). Ve for cyklech opět vytvoříme proměnné a spojíme je pomocí tmp = bdd_apply(f1,var,bddop_and). Nepotřebné uzly dereferencujeme a do výsledné f1 přiřadíme naše požadované $x_1 \cdot x_2$. To samé provedeme s funkcí f2, kde navíc voláme bdd_not() pro vytvoření komplementu. Jednotlivé funkce f1 a f2 voláním f=bdd_apply(f1,f2,bddop_or)) spojíme do f. Práci s balíčkem ukončíme pomocí bdd_done.

4.2.4 Vstupy

Načítání BDD je umožněno pouze z vnitřního formátu uloženého pomocí funkce bdd_save. PLA není podporován.

4.2.5 Výstupy

Výstup BDD lze zapsat do souboru, jak je zmíněno výše. Je podporován výstupní formát pro program Dot a lze exportovat i různé pravdivostní tabulky.

4.2.6 Závěr

Buddy je kompletní balík pro práci s Binárními rozhodovacími diagramy. Obsahuje poměrně rozsáhlou škálu funkcí pro manipulaci s BDD, podporuje řazení proměnných. K dispozici jsou rozhraní pro manipulaci s konečnými doménami (umožňující reprezentaci integerovských hodnot místo booleovských). Podporuje práci s Booleovskými vektory. Zajímavé je velmi dobře propracované C++ rozhraní, které zefektivňuje práci s tímto balíčkem. Nepodporuje negované hrany.

4.3 CAL

Tento balíček napsali autoři Raajev Ranjan a Jagesh Sanghavi. Jedná se o volně distribuovaný balíček pro práci s BDD založený na algoritmu do šířky. Garbage Collector je založen na počítaní referencí. V zájmu zmenšení paměťových nároků a udržení velikosti struktury node na 4 wordech, je použita technologie "bit tagging" (jedná se o vymaskování adresy uloženého uzlu) pro určování počtu referencí.

4.3.1 Struktura uzlu

```
struct CalBddNodeStruct {
  CalBddNode_t *nextBddNode
  CalBddNode_t *thenBddNode;
  CalBddNode_t *elseBddNode;
  Cal_BddId_t thenBddId;
  Cal_BddId_t elseBddId;
};
```

4.3.2 Příklad vytvoření jednoduchého BDD v CAL

```
#include <stdio.h>
#include "cal.h"

int main() {
Cal_BddManager manager;
Cal_Bdd  f1,f2,f,tmp,var;
int i;

manager=Cal_BddManagerInit();

f1=Cal_BddOne(manager);
Cal_BddUnFree(manager, f1);

for(i=1;i>=0;i--){
Cal_BddManagerCreateNewVarAfter(manager,var);
tmp=Cal_BddAnd(manager, var,fAdd);
Cal_BddUnFree(manager, tmp);
 Cal_BddFree(manager, f1);
f1=tmp;
}

f2=Cal_BddZero(manager);
```

```
Cal_BddUnFree(manager, f2);

for(i=1;i>=0;i--){
Cal_BddManagerCreateNewVarAfter(manager,var);
tmp=Cal_BddAnd(manager, Cal_BddElse(manager, var),fAdd);
Cal_BddUnFree(manager, tmp);
 Cal_BddFree(manager, f2);
F2=tmp;
}

f=Cal_BddOr(manager,f1,f2);
Cal_BddUnFree(manager,f);
Cal_BddFree(manager, f1);
Cal_BddFree(manager, f2);
Cal_BddManagerQuit(manager);
return 0;
}
```

4.3.3 Závěr

Cal je dalším reprezentantem v rodině velkých balíků. Obsahuje velké množství funkcí pro manipulaci s Binárními Rozhodovacími Diagramy. Bohužel velkým nedostatkem tohoto balíku je absence dokumentace. Nejsou k dispozici ani ukázkové příklady, jak vytvořit BDD.U takto velkého balíku bych dokumentaci přinejmenším očekával. Zajímavostí je výstavba BDD algoritmem do šířky, která by mohla přinést jistou úsporu pamětových nároků.

