České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Minimalizace neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů

Jan Bílek

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika kombinovaný magisterský

Obor: Výpočetní technika

Leden 2007

Poděkování

Velké díky Zdeňkovi P., Petrovi F. a Aleně F. za toleranci v zaměstnání, nezištnou odbornou pomoc, korekturu, pochopení a podporu.

Prohlášení

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

V Praze dne 18.1. 2007 ....

Abstract

This work implements modified binary decision diagrams (MBDs) and uses their structure to minimize incompletely specified logic functions. The product of this work is a software package, which is able to load logic functions from PLA files, minimize them and save them back to PLA file. Beside the minimization itself the package implements various static and dynamic variable ordering methods. This work introduces brand-new method for static variable ordering - STAT. Results of MBD minimization is experimentally compared with results of famous heuristic minimizer ESPRESSO.

Abstrakt

Tato práce se zabývá modifikovanými binárními rozhodovacími diagramy (MBD) a využitím jejich specifických vlastností při minimalizaci neúplně určených logických funkcí. Výsledkem práce je softwarový balík, který dokáže načítat logické funkce z PLA souborů, minimalizovat je a zpět uložit do PLA souboru. Vedle samotné minimalizace jsou v balíku implementovány metody na statické i dynamické řazení vstupních proměnných. Je zde představena nová metoda na statické řazení proměnných nazvaná STAT. Účinnost minimalizace logických funkcí pomocí MBD je porovnána s výsledky minimalizátoru ESPRESSO. Zároveň je vzájemně porovnána účinnost všech implementovaných technik na řazení proměnných.

Obsah

Seznam obrázků **xvii**

Seznam tabulek **xix**

1 Úvod **1**

1.1 Účel projektu a jeho pozadí 1

2 Základní definice **3**

2.1 Booleova algebra 3

2.2 Vyjádření logických funkcí 3

2.3 Formát PLA 6

3 Binární rozhodovací diagramy **7**

3.1 Binární rozhodovací diagramy 7

3.1.1 Konstrukce BDD 8

3.2 OBDD - uspořádané BDD 8

3.3 ROBDD - redukované uspořádané BDD 9

3.3.1 Vlastnosti ROBDD 9

3.4 Operace s ROBDD 11

3.4.1 Operace APPLY 11

3.4.2 Operace RESTRICT 11

3.5 Způsoby vyjádření neúplně určených funkcí pomocí BDD 11

3.5.1 Základní způsoby reprezentace neúplně určených funkcí pomocí ROBBD . 12

3.5.2 Reprezentace neúplně určených logických funkcí pomocí BDD_for_CF . . . 13

4 Modifikovaný binární diagram **15**

4.1 MBD 15

4.1.1 Vlastnosti MBD 15

4.1.2 Zvláštnosti MBD 17

5 Řazení vstupních proměnných **19**

5.1 Metody používané pro řazení proměnných 19

5.1.1 Statické metody řazení 19

5.1.2 Dynamické metody řazení 21

6 Minimalizace logických funkcí **23**

6.1 Klasické způsoby minimalizace 23

6.1.1 Minimalizace pomocí Booleovy algebry 23
6.1.2 Minimalizace s využitím map 24
6.1.3 Systematické minimalizace 24
6.2 Minimalizace pomocí MBD 25
6.2.1 Využití struktury MBD pro minimalizaci 27
6.2.2 Využití DCsetu při minimalizaci 27

7 Popis řešení **31**
7.1 Posloupnost operací při minimalizaci logické funkce pomocí MBD 31
7.2 Načítání a redukce MBD 32
7.3 Dynamické řazení proměnných 33
7.4 Přesměrování hran vedoucích do terminálního uzlu s hodnotou X 33
7.5 Boolevská minimalizace 33
7.6 Problém překrývajících se termů 34

8 Experimenty **35**
8.1 Řazení proměnných 35
8.1.1 Četnost výskytu počtu uzlů 35
8.1.2 Časová složitost načítání MBD 36
8.1.3 Účinnost řadících technik 36
8.1.4 Zhodnocení 39
8.2 Minimalizace logických funkcí 40
8.2.1 Minimalizace pomocí DCsetu 40
8.2.2 Celková minimalizace pomocí MBD 41
8.2.3 Zhodnocení 41

9 Implementace **43**
9.1 Použité nástroje a filozofie programu 43
9.1.1 Podporované platformy 43
9.2 Struktura programu 43
9.2.1 Node 44
9.2.2 MBD 45
9.2.3 MBDLIB 48
9.2.4 PLA 49

10 Vizualizace MBD **51**
10.1 Jazyk DOT 51
10.2 Graphviz 52
10.3 Nic není dokonalé 54
10.4 Jiné způsoby vizualizace 54

11 Závěr **55**

Seznam obrázků

2.1 Příklad PLA souboru . . . 6

3.1 Binární rozhodovací diagram funkce . . . 8
3.2 Redukce BDD . . . 10

4.1 MBD logické funkce a jeho pravdivostní tabulka . . . 16
4.2 Dva BDD pro logickou funkci s DCsetem . . . 16
4.3 Logické operace s DCsetem . . . 17

5.1 MBD pro různá pořadí vstupních proměnných . . . 20
5.2 Hlavní smyčka metody swap_down . . . 22

6.1 Karnaughova mapa logické funkce . . . 24
6.2 Základní kroky heuristiky ESPRESSO. . . . 26
6.3 Minimalizace s MBD . . . 28
6.4 Možnosti přesměrování hran vedoucích do X . . . 28

7.1 Implementace operace APPLY . . . 32
7.2 Implementace booleovské minimalizace . . . 34
7.3 Řešení překrývání termů z f_{on} a f_{dc} . . . 34

8.1 Četnost výskytu počtu uzlů - soubor 20i-1o-200t-0dc.pla . . . 36
8.2 Četnost výskytu počtu uzlů - soubor 20i-1o-200t-20dc.pla . . . 37
8.3 Četnost výskytu počtu uzlů - soubor 20i-1o-200t-30dc.pla . . . 37
8.4 Porovnání četností výskytu počtu uzlů pro všechny tři funkce . . . 38
8.5 Časová složitost načítání MBD . . . 38

9.1 Definice třídy Node . . . 45
9.2 Definice třídy MBD . . . 46
9.3 Definice metody MBD::get_node . . . 47
9.4 Definice třídy MBDLIB . . . 48
9.5 Definice třídy PLA . . . 50

10.1 MBD a jeho popis pomocí jazyka DOT . . . 53

Seznam tabulek

2.1 Zákony Booleovy algebry 4
2.2 Zápis Booleových operací 4
2.3 Pravdivostní tabulka pro logický výraz $f(a, b, c) = abc + \bar{a}bc + a\bar{b}\bar{c}$ 5

8.1 Účinnost řadících technik 39
8.2 Nahrazování hran vedoucích od uzlu s hodnotou X 40
8.3 Minimalizace pomocí MBD 42

9.1 Základní třídy programu 44

11.1 Parametry MBD programu 57

Kapitola 1

Úvod

Základním tématem následujících stránek jsou logické funkce a jejich reprezentace pomocí modifikovaných binárních diagramů (MBD). Úkolem bylo vytvořit vlastní softwarový balík, který bude pracovat s MBD a implementovat funkce, které by využily specifických vlastností MBD k minimalizaci logických funkcí.

1.1 Účel projektu a jeho pozadí

Minimalizace logických funkcí je obor, který se rozvinul s prvním praktickým uplatněním Booleovy algebry v praxi. Z algebraicky zadaných logických funkcí se přímo vychází nejen při návrhu číslicových obvodů, ale i v řadě jiných oborů, které s elektronikou, s níž jsou nejčastěji dávány do souvislosti logické funkce, nemají příliš mnoho společného. Prakticky všechny tyto obory mají požadavek, aby vyjádření logické funkce bylo co nejúspornější. Na tvaru logické funkce jsou přímo závislé velikosti struktur, které z ní vycházejí. Čím je její zápis menší, tím je menší i číslicový obvod nebo MBD.

Existuje řada různých *klasických* vyjádření logických funkcí jako je algebraický zápis, pravdivostní tabulka nebo mapy. V posledních dvaceti letech se s rozmachem výpočetní techniky začínají prosazovat tzv. *Binární rozhodovací diagramy*, z nichž vychází i předmět této práce - MBD. Využití tohoto způsobu vyjádření logických funkcí je nepřeberné, nicméně minimalizace nepatří mezi ty nejčastěji používané.

Úkolem této práce je nejen prozkoumat možnosti minimalizace pomocí MBD, ale také zjistit více o samotných MBD, které jsou v oblasti binárních rozhodovacích diagramů rovněž na okraji zájmu. Existuje řada programových balíků, které pracují s BDD jako CUDD [6] nebo BuDDy [7], ovšem ani jeden z nich přímo nepracuje s neúplně určenými funkcemi. Samozřejmě, že s těmito produkty je možné také neúplně určené funkce vyjádřit, ovšem pouze za použití méně efektivních postupů. V této práci je naopak zvolena přímá forma reprezentace neúplně určených funkcí, kterou představuje MBD.

Vytvářet úplně nový softwarový balík implementující MBD není triviální záležitostí, tudíž je vhodné předem upozornit, že snahou této práce není konkurovat výše zmiňovaným knihovnám. Při práci na MBD knihovně s podporou minimalizace logických funkcí bylo nutné vyřešit řadu poměrně složitých úkolů, jako je načítání MBD a jeho redukce, statické a dynamické řazení proměnných nebo například nahrazování uzlů. Vyřešit takový úkol za půl roku na 100% je pro jednoho člověka nemožné, proto nebylo u řady dílčích problémů zvoleno úplně nejlepší možné řešení[1], ale řešení, které se snáze implementuje. Díky takovým úlevám není pak možné tuto implementaci plně porovnávat s ostatními BDD balíky. I přesto, že je tato implementace MBD na půl cesty ke kompletnímu řešení minimalizace logických funkcí pomocí

[1] *Nejlepší* z pohledu časové složitosti nebo kvality výstupu.

MBD, se podařilo dosáhnout řady velmi zajímavých nebo překvapivých výsledků - a to jak v oblasti řazení proměnných, tak i u minimalizace logických funkcí. Při práci na tomto balíku byla dokonce objevena nová, dosud nikde nepopsaná metoda statického řazení proměnných - STAT.

Závěrem je nutné zdůraznit, že tento text vychází z teoretických základů položených v [1], kde byl MBD poprvé popsán důkladně MBD a kde byly řešeny a teoreticky zpracovány problémy jako minimalizace logických či řazení vstupních proměnných v MBD. Celá implementace MBD, o které je celá tato práce, je však vytvořena autorem naprosto samostatně a bez použití jakýchkoliv jiných knihoven nebo spolupracovníků.

Kapitola 2

Základní definice

Jak napovídá název této kapitoly, následující řádky budou věnovány základním definicím a používaným termínům. Účelem tohoto souhrnu je seznámit čtenáře se základními pojmy a především *přesně* vysvětlit, v jakém smyslu budou používány v dalších kapitolách - nikoliv ho tuto látku naučit.

2.1 Booleova algebra

Logické funkce, které využívají dvouhodnotovou logiku [1] se označují jako *Booleovy funkce*. Matematickým nástrojem k práci s Booleovými funkcemi je *Booleova algebra*.
Booleova algebra B je definována jako distributivní a komplementární svaz, který se skládá z:

- Konečné množiny logických proměnných ($a, b, c, d, \ldots, x, y, z$),
- Binárních operací logického součtu (označovaného + nebo $\vee$) a logického součinu (. nebo $\wedge$),
- Unární operace negace ($\bar{\ }$nebo $\neg$),
- a dvou nulárních operací v podobě logických konstant 0 a 1.

Pro takto definovanou Booleovu algebru platí základní zákony zobrazené v tabulce 2.1. Z nich se nechá odvodit řada dalších zákonů [8] včetně *Shannonova expanzního teorému*, o kterém bude řeč v dalších kapitolách.

Z předchozích řádků je víceméně patrné, jaký zápis logických operátorů byl v této práci použit, pro úplnost je kompletní seznam možností zápisu jednotlivých logických operací uveden v tabulce 2.2.

2.2 Vyjádření logických funkcí

Logická (Booleova) funkce n proměnných je definována jako $f : B^n \rightarrow B$, je-li B Booleova algebra. Logickou funkci lze zapsat několika způsoby. Dále v textu budeme hovořit o těchto:

- pravdivostní tabulka,
- logický výraz,
- mapa,

[1] Obor hodnot je 0, 1 .

Tabulka 2.1: Zákony Booleovy algebry

Zákon	Vztah
Komutativita	$a + b = b + a$ $a.b = b.a$
Asociativita	$(a + b) + c = a + (b + c)$ $(a.b).c = a.(b.c)$
Distributivita	$a + (b.c) = (a + b).(a + c)$ $a.(b + c) = (a.b) + (a.c)$
Neutralita 0 a 1	$a + 0 = a$ $a.1 = a$
Vlastnosti komplementu	$a + \bar{a} = 1$ $a.\bar{a} = 0$

Tabulka 2.2: Zápis Booleových operací

Operace	Zápis
logický součet	$a + b$
logický součin	$a.b$ nebo je vynechán - ab)
negace	$\bar{}$ ($\bar{a}$)

- vícerozměrná jednotková krychle,
- modifikované binární diagramy (MBD).

