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

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Bakalářská práce:

Port programového balíku CUDD pod platformu Windows

Vypracoval: Ondřej Kološ

Vedoucí práce: Ing. Petr Fišer

2006

Poděkování

Mé poděkování patří Ing. Petru Fišerovi za odborné vedeni, trpělivou pomoc a veškerý čas, který mi věnoval při zpracovávání této bakalářské práce. Také mu děkuji za jeho četné připomínky, které přispěly ke zkvalitnění textu.

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. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 20. 6. 2006

.....
podpis

Anotace

Tato bakalářská práce se zabývá převodem balíku CUDD (Colorado University Decision Diagram package) pod operační systém Microsoft Windows. Balík CUDD umožňuje pomocí mnoha funkcí manipulaci s binárními rozhodovacími diagramy (BDD), algebraickými rozhodovacími diagramy (ADD) a binárními rozhodovacími diagramy s potlačenou nulou (ZDD). V práci je nejprve probrána teorie binárních rozhodovacích diagramů, dále je zde i krátká zmínka o binárních rozhodovacích diagramech s potlačenou nulou a o algebraických rozhodovacích diagramech. Dále v práci je popis CUDDu a práce s ním. Nejdůležitější částí je popis práce při zadaných úkolech – převod CUDDu pod operační systém Microsoft Windows, rozšíření objektového rozhranní a implementace funkce pro nahrávání souborů ve formátu *pla*.

Abstract

This bachelor project deals with a conversion of the CUDD package (Colorado University Decision Diagram Package) under Microsoft Windows operating system. The CUDD package provides routines for manipulation with binary decision diagrams (BDDs), algebraic decision diagrams (ADDs) and zero-suppressed binary decision diagrams (ZDDs). The theory on binary decision diagrams is discussed first, which is followed by a short reference to zero-suppressed binary decision diagrams and algebraic decision diagrams. Description of the CUDD package and of the way how to work with it is presented below. The most important part is the description of my work on given tasks – conversion of CUDD package under Microsoft Windows operating system, object interface extension and implementation of a function for reading *pla* files.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Balík CUDD (Colorado University Decision Diagram package) poskytuje mnoho funkcí pro manipulaci s binárními rozhodovacími diagramy (BDD), algebraickými rozhodovacími diagramy (ADD) a binárními rozhodovacími diagramy s potlačenou nulou (ZDD – Zero-suppressed decision diagrams). Skupina pracovníků kolem Fabia Sommenzi z Coloradské Univerzity začala na CUDDu pracovat v roce 1996. Celý CUDD je napsán v programovacím jazyku C a bylo k němu také naprogramováno objektové C++ rozhranní. Blíže o CUDDu viz. kapitola 4.

CUDD je původně určen pod platformu Linux. Mým úkolem bylo upravit zdrojové kódy tak, aby byl CUDD kompilovatelný po operačním systémem Windows a vytvořit knihovny (statické a dynamické). Popis práce při převodu je v kapitole 5.

Ve své bakalářské práci se zaměřuji na binární rozhodovací diagramy. Ty se ukázaly jako velmi výhodná reprezentace booleovských funkcí. Funkce jsou reprezentovány pomocí kořenových orientovaných acyklických grafů, které se skládají z tzv. rozhodovacích vnitřních uzlů a terminálních uzlů, které nabývají hodnot 0 a1. Binárními rozhodovacími diagramy se zabývám více v kapitole 2.

Nejprve se chci zabývat teorií vztahující se k binárním rozhodovacím diagramům.

1.1 Výroková logika

Výroková logika se zabývá usuzováním, zda je nějaký výrok pravdivý nebo nepravdivý. Výrokem je věta, o které má smysl říci, že je pravdivá nebo nepravdivá (přiřadíme jí pravdivostní hodnotu). Pro označení pravdivého výroku používáme symbol 1, pro označení nepravdivého výroku používáme symbol 0. Výroky můžeme spojovat pomocí logických spojek do složených výroků a podle jednotlivých výroků usuzovat, zda je celý složený výrok pravdivý či nikoliv. Používáme tyto spojky (v závorce je označen příslušný symbol): negace (¬, v textu čárka nad proměnnou – $\bar{x}$), konjunkce (∧, &, ·), disjunkce (∨, +), implikace ($\Rightarrow$,$\rightarrow$) a ekvivalence ($\Leftrightarrow$,$\leftrightarrow$). V tabulkách č. 1-5 jsou pravdivostní tabulky pro tyto spojky.

	¬
0	1
1	0

∧	0	1
0	0	0
1	0	1

∨	0	1
0	0	1
1	1	1

Tabulky č. 1-3: *Pravdivostní tabulky*

⇒	0	1
0	1	1
1	0	1

⇔	0	1
0	1	0
1	0	1

Tabulky č. 4-5: *Pravdivostní tabulky*

Formule výrokové logiky je řetězec (konečná posloupnost) logických proměnných a logických spojek a závorek, jestliže vznikla podle těchto pravidel:

- každá logická proměnná je formule
- jsou-li *α, ß* formule, je $(\overline{a})$, $(\alpha \wedge ß)$, $(\alpha \vee ß)$, $(\alpha \Rightarrow ß)$, $(\alpha \Leftrightarrow ß)$ také formule

Jestliže ve formuli α známe pravdivostní hodnoty všech logických proměnných, tak 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. $\alpha \Leftrightarrow \alpha$). Naopak kontradikce je formule nepravdivá ve všech pravdivostních ohodnoceních (např. $\alpha \wedge \overline{a}$). Splnitelná formule je taková formule, která je pravdivá alespoň v jednom pravdivostním ohodnocení (např. $\alpha \wedge ß$).

1.2 Normální formy

Pro normální formy (NF) nejprve zavedeme několik pojmů.

Literál je logická proměnná nebo negace logické proměnné. Minterm je logický výraz *n* proměnných, skládající se pouze z konjunkcí literálů. (např. $\overline{a}$, $\alpha \wedge ß$). Maxterm je logický výraz *n* proměnných, skládající se pouze z disjunkcí literálů (např. $\overline{a}$, $\alpha \vee ß$).

1.2.1 DNF

Booleovský výraz je v disjunktivní normální formě (DNF), jestliže se skládá z disjunkce konečně mnoha mintermů, tj. skládá se z disjunkce konečně mnoha konjunkcí skládajících se z logických proměnných a z negací logických proměnných:

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

kde každé t_{nm} je logická proměnná x_{nm} nebo negace logické proměnné $\overline{x_{mn}}$. Například $(x \wedge \overline{y}) \vee (\overline{x} \wedge y)$.

Jakýkoliv booleovský výraz můžeme zapsat v disjunktivní normální formě.

1.2.2 CNF

Booleovský výraz je v konjunktivní normální formě (CNF), jestliže se skládá z konjunkce konečně mnoha maxtermů:

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

kde každé t_{nm} je logická proměnná x_{nm} nebo negace logické proměnné $\overline{x_{mn}}$. Například $(x \vee \bar{y}) \wedge (\bar{x} \vee y)$.

Jakýkoliv booleovský výraz můžeme zapsat v konjunktivní normální formě.

1.2.3 If-Then-Else NF

Booleovský výraz je v If-Then-Else normálové formě (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:

$$t = x_1 \rightarrow t_1, t_0$$

$$t_0 = y_1 \rightarrow 0, t_{00}$$

$$t_1 = y_1 \rightarrow t_{11}, 0$$

$$t_{00} = x_2 \rightarrow t_{001}, t_{00}$$

$$t_{11} = x_2 \rightarrow t_{111}, t_{110}$$

$$t_{000} = y_2 \rightarrow 0,1$$

$$t_{001} = y_2 \rightarrow 1,0$$

$$t_{110} = y_2 \rightarrow 0,1$$

$$t_{111} = y_2 \rightarrow 1,0$$

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

Z příkladu vidíme, že některé výrazy jsou identické, například t_{001} a t_{111}, a daly by se sloučit do jednoho. O těchto úpravách a také o tom, jak se podle INF vytvářejí binární rozhodovací diagramy se budu zabývat v kapitole 2.

1.3 Shanonův expanzní teorém

Shanonnův expanzní teorém je založen na myšlence, že každá booleovská funkce může být rozložena takto:

$$f(x_1,x_2,...,x_n) = x_1 \cdot f_{x_1}(1,x_2,...x_n) + \overline{x_1} \cdot f_{x_1}(0,x_2,...,x_n)$$

Kde f je jakákoliv booleovská funkce a f_{x_1} je funkce f rozložená podle x_1 – v prním případě jsou ve funkci f nahrazeny všechny výskyty proměnné x_1 logickou jedničkou, v druhém případě jsou nahrazeny logickou nulou.

2 Binární rozhodovací diagramy

Booleovská funkce může být reprezentována například pravdivostní tabulkou, součinem maxtermů, součtem mintermů, apod. Velmi výhodnou reprezentací těchto funkcí se ukazují binární rozhodovací diagramy (Binary Decision Diagrams – BDD) a především jejich modifikace - uspořádané binární rozhodovací diagramy (Ordered Binary Decision Diagrams – OBDD) a redukované upořádané binární rozhodovací diagramy (Reduced Ordered Binary Decision Diagrams – ROBDD).

2.1 Historie

Poprvé byly binární rozhodovací diagramy představeny C. Y. Leem ve článku "Representation of Switching Circuits by Binary-Decision Programs" v časopise Bell Systems Technical Journal z roku 1959. Dále byly studovány S. B Akersem, který v roce 1978 napsal publikaci "Binary Decision Diagrams".

Plný potenciál efektivnosti binárních rozhodovacích diagramů prozkoumal až R. E. Bryant z Carnegie Mellon University. Ten v roce 1986 napsal publikaci "Graph-Based Algorithms for Boolean Function Manipulation" a v roce 1992 napsal "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams". Jeho největším přínosem bylo rozšíření BDD o používání neměnného pořadí proměnných a o používání sdílených podgrafů.

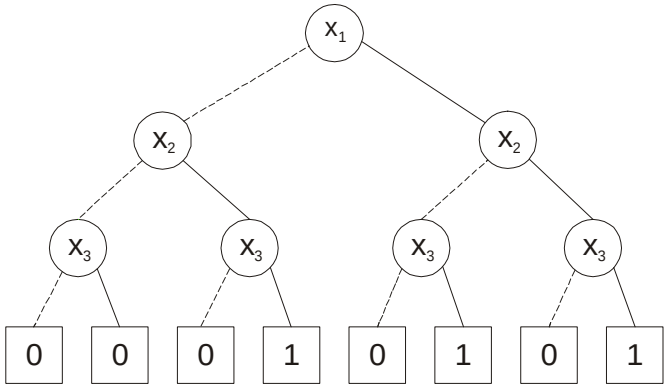
2.2 Binární rozhodovací diagramy

Binární rozhodovací diagram je kořenový orientovaný acyklický graf reprezentující logickou funkci. Šipky označující směr hrany se ovšem často nepíšou. BDD je speciálním případem rozhodovacího diagramu a skládá se z neterminálních "rozhodovacích" uzlů a jednoho nebo dvou typů terminálních uzlů.

Každý neterminální uzel představuje binární proměnnou *x* z logické funkce a má dvě hrany vedoucí k potomkům. Jedna hrana je označována jako *lo(x)* (zakreslována do grafu přerušovanou čarou) a odpovídá případu, kdy je proměnné přiřazena hodnota 0. Druhá hrana je označována jako *hi(x)* (zakreslována do grafu nepřerušovanou čarou). Tato hrana odpovídá případu, kdy je proměnné přiřazena hodnota 1. Někdy jsou větve také označovány jako *if-then* (případ, kdy je proměnné přiřazena hodnota 1) a *else* (proměnné je přiřazena 0).

Terminální uzly nabývají hodnot 0 nebo 1.

Příklad binárního rozhodovacího diagramu je na obrázku č. 1. Jeho pravdivostní tabulka je uvedena v tabulce č. 6.

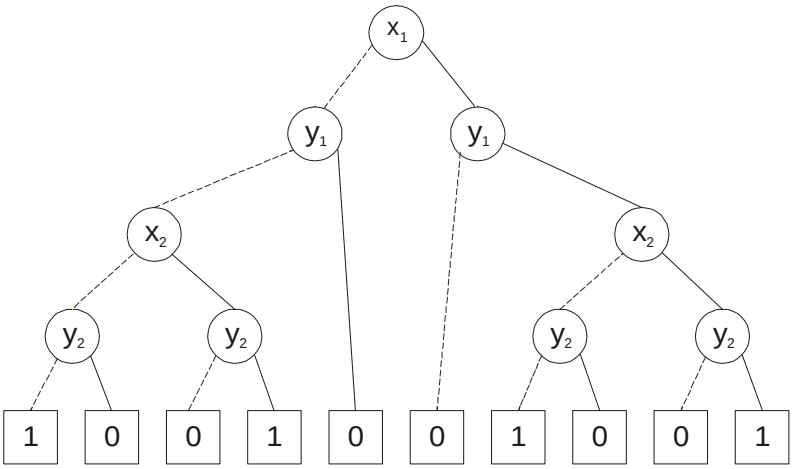


Obrázek č. 1: *Ukázka binárního rozhodovacího diagramu*

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. 6: *Pravdivostní tabulka logické funkce*

Pokud použijeme výraz z příkladu v odstavci 1.2.3 $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ převedený do if-then-else normálové formy, bude jeho binární rozhodovací diagram vypadat takto:



Obrázek č. 2: *Binární rozhodovací diagram výrazu* $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$

Tento binární rozhodovací diagram odpovídá také výrazu $\overline{(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)}$. Binární rozhodovací diagram tedy může reprezentovat různé výrazy (logické funkce). Ve skutečnosti jich může reprezentovat nekonečně mnoho, takže vytvořením binárního rozhodovacího diagramu ztrácíme informaci o syntaxi výrazu. Naproti tomu je v diagramu uchována kompletní sémantická informace. Například pro ohodnocení proměnných ve výrazu, jejíž rozhodovací diagram je na obrázku č. 2, procházíme diagram od kořene: podle hodnoty každé proměnné se rozhodneme do kterého z jejich potomků vstoupíme. Takto pokračujeme než dorazíme do jednoho z listů. Hodnota tohoto listu je ohodnocení celého výrazu pro dané hodnoty proměnných. Například pro hodnoty x_1=1, x_2=0, y_1=1, y_2=1 je ohodnocení celého výrazu rovno 0. Pokud dorazíme během průchodu diagramem do listu aniž bychom prošli všechny proměnné (v tomto případě například pro hodnoty x_1=1, x_2=0, y_1=0, y_2=1), znamená to, že hodnota celého výrazu záleží v tomto případě pouze na těch proměnných, jejichž uzly jsme prošli (x_1, y_1) a zbylé proměnné mohou nabývat jakýchkoliv hodnot, tzv. don't care hodnot – neurčené stavy (x_2, y_2).