4.4 CMU

Tento menší balíček pro práci s BDD vytvořili na univerzitě Carneige Mellon. Obsahuje necelých 100 funkcí pro manipulaci s BDD.

4.4.1 Ukázka práce s CMU

```
Int main {
bdd_manager bddm;
 bdd_init(bddm);

bdd f, var1, var2;
f = bdd_one;
var = bdd_var_with_index(bddm, 0);
var = bdd_var_with_index(bddm, 1);

f= bdd_or(bddm, var1, var2)

bdd_quit(bddm);
return 1;
}
```

4.4.2 Závěr

Balíček podporuje všechny základní operace, jsou zde implementovány metody pro variable ordering. Zajímavostí je podpora vícestavových diagramů MTBDD (Multi Terminal Binary Decision Diagram) (tedy s listy, které můžou nabývat i hodnot různých od 0 a 1). Podporuje funkce pro ukládání a nahrávání BDD ze souboru. PLA není podporován. Je velmi malý a poměrně rychlý. Je kompilovatelný pod platformami Unix. K dispozici je pouze seznam funkcí s velmi slabou dokumentací.

4.5 Další zajímavé balíky

ABCD:

Autor: Armin Biere

Jedná se o experimentální balíček pro manipulaci s BDD. Zajímavosti, které tento balíček nabízí, jsou jinak řešený Garbage Collector, odlišné integrace unikátních haschovacích tabulek uzlů pomocí otevřeného adresování a indexové (nikoliv pointrové) odkazy na

jednotlivé uzly. To umožnilo snížit velikost struktury uzlu na polovinu (2 word misto 4). Navíc, aby se vyhnul "shlukování" (clustering), je použito kvadratické posloupnosti pro řešení hashovacích kolizí.

JAVABDD:

Autor: John Whaley

Experimentální balíček pro práci s BDD kompletně napsaný v Javě. Vychází z balíčku BuDDY a může sloužit i jako interface k balíčkům: BuDDY, CUDD, CAL. Zajímavostí je, že na rozdíl od jiných java implementací si udržuje poměrně dobrou paměťovou náročnost.

5 PLA

Dalším z úkolu byla úprava balíků pro načítání BDD ze souboru ve formátu PLA (Programable Logic Array). Tento formát je používán programy pro popis fyzické implementace.

5.1 Popis PLA

Řádky začínající znakem # jsou komentáře. Řádky začínající tečkou („.“) nesou informaci o PLA.

.i - udává počet vstupních proměnných
.o - udává počet výstupních proměnných

.p - udává počet termů.

.ilb - implicitní pojmenování vstupních proměnných
.ob - implicitní pojmenování výstupních funkcí.

Předchozí 3 parametry nejsou povinné.
Poté obsahuje pravdivostní tabulku reprezentující jednotlivé proměnné a funkce. V tabulce proměnných nalézáme tři stavy 0,1 a -. „-„ reprezentuje takzvaný DC (doncare stav). Daná proměnná není použita ve výrazu funkce, při „1“ je použita v normálním a při „0“ v negovaném stavu.

.e - konec PLA

5.2 Imlementace načítání

Načítání jsem naprogramoval v jazyce C/C++, aby jej bylo možné použít pro všechny balíky nehledě na prostředí a bylo pro tento úkol mnohem efektivnější. Pomocí jednoduchého lexikálního analyzátoru rozpoznávám jednotlivé elementy PLA souboru a tyto informace ukládám do vlastní pomocné struktury PlaData.

```
typedef struct {
        int i;
        int o;
        int p;
        int allocated;

        unsigned short** iArray;
        unsigned short** oArray;

} PlaData;
```

Intuitivně proměnná i značí počet vstupních proměnných, o počet výstupních funkcí, p počet termů. Proměnná alocated je pouze pomocná proměnná pro načítání PLA souborů s neznámým počtem termů. Pole iArray je maticový zápis pravdivostních hodnot vstupů, kde 2 značí DC stav. Podobně oArray je maticový zápis výstupů.