O mapách a krychlích zde bude zmínka jen okrajová. Hlavní pozornost bude věnována logickým výrazům, pravdivostním tabulkám a pochopitelně hlavnímu tématu této práce - MBD. Příklad jedné funkce zapsané pomocí pravdivostní tabulky a logického výrazu je v 2.3. Následuje seznam používaných výrazů a jejich význam.

literál Název pro logickou proměnnou a její doplněk (a a $\bar{a}$). Jde o logický výraz tvořený pouze jednou proměnnou.

term Term logický výraz tvořený pouze literály a jen jednou binární logickou operací (např. $a+b+c$ nebo abc). **Pozor**: Bude-li v dalším textu použit výraz *term*, pak jím bude myšlen pouze term tvořený proměnnými a logickými *součiny*.

logický výraz Bude-li řeč o logické funkci (nebo výrazu), vždy tím bude myšlena funkce (výraz), ve které se logické součiny vyskytují ve dvou úrovních. Jde o tzv. *dvouúrovňovou logiku*. Příklad:

$$ab + \bar{b}c + \bar{a}c$$

Tabulka 2.3: Pravdivostní tabulka pro logický výraz $f(a, b, c) = abc + \bar{a}bc + a\bar{b}\bar{c}$

Vstupy			Výstup
a	b	c	f(a,b,c)
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

Logický výraz může být zapsán řadou způsobů, z nichž v této práci bude nejpoužívanější tzv. *disjunktivní normální forma* (DNF). Pod tímto termínem se skrývá logický výraz tvořený logickými součty termů logických součinů (viz předchozí příklad).

logické proměnné V dalším textu budou používány výhradně logické proměnné s písmenem x a odlišené kladnými indexy od 0 do n - tedy: $x_0, x_1, x_2, ..., x_{n-1}, x_n$.

ONset, OFFset, DCset *ONsetem*, *OFFsetem*, *DCsetem* jsou myšleny množiny termů logické funkce, jejichž výstupem je logická 1 (ONset), 0 (OFFset) nebo jejichž výstup není určen (DCset). Tyto množiny se pro danou logickou funkci f značí jako f_{on}, f_{off} a f_{dc}.

implikant logické funkce Součinový term, který definuje logickou funkci. Každý term z množiny ONset je implikantem logické funkce.

přímý implikant Takový *implikant* logické funkce, který už není možné více zjednodušit, např. odebráním redundantního literálu.

Shannonův expanzní teorém Stojí na myšlence, že každá logická funkce f může být rekurzivně rozložena (expandována) následujícím způsobem:

$$f(x_0, x_1, ...x_n) = x_0.f_{high}(1, x_1, ...x_n) + \bar{x_0}.f_{low}(0, x_1, ...x_n)$$

V každém kroku se funkce rozpadne na dvě podfunkce f_{high} a f_{low}. Z každé z nich je vyloučena vybraná proměnná x_i a nahrazena 1 (f_{high}) nebo 0 (f_{low}). Výsledná funkce je pak určena součtem násobků těchto podfunkcí s literálem nahrazené proměnné - x_i pro f_{high} a $\bar{x}_i$ pro f_{low}.

if-then-else normálová forma Pokud se logický výraz úplně rozloží pomocí Shannonova expanzního teorému, pak je v takzvané „if-then-else normálové formě". Z toho vyplývá i existence *if-then-else* operátoru (zkráceně *ite*), pomocí něhož lze vyjádřit všechny logické operace. Například logický součin funkcí f a g se vyjádří jako $ite(f, g, 0)$ a logický součet jako $ite(f, 1, g)$.

```
.i 6
.o 1
.p 3
1--0-- 1-
-0--1- 01
--1--0 10
.e
```

Obrázek 2.1: **Příklad PLA souboru.** PLA soubor pro logickou funkci f se dvěma výstupy f^0 a f^1. Kde $f^0_{on} = (x_0\bar{x_3}, x_2\bar{x_5})$, $f^0_{dc} = \emptyset$; $f^1_{on} = (\bar{x_1}x_4)$, $f^1_{dc} = (x_0\bar{x_3})$.

2.3 Formát PLA

Formát PLA byl použit na načítání a ukládání MBD. Jde o velmi jednoduchý a přímý způsob dvouúrovňové logické funkce. Zde budou popsány jen jeho nejdůležitější použité vlastnosti. Více je možné najít v [19].

Funkce uložená pomocí PLA formátu je zapsána textovém souboru s koncovkou pla. Soubor se načítá po jednotlivých řádcích a každý řádek souboru pak reprezentuje určitý parametr nebo vlastnost logické funkce. Podporované parametry jsou:

.i [číslo] Číslo uvedené za parametrem „.i" udává počet vstupních proměnných.

.o [číslo] Číslo uvedené za parametrem „.u" udává počet výstupů logické funkce.

.p [číslo] Číslo uvedené za parametrem „.p" udává počet termů.

.type [typ] Typ definuje způsob definice logické funkce. V této práci je podporován pouze typ „fd". Ten označuje funkci zadanou pomocí ONsetu a DCsetu.

.e Označuje konec definice funkce.

Samotná funkce je definována pomocí seznamu termů, kde každý term je zadán takto:

-10-101 1-0,

kde část před mezerou určuje vstupní proměnné funkce a část za mezerou určuje výstup termu. Znak – označuje neurčený vstup/výstup, 0 ve vstupní části označuje negovaný literál proměnné a 1 označuje nenegovaný literál. 1 ve výstupní části určuje skutečnost, že daný term patří do ONsetu pro daný výstup. 0 ve výstupní části naopak říká, že term patří do OFFsetu daného výstup - s touto informací se však nijak nepracuje, jelikož program pracuje s množinami ONset a DCset. Příklad pla souboru je na obrázku 2.1.

Kapitola 3

Binární rozhodovací diagramy

Binární rozhodovací diagramy (anglicky Binary decision diagrams - BDD) jsou jedním ze způsobů reprezentace logické funkce. Jsou základním stavebním kamenem z nich odvozených struktur jako OBDD, ROBDD a pochopitelně i MBD. V této kapitole bude důkladně popsána struktura BDD i její vyspělejší formy OBDD a ROBDD.

Teoretické zázemí, které vedlo k současné definici BDD, se začalo vyvíjet už na přelomu 50. a 60. let minulého století. Velký pokrok v této oblasti nastal na konci 70. let, kdy se poprvé objevil název BDD v souvislosti s vyjádřením logických funkcí. Za otce praktického využití BDD je považován R. E. Bryant, který v polovině 80. let definoval BDD [2] tak, jak se používají prakticky dodnes.

3.1 Binární rozhodovací diagramy

BDD je kořenový orientovaný acyklický graf. Jeho uzly se dělí do tří kategorií:

- **Kořen**: Je pouze jeden - nevedou do něho žádné hrany.
- **Neterminální uzly**: Mají předky i potomky - hrany vedou z nich i do nich. Také se jim říká *rozhodovací* nebo vnitřní uzly. V dalším textu bude pod pojem *neterminální uzel* spadat i kořen.
- **Terminální uzly**: Hrany vedou pouze do nich.

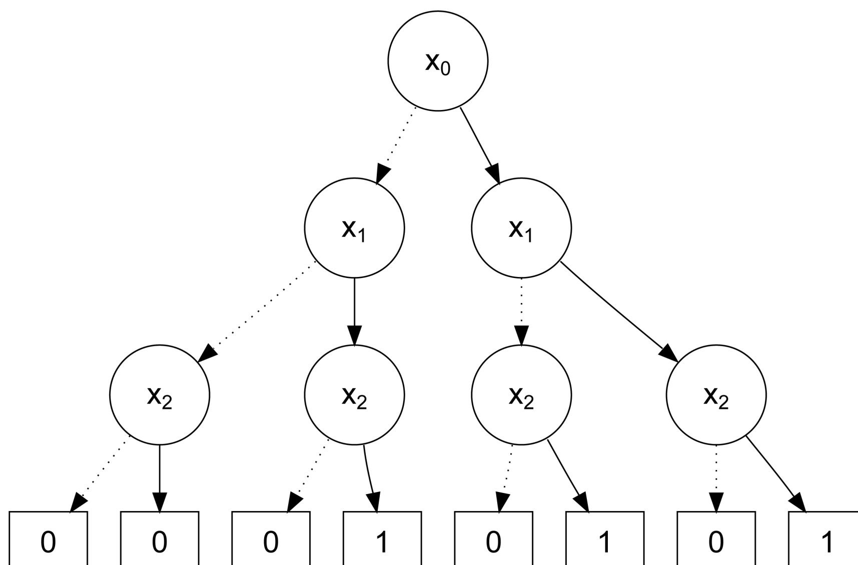
BDD je speciálním případem rozhodovacích diagramů, ve kterém má každý vnitřní uzel a kořen vždy dva potomky. Příklad jednoduchého BDD je znázorněn na obrázku 3.1.

Každý neterminální uzel představuje jednu vstupní logickou proměnnou a vedou z něj dvě hrany označované jako *high* a *low*. Hrana *high* odpovídá tomu, když hodnota proměnné uzlu je 1, a hrana *low*, když proměnná je nastavena na 0. Terminální uzly nabývají pouze dvou hodnot - 0 a 1.

S BDD se pracuje tak, že se prochází od kořene k terminálním symbolům, a podle toho, jakou má hodnotu terminální uzel, ve kterém takový průchod skončí, se určí výstupní hodnota funkce.

Pokud bychom chtěli z BDD na obrázku 3.1 získat hodnotu funkce pro vstupní proměnné $x_0 = 0, x_1 = 0, x_2 = 1$, pak bychom postupovali takto:

1. Začali bychom od kořene. Tomu patří proměnná x_0, která je rovná 0. Proto bychom se vydali po hraně *low* (představované tečkovanou čarou).



Obrázek 3.1: **Binární rozhodovací diagram funkce f.** $f(x_0, x_1, x_2) = \bar{x_0}x_1x_2 + \bar{x_0}x_1\bar{x_2} + x_0\bar{x_1}\bar{x_2}$. Tečkované čáry představují hrany *low*, plné čáry jsou hrany *high*.

2. V dalším uzlu s proměnnou x_1 (první zleva) se vydáme opět po hraně *low*, jelikož vstupní hodnota x_1 je také 0.

3. V uzlu s proměnnou x_2 (opět první zleva) se vydáme po hraně *high*, protože vstupní hodnota x_2 je 1.

4. Cesta končí v terminálním uzlu s hodnotou 0 (druhý zleva), což je výstupní hodnota pro zadané vstupní hodnoty.

3.1.1 Konstrukce BDD

BDD se vytváří rekurzivní aplikací Shannonova expanzního teorému. V každém uzlu se logická funkce rozloží podle proměnné, které uzel patří. Hrana *low* uzlu se nastaví na sčítanec, který se vytvoří nahrazením proměnné hodnotou 0, a hrana *high* povede do sčítance, kde byla proměnná nahrazena hodnotou 1.

3.2 OBDD - uspořádané BDD

Velmi záhy se v praxi ukázalo, že velikost BDD závisí na pořadí, v jakém se proměnné pří vytváření BDD zpracovávají. Pořadí proměnných se tedy stalo jedním z parametrůvlastností BDD. BDD, u nějž se vychází z nějakého pořadí, se říká *Ordered Binary Decision Diagram*[1] - OBDD.

Jsou-li proměnné x_i a x_{i+1} v uspořádání $x_i < x_{i+1}$, musí být v OBDD uzel patřící x_i *vždy* rodičem uzlu patřícímu x_{i+1} a nikdy naopak. Tato vlastnost je důležitým důsledkem uspořádání OBDD. Problematice řazení proměnných je věnována celá kapitola 5.

[1] Uspořádaný binární rozhodovací diagram.

3.3 ROBDD - redukované uspořádané BDD

Na obrázku 3.1 je možné si všimnout, že některé uzly jsou zbytečné. Logická funkce by se nezměnila, pokud by byly tyto uzly z BDD odstraněny. Například jde o první terminální uzel zleva s hodnotou 1 nebo první neterminální uzel zleva patřící proměnné x_2. Tyto zbytečné (redundantní) uzly je možné skutečně odstranit. Tomu se říká *redukce*. OBDD, v němž nejsou žádné redundantní uzly, se říká Reduced Ordered Binary Decision Diagram[2] - ROBDD.
Problém nadbytečných uzlů a jejich redukce byl důkladně popsán v [2]. Podle této práce lze redundantní uzly rozdělit do těchto kategorií:

- Terminální uzly se stejnou hodnotou.
- Neterminální uzel, jehož *high* a *low* hrany směřují do stejného uzlu (symbolicky pro uzel u: $high(u) = low(u)$).
- Dva či více podgrafů BDD jsou stejné. Pro dva stejné podgrafy s kořeny u a v v BDD platí, že musí patřit stejné proměnné ($var(u) = var(v)$) a jejich *high* a *low* hrany musí směřovat do stejných uzlů ($high(u) = high(v)$ a $low(u) = low(v)$).

Bryant rovněž stanovil postup, kterým se redundantní uzly odstraní. Postupuje se v těchto krocích:

1. **Odstraní se všechny zbytečné terminální uzly**: Na konci zbudou jen dva terminální uzly s hodnotami 0 a 1. Hrany vedoucí do ostatních terminálních uzlů se do nich přesměrují - všechny, které vedly do 0 (1), nyní povedou do jediného uzlu s hodnotou 0 (1).
2. **Odstranění stejných podgrafů**: Najdou-li se dva uzly u a v, pro něž zároveň platí: $var(u) = var(v)$, $high(u) = high(v)$ a $low(u) = low(v)$, pak se jeden z těchto uzlů odstraní, a všichni jeho rodiče se přesměrují do druhého[3].
3. **Odstranění všech uzlů se stejnými potomky**: Uzel u, pro něhož platí $high(u) = low(u)$, se úplně odstraní a všichni jeho rodiče se přesměrují do $high(u)$.
4. Kroky 2. a 3. se opakují do té doby, dokud je v BDD nějaký redundantní uzel. Odstraňování jednotlivých zbytečných neterminálních uzlů může vést ke vzniku nových zbytečných uzlů.

Jednotlivé kroky redukce jsou zobrazeny na obrázku 3.2. Výchozím stavem je BDD z obrázku 3.1.

3.3.1 Vlastnosti ROBDD

Výhody ROBDD jsou celkem zřejmé. ROBDD má méně uzlů, což nejen snižuje paměťové nároky na reprezentaci takové struktury[4], ale díky vypadnutí některých uzlů je i cesta od kořene k terminálnímu symbolu kratší.