2.3 Tvorba binárního rozhodovacího diagramu

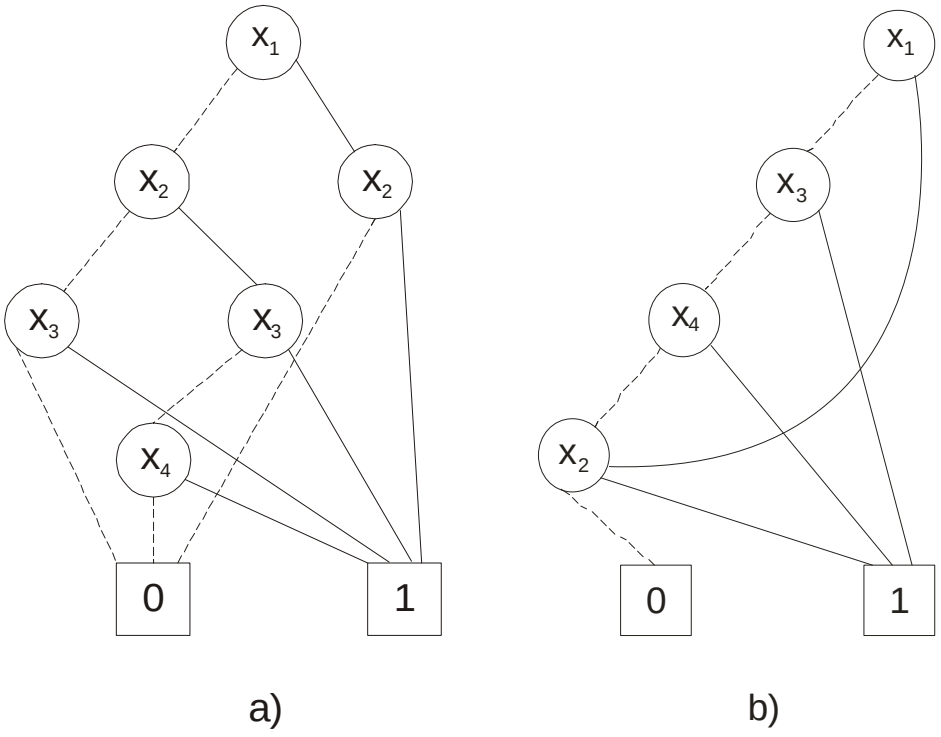
Binární rozhodovací diagram se tvoří pomocí aplikace Shannonova expanzního teorému na nějakou logickou funkci. Jak bylo vidět v příkladu z kapitoly 1.2.3 a jeho pokračování v kapitole 2.2 mohou vznikat v binárním rozhodovacím diagramu redundance. Tyto redundance vznikají jak na terminálech, tak na uzlech. V následujích kapitolách se budu zabývat odstraňováním těchto redundancí a uspořádáním proměnných v BDD.

2.3.1 Uspořádání proměnných – vznik OBDD

Ze zkušenosti je známo, že velikost binárního rozhodovacího diagramu je silně závislá na upořádání proměnných. Mějme neměnné uspořádání (<) proměnných v binárním rozhodovacím diagramu *d*. Říkáme, že *d* zachovává uspořádání <, jestliže pro každé dvě proměnné, pro které platí uspořádání $x_1 < x_2$, narazíme při všech možných průchodech od kořene k listům binárního rozhodovacího stromu na uzel, který je označený jménem proměnné x_1 dříve, než na uzel označený jménem proměnné x_2. Tento rozhodovací diagram nazýváme uspořádaný rozhodovací diagram.

Na obrázku č. 3 jsou dva diagramy funkce $f = (x_1 \cdot x_2) + (\overline{x_1} \cdot x_3) + (x_2 \cdot \overline{x_3} \cdot x_4)$, které se liší pouze uspořádáním proměnných. V případě a) jsou proměnné

uspořádány $x_1 < x_2 < x_3 < x_4$. V případě b) jsou proměnné uspořádány $x_1 < x_3 < x_4 < x_2$.



Obrázek č. 3: *Binární rozhodovací diagramy lišící se pouze upořádáním proměnných*

Jak je vidět z obrázku č. 3, v prvním případě obsahuje binární rozhodovací diagram šest neterminálních uzlů. Podle druhého uspořádání vzniknul diagram, který má pouze 4 vnitřní uzly, tj. dochází zde k poklesu počtu neterminálů o 33%.

Obecně neexistuje pravidlo, podle kterého bychom mohli určit pořadí proměnných tak, abychom sestrojili diagram s minimálním počtem vnitřních uzlů. Většinou se pořadí určuje ručně, dynamicky nebo pomocí heuristik (podrobněji v [7]). Pokud ovšem máme funkci proměnných $a_1....a_n$ a $b_1...b_n$ ve tvaru:

$$(a_1 \cdot b_1) + (a_2 \cdot b_2) + ... + (a_n \cdot b_n)$$

pak můžeme říci, že uspořádání $a_1 < b_1 <...< a_n < b_n$ vede na minimální počet vnitřních uzlů – $2n$, tj. na jednu proměnnou připadá jeden vnitřní uzel. Na opak například uspořádání $a_1 < ...< a_n < b_1...< b_n$ vede na binární rozhodovací diagram, který má $2\cdot(2^n-1)$ neterminálů. S rostoucím počtem proměnných roste prudce rozdíl náročnosti na uložení BDD a velmi se projevuje to, jak jsou proměnné uspořádány. Více o problému nalezení nejvýhodnějšího pořadí proměnných například v [7] nebo [10]

2.3.2 Redukce binárního rozhodovacího diagramu – vznik RBDD (ROBDD)

Pokud máme binární rozhodovací diagram nějaké logické funkce může v něm docházet k redundancím. Některé části BDD se opakují. Toto je z hlediska velikosti BDD jako datové struktury, která má být někde uložena, a z hlediska efektivnosti operací nad diagramem velmi nepříjemné. Ve stromě nebo BDD může dojít k těmto redundancím:

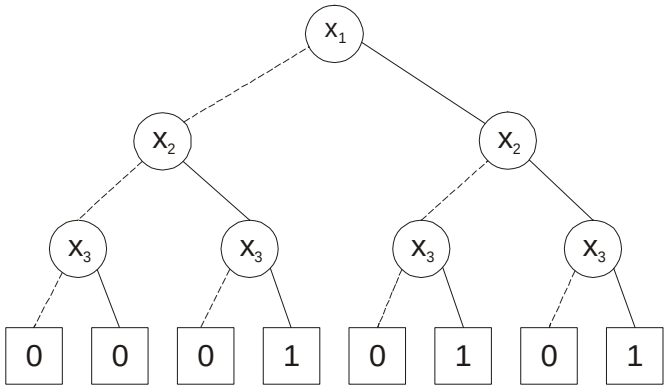
- opakování terminálních uzlů se stejnou hodnotou
- levý i pravý potomek uzlu je ten samý uzel; z hlediska sémantiky nemá tento uzel žádný význam
- uvnitř stromu nebo diagramu mohou být určité podstromy identické; jedna informace o funkci je ve stromu nebo diagramu zanesena vícekrát

Pro odstranění těchto redundancí definujeme tři transformační pravidla, která nemění sémantiku reprezentované funkce:

- odstranění opakujících se terminálů – odstraň všechny, až na jeden, terminální uzly, jejichž hodnoty jsou stejné a hrany vedoucí do odstraněných uzlů přesměruj do zbývajícího uzlu
- odstranění nadbytečných vnitřních uzlů – pokud vnitřní uzel v má levého i pravého potomka stejný uzel ($lo(v)=hi(v)$), pak tento uzel odstraň a všechny hrany vedoucí do tohoto uzlu přesměruj do levého potomka (pravého potomka).
- odstranění opakujících se vnitřních uzlů – pokud levý potomek uzlu u a levý potomek uzlu v je stejný a zároveň pravý potomek uzlu u a pravý potomek uzlu je stejný ($lo(u)=lo(v)$ a zároveň $hi(u)=hi(v)$), pak odstraň jeden z těchto uzlů a všechny hrany vedoucí do tohoto uzlu přesměruj do zbylého uzlu.

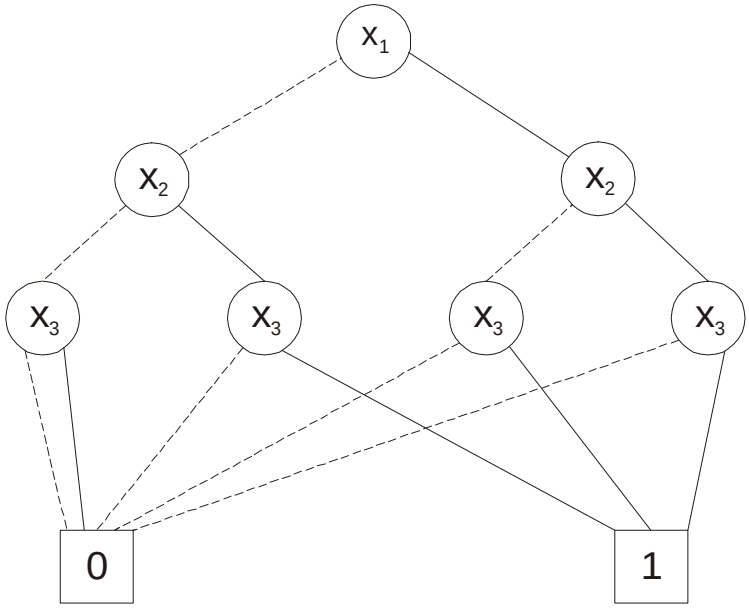
Tyto pravidla se při redukci binárního rozhodovacího diagramu používají opakovaně do té doby, dokud jejich opakování přináší nějaký pozitivní výsledek a je možné použít jedno z těchto pravidel.

Postup redukce ukážu na binárním rozhodovacím diagramu z obrázku č. 1. Pro lepší orientaci ještě jednou, jak tento diagram vypadá.



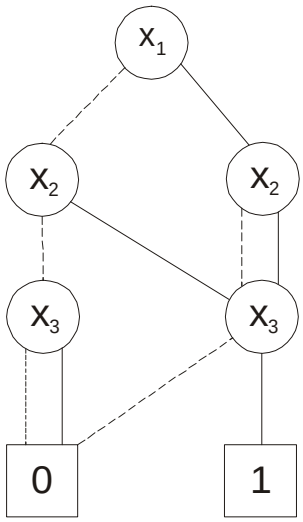
Obrázek č. 4: *Binární rozhodovací diagram*

Tento diagram zachovává uspořádání proměnných $x_1 < x_2 < x_3$. Proto po aplikaci všech "redukčních" pravidel vznikne ROBDD. Nejprve odstraníme všechny nadbytečné terminály:



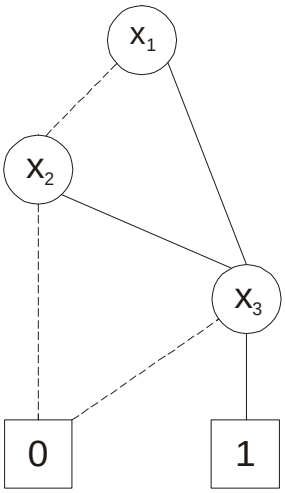
Obrázek č. 5: *Odstranění nadbytečných terminálu*

Na obrázku č. 5 je vidět výsledek celé operace. Z původních osmi terminálů zůstaly pouze dva. Dále aplikujeme pravidlo na odstranění opakujících se vnitřních uzlů. Vidíme, že pravé tři uzly označené jako x_3 mají stejné potomky. Proto budou dva z nich v tomto kroku odstraněny:



Obrázek č. 6: *Odstranění opakujících se vnitřních uzlů*

V posledním kroku aplikujeme poslední pravidlo a odstraníme ty uzly, které ze sémantického hlediska nemají žádný význam:



Obrázek č. 7: *Odstranění nadbytečných vnitřních uzlů*

Z původních patnácti uzlů (vnitřních i listů) byl diagram redukován na redukovaný uspořádaný rozhodovací binární diagram s pěti uzly. Velikost úspory je více než zřejmá.

ROBDD představují kanonickou reprezentaci logických funkcí. Pro jakoukoliv funkci $f: B^n \rightarrow B$ existuje právě jeden redukovaný uspořádaný binární rozhodovací diagram d s uspořádáním proměnných $x_1 < x_2 < ... < x_n$. Důkaz tohoto tvrzení můžeme najít například v [1]. Tato vlastnost ROBDD má několik důležitých důsledků. Funkce je splnitelná, právě tehdy, když ROBDD není reprezentován pouze terminálním uzlem 0. Dalším důsledkem je, že reprezentace tautologické funkce

pomocí ROBDD obsahuje pouze jeden uzel – terminální uzel 1. Pokud je funkce nezávislá na proměnné *x*, ROBDD reprezentace neobsahuje uzel označený touto proměnnou. Pokud je tedy vytvořen redukovaný uspořádaný binární rozhodovací diagram logické funkce, může být testováno spousta vlastností funkce.

2.4 Charakteristiky složitostí ROBDD

Redukované uspořádané binární rozhodovací diagramy poskytují praktické využití při symbolické manipulaci s logickými funkcemi pouze tehdy, pokud velikost diagramu asymptoticky roste s počtem proměnných mnohem pomaleji, než je nejhorší možný případ – exponenciální růst. Jak již bylo ukázáno v odstavci 2.3.1, velikost ROBDD některých funkcí je velmi závislá na uspořádání proměnných. Navíc, jak praxe ukazuje, existuje pro mnoho funkcí používaných v reálných aplikacích kompaktnější forma reprezentace, než jsou ROBDD.

Možnosti a omezení redukovaných uspořádaných binárních rozhodovacích diagramů se dají odvodit ze složitostí některých základních druhů logických funkcí. Následující tabulka ukazuje asymptotické složitosti OBDD reprezentací nejběžnějších druhů funkcí v závislosti na typu funkce a na uspořádání proměnných (nejhorší/nejlepší možné).

Druh funkce	Složitost	
	Nejlepší	Nejhorší
Symetrická	lineární	kvadratická
Disjunktivní forma	lineární	exponenciální
Konjunktivní forma	exponenciální	exponenciální

Tabulka č. 7: *Složitost OBDD reprezentací nejběžnějších funkcí*

Symetrické funkce, u kterých hodnota funkce závisí pouze na počtu argumentů, které jsou rovny 1, jsou nezávislé na uspořádání proměnných. Kromě konstantních funkcí, se složitost reprezentace těchto funkcí pohybuje od lineární (např. parita) do kvadratické (např. alespoň polovina proměnných je rovna 1).

Výstup z n-bitové sčítačky můžeme považovat jako logickou funkci (v disjunktivní formě) s proměnnými a0, a1,...,an-1 (první operand sčítačky) a b0, b1, ..., bn-1 (druhý operand sčítačky). Jak již bylo v odstavci 2.3.1 naznačeno, je složitost reprezentace OBDD funkce při uspořádání proměnných $a_0 < b_0 <...< a_{n-1} < b_{n-1}$ lineární (pro všechny bity) a pro uspořádání $a_0 < ...< a_{n-1} < b_0...< b_{n-1}$ exponenciální.

Logické funkce reprezentující celočíselné násobení jsou při reprezentaci jako OBDD problematické. Bez ohledu na uspořádání proměnných, kterékoliv dva

výstupy (kromě krajních výstupů) z n-bitové násobičky mají (podle [8]) exponenciálně složitou OBDD reprezentaci.

Odvození horní hranice složitosti ostatních funkcí může být založeno na vlastnostech logických sítí, které je realizují. C. L. Berman a K. L. McMillan odvodili hranice složitosti pro několik druhů "svázaných" sítí.

Mějme síť skládající se z *m* logických bloků s *n* počátečními vstupy a jedním primárním výstupem. Každý logický blok může mít několik vstupů a několik výstupů. Počáteční vstupy jsou reprezentovány pomocí bloků, které nemají žádné vstupy a jeden výstup. Dále definujeme lineární uspořádání sítě jako očíslování bloků od 1 do *m* tak, že blok, ze kterého vychází primární výstup je očíslován jako poslední. Mějme *x* jako počet spojů vedoucích z bloku *j* (kde $j<i$) do bloku *k* (kde $i \leq k$) přes blok *i*. Dále mějme *y* jako jako počet spojů vedoucích z bloku *j* (kde $j>i$) do bloku *k* (kde $i \geq k$) přes blok *i*. Označme w_f jako maximální *x* ze všech bloků sítě a w_r jako maximální *y*. Pokud známe w_f a w_r, můžeme říct, že danou obvodovou funkci můžeme reprezentovat pomocí ROBDD se složitostí nejvýše $n \cdot 2^{w_f 2^{w_r}}$.

2.5 Manipulace s ROBDD

Řada symbolických operací s booleovskými funkcemi může být implementována pomocí grafových algoritmů aplikovaných na ROBDD. Tyto algoritmy zachovávají jednu důležitou vlastnost – výsledné ROBDD zachovává stejné pořadí proměnných jako ROBDD, které algoritmus dostal jako argumenty operace. Díky tomu můžeme implementovat složitou operaci pomocí sekvence jednodušších operací.