5.3 Vytvoření BDD z vnitřní formy

Vytváření probíhá velice podobně ve všech balíčcích, liší se pouze v názvu volaných funkcí. Nejprve si vytvořím pole, ve kterém jsou uloženy ukazatele na jednotlivé výstupní funkce a nastavím je na hodnotu log. 0. Poté už pouze pro každý řádek vstupní matice vytvářím jednotlivé malé BDD. To si uložím do dočasné proměnné a podívám se do výstupní matice. Je-li na daném řádku pro zvolenou funkci uložena hodnota log. 1, tak tento malý BDD pomocí operátoru Apply_OR() sloučím s příslušnou funkcí. V případě že narazím na hodnotu log. 0, neprovádím nic. Takto postupně vystavím BDD reprezentující jednotlivé funkce.

5.3.1 Příklad vytvoření BDD pomocí CUDDu psaném v C

```
int Cudd_PlaLoad(DdManager *manager, char *path, BddArray *result){

DdNode *tmp,*var,*fAdd;
PlaData data;
PlaOpen(path, &data);
result->fRoot = (DdNode**)malloc(data.o*sizeof(DdNode));
result->i = data.i;
```

```
result->o = data.o;
for(int i=0;i<data.o;i++) result->fRoot[i] = Cudd_ReadLogicZero(manager);
  for (i=0; i<data.p;i++){
  fAdd=Cudd_ReadOne(manager);
  Cudd_Ref(fAdd);
      for (int j=0;j<data.i;j++){
      if (data.iArray[i][j]==1) tmp=Cudd_bddAnd(manager, var,fAdd);
      else if (data.iArray[i][j]==0) tmp=Cudd_bddAnd(manager, Cudd_Not(var),fAdd);


      if(data.iArray[i][j]== 1 || data.iArray[i][j]==0) {
        Cudd_Ref(tmp);
        Cudd_RecursiveDeref(manager, fAdd);
        fAdd = tmp;
        }
      }
        for(j=0;j<data.o;j++){
                if(data.oArray[i][j] == 1) {
                tmp = Cudd_bddOr(manager,result->fRoot[j],fAdd);
        Cudd_Ref(tmp);
                Cudd_RecursiveDeref(manager, result->fRoot[j]);
                result->fRoot[j] = tmp;
        }
        }
        Cudd_RecursiveDeref(manager, fAdd);
    }
        return 1;
}
```

V následující kapitole Testy uvidíme, že mnou implementovaná metoda není zcela vhodná pro tento balíček. Efektivnější implementaci uvedl Ondřej Kološ ve své bakálářské práci „Port programového balíku pod platformu Windows“.

5.3.2 Příklad vytvoření BDD pomocí BuDDy s využitím C++

```
int BuddyPlaLoad(char *path, BddArray *result){

bdd tmp,var,fAdd;
        PlaData data;
        PlaOpen(path, &data);
        result->fRoot = new bdd[data.o];

        fAdd=bdd_true();
        result->i = data.i;
        result->o = data.o;
        bdd_setvarnum(result->i+10);
```

```
        for(int i=0;i<data.o;i++) {
                result->fRoot[i] = bdd_false();
        }
        for (i=0; i<data.p;i++){
                fAdd = bdd_true();
                for (int j=0;j<data.i;j++){
                        var = bdd_ithvarpp(j);
                        if (data.iArray[i][j]==1) tmp= var &fAdd;
                        else if (data.iArray[i][j]==0) tmp=bdd_not(var) & fAdd;

                        fAdd = tmp;
                }
                for(j=0;j<data.o;j++){
                        if(data.oArray[i][j] == 1)  result->fRoot[j] |= fAdd;
                }
        }
        return true;
}
```

Pro ukázku rozhraní C++ jsem si zvolil baliček BuDDy. V příkladu vidíme značné úspory kódu. Dále pozorujeme využití operátorů nahrazující zápis pomocí volání funkcí. V neposlední řadě výhoda C++ spočívá v tom, že odpadl problém s počítáním referencí.