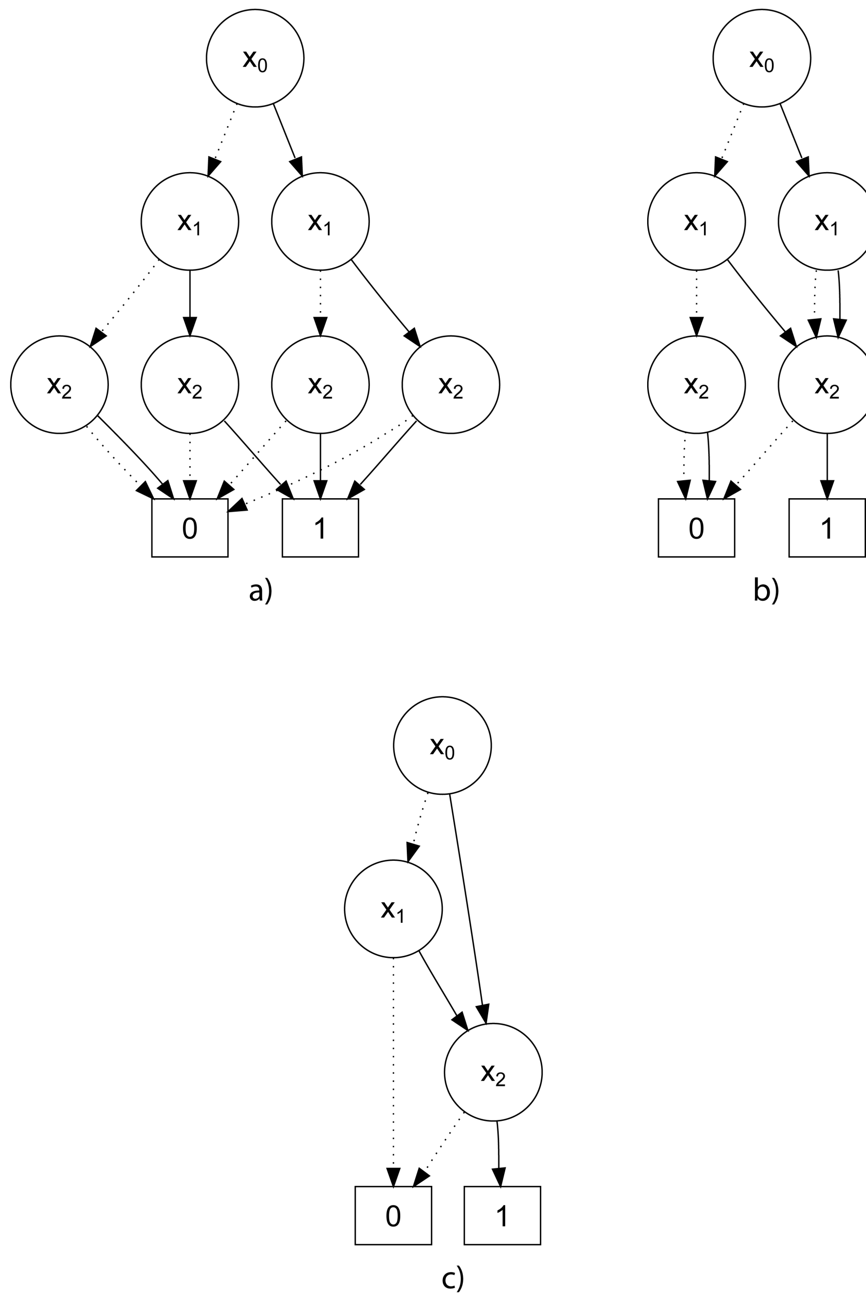
Přes to všechno je nejdůležitější vlastností ROBDD jeho kanonicita. Dva ROBDD vytvořené z jedné logické funkce musí být izomorfní [2]. Kanonicita ROBDD přináší řadu zajímavých aplikací ROBDD. Například je možné snadno se složitostí $O(1)$ provádět testy na tautologii či splnitelnost logických funkcí.

---

[2] redukovaný uspořádaný binární rozhodovací diagram

[3] Najednou se odstraňuje jen **jeden** uzel, nikoliv celý podgraf. Jsou-li dva grafy stejné, pak každý uzel v jednom grafu má svůj odpovídající uzel v druhém grafu. Všechny tyto uzly budou tedy nalezeny a odstraněny *po jednom*.

[4] V případě obrázku 3.2 je ROBDD o 10 uzlů menší než jeho neredukovaná verze.



Obrázek 3.2: **Redukce BDD.** Výchozím stavem je BDD z obrázku 3.1. Obrázek *a)* ukazuje stav BDD po odstranění zbytečných terminálních uzlů. Obrázek *b)* uvádí stav po odstranění stejných uzlů (2. a 3. uzel proměnné x_2). Obrázek *b)* předvádí konečný stav po odstranění uzlů, jejichž obě hrany vedou do jednoho uzlu.

3.4 Operace s ROBDD

Samotné ROBDD jsou v podstatě uzavřeným světem. Aby s nimi bylo možné lépe pracovat, bylo nutné definovat základní operace tak, aby bylo možné ROBDD spojovat, rozdělovat, případně přehazovat nebo odstraňovat proměnné.

Dvě základní operace byly definovány už v [2] - jde o APPLY a RESTRICT, které budou popsány bezprostředně na dalších řádcích. V kapitole 5 bude následně představena operace *swap*.

3.4.1 Operace APPLY

Operace APPLY pracuje tak, že mezi dvěma logickými funkcemi reprezentovanými dvěma ROBDD provede zadanou logickou operaci.

$$APPLY(f, g, op) = f < op > g$$

Důležité je připomenout, že výstupem operace APPLY je *jeden nový ROBDD* a nikoliv logická hodnota. Pravděpodobně vyvstanou námitky, že je možné získat hodnotu funkce f[5] i funkce g a pak mezi těmito provést zadanou operaci. Ano, to možné je, ale v praxi se ukázalo jako výhodnější mít co nejvíc částí jedné logické funkce v jednom ROBDD[6]. Důvodem je především rychlost takto prováděné operace. Pokud by každá část funkce (např. f_{on} a f_{dc}) byla oddělena ve svém ROBDD, muselo by se udělat tolik průchodů ROBDD, kolik by bylo částí funkcí. U částí funkcí *spojených* operací APPLY se provádí jen jeden průchod[7].

Operace APPLY hraje také zásadní roli při načítání ROBDD. Je v podstatě jediným snadným způsobem jak ROBDD načíst. Více se uplatněním APPLY při načítání ROBDD zabývá kapitola 7. Pro důkladnější popis APPLY je možné nahlédnout do [2] nebo [14]

3.4.2 Operace RESTRICT

Operace RESTRICT slouží k odstranění proměnné se známou (konstantní) hodnotou z ROBDD. Odstranění proměnné samozřejmě znamená vyjmout z ROBDD všechny uzly, které patří dané proměnné. Hrany vedoucí do vyjímaného uzlu se přesměrují podle toho, jaká je hodnota proměnné, které patří uzel v. Je-li rovna 1, pak se všichni rodiče přesměrují do $high(v)$, je-li rovna 0, přesměrují se do $low(v)$. Více informací i s obrázky je v [2] nebo [14].

3.5 Způsoby vyjádření neúplně určených funkcí pomocí BDD

Jak plyne z názvu této práce, zabývá se neúplně určenými logickými funkcemi. Bohužel ROBDD, tak jak byly popsané v předchozím textu, jsou navrženy jen pro úplně určené funkce. Chceme-li pomocí ROBDD vyjádřit neúplně určenou funkci, musíme ho zákonitě upravit. Tato podkapitola je věnována známým modifikacím ROBDD, s nimiž lze neúplně určené funkce vyjádřit.

Nejprve budou popsány základní způsoby reprezentace neúplně určených funkcí a pak bude vysvětlen jeden velmi zajímavý způsob vycházející z reprezentace logické funkce pomocí charakteristické funkce.

---

[5]pro dané vstupní hodnoty

[6]Každý term funkce může být reprezentován jedním ROBDD, ovšem není to efektivní.

[7]Časová složitost jednoho průchodu ROBDD je $O(n)$, kde n je počet proměnných.

3.5.1 Základní způsoby reprezentace neúplně určených funkcí pomocí ROBBD

V literatuře se vyskytují tři základní způsoby reprezentace neúplně určených funkcí pomocí ROBBD:

- Dvojice ROBDD reprezentující funkce f_{on} a f_{dc}.
- Přidání třetí výstupní hodnoty pro DC.
- Zavedení speciální proměnné na vyjádření DC.

Neúplně určené funkce pomocí dvou ROBDD Jde o základní možnost popisu neúplně určené funkce f pomocí ROBDD. Je-li logická funkce f *úplně* určená, pak f_{on} (logická funkce, která popisuje množinu takových ohodnocení vstupních proměnných, pro které logická funkce f vrací 1) stačí na vytvoření ROBDD a vyjádření logické funkce. Ovšem pokud je funkce určena neúplně, a vlastně do *hry* vstupuje třetí terminální hodnota pro neurčenou výstupní hodnotu, pak se pro vyjádření všech výstupních hodnot konstruuje druhý ROBDD pro množinu f_{dc}. Získání výstupní hodnoty (0, 1 nebo X) pak lze například pomocí *ite* (if-then-else) operátoru:

$$ite(f_{dc}, X, f_{on}) = (f_{dc} \wedge X) \vee (\neg f_{dc} \wedge f_{on}).$$

Tato metoda je popsána v [10]. Její nespornou výhodou je, že pro práci s neúplně určenými funkcemi lze použít naprosto stejné mechanismy jako pro práci s úplně určenými funkcemi. Prostě se jen pracuje se dvěma ROBDD a jejich výsledky se pak ještě vyhodnocují. Nevýhodami jsou velká redundance dat a omezená možnost redukce - snáze se redukuje jeden velký BDD než dva menší a oddělené.

Neúplně určené funkce pomocí třetí terminální hodnoty Tato metoda je jednoznačně nejjednodušší i nejelegantnější. Spočívá pouze v tom, že místo dvou terminálních hodnot 0 a 1 se zavedou tři: 0, 1 a X. Kde X představuje neurčenou výstupní hodnotu - *don't care*. Velkou výhodou tohoto způsobu je fakt, že množiny f_{on} a f_{dc} jsou v jednom grafu, což přispívá k lepší možnosti graf redukovat. Nevýhodou pochopitelně je, že funkce vrací tři hodnoty, s čímž musí počítat struktury, které s jejími výstupy pracují. Větší nevýhodou však je, že používané softwarové balíky pro práci s ROBDD takto definované ROBDD nepodporují. Tento způsob reprezentace neúplně určené funkce pomocí ROBDD byl představen v [12] a do hloubky se tím zabývá [1]. Celý tento text pracuje právě s touto reprezentací neúplně určených logických funkcí.

Neúplně určené funkce se zavedením speciální proměnné z Tento způsob je asi nejméně přímočarý a průhledný. Pro neúplně určenou funkci ff se zavádí speciální rozšířená (*extended*) funkce: $ext(ff) = (z \wedge ff_{dc}) \vee (\neg z \wedge ff_{on})$, kde z je rozšiřující (*extending*) proměnná představuje neurčené výstupní hodnoty. V ROBDD pro $ext(ff)$ pak proměnné z odpovídají uzly z_i (podle toho, v jaké výšce je). Na takto vytvořený BDD se použije operace *remove_Z*, která uzly z_i vyjme. Výsledný ROBDD je pak reprezentací neúplně určené funkce ff. Funkce *remove_Z* pracuje takto:

- Pro každý uzel z_i najdi jeho pokrytí $f(z_i)$ a vyjádři ho pomocí ROBBD.
- Všechny hrany, které vedly do uzlu z_i, pak přesměruj do ROBDD pro $f(z_i)$.

Výhodou této metody je její ohebnost. S proměnnou z lze různě manipulovat. Především je možné ji při vytváření ROBDD zpracovat v jiném pořadí, což pak podobně jako při zpracovávání obyčejného ROBDD může vést k jeho menší výsledné velikosti. Nevýhodou této metody je na rozdíl od předchozích dvou postupů její složitost. Tento postup byl důkladně vysvětlen v [11].

3.5.2 Reprezentace neúplně určených logických funkcí pomocí BDD_for_CF

V závěru je vhodné se alespoň okrajově zmínit o poměrně odlišném přístupu k této problematice. Stále se pro reprezentaci logické funkce používá BDD, ale to, z čeho se vytváří, je odlišné. BDD se nevytváří ze samotné logické funkce (určené pravdivostní tabulkou nebo součtem termů), ale z *charakteristické funkce* (characteristic function) dané logické funkce. Charakteristická funkce χ pro vícehodnotovou neúplně určenou funkci je definována takto:

$$\chi(\mathbf{X}, \mathbf{Y}) \ = \ \bigwedge_{i=1}^{m}((\neg y_i \wedge f_{off}^i) \ \vee \ (y_i \wedge f_{on}^i) \ \vee \ f_{dc}^i),$$

kde f_{set}^i představují jednotlivé výstupní funkce s indexem i, y_i je výstupní proměnná funkce f_{set}^i.

Z této funkce vznikne speciální struktura nazvaná BDD_for_CF, která je velmi podobná klasickému BDD, ovšem má pouze jeden kořen (pro vícehodnotovou funkci). Při jeho vyhodnocování se používají speciální pravidla (nestačí ho jen projít a vrátit hodnotu nalezeného terminálu). To je především proto, že některé uzly obsahují výstupní proměnné y_i. Více informací je v [13].

Kapitola 4

Modifikovaný binární diagram

V kapitole 3 byla popsána struktura ROBDD a také několik jejích modifikací umožňujících reprezentaci neúplně určených funkcí. V této kapitole bude hlouběji popsána struktura modifikovaného binárního diagramu - MBD, která byla zvolena pro vyjádření neúplně určených logických funkcí.

Hlavním důvodem, proč pracujeme právě s MBD, je jeho přímočarost a průhlednost, což je dáno tím, že jde o nejmenší úpravu ROBDD. Bude-li na dalších stránkách použit termín BDD, pak jím bude vždy automaticky myšlen ROBDD. Stejně tak pojem MBD bude přesně znamenat *Modifikovaný redukovaný uspořádaný binární rozhodovací diagram* - místo zkratky MROBDD bude použito jen MBD.