2.5.1 Operace APPLY

Operace APPLY generuje booleovskou funkci tak, že vezme dvě jiné booleovské funkce jako argumenty a provede s nimi nějakou operaci. Mějme funkci *f* a funkci *g* jako argumenty a binární booleovský operátor [op] (například AND, OR, atd.). APPLY vrací funkci *f* [op] *g*. Například můžeme negovat funkci vypočítáním $f \oplus 1$. Dále můžeme například testovat rovnost funkcí *f* a *g*, kde neurčené hodnoty jsou vyjádřeny funkcí *d*, vypočítáním $(f \overline{\oplus} g)+d$ a otestováním, zda výsledek je konstantní funkce 1.

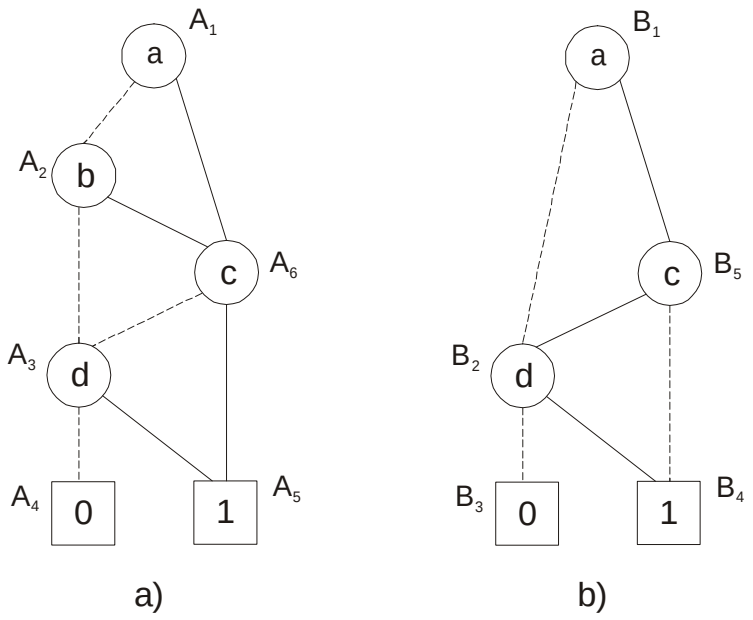
APPLY algoritmus prochází grafy, které jsou argumenty operace, do hloubky (depth-first) a udržuje si dvě hashovací tabulky – jednu pro zlepšení efektivity výpočtu, druhou pro výpomoc při tvorbě maximálně redukovaného grafu. Algoritmus

se opírá o fakt, že můžeme pomocí Shannonovy expanze obě funkce rozložit podle stejné proměnné a po té provést mezi částmi původní funkce provést požadovanou operaci:

$$f\ [op]\ g = (\bar{x} \cdot (f|x \leftarrow 0\ [op]\ g|x \leftarrow 0)) + (x \cdot (f|x \leftarrow 1\ [op]\ g|x \leftarrow 1))$$

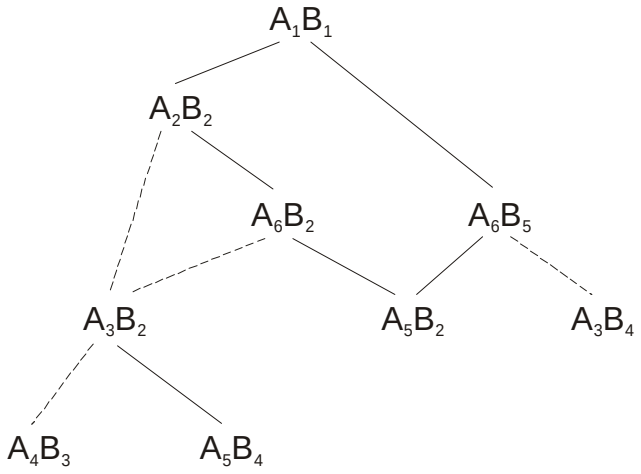
Tato rovnice je základem rekurzivní procedury pro výpočet ROBDD reprezentace f [op] g.

Ukažme si použití APPLY operace na funkcích $f(a,b,c) = ((a + b) \cdot c) + d$ (obrázek č. 8a) a $g(a,b,c)=(a \cdot \bar{c}) + d$ (obrázek č. 8b). Jako binární booleovský operátor [op] použijeme OR. Na obrázku č.8 vidíme diagramy obou funkcí:



Obrázek č. 8: *Diagramy funkce f a g*

Na dalším obrázku vidíme průběh rekurzivní procedury vytváření ROBDD z našeho příkladu. Pokud procedura narazí na terminál, rekurze se ukončí a funkce vrátí příslušný terminální uzel. Pokud procedura narazí na neterminál, vybere proměnnou *x*, která je v uspořádání proměnných nejníže a provede podle ní Shannonovu expanzi. Totéž provede rekurzivně ve funkcích vzniklých dosazením 0 a 1 za proměnnou *x* do původní funkce.

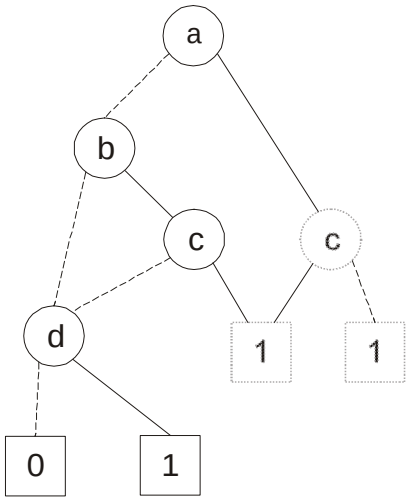


Obrázek č. 9: *Průchod procedury APPLY při vytváření ROBDD z funkcí f a g*

Abychom zlepšili efektivnost APPLY operace, přidáme do procedury popsané výše ještě další dvě vylepšení.

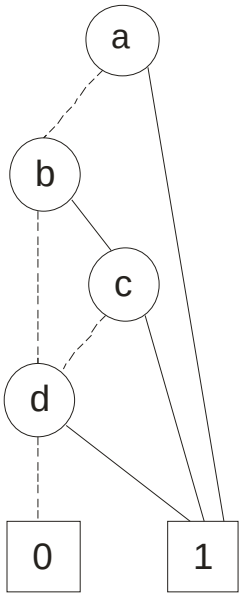
Za prvé, když narazíme na podmínku kdy jeden z argumentů je terminální uzel, který reprezentuje pro operaci "dominantní" hodnotu (např. 0 pro AND a 1 pro OR), rekurze je ukončena a je vrácen příslušný terminální uzel.

Za druhé, abychom se vyhnuli několikanásobnému volání na stejné páry argumentů, udržujeme si hashovací tabulku, ve které každý záznam má jako klíč dvojici uzlů (kořeny argumentů) a jako hodnotu uzel v generovaném grafu. Vždy se tedy nejprve podíváme do tabulky, zda jsme již pro tuto dvojici uzlů operaci neprováděli. Pokud ano, ukončíme rekurzi a vrátíme uzel z generovaného grafu, který je uložen jako hodnota u této dvojice uzlů. Každý krok rekurze tedy vrací jeden uzel z výsledného diagramu. Výsledek operace vidíme na obrázku č. 10.



Obrázek č. 10: *Výsledek operace APPLY bez redukce nadbytečných uzlů*

Jak je z obrázku vidět, tímto postupem vytváříme pouze neredukovaný uspořádaný binární rozhodovací diagram. Abychom dostali jako výsledek ROBDD, musíme v každém kroku rekurze testovat, zda-li uzel, který vracíme, nepatří mezi nadbytečné uzly (viz. kapitola 2.3.2). K tomu použijeme druhou hashovací tabulku, ve které jsou uloženy již vypočtené uzly z výsledného ROBDD. Konečný výsledek je vidět na obrázku č. 11.



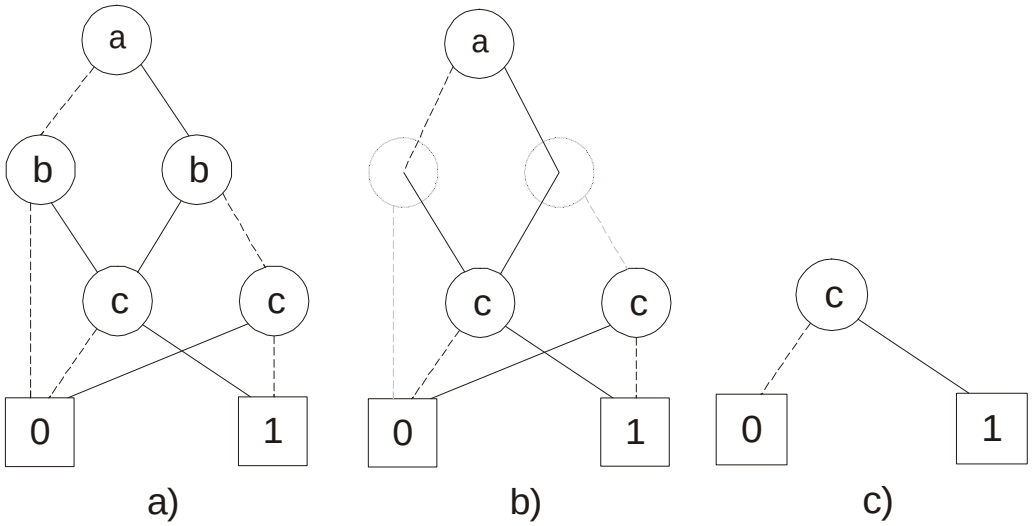
Obrázek č. 11: *Konečný výsledek operace APPLY*

Problémem ovšem zůstává, jak efektivně implementovat operaci APPLY. Jedním z řešení je operátor ITE (If-Then-Else) – viz. kapitola 2.6.3.

2.5.2 Operace RESTRICT

Pokud známe hodnotu některých vstupních proměnných, které jsou neměnné, můžeme ROBDD dále zjednodušit – fixujeme hodnotu určité proměnné a použijeme tuto operaci. Výpočet omezení na funkci, která je reprezentována BDD je zřejmé. Pokud omezíme proměnnou *x* pouze na hodnotu *k*, jednoduše přesměrujeme všechny hrany jdoucí do uzlu *v*, který je označený *x*, buď do potomka *lo(v)* kdy *k=0*, nebo do potomka *hi(v)* kdy *k=1*. Na obrázku č. 12 je diagram funkce $(b \cdot c) + (a \cdot \bar{b} \cdot \bar{c})$ (a), a diagram téže funkce, pokud proměnná *b* je omezená na hodnotu 1 (b). Z diagramu lze vidět, že implementace této operace, bez přidání nějakých dalších vylepšení, přináší jako výsledek pouze neredukovaný binární rozhodovací diagram. Proto si při tvorbě výsledného ROBDD budeme opět udržovat hashovací tabulku. Operace je implementována opět procházením původního grafu do hloubky. Každé rekurzivní volání má jako parametr uzel v původním grafu a vrací jako výsledek uzel ve

vytvářeném grafu, který je ukládán do hashovací tabulky. Tím je zajištěno, že výsledkem operace bude ROBDD (obrázek č. 12. c).



Obrázek č. 12: *Průběh operace RESTRICT*

2.6 Datová reprezentace

Efektivnost všech aplikací založených na operacích s ROBDD na závisí téměř výhradně na efektivnosti základních operací s ROBDD a požadavky na výkonnost jsou vysoké. Proto se mnoho výzkumů snažilo implementovat tyto operace do počítačových programů tak, aby byly rychlé a efektivně využívaly paměť.

Většina programů zabývajících se manipulacemi s BDD je navržena následovně. Základní datovou strukturou pro ROBDD je uzel. Jeho strukturu můžeme vidět v tabulce č. 8.

Proměnná	Velikost
Index	2B
High	4B
Low	4B

Tabulka č. 8: *Struktura datové reprezentace uzlu ROBDD*

Index je index *i* proměnné x_i. *High* je ukazatel na potomka *then*, *low* je ukazatel na potomka *else*. Velikosti pro jednotlivé proměnné jsou pro 32-bitové prostředí. Zvolená velikost indexu umožňuje vytvořit až 65536 různých proměnných.

Toto je pouze základní struktura uzlu. Aby operace s ROBDD a jeho uložení v paměti bylo co nejefektivnější, musíme do základní struktury přidat ještě další informace. Například v balíku CUDD, který bude podrobněji popsán v kapitole 4, je

ve struktuře *DdNode* (která představuje jak neterminální, tak terminální uzel diagramu) proměnná *ref*, ve které je uložen počet referencí na tento uzel. Velikost této promenné je 2 bajty (unsigned short).

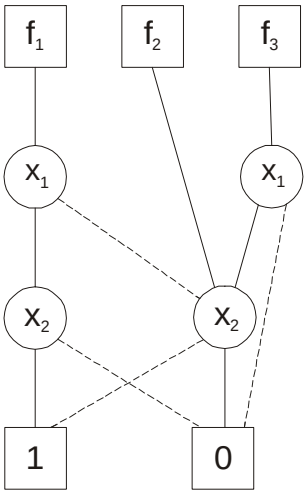
S přidáváním dalších informací musíme být opatrní, protože použití ROBDD v praktických aplikacích může vyžadovat až milióny uzlů a každá další informace ve struktuře uzlu velmi zvyšuje nároky na paměť. Většinou je tedy dosaženo optimálního výsledku pomocí různých kompromisů a chytrých vylepšení, například pomocí sdílených ROBDD, pomocí hashovací tabulky (tzv. unique table), ITE algoritmu, tabulky výpočtů (computed table, použití viz. operace APPLY) a negovaných hran (complemented edges). Těmito technikami se budu zabývat v několika následujících odstavcích.

2.6.1 Sdílené ROBDD

Několik funkcí může být reprezentováno pomocí jednoho orientovaného acyklického grafu s několika kořeny. Takto jsou podgrafy, které se vyskytují ve více ROBDD, reprezentovány pouze jednou. Takový to druh reprezentace funkcí je nazýván jako sdílené ROBDD.

Sdílené ROBDD ušetří spoustu času a kapacity v porovnání s tím, kdybychom měli každý ROBDD zvlášť. Pokud dosáhneme toho, že každá podfunkce je ve sdíleném ROBDD reprezentována pouze jednou, pak je kanonická reprezentace funkce převedena do silné kanonické formy. To znamená, že pokud máme dvě stejné funkce *f* a *g*, nereprezentujeme je dvěma stejnými ROBDD, ale pouze ukazateli, které ukazují do stejného místa v paměti.

Příklad sdíleného ROBDD funkcí $f_1=(x_1 \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$, $f_2=\overline{x_2}$, $f_3=x_1 \wedge \overline{x_2}$ je na následujícím obrázku.