5.3.3 Příklad vytvoření BDD pomocí Cal v C

```
int CalPlaLoad(Cal_BddManager manager, char *path, BddArray *result){

    Cal_Bdd tmp,var,fAdd;
    PlaData data;
    PlaOpen(path, &data);
    result->fRoot = new Cal_Bdd[data.o];

    fAdd=Cal_BddOne(manager);
    result->i = data.i;
    result->o = data.o;

    for(int i=0;i<data.o;i++) {
        result->fRoot[i] = Cal_BddZero(manager);
```

```
    }
    for (i=0; i<data.p;i++){
        fAdd=Cal_BddOne(manager);
        Cal_BddUnFree(manager, fAdd);
        for (int j=0;j<data.i;j++){
                Cal_BddManagerCreateNewVarAfter(manager,var);
                if (data.iArray[i][j]==1) tmp=Cal_BddAnd(manager, var,fAdd);
                else if (data.iArray[i][j]==0) tmp=Cal_BddAnd(manager,
Cal_BddElse(manager, var),fAdd);
                if(data.iArray[i][j]== 1 || data.iArray[i][j]==0) {
                        Cal_BddUnFree(manager, tmp);
                        Cal_BddFree(manager, fAdd);
                        fAdd = tmp;
                }
        }
        for(j=0;j<data.o;j++){
                if(data.oArray[i][j] == 1) {
                        tmp = Cal_BddOr(manager,result->fRoot[j],fAdd);
                        Cal_BddUnFree(manager, tmp);
                        Cal_BddFree(manager, result->fRoot[j]);
                        result->fRoot[j] = tmp;
                }
        }
        Cal_BddFree(manager, fAdd);
    }
    return 1;
}
```

6 Testy

V této sekci uveřejním testy provedené s jednotlivými balíčky spustitelnými pod operačním systémem Windows. Je velice těžké vymyslet nějakou metodu, která by efektivně porovnala jednotlivé balíky mezi sebou. Obsahují totiž velkou škálu různých přepínačů a nastavení, jež ovlivňují výslednou výstavbu a reprezentaci jednotlivých BDD. Počínaje velikostmi a nastavením Cache, metodami řazení a nastaveními různých typů Garbage Collectingu konče. Rozhodl jsem se proto, že porovnám pouze dobu výstavby jednotlivých BDD pomocí implementované metody pro načítání PLA a spočítám počet všech uzlů reprezentující tento diagram. Vše s defaultním nastavením všech možných proměnných. Testovací soubory přikládám na CD. Testy byly provedeny na počítači AMD Athlon XP 2400 + s operační pamětí 1024 RAM a prostředí MS Windows XP SP2 Professional.