4.1 MBD

Velmi podrobný popis MBD je možné nalézt v [1], proto se tento text omezí jen na nejnutnější vztahy a definice. Je důležité zdůraznit, že úplná definice MBD pracuje s vícehodnotovými výstupy, zatímco implementace MBD v této práci pracuje jen s jednohodnotovým výstupem.

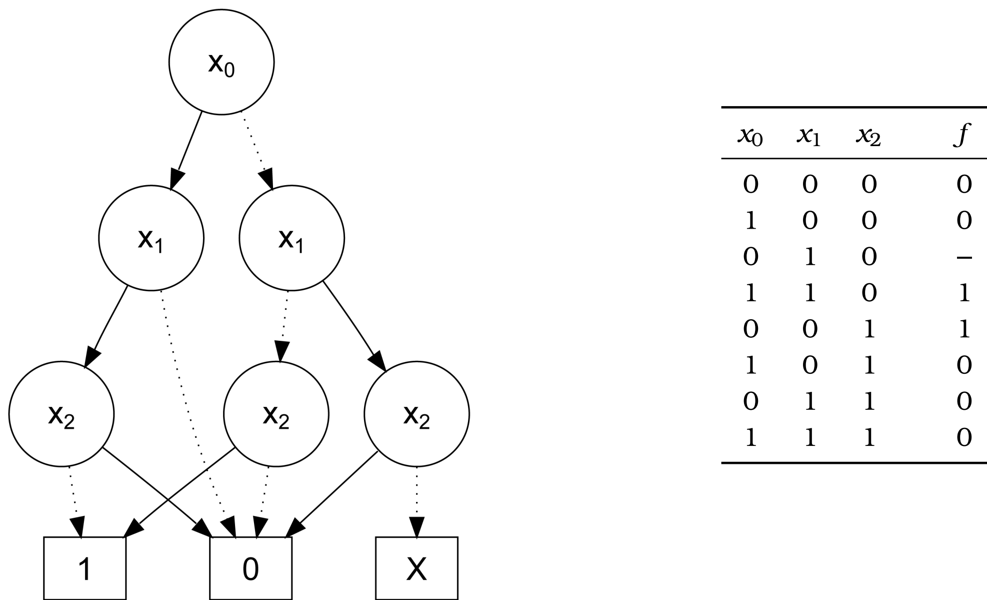
Zásadní změnou, kterou MBD přináší do BDD, je přidání třetího terminálu, který reprezentuje neurčené výstupy a označuje se X. Příklad MBD je na obrázku 4.1. Přítomnost terminálního uzlu X, je vynucena tím, že v MBD nejsou uloženy pouze termy z f_{on}, ale také termy z f_{dc}.

Výhodou takového řešení[1] je to, že se uspoří paměťový prostor potřebný na reprezentaci dvou BDD, ale především to, že integrace dvou BDD do jednoho může přinést nové redundance, které se odstraní a celkový počet uzlů se sníží. Tuto vlastnost dokumentuje obrázek 4.2. O tomto fenoménu byla zmínka i v kapitole 3 v souvislosti s operací APPLY. Tato shoda však není nijak překvapující, protože MBD se z f_{on} a f_{dc} vytváří pomocí APPLY s funkcí logický OR: $MBD(f) = APPLY(f_{on}, f_{dc}, OR)$.

4.1.1 Vlastnosti MBD

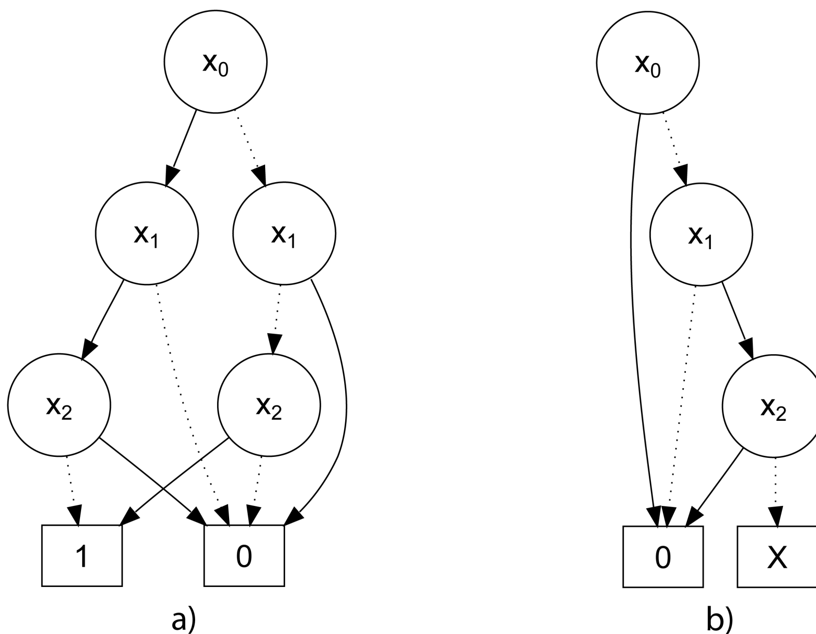
Ve zkratce lze říci, že vlastnosti a použití MBD jsou totožné s vlastnostmi a použitím BDD - pochopitelně s přihlédnutím k faktu, že MBD má navíc jeden terminální symbol. Jinými slovy se s MBD pracuje úplně stejně jako s BDD. Více je tato podobnost rozebrána v [1]

[1] Neúplně určené funkce je možné vyjádřit pomocí dvou různých BDD, kde jeden se vytvoří pro f_{on} a druhý pro f_{dc} - více v kapitole 3.



x_0	x_1	x_2	f
0	0	0	0
1	0	0	0
0	1	0	–
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	0
1	1	1	0

Obrázek 4.1: **MBD logické funkce** $f_{on} = x_0 x_1 \bar{x_2} + \bar{x_0}\bar{x_1} x_2$; $f_{dc} = x_0 \bar{x_1} x_2$ a jeho pravdivostní tabulka.



Obrázek 4.2: **Dva BDD pro logickou funkci s DCsetem.** BDD na obrázku *a)* zobrazuje f_{on} a BDD na obrázku *b)* zase f_{dc}. Jde o stejnou logickou funkci jako na předchozím vyobrazení. Je možné si všimnout, že celkový počet uzlů je zde o tři vyšší než u MBD.

AND	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

NEG	
0	1
1	0
X	X

Obrázek 4.3: **Logické operace s DCsetem.** Místo logických symbolů bylo použito anglických názvů. *AND* je logický součin, *OR* je logický součet a *NEG* je logická negace.

4.1.2 Zvláštnosti MBD

To, že se v MBD pracuje i s f_{on} a f_{dc}, má však určité nároky. Při provádění všech logických operací s terminálními uzly [2] se musí počítat s logickou hodnotou X. Tyto operace jsou vyjádřeny na obrázku 4.3.

Komplikacím, které se vyskytují při implementaci MBD, bude věnována část kapitoly 7.

[2] Například během operace APPLY s nějakou logickou funkcí - např. OR.

Kapitola 5

Řazení vstupních proměnných

Jak už bylo řečeno dříve, velikost výsledného MBD (tj. počet jeho vnitřních uzlů), je silně závislý na pořadí, v jakém se jednotlivé vstupní proměnné zpracovávají při načítání MBD.

Bez nadsázky lze říci, že nalezení alespoň dobrého, když ne nejlepšího pořadí, je největší problém, který se v této oblasti vyskytuje. Rozdíl ve velikosti MBD při různém pořadí vstupních proměnných ilustruje obrázek 5.1.

Maximální možná velikost MBD je $o(2^n)$, kde n je počet vstupních proměnných. Taková situace může nastat v případě, že je zvoleno nevhodné pořadí proměnných. Důsledky objemného MBD jsou zjevné. MBD se díky jeho paměťové náročnosti nemusí podařit načíst a velký MBD generuje velký počet výstupních termů, což je v rozporu s úkoly kladenými na minimalizaci logické funkce. V této souvislosti je vhodné připomenout, že velikost MBD se při použití jakkoliv nevhodného pořadí vstupních proměnných zvětšuje pouze do šířky a nikoliv do výšky. Maximální výška MBD s jakkoliv špatným pořadím vstupních proměnných je právě rovna počtu vstupních proměnných.

Problém, který způsobuje vznik velkých MBD, se nazývá *partitioning* [4] - česky *oddělení, rozdělení*. Velmi stručně řečeno jde o to, že proměnné, které jsou obsaženy v jednotlivých termech, jsou při načítání MBD zpracovávány ve velkém vzájemném odstupu, což dokumentuje obrázek 5.1. Úkolem metod na určení pořadí proměnných je zajistit, aby pořadí proměnných, které se vyskytují v jednom termu, nebylo příliš odlišné, aby proměnné nebyly příliš vzdálené.

Problém *oddělení* se vyskytuje především u funkcí, jejichž jednotlivé termy nejsou tvořeny literály *všech* proměnných.

5.1 Metody používané pro řazení proměnných

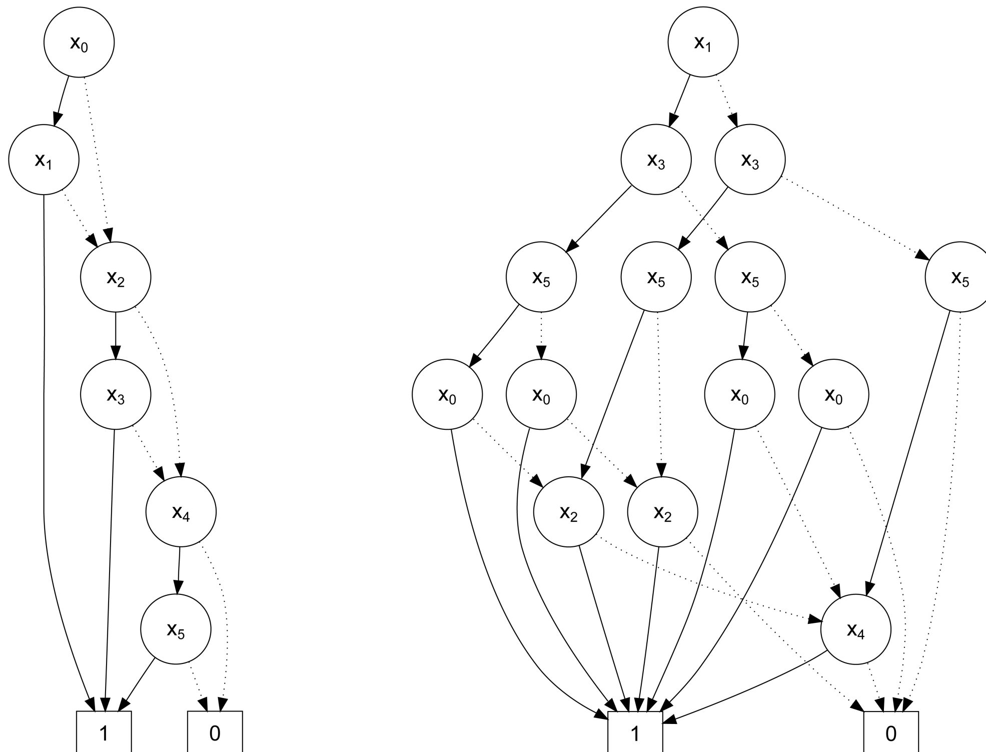
Metody, které se zabývají určováním pořadí vstupních proměnných, se dělí do dvou základních skupin: na *statické* a *dynamické*.

Statické metody určují pořadí proměnných přímo ze zadání logické funkce. Naproti tomu *dynamické* metody pracují s již vytvořeným MBD a manipulací s ním se ho snaží zmenšit.

5.1.1 Statické metody řazení

Ačkoliv nejsou statické metody tak účinné jako dynamické, jsou nezbytnou součástí každého softwarového balíku na práci s MBD. Jejich úkolem je nalézt nějaké vyhovující pořadí, které by umožnilo MBD alespoň načíst.

Složitost nejlepšího známého algoritmu určujícího nejlepší pořadí je $O(n^2 3^n)$ [3], což je podstatné vylepšení oproti hledání pořadí hrubou silou, jehož časová složitost je $O(n!2^n)$. Bohužel



Obrázek 5.1: **MBD pro různá pořadí vstupních proměnných.** Logická funkce $y = x_0x_1 + x_2x_3 + x_4x_5$ je načtena nejdříve s pořadím proměnných $P_1 = (x_0, x_1, x_2, x_3, x_4, x_5)$ a následně s pořadím proměnných $P_2 = (x_1, x_3, x_5, x_0, x_2, x_4)$.

časová složitost této metody je stále příliš vysoká na to, aby ji bylo možné použít v praxi.

Existuje řada heuristik, které tento problém řeší s polynomiální nebo dokonce lineární složitostí. Často používaná a citovaná je například metoda MINCE [4]. Její implementace je však velmi složitá. V softwarovém balíku, jehož se týká tento dokument, jsou implementovány dvě implementačně snadné heuristiky: FORCE [5] a STAT, která byla objevena při práci na tomto softwarovém balíku.

Pozn.: Autorovi této práce nebylo v době jejího vytváření známo, že by někdo takovou metodu dříve implementoval či popsal.

Metoda FORCE Tento přístup vychází z *nějakého* prvotního pořadí proměnných, které se pomocí iterací pokouší vylepšit. Jako metriku používá tzv. celkový *span* (česky *rozpětí*). Span je definován jako největší rozdíl pořadí dvou proměnných v jednom termu. Celkový span je pak součet spanů všech termů. Například pro funkci y z obrázku 5.1 je její celkový span pro pořadí P_1 $Span(y, P_1) = 1 + 1 + 1 = 3$ a pro P_2 je $Span(y, P_2) = 3 + 3 + 3 = 9$.

Algoritmus funguje tak, že nejdříve pro každý term spočítá tzv. *center of gravity* (COG - česky střed gravitační síly) a pak pro každou proměnnou vypočítá z jednotlivých COG její lepší pořadí. Tento postup se pak opakuje do té doby, kdy se celkový span přestane zlepšovat. Detailní popis metody je v [5].