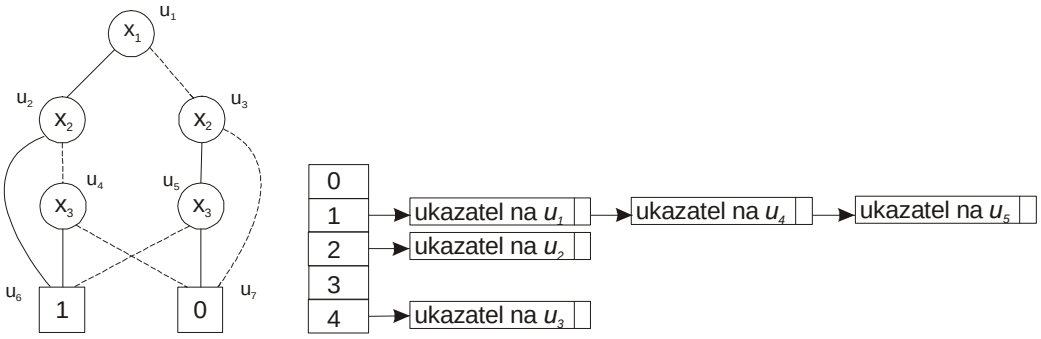
Obrázek č. 13: *Příklad sdíleného ROBDD*

2.6.2 Hashovací tabulka (Unique table)

Silná kanonická forma, které je možné dosáhnout reprezentací funkcí pomocí sdílených ROBDD, umožňuje udržovat ROBDD ve každé části výpočtu redukované. ROBDD se vytváří ze spodu. Proto pokaždé, když je přidáván do ROBDD nový uzel, musí být kromě jeho indexu (označení jména proměnné) také specifikován levý i pravý potomek. Nejprve se zjistí, zda-li už tento uzel existuje. Pokud ano, nevytváří se nový, ale použije se již existující uzel. To jestli uzel s trojicí vlastností (x_i, ***high***, *low*) již existuje, zjistíme z hashovací tabulky. Tato trojice je pomocí nějaké hashovací funkce uložena do hashovací tabulky. Používá se zřetězené hashování. Proto, pokud nastane kolize a na stejné místo je potřeba uložit více ukazatelů na uzly, je použit spojový seznam. Příklad takovéto tabulky pro uložení ROBDD funkce třech proměnných $f(x_1,x_2,x_3)$ je na obrázku č. 14. Jednotlivé vnitřní uzly označíme $u_1,u_2,...,u_5$, abychom je mohli od sebe odlišit. V praxi se používá například adresa uzlu v paměti. Hashovací tabulka může například být pole velikosti 5 s indexy 0 – 4. Dále potřebujeme hashovací funkci. Ta může vypadat například takto:

$$h(x_i,u_j,u_k)=(i+k+j)\%5$$

pozn.: % je označení pro funkci modulo



Obrázek č. 14: *Příklad hashovací tabulky pro uložení ROBDD*

2.6.3 ITE algoritmus a tabulka výpočtů (computed table)

Tabulka výpočtů je datová struktura sloužící k uchování výsledků výpočtů. Slouží k tomu, aby stejné výsledky nemusely být počítány vícekrát. Funguje na principu cache-hash tabulky, to znamená, že se uchovává jen podmnožina výsledků a to podle nějakého pravidla – např. LRU (Least Recently Used – výsledky, které jsou znovu málo používány jsou odstraněny a nahrazeny těmi, které jsou používány častěji). Ukládání a hledání v tabulce probíhá podle principů běžné hashovací tabulky.

V kapitole 2.6.1 jsem popsal základní myšlenku výpočtu binárních operací nad ROBDD. Abychom mohli pracovat se všemi binárními operacemi nějakým sjednoceným způsobem, vymysleli (jak je uvedeno v [7]) K. S. Brace, R. L. Rudell a R. E. Bryant If-Then-Elese operátor (ITE).

ITE je ternární operátor se vstupy *f, g, h*, který počítá tuto funkci: ***If f, then g, else h.*** Toto můžeme zapsat jako:

$$ITE(f,g,h)=(f \cdot g) + (\overline{f} \cdot h)$$

V tabulce č. 9 je ukázáno, jak se dají jednotlivé binární operace reprezentovat pomocí operátoru ITE.

	Operace	**ITE výraz**
0000	0	0
0001	$f \cdot g$	ITE(f,g,0)
0010	$\overline{f \Rightarrow g}$	ITE(f, $\overline{g}$,0)
0011	f	f
0100	$\overline{f \Leftarrow g}$	ITE(f,0,g)
0101	g	g
0110	$f \oplus g$	ITE(f, $\overline{g}$,g)
0111	$f + g$	ITE(f,1,g)
1000	$\overline{f + g}$	ITE(f,0, $\overline{g}$)
1001	$\overline{f \oplus g}$	ITE(f,g, $\overline{g}$)
1010	$\overline{g}$	ITE(g,0,1)
1011	$f \Leftarrow g$	ITE(f,1, $\overline{g}$)
1100	$\overline{f}$	ITE(f,0,1)
1101	$f \Rightarrow g$	ITE(f,g,1)
1110	$\overline{f \cdot g}$	ITE(f, $\overline{g}$,1)
1111	1	1

Tabulka č. 9: *Přehled výrazů vyjádřitelných pomocí ITE operátoru*

Nechť *f, g* a *h* jsou logické funkce a x_i je proměnná, podle které se expanduje (leading variable). Při počítání ITE(*f,g,h*) můžeme použít následující rekurzívní rozklad:

$$
\begin{aligned}
ITE(f,g,h) &= f \cdot g + \overline{f} \cdot g = \\
&= x_i \cdot (f \cdot g + \overline{f} \cdot h)_{xi} + \overline{x_i} \cdot (f \cdot g + \overline{f} \cdot h)_{\overline{xi}} = \\
&= x_i \cdot (f_{xi} \cdot g_{xi} + \overline{f_{xi}} \cdot h_{hi}) + \overline{x_i} \cdot (f_{\overline{xi}} \cdot g_{\overline{xi}} + \overline{f}_{\overline{xi}} \cdot h_{\overline{xi}}) = \\
&= ITE(x_i, ITE(f_{xi}, g_{xi}, h_{xi}), ITE(f_{\overline{xi}}, g_{\overline{xi}}, h_{\overline{xi}})) = \\
&= \underline{(x_i, ITE(f_{xi}, g_{xi}, h_{xi}), ITE(f_{\overline{xi}}, g_{\overline{xi}}, h_{\overline{xi}}))}
\end{aligned}
$$

Výsledkem úprav je trojice, která má stejný význam, jako operace APPLY definovaná pomocí Shannova expanzního teorému (viz. sekce 2.6.1). Úpravami jsme dostali uzel proměnné x_i a jeho dva rekurzivně definované potomky.

Algoritmus zapsaný v pseudokódu vypadá takto:

```
ITE(f,g,h){
  if(terminal_case){
    return (vysledek terminal_case);
  }
  else if(f,g,h)∈ ComputedTable{
    return ComputedTable(f,g,h);
  }
  else{
    Nechť xi je vedoucí  proměnná v pořadí proměnných pro f,g a
      h;
    t=ITE(fxi,gxi,hxi);
    e=ITE(fx̄i,gx̄i,hx̄i);
    if(t==e){
      return e;
    }
    r=najdi_nebo_pridej_do_UniqueTable(xi,t,e);
    vloz_do_ComputedTable({f,g,h},r);
    return r;
  }
}
```

Vstupem tohoto algoritmu jsou ROBDD *f*, *g*, *h* se stejným uspořádáním proměnných. Výstupem je ROBDD podle ITE(*f*,*g*,*h*). Terminální případy (`terminal_case`) mohou být tyto:

- ITE(1,*f*,*g*) = *f*
- ITE(0,*f*,*g*) = *g*

Dále se rekurze může ukončit v těchto případech:

- ITE(*f*,1,0) = *f*
- ITE(*f*,*g*,*g*) = *g*

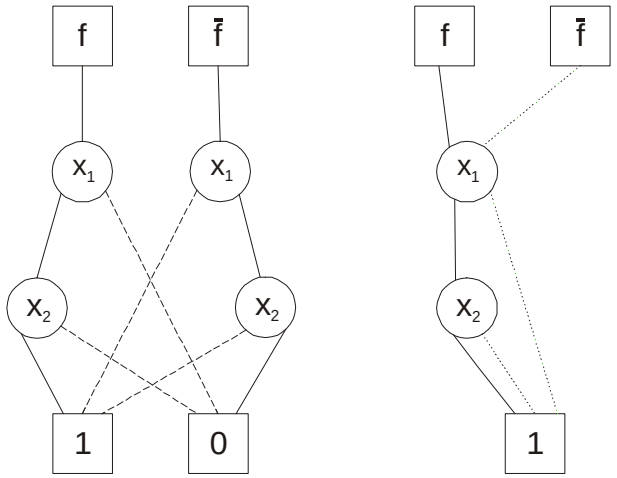
Funkce `najdi_nebo_pridej_do_UniqueTable()` nejprve zjistí, zda trojice, kterou dostala jako parametr už nebyla realizována jako uzel ve výsledném ROBDD. Pokud ano, vrátí ukazatel na tento uzel. V opačném případě je vytvořen nový uzel a je

vrácen ukazatel na něj. Funkce `vloz_do_ComputedTable()` vloží vypočítaný mezivýsledek do `ComputedTable`.

Složitost ITE algoritmu je ohraničená *O*(velikost(*f*)·velikost(*g*)·velikost(*h*)). Pro binární operace je ovšem složitost algoritmu nižší – je ohraničená kvadratickou funkcí

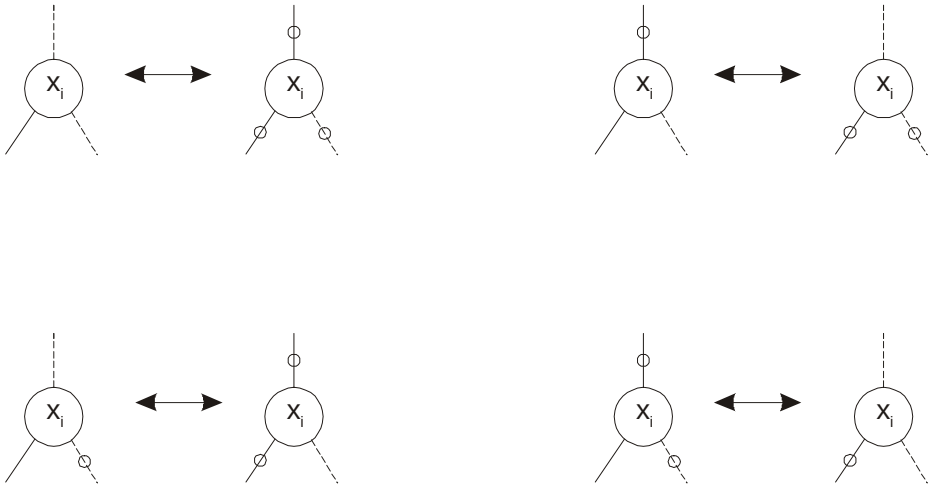
2.6.4 Negované hrany (Complemented edges)

Velmi efektivním vylepšením je použití tzv. negovaných hran. Tato technika je založena na faktu, že redukované uspořádané binární diagramy funkce *f* a její negace $\overline{f}$ se liší jen v tom, že hodnoty jejich uzlů jsou navzájem vyměněné. Přidáním nové jednobitové informace o každé hraně docílíme značné úspory paměti. Pokud tento bit není nastaven, je podgraf ROBDD interpretován normálně (jako *f*). Pokud tento bit je nastaven, pak je podgraf ROBDD interpretován negovaně (jako $\overline{f}$). Díky tomu mohou být funkce *f* a $\overline{f}$ reprezentovány pomocí jednoho ROBDD. Také stačí pouze jeden terminální uzel (uzel 1). Příklad ROBDD s použitím negovaných hran je na následujícím obrázku (negované hrany jsou tečkované).



Obrázek č. 15: *ROBDD bez použití a s použitím techniky negovaných hran*

Hlavním problémem je, že při výskytu negovaných hran v ROBDD ztrácíme kanonickou reprezentaci. Jednou z množností, jak ji obnovit, je omezení možných pozic negovaných hran. Existují čtyři páry kombinací (viz. obrázek č. 16), které reprezentují různé funkce a funkce v páru jsou ekvivalentní. Hrany, které jsou negované jsou na obrázku označeny kroužkem.



Obrázek č. 16: *Ekvivalentní kombinace výskytu negovaných hran*

Abychom zajistili kanonickou reprezentaci, nesmí být hrana *then* nikdy negována. Z tohoto důvodu si vždy vybereme levou možnost (podle obrázku č. 16) z ekvivalentních kombinací.

Vzhledem k tomu, že některé trojice ITE(*f,g,h*) jsou ekvivalentní – například pro funkci $f+g$ existují tyto tři možné kombinace:

$$\text{ITE}(f,f,g)=\text{ITE}(f,1,g),=\text{ITE}(g,1,f)=\text{ITE}(g,g,f)$$

definujeme tzv. standardní trojice. První čtveřice pravidel říká, že vždy, když je to možné, nahradíme funkci konstantou:

$$\text{ITE}(f,g,g) \Rightarrow \text{ITE}(f,1,g)$$

$$\text{ITE}(f,g,f) \Rightarrow \text{ITE}(f,g,0)$$

$$\text{ITE}(f,g,\overline{f}) \Rightarrow \text{ITE}(f,g,1)$$

$$\text{ITE}(f,\overline{f},g) \Rightarrow \text{ITE}(f,0,g)$$

Dále existují ekvivalentní trojice, které využívají komutativnosti ITE operátoru, jestliže je druhý nebo třetí argument konstanta, nebo jestli jsou navzájem negované.

$$\text{ITE}(f,1,g) \Rightarrow \text{ITE}(g,1,f)$$

$$\text{ITE}(f,g,0) \Rightarrow \text{ITE}(g,f,0)$$

$$\text{ITE}(f,g,1) \Rightarrow \text{ITE}(\overline{g},\overline{f},1)$$

$$\text{ITE}(f,0,g) \Rightarrow \text{ITE}(\overline{g},0,\overline{f})$$

$$\text{ITE}(f,g,\overline{g}) \Rightarrow \text{ITE}(g,f,\overline{f})$$

Je dobré vybrat tu trojici, jejíž první argument je závislý na proměnné, která se v pořadí proměnných vyskytuje dříve.

2.6.5 Správa paměti

Při správě paměti musíme čelit několika problémům:

- obecně má vnitřní uzel ROBDD více než jednoho předchůdce, a kvůli nárokům na paměť nemůžeme ve struktuře uzlu ukládat reference na předchůdce
- na uzly je odkazováno z unique table, computed table a z ostatních uzlů
- ze zkušenosti víme, že není příliš výhodné odstranit dočasné uzly (uzly mezivýpočtů) jakmile to jde; mohou být důležité pro mechanizmus computed table, tj. cache paměti

Z těchto důvodů se ukazuje, že není dobré uvolnit paměť po dočasných uzlech ihned. Lépe je vyčkat na více takových událostí. K tomu se používá mechanizmus zvaný garbage collection.

K tomu se využívá počítání referencí na jednotlivé uzly. Mějme uzel *v*. Kdykoliv je vytvořen nějaký jiný uzel, jehož hrana *then* nebo hrana *else* končí v tomto uzlu, je počet referencí na tento uzel zvýšen o jedna. Naopak když se funkce, která je reprezentována tímto uzlem, přestane používat je počet referencí dekrementován. Pokud počet referencí klesne na nulu, pak jsou v reference dekrementovány rekurzivně i v jeho potomcích. Uzel, který má 0 referencí je prohlášen jako mrtvý (dead). Když počet mrtvých uzlů dosáhne určitého množstvím je spuštěna správa paměti – garbage collector – a paměť, kterou zabírají mrtvé uzly je uvolněna.

Struktura uzlu je v následující tabulce.

Proměnná	Velikost
Index	2B
High	4B
Low	4B
Next	4B
Refcount	cca. 2B
Mark	

Tabulka č. 10: *Konečná podoba struktury uzlu*

Proměnné *Index, high, low* jsem již vysvětlil na začátku kapitoly. Proměnná *next* je ukazatel v hashovaci tabulce (unique table) na další uzel, u nějž hashovací funkce vrací stejný výsledek. *Refcount* je počet referencí na uzel. Některé aplikace vyžadují další informaci, například o tom, jestli uzel byl již navštívený. Pro to, je vyhrazena proměnná *mark.*

Z tabulky vidíme, že jeden uzel zabírá v paměti cca. 16 bajtů. Po započítání dalších paměťových míst, které tento uzel nepřímo zabírá (ukazatel v unique tabulce na tento uzel, záznam v computed table) dostáváme hodnotu přibližně 21 bajtů. Uložení miliónů uzlů v paměti tedy zabere asi 21MB.