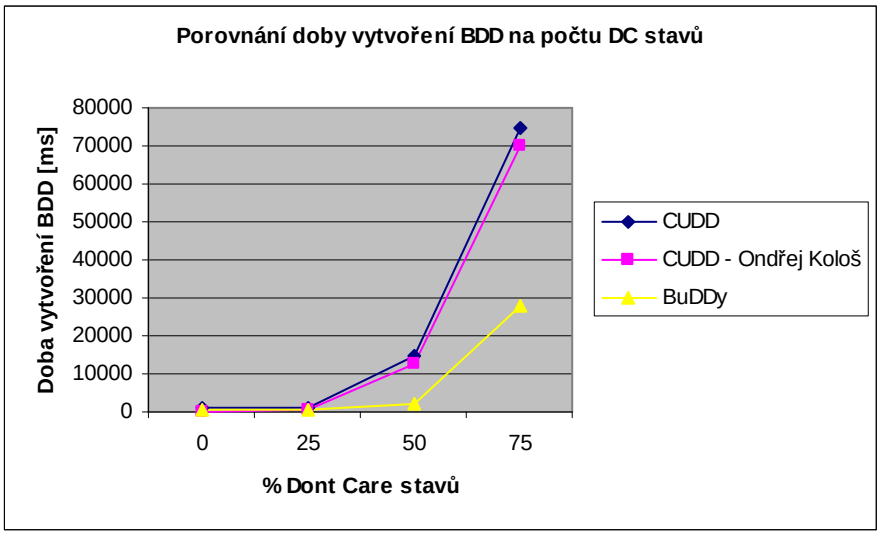
Bohužel se mi nepodařilo zprovoznit porty balíčků CAL a CMU pod operačním systémem Windows. Využíval jsem knihovnu winglu 1.4 (port do MS VSC++6), kterou jsem rozšířil o metody načítání PLA souborů. Nicméně při vytváří samostatné knihovny jsem narazil na problém s knihovnou mem (storage management library) využívající linuxové funkce. Pro úspěšný port by bylo nutné celou mem přepsat, což přesahuje rámec této BP.
V testování jsem se omezil tedy pouze na balíčky BuDDy a CUDD, u kterých proběhl port úspěšně. Do testů jsem zahrnul i práci Ondřeje Kološe, který portoval CUDD pod windows a rovněž implementoval načítání PLA.

6.1 Test náročnosti načítání PLA souboru do vnitřní formy

Pomocí nástroje pro generování PLA souborů PLA Generátor 2.0 jsem vytvořil testovací soubory. Změřil jsem rychlosti načítání do vnitřní normy a zjistil, že ve většině případů načítání nešlo změřit, anebo trvalo zanedbatelně dlouho oproti době výstavby jednotlivých BDD.

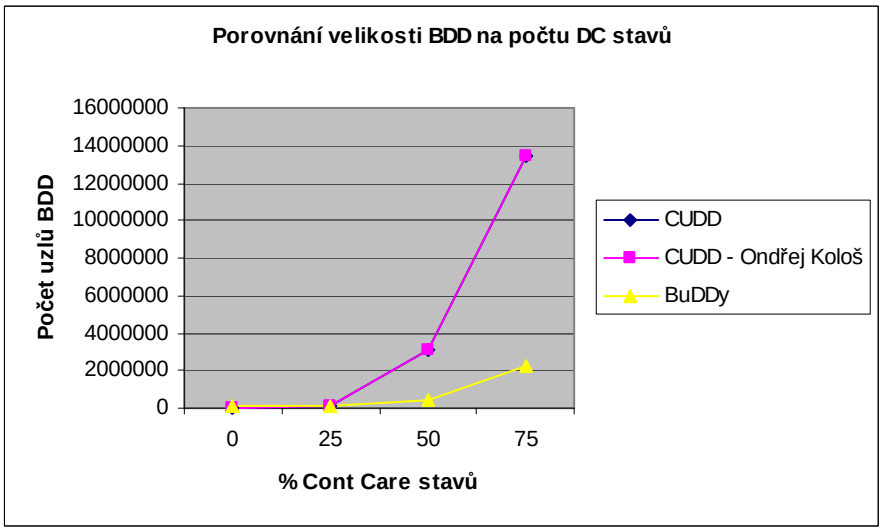
6.1.1 Testovací sada DC

Test odráží načítání PLA s % zastoupením Dont-Care stavů na vstupech (0%, 25%, 50%, 75%). Vstupní soubory PLA obsahují 50 vstupů, 15 výstupů a 1000 termů.



Obrázek 10: Graf závislosti doby vytvoření BDD na % DC stavů

Z grafů je vidět, že náročnost načítání PLA se zastoupením Dont-Care stavů větším než 50% vyvolává značnou výpočetní zátěž. To je způsobeno obrovským nárůstem počtu uzlů reprezentující dané BDD, což nám ukazuje následující tabulka.