Časová složitost FORCE je $O((|K|+|V| \log |V|) \log |V|)$, kde $|K|$ je počet termů a $|V|$ počet vstupních proměnných.

Metoda STAT Tento přístup se na celý problém dívá statistickým pohledem. Velmi zjednodušeně řečeno, algoritmus STAT pracuje tak, že proměnné, jejichž literály se nacházejí v největším počtu termů, dává co nejblíže k sobě a co nejblíže k začátku pořadí. V praxi se postupuje tak, že se pro každou proměnnou určí počet termů, v nichž se vyskytuje její literál, a proměnné se seřadí sestupně podle počtu výskytu literálů.

Algoritmus STAT:

1. $C_n = |K_n|$;
2. $order(n) = sort(C_1, ..., C_n)$,

kde K_n je množina termů obsahujících proměnnou s indexem n, C_n je počet výskytů proměnné s indexem n, $sort(...)$ provádí sestupné řazení proměnných podle počtu jejich výskytů a $order(n)$ je výsledné pořadí proměnných. Tato metoda vychází z předpokladu, že proměnné, které se nejčastěji vyskytují v termech logické funkce, budou s největší pravděpodobností i velmi blízcí sousedé.

Obrovskou výhodou této metody je, že nevychází z žádného původního pořadí a tudíž nemůže uvíznout v žádném *mělkém* lokálním minimu. Pro stejnou reprezentaci logické funkci určí vždy stejné pořadí. Problém takto definované metody STAT je v tom, že nedokáže určit pořadí proměnných, pokud mají stejné četnosti výskytu. Například pro funkci z obrázku 5.1 by vhodné pořadí určit nedokázala.

Z toho důvodu je STAT vylepšena o jednu nadstavbu, které se říká *Výběr nejlepšího souseda* (*Best neighbour selection* - BNS). Ta pracuje tak, že v množině už seřazených proměnných ještě přerovná ty proměnné, které mají stejný počet výskytů. Algoritmus postupuje od proměnných s nejnižším pořadím k proměnným s nejvyšším pořadím a za každou proměnnou se snaží vložit takovou proměnnou, která s nimi byla nejčastěji obsažena ve stejných termech. Toto vylepšení už získá dobré pořadí proměnných i pro funkci y z obrázku 5.1.

Algoritmus STAT - BNS:

1. $i = 0$
2. $s = BNS(order(i))$
3. $if(C_i == C_s)$ *then* $order(i + 1) = s$
4. $i = i + 1$
5. $if(i < n)$ *then goto* 2.,

kde funkce $BNS(v)$ určí pro každou proměnnou v jejího nejlepšího souseda.

Celková složitost upraveného algoritmu STAT je $O(|K||V| + |V| \log |V| + |V|)$, což je, byť zanedbatelně, horší než u FORCE.

5.1.2 Dynamické metody řazení

Dynamické metody pracují už s hotovým MBD, jehož velikost postupně snižují. Základní operací, která se v této oblasti používá, je tzv. *swap* (prohození) proměnných, které jsou v pořadí vedle sebe. Kvalita prohození se měří počtem vnitřních uzlů v MBD - čím je jich méně, tím je výsledek lepší.

Na operaci prohození jsou založeny všechny dynamické metody. Nejznámější heuristikou

```
while (improved)
{
    improved = false; old_size = mbd->size();
    for (i=0; i < mbd->get_input_var_num()-1; i++)
    {
        mbd->swap(i, i+1);          // prohození sousedních proměnných

        if (mbd->size() < old_size)
        {
            improved = true;        // nový MBD je menší
            old_size = mbd->size();
        }
        else mbd->swap(i, i+1);     // nový MBD je větší, prohoď nazpět
    }
}
```

Obrázek 5.2: **Hlavní smyčka metody swap_down.**

je tzv. sifting (prosévání) [9], který je implementován v balíku CUDD [6]. Vznikla i řada dalších metod založených na genetických algoritmech, simulovaném ochlazování a dalších stochastických metodách [1], které mají za úkol zamezit uvíznutí algoritmu v lokálním minimu. V této práci byl zvolen tzv. hladový přístup, který se pádu do lokálního minima nebrání a naopak do něho rychle směřuje.

Hladový přístup Tento elementární postup byl popsán důkladně v [1]. Je variantou známého hladového algoritmu [15], který v každém svém kroku vybírá stav s nejlepší ohodnocením.

V této práci je implementována metoda prohazování *swap_down* [1], jejíž hlavní část je pomocí pseudo C++ kódu zobrazena na obrázku 5.2. Skládá se ze dvou smyček, z nichž vnitřní (for) provádí postupné prohazování sousedních proměnných[1] v MBD (*swap*) od první v pořadí do poslední. Pokud takové prohození zmenší velikost MBD, pak se tento stav ponechá, jinak se vrací do staré „neprohozené" podoby. Vnější smyčka (*while*) pracuje do té doby, dokud ta vnitřní má nějaký účinek.

[1]Je-li index i i proměnné x^i její pořadí v seznamu proměnných, pak proměnné x^{i-1} a x^{i+1} jsou jejími *sousedy* (samozřejmě pokud vůbec existují).

Kapitola 6

Minimalizace logických funkcí

Jak už bylo řečeno v Úvodu, minimalizace logických funkcí je základním krokem v návrhu každého logického obvodu. Proto je této oblasti výzkumu věnována poměrně slušná pozornost.

V této kapitole budou nejdříve připomenuty klasické metody minimalizace a potom bude zvýšený zájem věnován minimalizaci neúplně určených logických funkcí pomocí MBD. Závěrem bude provedeno srovnání klasických způsobů minimalizace s metodou prováděnou na MBD. Ještě jednou je vhodné připomenout, že pojmem logická funkce se zde myslí disjunktivní normální forma (součin termů). Ta je jak vstupem pro minimalizaci, tak i jejím výstupem. Jedním dechem je dobré doplnit, že zminimalizovaný DNF zápis logické funkce nemusí vést k nejúspornější fyzické realizaci logického obvodu.

6.1 Klasické způsoby minimalizace

Prakticky všechny minimalizační metody pracují ve dvou hlavních krocích:

1. Nalezení množiny přímých implikantů[1].

2. Výběr minimálního počtu přímých implikantů nutných pro úplné pokrytí logické funkce.

Ať už jednotlivé metody pracují jakýmkoliv způsobem, vždy jde o to nalézt co nejméně nejmenších přímých implikantů.

6.1.1 Minimalizace pomocí Booleovy algebry

Tento způsob je vlastně uplatněním základních Boolevých zákonů na funkci zapsanou pomocí logického výrazu. Nejlépe celý postup představí menší příklad:
Je zadána funkce:

$$f = x_0 x_2 \bar{x}_3 + \bar{x}_0 x_1 x_2 + \bar{x}_2 x_3 + x_2 x_3.$$

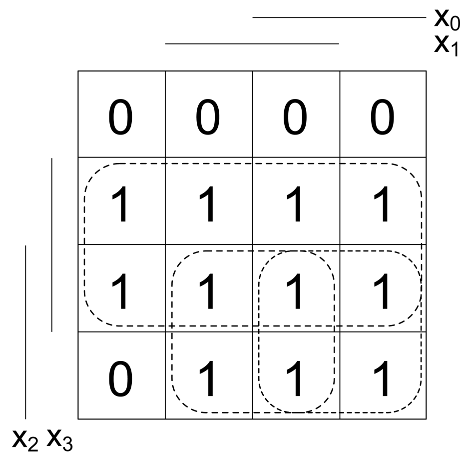
Uplatněním zákona sporu ($ab + \bar{a}b = b$) se upraví dva poslední členy:

$$f = x_0 x_2 \bar{x}_3 + \bar{x}_0 x_1 x_2 + x_3.$$

Uplatněním zákona absorbce negace ($a + \bar{a}b = a + b$) se zpracuje první a třetí člen:

$$f = x_0 x_2 + \bar{x}_0 x_1 x_2 + x_3.$$

[1] více o přímých implikantech v kapitole 1



Obrázek 6.1: **Karnaughova mapa logické funkce** $f = \bar{x}_0\bar{x}_1\bar{x}_2x_3 + \bar{x}_0x_1\bar{x}_2x_3 + x_0x_1\bar{x}_2x_3 + x_0\bar{x}_1\bar{x}_2x_3 + \bar{x}_0\bar{x}_1x_2x_3 + \bar{x}_0x_1x_2x_3 + x_0x_1x_2x_3 + x_0\bar{x}_1x_2x_3 + \bar{x}_0x_1x_2\bar{x}_3 + x_0x_1x_2\bar{x}_3 + x_0\bar{x}_1x_2\bar{x}_3$. Smyčky vybraných přímých implikantů jsou zvýrazněny přerušovanou čarou. Výsledná minimalizovaná funkce má tvar $f_{min} = x_3 + x_0x_2 + x_1x_2$.

Pomocí distributivního zákona lze z prvních dvou členů vytknout x_2 a následně obsah závorky zredukovat užitím zákona absorbce negace:

$$f = (x_0 + \bar{x}_0x_1)x_2 + x_3 = (x_0 + x_1)x_2 + x_3.$$

Po roznásobení závorky vznikne konečná minimalizovaná podoba funkce:

$$f = x_0x_2 + x_1x_2 + x_3.$$

Tento způsob je sice poměrně elegantní, ale není příliš systematický a nehodí se pro automatické zpracování nějakým počítačovým programem.

6.1.2 Minimalizace s využitím map

Metoda map je poměrně intuitivní grafický přístup, kdy se funkce zapíše do mapy. Příklad Karnaughovy mapy a minimalizace je na obrázku 6.1. Přímé implikanty se zde vyskytují v podobě smyček - účelem je najít co nejmenší počet co největších smyček.

Tato metoda je nevhodná pro automatické zpracování a při manuálním výpočtu začíná být silně nepřehledná od sedmi a více proměnných.

6.1.3 Systematické minimalizace

Předchozí metody mohou vypadat elegantně, ovšem ve skutečnosti nejsou příliš použitelné - nejsou totiž systematické. Za *systematické metody* lze považovat pouze takové postupy, které pracují přesně podle předem určeného algoritmu a nespoléhají na dobrý zrak, zkušenosti a intuici toho, kdo provádí minimalizaci. Jiný slovy jde o metody, které lze relativně snadno převést do počítačové podoby.

Metoda Quine-McCluskey Jde o první systematickou metodu, která sice není tak elegantní jako mapy, ale za to může pracovat s více proměnnými. Její velkou výhodou je, že vždy nalezne *přesné* řešení tj. nejmenší možnou funkci. Na druhé straně je poměrně časově náročná.

Metoda Quine-McCluskey pracuje takto:

1. Nejdříve nalezne množinu přímých implikantů.

2. Z nich pomocí tzv. *tabulky pokrytí* vybere takové, aby jich byl co nejnižší počet.

Ačkoliv je tato metoda poměrně dobře zpracovatelná na počítači[2], jde o NP-Těžký problém [16], což ji velmi limituje.

Heuristika ESPRESSO Jestliže je exaktní metoda špatně použitelná, pak musí dostat slovo heuristický přístup. Zřejmě nejúspěšnější minimalizační heuristikou je algoritmus nazvaný ESPRESSO, který byl vyvinutý ve spolupráci firmy IBM a univerzity v Berkeley v 80. letech minulého století. Jako každá heuristika nemá za cíl nalézt nejlepší možné řešení, ale alespoň nějaké *dobré* či *vyhovující*. ESPRESSO je určeno pro počítačové zpracování, a tak není příliš vhodné pro manuální minimalizaci.

Algoritmus pracuje při minimalizaci funkce f ve třech základních krocích:

1. Nejdříve se vytvoří nějaké (náhodné) pokrytí funkce f z několika vybraných (náhodně) přímých implikantů.

2. Toto pokrytí je pak *nějak* upraveno a následně zredukováno na minimální počet přímých implikantů.

3. Uloží se nejlepší nalezené pokrytí f a pokračuje se znovu v bodu 1, je-li potřeba.

Kouzlo metody ESPRESSO spočívá v tom, co činí ve druhém bodu algoritmu. *Úprava* daného pokrytí sestává ze tří kroků:

1. **Redukce** původního pokrytí.

2. **Expanze** pokrytí vytvořeného v bodu 1.

3. **Nalezení** minimálního počtu **přímých implikantů**.

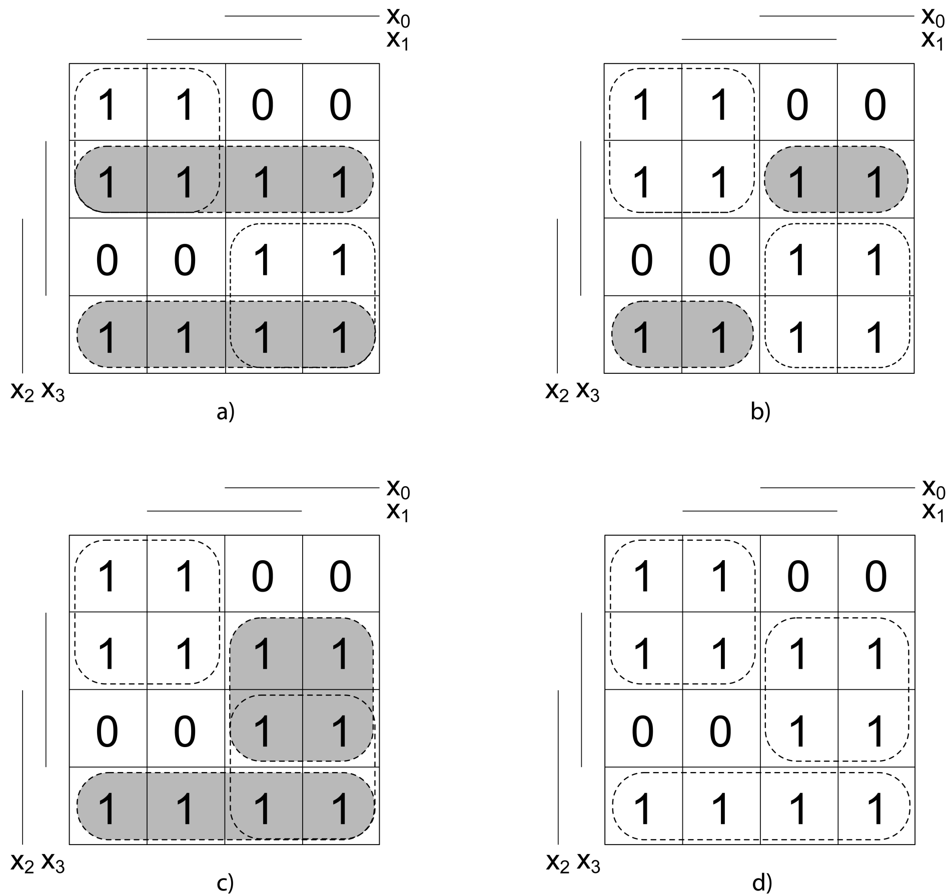
Celý algoritmus lépe popisuje obrázek 6.2. Pochopitelně, že tato hrubá kostra algoritmu ESPRESSO nemůže stačit na jeho úplné pochopení[3], nicméně jako úvod do moderního způsobu minimalizace by předchozí řádky stačit měly. Vzhledem k tomu, že ESPRESSO hraje významnou úlohu i v celé této práci, byla zmínka o něm nezbytná.