2.7 Použití BDD

Binární rozhodovací diagramy mají v praxi široké využití. Pomocí ROBDD bylo vyřešeno spoustu problémů například v oblasti návrhu číslicových systémů, v oblasti umělé inteligence, v oblasti konečných automatů, matematické logice a dalších oblastech.

Například v oblasti číslicových systémů se ROBDD používají například při ověřování, testuje se například ekvivalence dvou kombinačních logických obvodů. ROBDD se také využívají při opravování konstrukčních chyb, v citlivostních analýzách, pravděpodobnostních analýzách, atd.

3 Jiné rozhodovací diagramy

3.1 Binární rozhodovací diagramy s potlačenou nulou

S. Minato představil binární rozhodovací diagramy s potlačenou nulou (zero-suppressed decision diagrams) ve své publikaci Binary Decision Diagrams and Applications for VLSI CAD z roku 1996 [17]. Tyto diagramy umožňují uložit množinu některých kombinací mnohem efektivněji a používají se především k ukládání matic a vektorů. Zaobíral se především strategiemi, které jsou použity při redukci BDD.

BDD i ZDD mohou být považovány jako druh rozhodovacích stromů, které jsou zjednodušené použitím pravidel uvedených v odstavci 2.3.2. Tyto operace zaručují, že výsledný diagram bude kanonický. BDD a ZDD se liší v pravidle eliminace uzlů, které nemají z hlediska diagramu žádný sémantický význam.

U binárních rozhodovacích diagramů jsou odstraněny ty uzly, jejichž levý i pravý potomek je stejný uzel. U binárních rozhodovacích diagramů s potlačenou nulou je tomu jinak. Jsou odstraněny ty uzly, jejichž hrana ohodnocená 1 (*then* hrana) vede do terminálního uzlu s hodnotou 0. Tato úprava umožňuje uložení kombinací, ve kterých je málo prvky z původního počtu (tzv. řídké množiny – sparse sets), efektivněji než u binárních rozhodovacích diagramů.

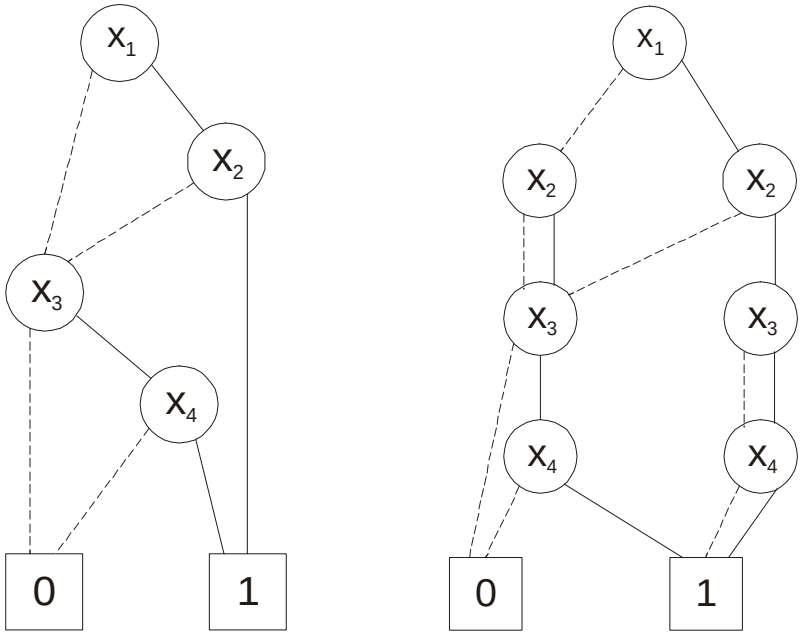
V následujících třech podkapitolách bych rád ukázal rozdíl mezi BDD a ZDD na dvou případech.

3.1.1 Reprezentace booleovské funkce

V BDD platí, že všechny cesty z kořene do terminálního uzlu s hodnotou 1, mohou být považovány jako krychle tvořící disjunktní pokrytí funkce. Proměnná je v krychli ohodnocená 1, pokud v cestě od kořene k terminálnímu uzlu 1 je hrana *then*, která vychází z uzlu vztahujícího se k této proměnné. Obdobně pro proměnnou ohodnocenou v krychli hodnotou 0. Proměnná, která se v krychli nevyskytuje, se v cestě nevyskytuje.

Pro tutéž funkci v ZDD taktéž platí, že všechny cesty z kořene do terminálního uzlu 1, reprezentují disjunktní pokrytí funkce. Také platí, že proměnná je v odpovídající krychli ohodnocená 1, pokud je v cestě hrana *then* vycházející z uzlu vztahujícího se proměnné. U proměnných ohodnocených 0 je tomu jinak – buď se v cestě vyskytuje hrana *else* vycházející z odpovídajícího uzlu nebo cesta vůbec neprochází tímto uzlem. Proměnná se v krychli nevyskytuje, jestliže cesta prochází

uzlem označeným jménem proměnné a obě hrany vycházející z tohoto uzlu směřují do stejného uzlu. Na obrázku č. 17 je jak BDD, tak binární rozhodovací diagram s potlačenou nulou pro funkci $F=(x1 \wedge x2) \vee (x3 \wedge x4)$.

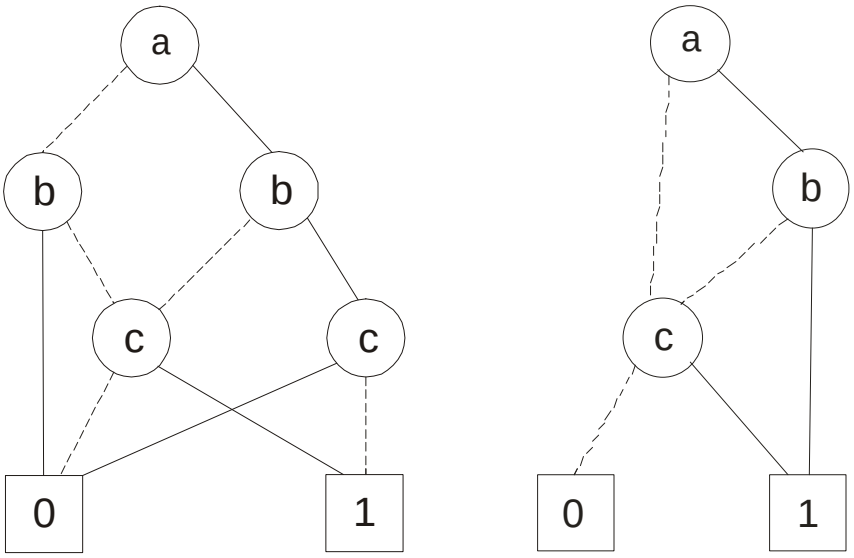


Obrázek č. 17:*BDD a ZDD pro funkci* $F=(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

Jak je vidět z těchto diagramů, je $\{x_1 \wedge x_2,\ \overline{x_1} \wedge x_3 \wedge x_4,\ x_1 \wedge \overline{x_2} \wedge x_3 \wedge x_4\}$ disjunktní pokrytí této funkce. Dále je vidět, že ZDD obsahuje daleko více uzlů, než BDD (téměř dvakrát tolik). Ze zkušenosti vyplývá, že binární rozhodovací diagramy s potlačenou nulou nejsou vhodné pro reprezentaci takovýchto typických booleovských funkcí.

3.1.2 Reprezentace množiny podmnožin

Mějme tři prvky (*a, b, c*) a množinu podmnožin z těchto prvků {{*a,b*},{*a,c*},{c}}. Pokud přiřadíme každý prvek ke stejně pojmenované binární proměnné, pak charakteristická funkce této množiny podmnožin je $F=(a \wedge b \wedge \overline{c}) \vee (a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge \overline{b} \wedge c)$. První minterm odpovídá první podmnožině, atd. BDD a ZDD této množiny podmnožin je vidět na obrázku č. 18.



Obrázek č. 18: *BDD a ZDD pro množinu podmnožin {{a,b},{a,c},{c}}*

V obou diagramech jsou tři cesty z kořene do terminálního uzlu 1, které odpovídají podmnožinám {*a,b*},{*a,c*},{c}. Z obrázku jde vidět, že velikost binárního rozhodovacího diagramu s potlačenou nulou je menší než velikost binárního rozhodovacího diagramu. Dá se dokázat, že horní hranice velikosti ZDD je celkový počet prvků ve všech podmnožinách zahrnutých do množiny. Této hranice je ovšem dosaženo zřídkakdy, většinou je velikost ZDD mnohem menší. Naproti tomu je horní hranice velikosti BDD dána počtem podmnožin násobeným počtem všech prvků, které se v nich mohou objevit. Z toho vyplývá, že reprezentace podmnožin pomocí ZDD je efektivnější než reprezentace pomocí BDD.

Pokud by se v množině podmnožin vyskytla prázdná podmnožina, byla by reprezentována mintermem $F = \bar{a} \cdot \bar{b} \cdot \bar{c}$. Pokud by celá množina byla prázdná, její charakteristická rovnice by vypadala takto: F=0, zatím co množina tvořená všemi možnými podmnožinami by měla charakteristickou rovnici F=1.

Dále se binární rozhodovací diagramy s potlačenou nulou používají k řešení problému pokrytí, jehož způsob řešení je analogický s řešením reprezentace množiny podmnožin. Blíže je tento problém vysvětlen např. v [9].

3.2 Algebraické rozhodovací diagramy

Algebraické rozhodovací stromu jsou obdobou více-terminálových binárních rozhodovacích diagramů (Multi-Terminal Binary Decision Diagrams – MTBDDs) a rozšiřují binární rozhodovací diagramy o další terminální uzly a jsou pro ně definovány nové operace.

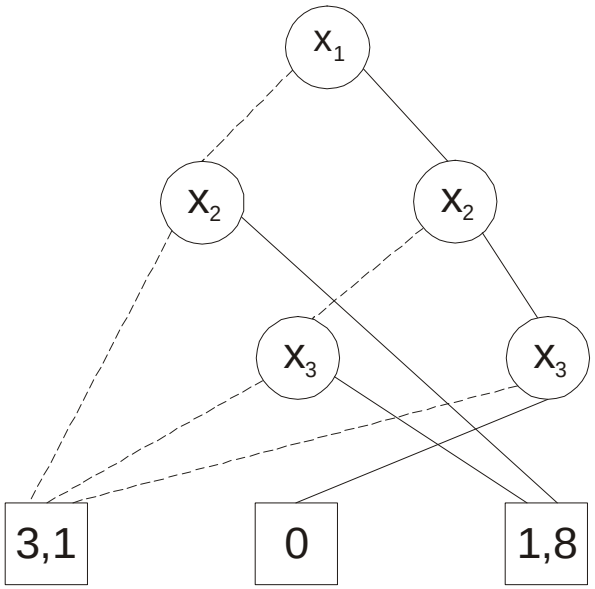
Struktura algebraických rozhodovacích diagramů je stejná jako u binárních rozhodovacích diagramů s výjimkou terminálních uzlů. Shannonův expanzní teorém pro ADD je velmi podobný. Nechť *v* je uzel proměnné *x*. Potom je Shannonův expanzní teorém pro algebraické rozhodovací diagramy definován takto:

$$fv := if\ x\ then\ fsucc1(v)\ else\ fsucc0(v)$$

ADD promítá každé ohodnocení proměnných funkce z jejího definičního oboru na hodnotu jednoho z terminálních uzlů. Ta je získána procházením ADD podle daného ohodnocení proměnných.

Na následujícím obrázku je příklad algebraického rozhodovacího diagramu pro funkci

$$f(x_1, x_2, x_3) = \begin{cases} 3{,}1 \ \ pro\ (x_1 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_2}) \\ pro\ x_1 \wedge x_2 \wedge x_3 \\ 1{,}8\ v\ ostatních\ případech \end{cases}$$



Obrázek č. 19: *Příklad algebraického rozhodovacího diagramu*

Vlastnosti algebraických rozhodovacích diagramů mohou být shrnuty takto:

- algebraické rozhodovací diagramy jsou redukované a uspořádané, z toho vyplývá, že jsou kanonickou reprezentací funkcí.

- některé vlastnosti, jako například tvorba doplňků, mohou mít omezenou použitelnost, protože doplněk v booleově logice nemusí mít význam vzhledem k množině terminálů.
- V porovnání s normálními řídkými datovými strukturami, algebraické rozhodovací diagramy poskytují jednotný přístupový čas – log (N), kde N je počet čísel uložených v ADD

Algebraické rozhodovací diagramy jsou využívány například k řešení:

- násobení matic
- hledání nejkratší cesty

4 Balík CUDD

Colorado University Decision Diagram balík byl vyvíjen Fabiem Somenzi a jeho pracovní skupinou na Coloradské univerzitě. Tento balík se postupně vyvíjí od roku 1996 a nyní je na internetových stránkách projektu (http://vlsi.colorado.edu/~fabio/CUDD/) k dispozici verze 2.4.1. Postupným a opatrným vylepšováním a rozšiřováním algoritmů z předešlých verzí, dosáhl Somenzi značných výkonnostních vylepšení a komplexnosti. K dispozici je přes 400 funkcí jen v základním balíku. K dispozici jsou také různá rozšíření, například pro načítání BDD ze souboru.

Výjimečnou vlastností tohoto balíku je velké spousta algoritmů, které vylepšují uspořádání proměnných. Dále, jak již bylo zmíněno v úvodu, tento balík podporuje operace s BDD, ADD a ZDD. V systému VIS (Veryfication Interacting with Synthesis) je balík CUDD nejpoužívanějším (více o VIS např. v [7]).

Celý projekt CUDD je napsán v jazyku C, ale existuje také C++ rozhranní.