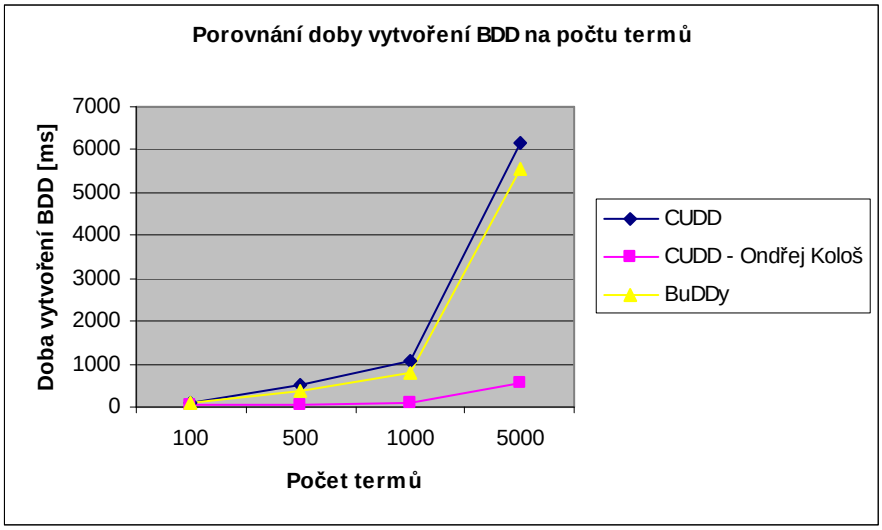


Obrázek 11: Graf závislosti velikosti BDD na počtu DC stavů

Vidíme, že reprezentace BDD s vysokým % DC stavů mnou navrhnutou metodou načítání činní balíčku CUDD nemalé výpočetní i paměťové problémy. Při 75% DC stavech CUDD reprezentuje dané BDD bezmála 13 milionů uzlů, kdežto u druhého balíčku BuDDy jen 2miliony.

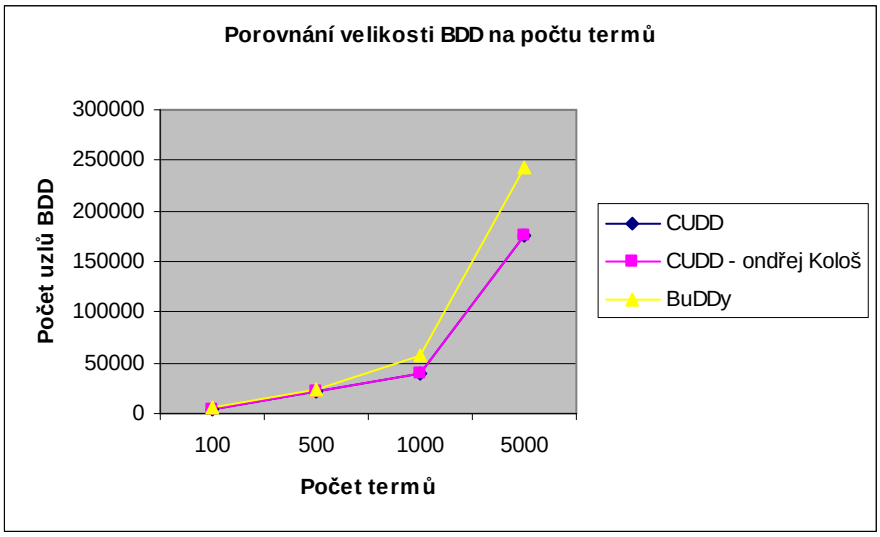
6.1.2 Testovací sada Termy

V tomto testu jsem podrobil balíky testům v závislosti na vzrůstajícím počtu termů. Vstupní soubory mají 50 vstupů, 15 výstupů a DC stav 0%.