6.2 Minimalizace pomocí MBD

MBD je jen jedním ze způsobů zápisu logické funkce, tudíž i minimalizace prováděná na něm se svým základním postupem neliší od ostatních metod. To bude detailněji popsáno na dalších řádcích. Tento přístup je jemně upravenou variantou metody popsané v [1].

[2] Ostatně k tomuto účelu byla i navržena.

[3] Za Redukcí a Expanzí se skrývá řada dalších zajímavých postupů.



Obrázek 6.2: **Základní kroky heuristiky ESPRESSO.** Obrázek *a*) popisuje počáteční množinu přímých implikantů. *b*) ukazuje redukci přímých implikantů, při níž nemusí nutně vzniknout *přímé* implikanty, nýbrž jen implikanty. *c*) předvádí expanzi implikantů na přímé implikanty. *d*) zobrazuje výběr minimální množiny přímých implikantů.

6.2.1 Využití struktury MBD pro minimalizaci

Pokud chceme z MBD dostat zpět množinu termů (f_{on}, *ONset*), potom musíme procházet MBD od terminálního uzlu s hodnotou 1 do kořene. Všechny cesty, které projdeme, představují jednotlivé termy.

V každém uzlu, kterým se v cestě za kořenem procházíme, se provádí zpětná verze Shannonova expanzního teorému[4]:

$$f_u = x_u.f_{high} + \bar{x_u}.f_{low},$$

kde x_u je proměnná, které patří uzel u, a funkce f_{low} a f_{high} jsou získané z podgrafů $high(u)$ a $low(u)$.

Tomuto postupu se říká *algebraické procházení MBD*. Bohužel to však nemá nic společného s minimalizací, jelikož takto vzniklý logický výraz má často více termů a literálů než ten původní, ze kterého MBD vzniklo[5]. Aby při posledním spojení v kořenu vznikla minimalizovaná funkce, je nutné procházet jej booleovským způsobem - tj. v každém uzlu provádět navíc *booleovskou minimalizaci* funkce f_u. Tento postup dokumentuje obrázek 6.3. To, jaká minimalizace se provádí, je věcí volby. V tomto projektu byl použit externí minimalizátor ESPRESSO, který se volá v každém uzlu pro funkce f_u.

Důvodem, proč je použito ESPRESSO a nikoliv vlastní minimalizátor, je několik. ESPRESSO je vyzkoušený kvalitní algoritmus, který by se jen těžko podařilo překonat, navíc by práce na vlastním, byť jednoduchém, minimalizátoru silně přesáhla rámec tohoto projektu. Pochopitelně, že volání externího programu v programu s časovou složitostí $O(2^n)$[6] není nijak optimální, nicméně pokud je nutné prozkoumat skutečné možnosti využití MBD při minimalizaci logických funkcí, nebyla jiná cesta možná. Samotné ESPRESSO navíc posloužilo jako srovnávací program, s jehož výsledky byly porovnávány výsledky minimalizace dosažené pomocí MBD.

6.2.2 Využití DCsetu při minimalizaci

Stejně jako u jiných způsobů zápisu logické funkce, tak i u MBD představuje DCset užitečnou doplňující informaci, kterou je možné použít například při minimalizaci. Při minimalizaci pomocí map se DCsetu využívá k vytváření větších smyček, a tím pádem i menších konečných funkcí [8]. U MBD se neurčených výstupů využívá podobně, ovšem s přihlédnutím ke specifickým vlastnostem MBD.

Stručně řečeno, všechny uzly, které vedou do terminálního uzlu s hodnotou X (don't care), je možné odstranit, a tím zmenšit velikost celého MBD a přispět k celkově menší funkci. Odstranění uzlu spočívá v tom, že se hrana, která vede do X, přesměruje do jiného uzlu tak, aby se původní uzel stal redundantní. Hranu uzlu u vedoucí do X je možné přesměrovat dvěma způsoby (viz obrázek 6.4):

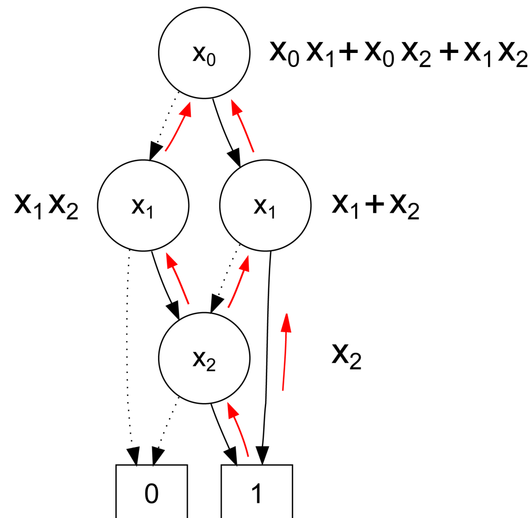
- Přesměrovat ji do stejného uzlu, do kterého vede druhá hrana uzlu u (vede-li $high(u)$ to X, pak se přesměruje do stejného uzlu jako hrana $low(u)$ nebo naopak). Tím, že obě hrany uzlu povedou do jednoho uzlu, stane se uzel u redundantním a je možné ho odstranit z MBD.

- Přesměrovat ji do jiného uzlu tak, aby z uzlu u vznikl uzel, který se už v MBD vyskytuje, čímž by se jeden z nich stal zbytečným.

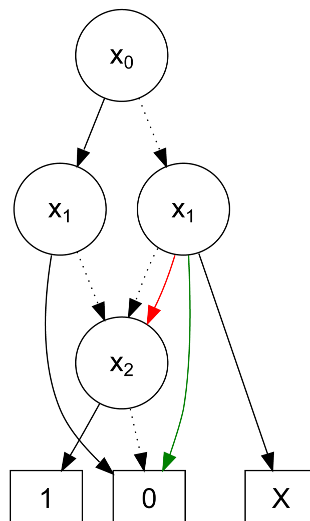
[4] tj. syntéza a nikoliv rozklad

[5] na vině je opět *partitioning* - viz kapitola 5

[6] Počet uzlů může být $O(2^n)$, ovšem v reálných situacích je mnohem menší.



Obrázek 6.3: **Minimalizace s MBD.** Booleovské generování termů - procházejí se všechny cesty od terminálu 1 do kořene a v každém uzlu se provádí booleovská minimalizace. Logické výrazy vedle uzlů ukazují minimalizovaný logický výraz pro daný podstrom. Červené šipky znázorňují směr průchodu MBD.



Obrázek 6.4: **Možnosti přesměrování hran vedoucích do** X**.** Červená hrana naznačuje možnost přesměrování hrany *high* do uzlu, kam vede hrana *low*. Zelená hrana naznačuje přesměrování do jiného uzlu tak, aby vznikl stejný uzel, jaký v MBD už existuje.

Ať se zvolí jakákoliv z těchto možností, vždy z MBD ubude alespoň jeden uzel. Není to málo, ale každá taková změna může dát vzniknout dalším redundantním uzlům. Vážnou otázkou tedy je, jaký krok v daném uzlu učinit[7]. Bohužel, neexistuje žádná metoda, jež by dokázala zjistit, která náhrada bude mít větší dopad na velikost MBD. Jedinou možností, jak se dostat k nejlepšímu možnému výsledku, je vyzkoušet všechny možné náhrady. Problém však je, že počet všech kombinací má faktoriální charakter, což není příliš vhodné pro běžné použití.

Úkolem této práce nebylo nalézt metodu vhodnou pro výběr náhrady za hranu vedoucí do X, nýbrž prozkoumat možnosti, které tyto náhrady skýtají. Více o tomto tématu prozradí výsledky experimentů, které jsou prezentovány v kapitole 8.

[7]Samozřejmě, že v MBD jsou také uzly, u nichž je možné provést jen první způsob nahrazení X, jelikož v MBD neexistuje uzel, na který by šlo původní uzel převést. Na druhé straně může být v MBD několik uzlů, které je možné převést původní uzel, a pak je znovu složitější si některou z možností vybrat.

Kapitola 7

Popis řešení

Vytvoření softwarového balíku na minimalizaci pomocí MBD je celkově velmi složitý problém a to zvláště v případě, když jde o originální výtvor bez použití jiných softwarových knihoven či jiných doplňků. Účelem této kapitoly je popsat postupy použité v této práci a zdůvodnit je. Jde o tedy o jakési nahlédnutí pod pokličku celé implementace. Pro jednoduchost zde nebudou popsány konkrétní používané datové struktury a zdrojové kódy, jejichž stručný komentovaný výpis je v kapitole 9.

7.1 Posloupnost operací při minimalizaci logické funkce pomocí MBD

V následujícím výčtu je posloupnost úkonů, které vykonává tato implementace při minimalizaci logické funkce. Zároveň jde o kroky, které musí vykonat program, který je výsledkem této práce (více o něm v kapitole 9). Některé body jsou podrobněji rozebrány v dalších oddílech této kapitoly.

Postup při minimalizaci:

1. **Načtení funkce ze souboru**: Funkce je načtena z PLA souboru do jeho vnitřní reprezentace. Vnitřní reprezentace PLA je popsána v kapitole 9.

2. **Určení pořadí vstupních proměnných**: Pomocí zvolené metody se určí pořadí vstupních proměnných, které pak bude použito k načtení MBD. Řazení je prováděno metodami popsanými v kapitole 5

3. **Načtení funkce do MBD**: Z vnitřní reprezentace PLA souboru je funkce načtena do MBD a rovnou redukována.

4. **Dynamické řazení proměnných**: Po načtení je použita metoda *swap_down* k přerovnání proměnných.

5. **Minimalizace funkce**: Je-li požadována minimalizace logické funkce, pak je provedena v této fázi. Do tohoto bodu spadá i přesměrování případných hran vedoucích do terminálního uzlu s hodnotou X.

6. **Uložení do souboru**: Funkce je načtena z MBD přes vnitřní reprezentaci logického výrazu a uložena do PLA souboru.

```
Node* APPLY(Node* u1, Node* u2)
{
    Node *low, *high, *n;

    // rekurzivní spuštění APPLY
    low = APPLY(u1->low, u2->low);
    high = APPLY(u1->high, u2->high);

    // redukce: 3. pravidlo
    if (low == high)
        return high;

    // redukce: 2. pravidlo
    if (n = node_hash_tab.find(u1.var_id, low, high))
        return n;

    // uzel je unikátní
    return new Node(u1.var_id, low, high);
}
```

Obrázek 7.1: **Implementace operace APPLY.**

7.2 Načítání a redukce MBD

Práce s MBD je poměrně ošemetná věc. Prakticky každý úkon v této oblasti má exponenciální složitost a výjimkou není načítání MBD. Zásadním a jedním z vůbec největších problémů, který je nutné při impelementaci MBD vyřešit, je načítání MBD. V každé literatuře je vznik MBD (BDD) popsán pomocí Shannonovy expanze. Bohužel, takový přístup je v praxi nepoužitelný. Jednak je jeho časová složitost velká[1], ale také jeho implementace by byla velmi složitá.

Pro účely této práce byl zvolen přístup založený na operaci APPLY. Každý term se nejprve samostatně převede do vlastního MBD a ty se pak cyklicky spojují dohromady pomocí operace APPLY s logickým součtem. Pochopitelně, že časová složitost je totožná s časovou složitostí Shannonovy expanze, ovšem implementačně jde o velmi jednoduchý úkol. Operace APPLY pracuje rekurzivně - v každém uzlu se znovu spouští pro oba syny a pak se výsledek dále zpracovává (jde o procházení grafu do hloubky).

Zásadní předností implementace MBD v této práci je to, že *redukce MBD se provádí* přímo při jeho načítání *v operaci APPLY*. Všechny tři kroky redukce popsané v kapitole 3 tedy není nutné provádět po načtení MBD. Výhodou tohoto postupu není jenom zmiňované odpadnutí sekundární redukce MBD, ale také to, že cyklická spouštěná rekurzivní operace APPLY v každém kroku vždy pracuje s redukovaným (tedy pro dané pořadí nejmenším možným) MBD, díky čemuž skutečná složitost načítání MBD je mnohem menší než $O(2^n)$. K této hranici se MBD skutečné logické funkce přiblíží jen zřídka. Zjednodušená verze operace APPLY je na obrázku 7.1.