Jak sám Somenzi píše v nápovědě k CUDDu, může být tento balík používán třemi způsoby:

- jako černá skříňka (black box) – v tomto případě program aplikace využívá pouze funkce, které jsou zpřístupněny. Seznam těchto funkcí je přiložen k nápovědě. Taková to aplikace se nemusí zabývat detaily uspořádávání/přeuspořádávání proměnných, které se děje v pozadí
- jako "průhledná" skříňka (clear box) – při psaní náročnějších aplikací, je možné napsat vlastní funkci (s využitím stávajících funkcí – jak exportovaných, tak vnitřních) a přidat ji do balíku
- pomocí rozhranní – objektově orientované jazyky jako C++ a Perl5 umožňují osvobodit programátora od správy paměti. C++ rozhranní je zahrnuto přímo v distribuci, Perl5 rozhranní existuje samostatně

4.1 Příklad vytvoření jednoduchého ROBDD

Součástí balíku je také dokumentace, která je v angličtině, ale je psána velmi srozumitelně. Proto se nebudu zabývat tím, jak tento balík používat. Uvedu pouze krátký příklad kódu, ze kterého bude vidět, jak se vytváří jednoduchý 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 manazeru
f1=Cudd_ReadOne(manager); //vytvareni bdd x0*x1*x2
Cudd_Ref(f1);

for(i=2;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_ReadOne(manager); //vyvareni bdd x0'*x1'*x2'
Cudd_Ref(f2);

for(i=2;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); //vytvoreni celeho bdd
Cudd_Ref(f);
Cudd_RecursiveDeref(manager,f1);
Cudd_RecursiveDeref(manager,f2);
Cudd_Quit(manager);
return 0;
}
```

V tomto příkladě vytvářím jednoduché ROBDD funkce $f=x_0 \cdot x_1 \cdot x_2 + \overline{x_0} \cdot \overline{x_1} \cdot \overline{x_2}$. V každém programu se nejprve musí pomocí funkce `Cudd_init()` zinicializovat *DdManager*, který se stará o správu paměti. *DdMangerem* se budu podrobněji zabývat v kapitole 4.3. Celý ROBDD je vytvářen tak, že se nejprve ve dvou for-cyklech vytvoří ROBDD jednotlivých termů a ty pak se spojí do výsledného diagramu.

ROBDD se tvoří od spodu, to znamená, že před začátkem každého for-cyklu vytvořím konstantní funkci 1 a pak ve for-cyklu vytvářím uzly proměnných (pokud již uzel existuje je vrácen ukazatel na tento uzel) a pomocí funkce `Cudd_bddAnd()` přidávám proměnné k již vytvořenému ROBDD. Mezivýsledky jsou ukládané do pomocné proměnné a musí být referencovány (při vzniku) a dereferencovány (když už nejsou nadále třeba). Nakonec oba ROBDD spojím pomocí funkce `Cudd_bddOr()` do jednoho. K dispozici je také funkce `Cudd_bddIte()`, která slouží jako ITE

operátor, ale pro tyto základní operace je trochu efektivnější použít výše zmíněné funkce.

Na závěr celého programu je dobré ukončit *DdManager* voláním funkce `Cudd_Quit()`.

4.2 Struktura DdNode

Všechny rozhodovací diagramy (BDD, ADD i ZDD) jsou složeny ze struktury DdNode. Její definice vypadá takto:

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

Proměnná index se týká jména proměnné uzlu. Index je neměnný atribut a odráží pořadí vzniku uzlů. Uzel s indexem 0 byl vytvořen jako první. Index je typu DdHalfWord, což je pro 32 bitové prostředí unsigned short.

Proměnná ref udržuje informaci o počtu referencí na tento uzel. Je to informace pro garbage collector, jestli tento uzel může odstranit či nikoliv.

Next je proměnná typu DdNode a ukazuje na další uzel v hashovací tabulce – unique table.

Poslední položkou je unie, která v případě neterminálního uzlu obsahuje ukazatele na potomky uzlu a v případě terminálního uzlu obsahuje hodnotu tohoto uzlu.

4.3 Správa paměti – DdManager

Všechny uzly používané v binární, algebraických a rozhodovacích diagramech s potlačenou nulou jsou udržovány v hashovací tabulce – unique table. BDD a ADD mají stejnou tabulku, zatímco ZDD mají svou vlastní. Principy této tabulky jsou popsány v kapitole 2.7.2. Tato tabulka, spolu s dalšími položkami tvoří strukturu DdManager. Vzhledem k tomu, že DdManager obsahuje přes 130 proměnných, nebudu zde jeho definici uvádět.

5 Převod CUDDu pod Windows

Balík CUDD je napsán pod operační systém Linux, kde se dá přeložit bez problémů, například pomocí makefilu, který je přiložen. Při pokusu o překlad pod operačním systémem Windows bez jakýchkoliv úprav kódu, byla výsledkem pouze spousta chyb a varování. Mým úkolem bylo tedy upravit jej tak, aby se dal kompilovat pod operačním systémem, nebo nejlépe pod oběma systémy.

5.1 Změny v kódu

Nejtěžší částí bylo zjistit, proč překlad nefunguje a pochopit strukturu celého balíku CUDD. Po domluvě se svým zadavatelem jsem se rozhodl, že balík CUDD budu spravovat jako projekt ve vývojovém prostředí Borland C++Builder 6. Podle původního makefilu vznikly 4 statické knihovny - *libcudd.a, libdddmp.a, libepd.a, libmtr.a, libst.a,* a *libutil.a*. Dále se také vytvořil spustitelný soubor *nanotrav*, kterým se spouští jednoduchý program na procházení stavů automatu. Já jsem tuto strukturu trochu pozměnil. Po překladu celého projektu vzniknou pouze 4 statické knihovny - *cudd .lib* (vznikla sloučením původní *libbcudd.a, libepd.a, libmtr.a, libst.a* a *libutil.a*), *dddmp.lib* (odpovídá původní *libdddmp.a*), *nanotrav.lib* (pro používání funkcí, které využíval program *nanotrav*) a *cuddObj.lib* (knihovna pro objektové rozhranní). Do projektu jsem nezahrnul obsah adresáře mnemosyne, protože se jedná o program, který nemá souvislost s CUDDem. Jedná se o program, který zjistí chyby se správou paměti v programu. Navíc je tento program plně určen pod operační systém Linux a jeho portování by možná vydalo na další bakalářskou práci.

Po prostudování struktury CUDDu jsem zjistil, že hlavním problémem, proč nejde projekt přeložit, je v začleňování (includování) hlavičkových souborů do kódů. Původně totiž byly hlavičkové soubory includovány například takto: `#include "util.h",` tj. bez zadání relativní cesty k hlavičkovému souboru, což u zdrojových kódů, které byly umístěny v jiném adresáři, než hlavičkový soubor dělalo problémy. Preprocesor, který vkládá obsah hlavičkového souboru do souboru, ve kterém je direktiva `#include`, tak hledal includovaný soubor nejprve v aktuálním adresáři (adresář, ve kterém je uložený soubor s kódem) a když ho nenašel, začal hledat tento soubor ve standardních adresářích pro hlavičkové soubory, které má nadefinované. Takže při překladu v operačním systému Windows docházelo ke dvěma chybám. Buď hlavičkový soubor nebyl nalezen (neexistoval ani ve standardních adresářích pro hlavičkové soubory) nebo byl nalezen nesprávný soubor (ve standardních adresářích pro hlavičkové soubory existoval jiný hlavičkový soubor se stejným jménem).

Konkrétně u uvedeného příkladu docházelo k tomu, že preprocesor vložil místo souboru $(*cudd)/util/util.h* soubor ze standardního adresáře *include*.

Proto jsem všechny hlavičkové soubory přesunul do jednoho adresáře – $(*cudd)/include* a upravil cesty k hlavičkovým souborům ve všech souborech se zdrojovými kódy. Podrobnější výpis změn ve všech souborech je uveden v příloze A. Například v souboru $(*cudd)/cuddInit.c* jsem změnil již zmíněnou direktivu include na `#include "../include/util.h"`.

Dále jsem do některých hlavičkových souborů přidal dvě direktivy include a to: `#include <stdlib.h>` a `#include <string.h>`. Překlad sice probíhal i bez nich, ale překladač hlásil příliš mnoho varování a nakonec překlad ukončil.

Posledními úpravami bylo nahrazení direktivy `#include <sys/time.h>` za `#include <time.h>`, odstranění direktivy `#include <sys/resource.h>` v souborech *cpu_stats* a *datalimit.c* a přidání direktivy `#include <stdio.h>` do souboru *mtr.h* a *cudd.h*.

Úpravy, které jsem zmínil v posledních dvou odstavcích jsem prováděl tak, aby bylo možné projekt přeložit i pod operačním systémem Linux s minimálními úpravami. Proto jsem použil direktivy podmíněného překladu: `#IF !LINUX == 1` *změněná část* `#ELSE` *původní část* `#ENDIF` (popřípadě bez *else*-části). To znamená, že pokud nebude definována konstanta *LINUX* s hodnotou 1, bude možno překládat projekt pod operačním systémem Windows. Pokud ano, projekt bude přeložitelný v Linuxu.

Dalším problémem, na který jsem při překladu narazil, byly některé warningy při překladu knihovny *dddmp.lib*. Týkalo se to warningů, podle kterých byly některé dvojice funkcí stejně pojmenované (se stejnými parametry) a umístěné v různých souborech. Překlad sice proběhl, ale vzhledem k tomu, že by jedna z funkcí pravděpodobně chyběla, rozhodl jsem se názvy funkcí rozlišit. Přibližším zkoumání jsem zjistil, že funkce se od sebe liší jen velmi málo, nebo skoro vůbec. Přesto jsem názvy funkcí upravil a to následovně.

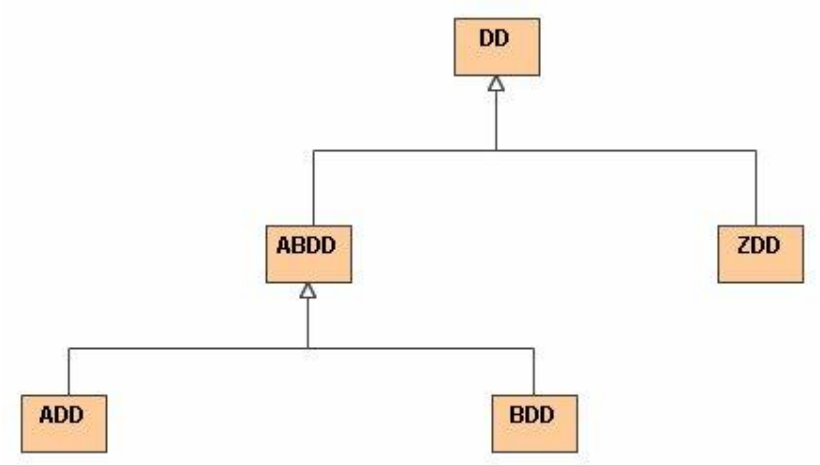
Funkce, které měly stejný název, se vyskytovaly pouze ve dvou souborech - *dddmpDdNodeCnf.c* a *dddmpNodeCnf.c*. Nepodařilo se mi zjistit, proč tomu tak je, protože i v dokumentaci, která je přiložena je ke zdrojovému kódu, mají oba soubory stejný popis. Vybral jsem si náhodně soubor *dddmpDdNodeCnf* a funkce, které měly shodný název s funkcemi v druhém souboru jsem přejmenoval tak, že jsem na konec jejich názvu přidal "2". Například jsem funkci `DddmpNumberDdNodesCnf()` přejmenoval na funkci `DddmpNumberDdNodesCnf2()`. Nakonec jsem přidal jejich novou deklaraci do souboru *dddmpInt.h*.

Po popsaných úpravách probíhá překlad bez problémů. Překladač hlásí spoustu warningů (cca 50). V nich upozorňuje, že některé proměnné nebyly použity. To je způsobeno tím, že tyto proměnné jsou používány podmíněně, např. pokud je definováno makro debugování.

Projektové soubory jsou spolu se zrojovými kódy umístěny na přiloženém CD v adresáři *Projekty/Bez dll/Cudd*. Výsledné knihovny jsou uloženy v adresáři ***Lib.***

5.2 Rozšíření objektového rozhranní

Druhým úkolem bylo rozšířit objektové rozhranní CUDDu o funkce, které využívá program nanotrav. Rozšířil jsem pouze část rozhranní, která se týkala BDD, tj. třídu BDD. Strukturu objektového rozhranní pro objekty ADD, BDD a ZDD ukazuje následující obrázek.

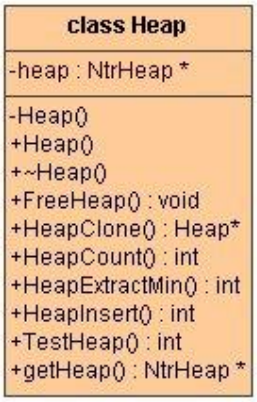


Obrázek č. 20: *Struktura části objektového rozhranní CUDDu*

Rozhodoval jsem se, zda přidat do třídy *BDD* nějaké privátní proměnné. Nejprve jsem chtěl přidat některé proměnné, které jsou často využívány funkcemi nanotravu. Uvědomil jsem si, že by to bylo zbytečné a bude stačit, když budou metodám třídy *BDD* předány jako parametry. Nakonec jsem se rozhodl, že přidám navíc pouze funkci *readBnetNetwork()*, která bude privátní. Tato funkce načítá booleovskou síť do struktury typu *BnetNetwork*, která slouží jako vstupní parametr funkcím z nanotravu. Jako vstup používá soubor ve formátu *blif*, ve kterém je uložena booleovská síť. Proto nejprve ve funkci *readBnetNetwork()* BDD uložím do *blif*-souboru a po té jej načtu do struktury *BnetNetwork* a předám jako parametr dané funkci.

Do třídy *BDD* jsem dále přidal téměř všechny funkce nanotravu (nyní již jako veřejně přístupné). Nepřidal jsem pouze funkce, které se týkaly diagramů s potlačenou nulou a funkce, které se týkaly operací se strukturami *NtrHeap* (pomocná prioritní fronta) a *NtrPartTR* (relace tranzitivity). Pro tyto struktury jsem vytvořil nové třídy – *Heap* a *PartTR* – a definoval jejich potřebné metody.

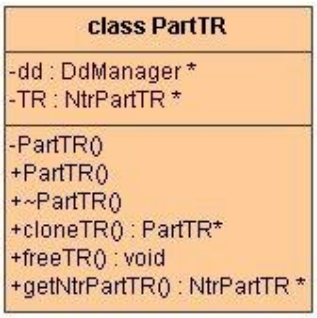
Struktura třídy *Heap* je na následujícím obrázku.



Obrázek č. 21: *Struktura třídy Heap*

Tato struktura má pouze jednu privátní proměnnou – *heap*, což je ukazatel na původní strukturu NtrHeap. Dále má jeden privátní a jeden veřejný konstruktor (lišící se v parametrech), destruktor a metodu *getHeap*, která zpřístupní privátní položku *heap*. Zbylé metody jsou pouze funkce převzaté z nanotravu.

Druhou třídou, kterou jsem přidal do objektového rozhranní CUDDu je třída PartTR. Její struktura je na obrázku č. 22.

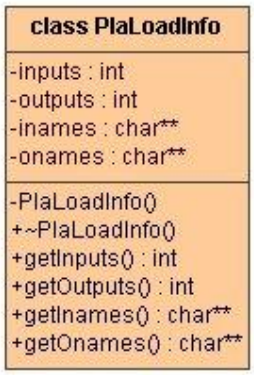


Obrázek č. 22: *Struktura třídy PartTR*

Tato struktura má dvě privátní proměnné – ukazatel na DdManager (správa paměti) a ukazatel na NtrPartTR (původní struktura nanotravu). Dále má jeden privátní a jeden veřejný konstruktor (lišící se v parametrech), destruktor a metodu

getNtrPartTR, která zpřístupní privátní položku *TR*. Zbylé metody jsou pouze funkce převzaté z nanotravu.

Kromě tříd zmíněných na obrázku č. 20 existují v objektovém rozhranní další třídy. Jednu z nich – *BDDvector*, jsem rozšířil o metodu *plaLoad(),* která je odvozená od funkce *cudd_PlaLoad()*. Bližší popis této funkce je v kapitole 5.3. Tato třída slouží k uchování pole binárních rozhodovacích diagramů. Vzhledem k tomu, že tato třída neobsahovala proměnné, ve kterých by bylo možné uchovat informaci, kterou vrátí funkce *plaLoad()* (s výjimkou počtu výstupů, které odpovídá proměnná udávající velikost vektoru), vytvořil jsem novou třídu – *PlaLoadInfo*. Instanci této třídy vrací funkce *plaLoad()*. Struktura *PlaLoadInfo* vypadá následovně.



Obrázek č. 23: *Struktura třídy PlaLoadInfo*

Tato třída obsahuje čtyři privátní proměnné – inputs (počet vstupů), outputs (počet výstupů), inames (ukazatel na pole jmen proměnných), onames (ukazatel na pole jmen proměnných). Tato třída je zvláštní tím, že má privátní konstruktor. Pro toto jsem se rozhodl, aby instance této třídy mohly být vytvořeny pouze v metodách třídy BDDvector (která je definovaná jako spřátelená třída – friend class) a zároveň nemusela být celá třída *PlaLoadInfo* uvnitř třídy *BDDvector*. Rozhodl jsem se tak, aby *PlaLoadInfo* nezabírala zbytečně místo v paměti při každém vytvoření instance třídy *BDDvector*. Dále třída obsahuje destruktor a metody umožňující přístup k privátním položkám.

Objektové rozhranní jsem rozšiřoval ve stejném stylu, jaký byl použit při jeho vytváření. To znamená, že metody se jmenují podobně jako v neobjektové verzi (bez přepony "ntr_") a parametry, které při volání dostanou, předají funkcím v neobjektové verzi.

Seznam všech tříd a metod, které jsem přidal do objektového rozhranní je uveden v příloze B.

5.3 Ověření funkčnosti

K ověření funkčnosti celého CUDDu a k němu přidružených knihoven po portování pod operační systém Windows jsem využil ukázkové programy, které byly součástí distribuce balíku. Mezi tyto ukázkové programy patří:

- cuddTest – otestuje logickou funkčnost některých funkcí – načte matici s reálnými koeficienty a transformuje ji do ADD. Po té provede různé operace nad ADD a BDD. Nakonec otestuje funkce vztahující se k násobení matic a k operacím s Walshovou maticí.
- dddmpTest – program na testování funkčnosti dddmp knihovny. Pomocí jednoduchého příkazového řádku zadává uživatel vybrané příkazy sloužící k operacím s BDD, ADD jako je nahrávání nebo ukládání ze/do souboru.
- mtrTest – program na otestování funkcí mtr balíku
- nanotravTest – program sloužící k procházení stavů automatu
- util_resTest – program na výpis proměnných prostředí
- util_savTest
- cuddObjTest – program na otestování objektového rozhranní CUDDu

Tyto programy se mi podařilo s pomocí "nových" statických knihoven určených pod operační systém Windows podařilo bez větších úprav přeložit. Upravil jsem pouze cesty k hlavičkovým souborům. Tyto ukázkové programy jsou umístěné v adresáři *Programy/Bez dll.* Vše jsem dělal opět formou projektu v prostředí Borland C++ Builderu 6. Projekty jsou pro jednoduchou manipulaci sloučené do jedné projektové skupiny. Soubory projektů jsou spolu se zdrojovými kódy umístěny v adresáři *Projekty/Bez dll/Examples.*

5.4 Načítání BDD ze souboru typu pla

Třetím úkolem bylo napsat začlenit do CUDDu funkci, která načte binární rozhodovací diagram ze souboru typu *pla*. V tomto souboru je ve dvou úrovních popsána booleovská funkce. Nejprve je funkce pomocí klíčových slov a hodnot popsána obecně (počet vstupů, výstupů, atd.). Následuje matice jedniček, nul a pomlček, která specifikuje hodnoty vstupů a k nim hodnoty odpovídajících výstupů. Příklad *pla* souboru je na obrázku č. 24.

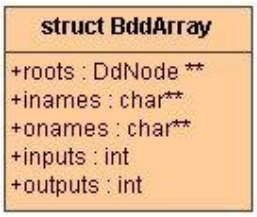
```
.i 3
.o 1