Obrázek 12: Graf závislosti doby vytvoření BDD na počtu termů

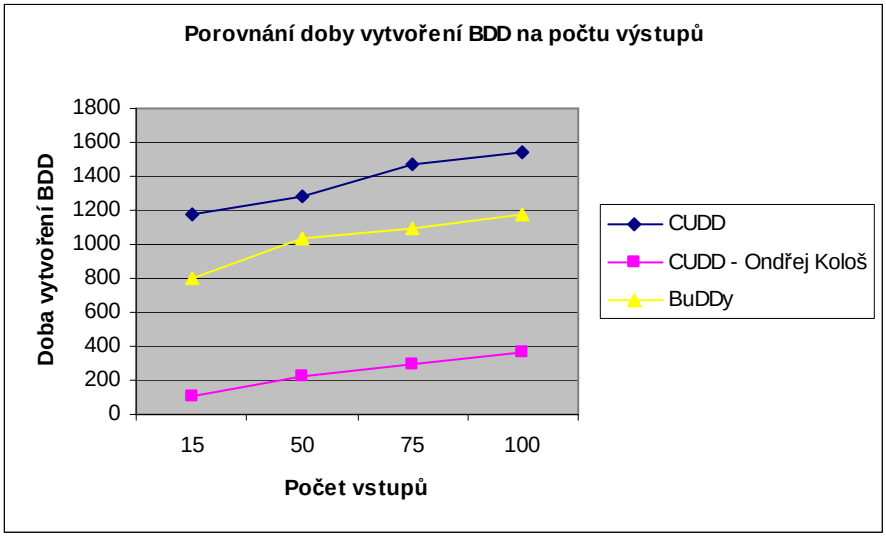
Zde jsou mimo jiné vidět nedostatky způsobené mým návrhem načítací funkce. Problém je především v tom, že při každé iteraci druhého vnořeného for cyklu vymažu proměnou tmp, kde mám uložený malý BDD reprezentující řádek v matici vstupů (tím nemohu v dalších krocích tento BDD využít, protože ho Garbage Collector smaže). Vhodnější by bylo si tyto řádky uschovat v nějakém poli a poté skládat jednotlivé funkce. Je zajímavé, že při použití souborů s hodnotou DC stavů 50% a více dosahují obě implementace načítání v CUDDu stejných hodnot.



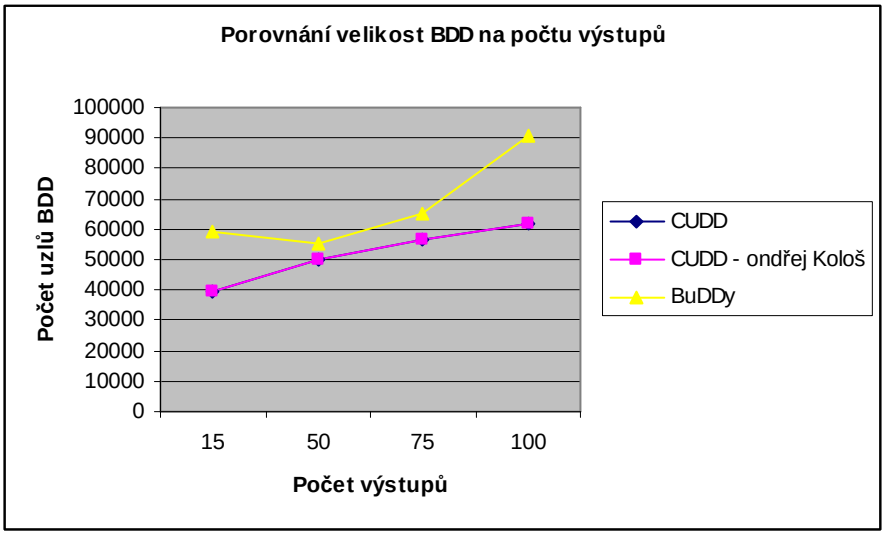
Obrázek 13: Graf závislosti velikosti BDD na počtu termů

Mírný nárůst počtu uzlů v balíčku Buddy pro plně určené funkce je způsoben tím, že neobsahuje implementaci negovaných hran.