Tato implementace operace APPLY ukazuje pouze dvě redukční pravidla. To první - odstranění redundatních terminálních uzlů, se zde taky provádí, ale bylo vynecháno[2]. Uplatnění třetího redukčního pravidla je, jak patrno z obrázku, rovněž triviální. Nejzajímavější částí operace APPLY je implementace druhého redukčního pravidla. Aby bylo možné v rozumném čase prohledat už vytvořený MBD a najít stejný uzel, byla k tomu použita hashovací

[1] $O(2^n)$

[2] Je triviální a čtenář si jej snadno domyslí - pokud ne, více je možné vyčíst v kapitole 9.

tabulka[3]. Klíče, s jakými jsou uzly do tabulky ukládány, se generují z identifikátoru proměnné, které uzel patří, a z ukazatelů na *high* a *low* podgrafy. Pokud se při operaci APPLY najde uzel s těmito stejnými ukazateli, pak operace APPLY vrací místo nově vytvořeného uzlu uzel nalezený. Výhodou hashovací tabulky není jen to, že umožňuje provádět velmi efektivní redukci MBD, ale také že jde o lineární reprezentaci, kde jsou jsou jednotlivé uzly uloženy v jednorozměrném vektoru. Hashovací tabulka se dále využívá při minimalizací DCsetu nebo při prohazování přehazování proměnných.

7.3 Dynamické řazení proměnných

K dynamickému řazení je použita metoda swap_down, jež je důkladně popsána v kapitole 5. Základní funkcí, jež je ve swap_down použita, je *swap*, jež slouží k prohození uzlů, které patří dvou sousedním proměnným. Při implementaci této metody se přítomnost hashovací tabulky ukázala jako velká výhoda.

Funkce pracuje tak, že při prohazování x_a^i x_b^{i+1} proměnných x_a^i x_b^{i+1} (s pořadím i a $i+1$) najde postupně všechny uzly proměnné x_a^i. Potom přesměruje jejich rodiče do uzlů proměnných x_b^{i+1} a jejich syny do synů uzlů x_b^{i+1}. Syny uzlů proměnné x_b^{i+1} přesměruje do uzlů x_a^i. Implementace této operace je ve skutečnosti poměrně složitá operace, ve které lze snadno udělat chybu. Díky hashovací tabulce je však vyhledávání uzlů velmi snadné.

Jiným vylepšením, které umožňuje snadnější implementaci swap, je to, že si každý uzel uchovává seznam odkazů na své rodiče - tj. ty uzly, které do něho směřují svou hranu *high* nebo *low*).

Zásadním poznatkem, který spolu se dvěma zmiňovanými vylepšeními umožňuje implementovat funkci swap se složitostí $O(m)$, kde m je počet uzlů MBD, je fakt, že změny, které po prohození proměnných v MBD vzniknou, se týkají pouze uzlů proměnných, které se účastnily prohození **??**. Jinými slovy lze říci, že po prohození uzlů proměnných x_a^i x_b^{i+1} mohou redundatní uzly vzniknout pouze mezi uzly proměnných x_a^{i+1} x_b^i, ale zbytek MBD zůstane nezměněn. Díky této skutečnosti není nutné po prohození proměnných provádět redukci celého MBD, ale stačí aplikovat redukční pravidla jen na uzly proměnných x_a^{i+1} a x_b^i, která se provádí přímo ve funkci swap.

7.4 Přesměrování hran vedoucích do terminálního uzlu s hodnotou X

V tomto úkolu není vůbec žádný zádrhel a jde o implementačně velmi jednoduchou úlohu. Jednoduše se najdou všechny uzly, jejichž jedna hrana směřuje do uzlu s hodnotou X, a provede se jedno z přesměrování popsaných v kapitole 6. Problémem zde je, že se musí následně provést redukce MBD, která odstraní vzniklé redundance.

7.5 Boolevská minimalizace

Boolevská minimalizace popsaná v kapitole 6 se provádí prakticky stejně, jak je zde napsáno. Jediný rozdíl je v tom, že MBD neprochází od terminálního uzlu s hodnotou 1, nýbrž rekurzivně

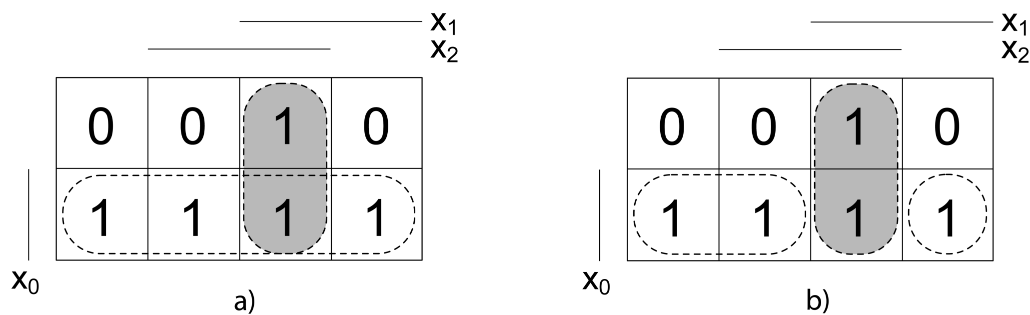
[3]Nejhorší časová složitost prohledání hashovací tabulky je díky kolizím $O(n)$ - tedy jako u prohledávání vektoru. Díky vhodně navrženému generátoru klíčů (aby generoval co nejméně kolizí) je reálná složitost $O(n/c)$, kde c je konstantní velikost hashovací tabulky. V případě MBD je nejhorší možné n známé předem, tudíž lze zvolit vhodné c, aby účinnost hashovací tabulky byla pro danou logickou funkci co nejvyšší.

```
void get_pla(PLA& pla, Node* n)
{
    PLA low, high;

    pla.get_pla(low, n->low);     // získá PLA pro podgraf low
    pla.get_pla(high, n->high);   // získá PLA pro podgraf high

    pla.merge(low, high);         // spojení obou PLA
    pla.minimize();               // boolevská minimalizace
}
```

Obrázek 7.2: **Implementace booleovské minimalizace.** Funkce ukládá logickou funkci do vnitřní reprezentace formátu PLA, jelikož minimalizátor ESPRESSO pracuje právě s formátem PLA. Minimalizátor ESPRESSO je spuštěn ve funkci PLA::minimize().



Obrázek 7.3: **Řešení překrývání termů z f_{on} a f_{dc}.** Obrázek *a)* ukazuje mapu funkce $f(x_0, x_1, x_2)$, kde $f_{on} = x_0$ a $f_{dc} = x_1x_2$ (zvýrazněno šedivě). Tímto způsobem *vidí* MBD logickou funkci. Průnik f_{on} a f_{dc} je v $f_p = x_0x_1x_2$. Obrázek *b)* zobrazuje vyloučení průniku z f_{on}, který se rozpadne na $f_{on} = x_0\bar{x_1} + x_0x_1\bar{x_2}$. Termům z f_{dc} byla dána přednost, tak se zde nic neměnilo.

od kořene pomocí algoritmu prohledávání do hloubky. Celý algoritmus je znázorněn na obrázku 7.2. Složitost tohoto procházení je $O(2^n)$, kde n je počet vstupních proměnných.

7.6 Problém překrývajících se termů

Při načítání funkcí s DCsetem do jednoho diagramu, se může objevit problém, který by se dal nazvat „překrývající se termy".

Máme-li funkci $f(x_0, x_1, x_2)$, která se skládá z $f_{on} = x_0$ a $f_{dc} = x_1x_2$, pak jeden z těchto termů zastíní při načtení funkce do MBD ten druhý. Při načítání MBD se totiž obě funkce chovají jako dvě plně určené logické funkce ($f = f_{on} + f_{dc}$), jediný rozdíl je v tom, že poslední hrana termu z f_{dc} vede do uzlu s hodnotou X místo 1. Tento rozdíl však MBD nebere v úvahu. V současné verzi je možné z překrývajících se termů vybrat pouze jeden, s tím že druhý úplně zanikne. Bohužel samotná struktura MBD jiné řešení nenabízí. V této práci byla dána přednost termům z DCsetu, které se v MBD prosadí na úkor termů z ONsetu.

Jedinou možností, jak získat plně korektní řešení, je analyzovat logickou funkci na vstupu, nalézt průniky mezi termy z f_{on} a f_{dc} a odstranit je - tj. rozdělit některé termy, aby průniky neobsahovaly. Pokud chceme upřednostnit termy z f_{dc}, pak průniky vyjmeme z f_{on}. Celou operaci demonstruje s využitím map obrázek 7.3.

Kapitola 8

Experimenty

V této kapitole budou předvedeny výsledky experimentů a testů dříve popsaných řadících a minimalizačních metod.

8.1 Řazení proměnných

Dříve, než budou předvedeny samotné výsledky měření, je nutné vysvětlit, co a za jakým účelem se měřilo. Zásadním úkolem testů bylo zjistit, jak se MBD chová při načítání logické funkce za různého pořadí vstupních proměnných, a tím zároveň prozkoumat a ověřit účinnost statických a dynamických metod popsaných v předchozím textu.

8.1.1 Četnost výskytu počtu uzlů

Tři vybrané logické funkce[1] s různým počtem neurčených vstupních proměnných byly opakovaně načítány do MBD s náhodně vygenerovaným pořadím. Výsledkem tohoto testu je pro každou funkci histogram udávající četnost výskytu počtu uzlů v MBD, který je silně závislý na pořadí proměnných. Do těchto grafů byly také vyneseny výsledky statických metod FORCE[2] a STAT.

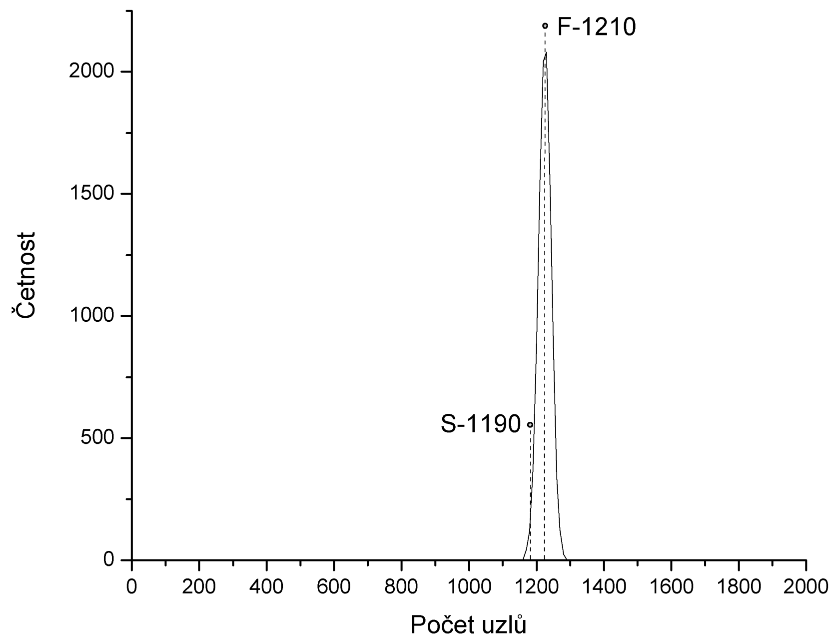
Byly použity tři PLA soubory, které měly všechny shodný počet vstupních proměnných - 20 a stejný počet termů - 200. Lišily se počtem neurčených vstupních proměnných. První soubor neměl žádné neurčené vstupní proměnné, druhý jich měl 20% a třetí 30% a všechny tři funkce byly plně určeny.

Testování probíhalo tak, že pro každou funkci bylo vygenerováno 10.000 náhodných pořadí a ty byly použity k načtení MBD. Pro každou funkci bylo rovněž použito statických řadících heuristik FORCE a STAT. STAT dává pro stejnou funkci vždy stejný výsledek, proto stačilo načíst funkci pouze jednou. FORCE vychází z náhodného pořadí, proto bylo nutné provést rovněž 10.000 načtení každé funkce s touto technikou řazení. Do grafů jsou vyneseny pouze průměrné hodnoty získané řazením FORCE. Získané charakteristiky jsou znázorněny na obrázcích 8.1, 8.2 a 8.3. Obrázek 8.4 nabízí srovnání všech charakteristik najednou.

Ze čtyř jmenovaných obrázků vyplývají dvě velmi zajímavé skutečnosti. Četnost výskytů počtů uzlů má přibližně průběh Gaussovy křivky, přičemž její nejvyšší bod (maximum) leží v průměrné hodnotě. Mnohem zajímavější však je, že s počtem neurčených vstupních hodnot se tato křivka nejen roztahuje a snižuje, ale také rapidně narůstá průměrný počet uzlů v MBD

[1] Funkce byly zadány v PLA souborech.

[2] Průměrná hodnota.



Obrázek 8.1: **Četnost výskytu počtu uzlů - soubor 20i-1o-200t-0dc.pla.** Funkce má 0% neurčených vstupních hodnot. „S" označuje hodnotu dosaženou algoritmem STAT a „F" metodou FORCE.

a s ním ruku v ruce stoupá i čas potřebný na načtení MBD. Zajímavé je i výkonnost řadící techniky STAT. U všech tří funkcí se její výsledek zobrazí do *levé paty* grafu, což znamená, že jde o velmi dobrý výsledek. U jedné z funkcí dosáhl algoritmus STAT absolutně nejlepšího výsledku z celkových 20.001 vyzkoušených pořadí[3].

8.1.2 Časová složitost načítání MBD

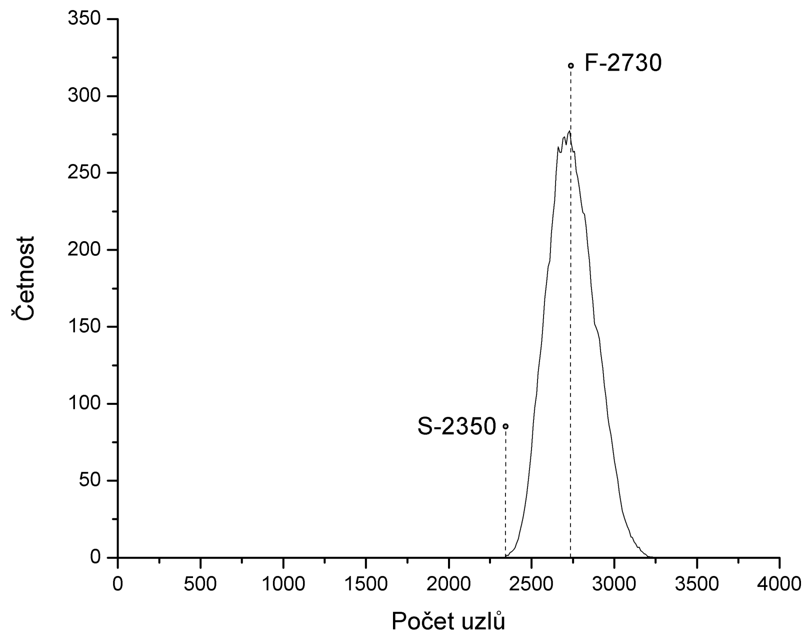
Tento experiment měl zjistit, jaká je složitost načítání MBD v závislosti na počtu uzlů. Byla vybrána jedna logická funkce, která byla opakovaně načítána, přičemž byl měřen čas této operace. Výsledek lze vidět na obrázku 8.5. Z něho se dá vyčíst, že rychlost načítání MBD v závislosti na jeho výsledném počtu uzlů je přibližně lineární. Pochopitelně, že z výsledků získaných měřením na jedné funkci, nelze činit žádné závěry.