000 1
001 0
010 0
011 0
100 0
101 0
110 0
111 1
```

Obrázek č. 24: *Ukázka pla souboru*

Klíčové slovo *.i* specifikuje počet vstupů, *.o* specifikuje počet výstupů. V následující matici jsou v prvních třech sloupcích hodnoty vstupů a ve čtvrtém sloupci (který je pro přehlednost oddělen mezerou) jsou odpovídající hodnoty výstupů. V tomto příkladě se jedná o funkci $f(x1,x2,x3) = x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$.

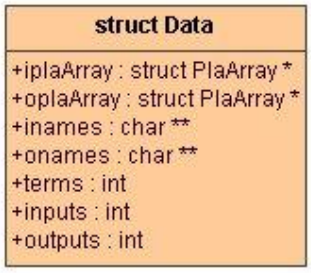
Funkci pro načítání *pla* souborů jsem pojmenoval *cudd_PlaLoad()* a umístil ji do souboru *cuddPlaLoad.c* do adresáře $(*cudd)/cudd*. Tato funkce má dva parametry – ukazatel na *DdManager* (správa paměti) a ukazatel na typ *FILE* (handler souboru). Funkce vrací ukazatel na strukturu *BddArray*, kterou jsem si pro tento účel nadefinoval. Její strukturu ukazuje následující obrázek.



Obrázek č. 25: *struktura BddArray*

Tato struktura obsahuje pole ukazatelů na kořeny jednotlivých binárních rozhodovacích stromů (*roots)*, ukazatel na pole jmen proměnných *(inames)*, ukazatel na pole jmen výstupů *(onames)*, počet vstupů *(inputs)* a počet výstupů *(outputs)*.

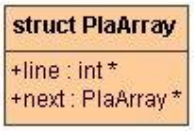
Funkce *cudd_PlaLoad()* využívá pro načtení *pla* souboru do vnitřního tvaru další funkci – *fileLoad()*. Tato funkce přebírá od funkce *cudd_PlaLoad()* ve formě parametru ukazatel na typ *FILE* – handler souboru, ve kterém je uložen popis booleovské funkce. Návratovou hodnotou je opět ukazatel na strukturu - *Data*, kterou jsem si nadefinoval takto:



Obrázek č. 26: *struktura Data*

Tato struktura obsahuje dva ukazatele na strukturu *PlaArray (iplaArray, oplaArray)*, jejíž popis uvedu v následujícím odstavci, ukazatel na pole jmen proměnných *(inames)*, ukazatel na pole jmen výstupů *(onames)*, počet termů *(terms)*, počet vstupů *(inputs)* a počet výstupů *(outputs)*.

Struktura *PlaArray* slouží pro uložení matice hodnot vstupů a výstupů ze souboru, její struktura vypadá následovně:



Obrázek č. 27: *struktura PlaArray*

Line představuje ukazatel na pole hodnot vstupů/výstupů pro jeden term. Logické jedničce odpovídá 1, logické nule odpovídá 0, neurčenému stavu (-) odpovídá -1. *Next* představuje ukazatel na další strukturu *PlaArray*. V podstatě jednomu řádku matice odpovídají dvě instance *PlaArray* – jedna pro hodnoty vstupů a jedna pro hodnoty výstupů. Definice všech struktur jsem umístil do souboru *plaLoad.h* do adresáře $(*cudd)/include*.

Nyní se vrátím k pomocné funkci *fileLoad()*. Pomocí jednoduchého lexikálního analyzátoru rozpoznávám jednotlivá klíčová slova a ukládám jejich hodnoty do odpovídajících proměnných. Pokud funkce narazí na nějaký neočekávaný symbol (znak), ukončí se načítání souboru, uvolní se dynamicky alokovaná paměť a vrátí nulový ukazatel *(null)*, jinak vrátí ukazatel na strukturu *Data*.

Funkce *cudd_PlaLoad()* použije obsah této struktury. Nejprve si vytvořím dočasné binární rozhodovací diagramy – pro každý term (řádek v souboru) jeden – abych je nemusel při opakovaném používání opakovaně vytvářet. Kořeny těchto dočasných BDD si ukládám do pole. Následně projdu po sloupcích celou matici výstupních hodnot a vytvářím konečná BDD – pro každý sloupec jedno. Pokud ještě

není pro daný sloupec (výstup) vytvořeno žádné BDD a narazím na hodnotu 1 (funkce má pro dané vstupy na výstupu logickou jedničku), uložím odpovídající dočasné BDD na patřičnou pozici v poli kořenů konečných BDD. Pokud již na patřičné pozici v poli kořenů existuje BDD, spojím toto BDD s odpovídajícím dočasným BDD pomocí funkce *cudd_bddOr()*. Jinak neprovádím nic. Takto projdu všechny sloupce v matici výstupních hodnot a vytvořím celé pole konečných BDD – proměnná *roots* ve struktuře BddArray. Hodnoty ostatních proměnných *BddArray* převezmu ze struktury *Data*, kterou vrátila funkce *fileLoad()*. Nakonec je uvolněna nepotřebná paměť a vrácen ukazatel na *BddArray*.

Funkce *cudd_PlaLoad()* je napsána v jazyku C, aby mohla být začleněna do celého projektu CUDD. Stačí tedy použít ve zdrojovém kódu direktivu `#include "include/cudd.h"` a funkce je k dispozici.

Funkce je také integrována do objektového rozhranní obdobným způsobem jako funkce z programu nanotrav – je přidána jako další metoda do třídy *BDDvector*.

Kód funkce je jednak umístěn v adresáři *$(cudd)/cudd* a jednak je umístěn jako samostatný projekt v adresáři *Projekty/Bez dll/PlaLoad*.

5.4.1 Ukázková aplikace

Abych ověřil funkčnost nahrávání *pla* souborů, naprogramoval jsem jednoduchou aplikaci, která využije knihovny. Abych také ověřil funkčnost objektového rozhranní CUDDu, napsal jsem program v jazyce C++. Program načte ze zadaného *pla* souboru pole BDD do struktury *BDDvector* a vypíše informace jako je počet vstupů, počet výstupů, popřípadě názvy vstupů, výstupů. Dále program vypíše počet uzlů v jednotlivých binárních rozhodovacích diagramech, čas načtení souboru a ukončí se. Časy načítání souborů jsem použil v následujícím odstavci.

Projekt ukázkové aplikace je uložen v adresáři *Projekty/Bez dll/ObjectPlaLoad*.

5.5 Vytvoření Dll knihoven

Posledním úkolem, který jsem dostal k vypracování, bylo pokusit se vytvořit dll (dynamic link library) knihovny CUDDu. Bohužel jsem na internetu nenašel nějaký návod, jak vytvořit dll knihovnu rozsáhlejšího projektu v prostředí Bordland C++ Builder. Všechny návody se týkaly pouze vytvoření jednoduché dll knihovny (např. pouze obsahovala jen jednu funkci). Proto jsem po několika neúspěšných pokusech zvolil následující postup, který ovšem nemusí být úplně korektní.

V celém projektu jsem upravil pouze všechny hlavičkové soubory. Hlavičkové soubory projektu a hlavičkové soubory, které budou později používány k vytváření jiných aplikací, se od sebe trochu liší (viz. dále). Do všech hlavičkových souborů projektu jsem vložil následující kód:

```
#ifndef __BUILDING_THE_DLL
#define __BUILDING_THE_DLL
#endif

#ifndef DLL
#define DLL
#ifdef __BUILDING_THE_DLL
#define __EXPORT_TYPE dllexport
#else
#define __EXPORT_TYPE dllimport
#endif
#endif
```

V první části nejprve definuji makro `__BUILDING_THE_DLL`, které určuje, zda se vytváří dll knihovna nebo je vytvářen jiný projekt. Toto makro použiji v následujícím podmíněném překladu. V něm definuji makro `__EXPORT_TYPE`, jehož hodnota závisí na tom, zda bylo definováno předešlé makro. Dále ještě definuji pomocné makro `DLL`, které využívám pouze k tomu, aby nedošlo k vícenásobným definicím.

Před deklarace funkcí z hlavičkových souborů jsem přidal následující kód - `extern "C" __declspec(__EXPORT_TYPE)`, před deklarace tříd pouze `__declspec(__EXPORT_TYPE)`.

Na konec hlavičkového souboru jsem přidal následující kód:

```
#ifndef DLL2
#define DLL2
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason,
                         void* lpReserved)
{
  return 1;
}
#endif
```

Ve něm je definována funkce *DllEntryPoint()*, která je volána, když je knihovna dynamicky linkována (odpojována).

Vše jsem opět dělal formou projektu v prostředí C++ Builderu. Celý CUDD je, stejně jako v případě statických knihoven, rozdělen do čtyřech částí a po překladu vzniknou čtyři dll knihovny – *cuddDll.dll*, *cuddObjDll.dll*, *dddmpDll.dll* a *nanotravDll.dll*. Dále vzniknou čtyři pomocné soubory - *cuddDll. lib*, *cuddObjDll.lib*, *dddmpDll. lib* a *nanotravDll. lib.* Tyto soubory se liší od předešlých statických knihoven a používají se pouze při vytváření nových projektů. Linker z nich čerpá

informace o tom, kde a ve které dll knihovně je kód funkce umístěn. Proto také musí být připojeny k jiným vytvářeným projektům.

Knihovny jsou na sobě závislé, viz následující tabulka:

dll knihovna	ke své funkci potřebuje
cuddDll.dll	---
dddmpDll.dll	cuddDll.dll
nanotravDll.dll	cuddDll.dll,dddmpDll.dll
cuddObjDll.dll	cuddDll.dll, nanotravDll.dll

Tabulka č. 11: *Závislosti dll knihoven*

Při vytváření projektů, které budou využívat dll knihovny je tedy třeba připojit k projektu ***.lib*** soubory (zkompilovaný projekt je již ovšem nepotřebuje, používá pouze dll knihovny). Dále jsou zapotřebí trochu jinak upravené hlavičkové soubory. Tyto hlavičkové soubory se liší od hlavičkových souborů, které byly použity při vytváření dll knihoven, následovně:

- není definováno makro `__BUILDING_THE_DLL`
- chybí definice funkce *DllEntryPoint()*

Vzhledem k využití podmíněného překladu podle makra `__BUILDING_THE_DLL` není třeba dál již nic měnit.

Abych otestoval funkčnost dll knihoven, použil jsem všechny předchozí příklady (včetně ukázkové aplikace) a modifikoval je tak, aby využívaly dll knihovny. Všechny projekty využívající dll knihovny, spolu s vlastním projektem dll knihoven, jsou umístěny na cd v adresáři *Projekty/Dll.* Include soubory jsou v adresáři *Include/include/Dll.* Samotné dll knihovny jsou spolu s pomocnými *lib* soubory umístěné v adresáři *Lib/Dll.* A konečně programy, které využívají dll knihovny, jsou umístěné v adresáři *Programy/Dll.*

5.6 Testy

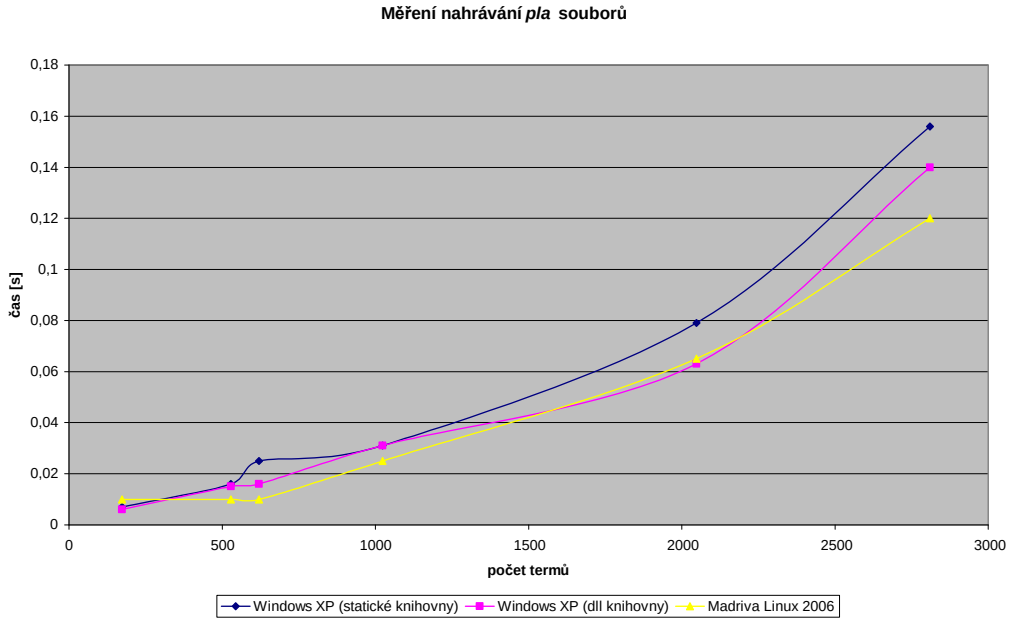
Výkonnost funkce jsem otestoval na pla souborech, které mi byly poskytnuty panem Ing. Petrem Fišerem. Všechny soubory jsou na přiloženém CD v adresáři *Testovací soubory.*

Testy jsem provedl na počítači s procesorem AMD AthlonXP 2200+, 512 MB DDR RAM a to jednak pod operačním systémem Windows XP a jednak pod operačním systémem Madriva Linux 2006. Výsledky jsou v následující tabulce.

soubor	počet vstupů	počet výstupů	počet termů	čas [s] –		
				Windows XP		Mandriva Linux
				statické knihovny	dll knihovny	
ibm.pla	48	17	173	0,007	0,006	0,010
soar.pla	83	94	529	0,016	0,015	0,010
ex4.pla	128	28	620	0,025	0,016	0,010
test3.pla	10	35	1024	0,031	0,031	0,025
test2.pla	11	35	2048	0,079	0,063	0,065
pdc.pla	16	40	2810	0,156	0,140	0,120

Tabulka č. 12 : *Srovnání výkonnosti - pod Windows XP × pod Mandriva Linux 2006*

Z tabulky je vidět, že doba načítání BDD závisí hlavně na počtu zapsaných termů (řádků v souboru). Co se týče rychlosti jsou výsledky pod operačním systémem Windows XP celkově mírně o něco horší než pod operačním systémem Madriva Linux 2006. To ovšem mohlo být způsobeno tím, že mívám pod operačním systémem Windows XP spuštěno více aplikací. Pro přehlednost ještě uvádím srovnání výkonností v následujícím grafu. Z něj lze vidět, že nárůst je přibližně lineární až polynomický.



Obrázek č. 28: *Graf porovnání výkonnosti pod různými operačními systémy*

6 Závěr

V této práci se mi podařilo nejprve přiblížit teorii týkající se rozhodovacích diagramů (BDD, ZDD a ADD) a následně popsat postup při převádění balíku CUDD pod operační systém Microsoft Windows a problémy, které byly s ním spojeny.

Co se týče práce samotné, pracoval jsem na několika dílčích úkolech. Nejprve se mi povedlo vytvořit statické knihovny balíku CUDD, následně jsem rozšiřoval objektové rozhranní CUDDu o funkce programu nanotrav a také jsem rozšířil celý CUDD o funkci, která slouží ke čtení souborů ve formátu pla. Nakonec jsem vytvořil dynamické dll knihovny CUDDu.

Při portování jsem se snažil, aby celý balík byl zpětně kompilovatelný s minimálními úpravami i pod operačním systémem Linux, což se mi díky direktivám podmíněného překladu povedlo.

Součástí celé práce jsou také aplikace, které ověřují funkčnost knihoven a to jak statických tak dynamických. Většinou jsem využil ukázkové aplikace, které byly k CUDDu přiloženy.

Na závěr jsem provedl několik testů funkce, kterou jsem naprogramoval. Tyto testy umožňují srovnat rychlost načtení pla souborů jednak pod operačním systémem Windows XP a jednak pod operačním systémem Mandriva Linux 2006. Při testech pod Windows XP využívám jak statické knihovny, tak dynamické. Z testů vyplývá, že rychlost funkce je za všech podmínek přibližně srovnatelná, pod Mandriva Linux 2006 o málo rychlejší.

V budoucnu by stálo za úvahu, zda celý balík učesat, sjednotit a vypracovat také obsáhlejší uživatelskou dokumentaci, která by zahrnovala všechny funkce ze všech částí CUDDu. Dále je možno balík rozšiřovat o další potřebné funkce, stačí pouze trochu proniknout do jeho struktury.

7 Seznam použité literatury

[1] H. R. Andersen, An introduction to Binary Decision Diagrams, 1997

[2] K. Korovin, Lecture noter for Logic in Computer Science, 2006

[3] M. Demlová, B. Pondělníček, Matematická logika, ČVUT Praha 1997

[4] B. Bollig, M. Sauerhoff, D. Sieling, I. Wegener, Binary Decision Diagrams

[5] F. Somenzi, CUDD: CU Decision Diagram Package Release 2.3.1, 2001

[6] Wikipedia - http://en.wikipedia.org/wiki/Binary_decision_diagram

[7] C. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design, 1998

[8] R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication, 1991

[9] A. Mishchenko, An Introduction to Zero-Suppressed Binary Decision Diagrams, 2001

[10] W. Lenders, Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams, 2004

[11] R.. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi, Algebraic Decision Diagrams and their Applications, 1993

[12] C. Gröpl, Binary Decision Diagrams for Random Boolean Functions, 1999

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

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

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

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

[17] S. Minato, Binary Decision diagrams and Applications for VLSI CAD, 1996.

[18] H. Schildt – Nauč se sám C (třetí edice), 2001

[19] H. Schildt – Nauč se sám C++ (třetí edice), 2001

8 Použité zkratky

ADD	Algebraic Decision Diagram *algebraický rozhodovací diagram*
BDD	Binary Decision Diagram *binární rozhodovací diagram*
CNF	Conjunctive Normal Form *konjunktivní normálová forma*
CUDD	Colorado University Decision Diagram (package) *Balík z Coloradské univerzity týkající se rozhodovacích diagramů*
DNF	Disjunctive Normal Form *disjunktivní normálová forma*
INF	If-Then-Else Normal Form *If-Then-Else normálová forma*
ITE	If-Then-Else (operator) *If-then-else (operátor)*
MTBDD	Multi-Terminal Binary Decision Diagram *více-terminálový binární rozhodovací diagram*
NF	Normal Form *normální forma*
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*

9 Přílohy

Příloha A

Výpis změn v kódu v jednotlivých souborech

adresář cudd:

cuddAddAbs.c – upravení cesty k hlavičkovým souborům
cuddAddApply.c – upravení cesty k hlavičkovým souborům
cuddAddFind.c – upravení cesty k hlavičkovým souborům
cuddAddInv.c – upravení cesty k hlavičkovým souborům
cuddAddIte.c – upravení cesty k hlavičkovým souborům
cuddAddNeg.c – upravení cesty k hlavičkovým souborům
cuddAddWalsh.c – upravení cesty k hlavičkovým souborům
cuddAndAbs.c – upravení cesty k hlavičkovým souborům
cuddAnneal.c – upravení cesty k hlavičkovým souborům
cuddApa.c – upravení cesty k hlavičkovým souborům
cuddAPI.c – upravení cesty k hlavičkovým souborům
cuddApprox.c – upravení cesty k hlavičkovým souborům
cuddBddAbs.c – upravení cesty k hlavičkovým souborům
cuddBddCorr.c – upravení cesty k hlavičkovým souborům
cuddBddIte.c – upravení cesty k hlavičkovým souborům
cuddBridge.c – upravení cesty k hlavičkovým souborům
cuddCache.c – upravení cesty k hlavičkovým souborům
cuddCheck.c – upravení cesty k hlavičkovým souborům
cuddClip.c – upravení cesty k hlavičkovým souborům
cuddCof.c – upravení cesty k hlavičkovým souborům
cuddCompose.c – upravení cesty k hlavičkovým souborům
cuddDecomp.c – upravení cesty k hlavičkovým souborům
cuddEssent.c – upravení cesty k hlavičkovým souborům
cuddExact.c – upravení cesty k hlavičkovým souborům
cuddExport.c – upravení cesty k hlavičkovým souborům
cuddGenCof.c – upravení cesty k hlavičkovým souborům
cuddGenetic.c – upravení cesty k hlavičkovým souborům
cuddGroup.c – upravení cesty k hlavičkovým souborům
cuddHarwell.c – upravení cesty k hlavičkovým souborům
cuddInit.c – upravení cesty k hlavičkovým souborům
cuddInt.h – upravení cesty k hlavičkovým souborům
cuddInteract.c – upravení cesty k hlavičkovým souborům
cuddLCache.c – upravení cesty k hlavičkovým souborům
cuddLevelQ.c – upravení cesty k hlavičkovým souborům,
cuddLinear.c – upravení cesty k hlavičkovým souborům
cuddLiteral.c – upravení cesty k hlavičkovým souborům
cuddMatMult.c – upravení cesty k hlavičkovým souborům
cuddPriority.c – upravení cesty k hlavičkovým souborům
cuddRead.c – upravení cesty k hlavičkovým souborům
cuddRef.c – upravení cesty k hlavičkovým souborům

cuddReorder.c – upravení cesty k hlavičkovým souborům
cuddSat.c – upravení cesty k hlavičkovým souborům
cuddSign.c – upravení cesty k hlavičkovým souborům
cuddSolve.c – upravení cesty k hlavičkovým souborům
cuddSplit.c – upravení cesty k hlavičkovým souborům
cuddSubsetHB.c – upravení cesty k hlavičkovým souborům
cuddSubsetSP.c – upravení cesty k hlavičkovým souborům
cuddSymmetry.c – upravení cesty k hlavičkovým souborům
cuddTable.c – upravení cesty k hlavičkovým souborům
cuddUtil.c – upravení cesty k hlavičkovým souborům
cuddWindow.c – upravení cesty k hlavičkovým souborům
cuddZddCount.c – upravení cesty k hlavičkovým souborům
cuddZddFuncs.c – upravení cesty k hlavičkovým souborům
cuddZddGroup.c – upravení cesty k hlavičkovým souborům
cuddZddIsop.c – upravení cesty k hlavičkovým souborům
cuddZddLin.c – upravení cesty k hlavičkovým souborům
cuddZddMisc.c – upravení cesty k hlavičkovým souborům
cuddZddPort.c – upravení cesty k hlavičkovým souborům
cuddZddReord.c – upravení cesty k hlavičkovým souborům
cuddZddSetop.c – upravení cesty k hlavičkovým souborům
cuddZddSymm.c – upravení cesty k hlavičkovým souborům
cuddZddUtil.c – upravení cesty k hlavičkovým souborům

adresář dddmp:

dddmpBinary.c – upravení cesty k hlavičkovým souborům
dddmpConvert.c – upravení cesty k hlavičkovým souborům
dddmpDbg.c – upravení cesty k hlavičkovým souborům
dddmpDdNodeBdd.c – upravení cesty k hlavičkovým souborům
dddmpDdNodeCnf.c – upravení cesty k hlavičkovým souborům, přejmenování některých funkcí
dddmpLoad.c – upravení cesty k hlavičkovým souborům
dddmpLoadCnf.c – upravení cesty k hlavičkovým souborům
dddmpNodeAdd.c – upravení cesty k hlavičkovým souborům
dddmpNodeBdd.c – upravení cesty k hlavičkovým souborům
dddmpNodeCnf.c – upravení cesty k hlavičkovým souborům
dddmpStoreAdd.c – upravení cesty k hlavičkovým souborům
dddmpStoreBdd.c – upravení cesty k hlavičkovým souborům
dddmpStoreCnf.c – upravení cesty k hlavičkovým souborům
dddmpStoreMisc.c – upravení cesty k hlavičkovým souborům
dddmpUtil.c – upravení cesty k hlavičkovým souborům

adresář epd:

epd.c – upravení cesty k hlavičkovým souborům

adresář include:

(všechny hlavičkové soubory z CUDDu byly přesunuty do tohoto adresáře)

bnet.h – upravení cesty k hlavičkovým souborům
cudd.h – upravení cesty k hlavičkovým souborům, `#include <stdio.h>`
cuddInt.h – upravení cesty k hlavičkovým souborům
cuddObj.hh – přejmenování na cuddObj.h, přidány funkce z nanotravu, přidání metody *plaLoad()*, přidání tří nových tříd – *Heap, partTR, PlaLoadInfo*
dddmp.h – upravení cesty k hlavičkovým souborům
dddmpInt.h – upravení cesty k hlavičkovým souborům, přidání prototypů některých funkcí
epd.h – bez změn
mtr.h – přidání `#include <stdio.h>`
mtrInt.h – upravení cesty k hlavičkovým souborům
ntr.h – upravení cesty k hlavičkovým souborům
st.h – bez změn
util.h – přidání `#include <stdlib.h>, #include <stdlib.h>` , odstranění `#include <unist.h>, #include <ctype.h>`

adresář mtr:

mtrBasic.c – upravení cesty k hlavičkovým souborům
mtrGroup.c – upravení cesty k hlavičkovým souborům

adresář nanotrav:

bnet.c – upravení cesty k hlavičkovým souborům
chkMterm.c – upravení cesty k hlavičkovým souborům
main.c – upravení cesty k hlavičkovým souborům
ntr.c – upravení cesty k hlavičkovým souborům
ntrBddTest.c – upravení cesty k hlavičkovým souborům
ntrHeap.c – upravení cesty k hlavičkovým souborům
ntrMflow.c – upravení cesty k hlavičkovým souborům
ntrShort.c – upravení cesty k hlavičkovým souborům
ntrZddTest.c – upravení cesty k hlavičkovým souborům
ucbqsort.c – upravení cesty k hlavičkovým souborům

adresář obj:

cuddObj.cc – přejmenování na cuddObj.cpp, implementace funkcí z nanotravu, implementace metod nových tříd, metoda na načítání pla souborů

adresář st:

st.c – upravení cesty k hlavičkovým souborům

adresář util:

cpu_stats.c – upravení cesty k hlavičkovým souborům, nahrazení `#include sys/time.h` za `#include <time.h>`, odstranění `#include<sys/resource.h >`
cpu_time.c – upravení cesty k hlavičkovým souborům
datalimit.c – upravení cesty k hlavičkovým souborům, nahrazení `#include sys/time.h` za `#include <time.h>`, odstranění `#include<sys/resource.h >`
getopt.c – upravení cesty k hlavičkovým souborům
pathsearch.c – upravení cesty k hlavičkovým souborům
pok.txt – upravení cesty k hlavičkovým souborům
prtime.c – upravení cesty k hlavičkovým souborům
ptime.c – upravení cesty k hlavičkovým souborům
safe_mem.c – upravení cesty k hlavičkovým souborům
saveimage.c – upravení cesty k hlavičkovým souborům
state.c – upravení cesty k hlavičkovým souborům
strsav.c – upravení cesty k hlavičkovým souborům
stub.c – upravení cesty k hlavičkovým souborům
texpand.c – upravení cesty k hlavičkovým souborům
tmpfile.c – upravení cesty k hlavičkovým souborům

Příloha B

Seznam přidaných tříd a metod do objektového rozhranní

Třídy:

```
Heap
PartTR
PlaLoadInfo
```

Metody – třída *BDD*:

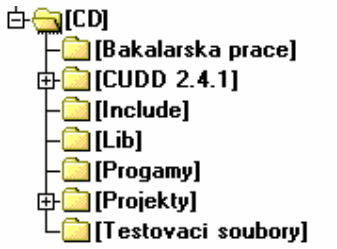
```
int ClosureTrav(NtrOptions * option)const
int Envelope(PartTR *TR, FILE *dfp, NtrOptions * option)const
int SCC(NtrOptions * option) const
int ShortestPaths(NtrOptions * option)
int TestClipping(const BDD * bdd,NtrOptions * option) const
int TestClosestCube(NtrOptions * option) const
int TestCofactorEstimate(NtrOptions * option) const
int TestDecomp(NtrOptions * option) const
int TestDensity(NtrOptions * option) const
int TestEquivAndContain(const BDD* bdd,NtrOptions * option) const
int TestMinimization(const BDD* bdd, NtrOptions * option) const
BDD TransitiveClosure(PartTR *TR, NtrOptions * option) const
int Trav(NtrOptions * option) const
int VerifyEquivalence(const BDD* bdd,NtrOptions * option) const
int buildDDs(const BDD * bdd,NtrOptions * option)
BDD getStateCube(char * filename, int  pr)const
BDD initState(NtrOptions * option) const
int maxflow(NtrOptions * option)
double maximum01Flow(DdNode * sx,DdNode * ty,DdNode * E,DdNode ** F,
                     DdNode ** cut, DdNode ** x,DdNode ** y, DdNode
                     ** z,  int  n,  int  pr)
```

Metody – třída *BDDvector*:

```
PlaLoadInfo* plaLoad(FILE *f)
```

Příloha C

Obsah přiloženého CD



Adresář	Popis
Bakalarska prace	Elektronická verze této práce
CUDD 2.4.1	Původní linuxová verze CUDDu
Include	Adresář s hlavičkovými soubory
Lib	Adresář s vytvořenými knihovnami pod operační systém Windows
Programy	Testovací a ukázkové programy
Projekty	Adresář s jednotlivými projekty
Testovaci soubory	Testovací soubory nahrávání *pla* souborů