6.1.3 Testovací sada Výstupy



Obrázek 14: Graf závislosti doby vytvoření BDD na počtu vstupů



Obrázek 15: Graf závislosti velikosti BDD na počtu výstupů

7 Závěr

V rámci této bakalářské jsem nejprve celkem podrobně rozebral teorii týkající se Binárních Rozhodovacích Diagramů. Dále jsem představil jednotlivé BDD balíčky. Stručně jsem popsal struktury v nich použité, uvedl krátký příklad práce s nimi a pokusil se v závěrech zhodnotit, čím se liší od jednotlivých implementací.

Dalším výsledkem této bakalářské práce je vznik dvou portů balíčků CUDD a BuDDy do prostředí Windows a v nich zahrnutá implementace načítací funkce z PLA souborů.

V závěru jsem pomocí knihoven těchto balíčků ověřil funkčnost načítání a výstavby BDD ze souborů.

Ve výsledném zhodnocení se nabízí hned několik možností rozšíření v budoucnosti:

- Zahrnout do studie a testů i balíčky pracující pod Platformou Unix, zaměřit se především na práci s pamětí.
- Vylepšit náročnost funkce pro načítání BDD z formátu PLA především pro funkce v vysokým počtem DC stavů.
- Portnout balíky CAL a CMU do operačního systému Windows nebo vytvořit vlastní implementaci.

8 Použitá literatura

[1] S. B Akers, Binary Decision Diagrams, 1978

[2] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, 1986

[3] R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, 1992

[4] K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, 1990

[5] H. R. Andersen, An Introduction to Binary Decision Diagrams, 1998

[6] CH. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design, 1998

[7] J. L. Nielsen, Binary Decision Diagram package Release 2.2, 2002

[8] F. Dimenzi, CUDD: CU Decision Diagram Package Release 2.4.1, 2005

[9] B. Yang, R. E. Bryant, D. R. O´Hallaron, A. Biere, O. Coudert, G. Janseen, R. K. Ranjan, F. Somenzi, A Performance Study of BDD-Based Model Checking

[10] R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, 1993

[11] A. Biere, Effiziente Modellprüfung des µ-Kalküls mit binären Entscheidungsdiagrammen, 1997

[12] R. P. Jakobi, A Study of the Aplication of Binary Decision Diagrams to Multi-level Logic Synthesi, 1993

[13] D. Beyer, A. Noack, CrocoPat 2.1 Introduction and Reference Manual, 2004

[14] C. Y. Lee, Representation of Switching Circuits by Binary-Deciosion Programs, 1959

[15] http://en.wikipedia.org/wiki/Binary_decision_diagram

[16] http://service.felk.cvut.cz/vlsi/prj/PLA_gen/

Seznam tabulek:

Seznam obrázků:

8. Seznam zkratek

ADD Algebraic Decision Diagram
algebraický rozhodovací diagram

BuDDy BDD balíček od J. L Nielsena

BDD Binary Decision Diagram
binární rozhodovací diagram

BDT Binary Decision Tree
Binární rozhodovací strom

CAL BDD balíček z univerzity v Berkeley

CMU BDD balíček z univerzity Carnegie Mellon

CUDD Colorado University Decision Diagram (package)
*Balík z Coloradské univerzity týkající se rozhodovacích diagram*ů

DC Dont-Care

DD Decision diagram
Rozhodovací diagram

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

OBDD Ordered Binary Decision Diagram
uspořádaný binární rozhodovací diagram

ROBDD Reduced Ordered Binary Decision Diagram
redukovaný uspořádaný binární rozhodovací diagram

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