8.1.3 Účinnost řadících technik

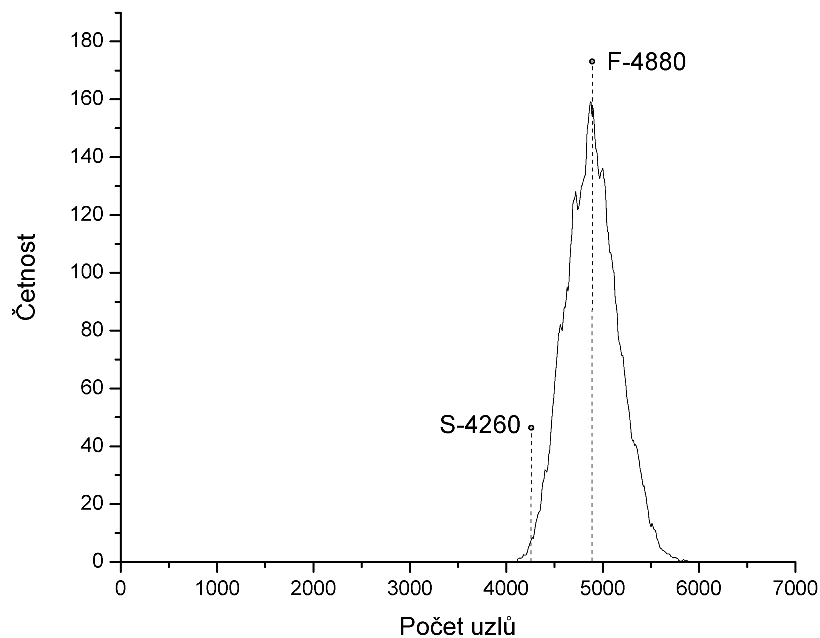
V tomto experimentu šlo o to porovnat jednotlivé techniky řazení - statické i dynamické. Účinnost je měřena počtem uzlů, které MBD obsahuje.

Experiment byl proveden pro několik různých funkcí, které měly různý počet vstupních proměnných, různý počet termů i různý počet neurčených vstupních hodnot. Byla testována jak náhodná volba pořadí, tak i statické metody FORCE a STAT. U náhodných pořadí a FORCE byla funkce načtena 50x a výsledkem byla její průměrná hodnota. Po statickém metodě a načtení MBD byla použita metoda dynamická. Výsledek experimentu znázorňuje tabulka 8.1.

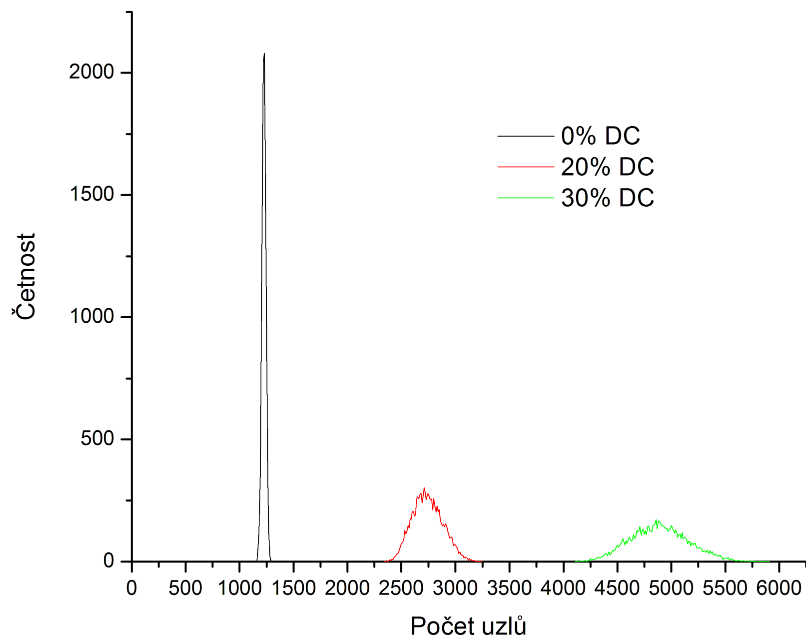
[3] 10.000 x Random, 10.000 x FORCE, 1 x STAT.



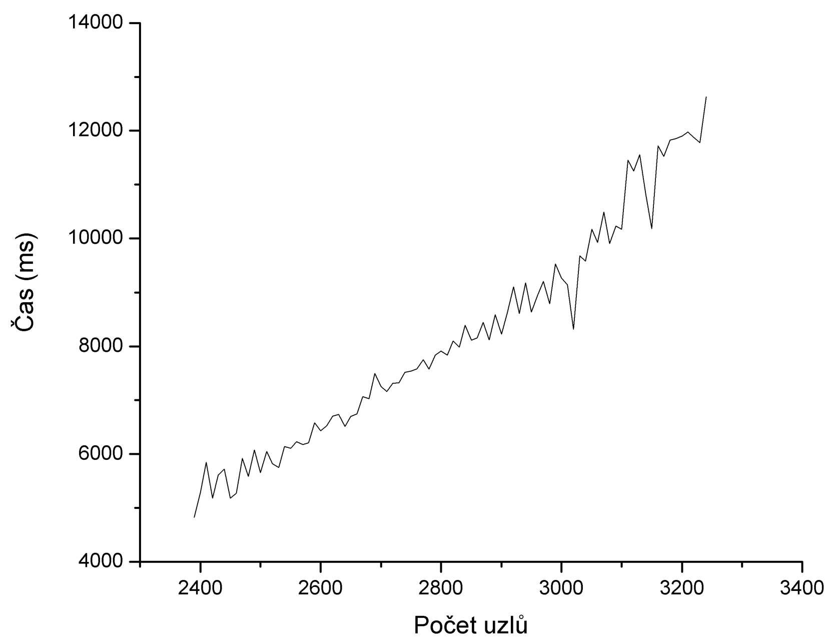
Obrázek 8.2: **Četnost výskytu počtu uzlů - soubor 20i-1o-200t-20dc.pla.** Funkce má 20% neurčených vstupních hodnot. „S" označuje hodnotu dosaženou algoritmem STAT a „F" metodou FORCE.



Obrázek 8.3: **Četnost výskytu počtu uzlů - soubor 20i-1o-200t-30dc.pla.** Funkce má 30% neurčených vstupních hodnot. „S" označuje hodnotu dosaženou algoritmem STAT a „F" metodou FORCE.



Obrázek 8.4: **Porovnání četností výskytu počtu uzlů pro všechny tři funkce.**



Obrázek 8.5: **Časová složitost načítání MBD** pro různé výsledné velikosti. Byla použita funkce uložená v souboru 20i-1o-200t-20dc.pla.

Tabulka 8.1: **Účinnost řadících technik.** *Random* - počet uzlů v MBD bez použití řadící techniky. *FORCE* - počet uzlů dosažených použitím statické heuristiky FORCE. *STAT* - výsledky dosažené použitím algoritmu STAT. *Normal* - počet uzlů v MBD bez použití dynamického řazení. *Swap* - výsledky dosažené s dodatečným dynamickým řazením. Tučně vytištěné hodnoty označují nejlepší dosažené výsledky pro jednotlivé funkce.

Soubor	Random		FORCE		STAT	
	Normal	Swap	Normal	Swap	Normal	Swap
6i-1o-10t-10dc.pla	19	**17**	21	19	20	19
10i-1o-100t-0dc.pla	132	128	132	**127**	136	131
10i-1o-100t-20dc.pla	189	183	190	187	186	**172**
10i-1o-100t-30dc.pla	167	162	166	**159**	166	161
12i-1o-80t-30dc.pla	453	429	459	434	416	**401**
12i-1o-100t-10dc.pla	316	303	317	305	304	**297**
15i-1o-100t-10dc.pla	552	526	552	525	510	**502**
15i-1o-200t-25dc.pla	1850	1746	1833	1748	1666	**1611**
18i-1o-200t-0dc.pla	883	**856**	884	854	913	870
18i-1o-100t-20dc.pla	1105	1010	1098	1013	992	**940**
20i-1o-200t-0dc.pla	1220	1184	1220	1182	1193	**1145**
20i-1o-200t-20dc.pla	2724	2510	2723	2470	2359	**2280**
20i-1o-200t-30dc.pla	4884	4401	4858	4382	4269	**4159**
30i-1o-300t-0dc.pla	4420	4337	4415	4329	4433	**4291**
30i-1o-300t-20dc.pla	6815	6451	6784	6438	6242	**6143**
30i-1o-300t-30dc.pla	11897	10665	11850	10769	10550	**9969**
50i-1o-300t-5dc.pla	10451	10342	10457	10348	10402	**10318**
Celkem	48077	45250	47959	45289	44757	**43409**
Celkem (%)	100	94,1	99,8	94,2	93,1	**90,3**

Z výsledku je dostatečně patrné, jak velký účinek má dynamické prohazování proměnných na snížení počtu uzlů v MBD. Ruku v ruce se statickou technikou STAT, bylo dosaženo absolutně nejlepšího celkového výsledku, když se podařilo snížit velikost MBD téměř o 10%. Propadákem byla metoda FORCE, jež dosáhla dokonce horších výsledků než při generovaném náhodném pořadí.

8.1.4 Zhodnocení

Výsledky dosažené v tomto oddíle jsou docela zajímavé, ale ne překvapivé. Už předem bylo víceméně jasné, že na pořadí vstupních proměnných hodně záleží, a zde provedené testy to jen dokazují. Jako vhodná volba řadících technik se zdá kombinace snadno implementovatelné a časově méně složité statické metody a vyspělejší dynamické metody. Ačkoliv zde představená dynamická technika měla své rezervy (uvíznutí v lokálním minimu), přesto ukázala svou nespornou sílu a potenciál. Statická metoda STAT, která vznikla při práci na tomto projektu, je velmi milým překvapením a alternativou ke všem ostatním, převážně mnohem složitějším metodám. Problémem metody FORCE je to, že používá velmi problematickou veličinu na určení kvality - *span*[4].

[4]Ponecháno bez přímého důkazu - jednoduše výsledky této metody hovoří za vše.

Tabulka 8.2: **Nahrazování hran vedoucích od uzlu s hodnotou X.** *DCset* - podíl termů funkce v %, které jsou v DCsetu. *X parents* - počet rodičů terminálního uzlu s hodnotou X. *Normal* - počet uzlů MBD po jeho načtení a redukci. *Brother* - výsledný počet uzlů MBD po přesměrování hran vedoucích do X do stejných uzlů, do kterých vedla druhá hrana uzlu. *Random* - výslednou velikost MBD, pokud byly hrany vedoucí do X přesměrovány do náhodně vybraného uzlu (jen do takového uzlu, aby vznikla nějaká redundance).

Soubor	DCset (%)	X parents	Normal	Random	Brother
5i-1o-20t-15dc-20odc.pla	20	4	14	7	3
8i-1o-40t-5dc-20odc.pla	20	3	67	54	50
8i-1o-40t-5dc-50odc.pla	50	3	63	37	37
10i-1o-50t-0dc-20odc.pla	20	2	109	95	95
10i-1o-100t-0dc-50odc.pla	50	4	160	120	102
16i-1o-100t-5dc-20odc.pla	20	5	600	486	464
16i-1o-100t-5dc-50odc.pla	50	2	596	397	362
20i-1o-200t-5dc-50odc.pla	50	4	1642	1061	933
20i-1o-200t-5dc-10odc.pla	10	4	1570	1394	1373
20i-1o-300t-15dc-20odc.pla	20	5	2985	2576	2483
30i-1o-200t-0dc-20odc.pla	20	2	3334	2765	2615
30i-1o-300t-0dc-20odc.pla	20	2	4617	3945	3765
30i-1o-300t-0dc-70odc.pla	70	2	4617	2317	1782
40i-1o-200t-0dc-20odc.pla	20	2	5297	4358	4240
40i-1o-200t-0dc-50odc.pla	50	2	5314	2001	1326
40i-1o-300t-0dc-20odc.pla	20	2	7681	6438	6183

Ze čtyř grafů plyne jeden zásadní, dříve ne tak patrný, závěr: Velikost MBD je nekompromisně přímo závislá na počtu neurčených vstupních hodnot. Čím je neurčených vstupních hodnot více, tím je MBD celkově větší a rozpětí jeho možných velikostí širší. U řídce definovaných funkcí[5] narůstá důležitost statických metod, jelikož rychlost dynamických technik je závislá na velikosti MBD - zde se každý předem ušetřený uzel velmi hodí.

8.2 Minimalizace logických funkcí

V tomto oddíle jsou prezentovány výsledky testů týkajících se minimalizace logických funkcí. První experiment se zabývá problémem nahrazování DCsetu a druhý komplexní minimalizací pomocí MBD.

8.2.1 Minimalizace pomocí DCsetu

Jak bylo napsáno již dříve v kapitole 6, při přesměrovávání hran vedoucích do terminálního uzlu s hodnotou Xm je možné si vybrat, zda bude hrana přesměrována do stejného uzlu jako její bratrská hrana, nebo jiného uzlu tak, aby v MBD vznikly dva stejné uzly. Účelem této operace je vytvoření jednoho nebo více redundantních uzlů, které by bylo možné odstranit.

Účelem tohoto experimentu bylo zjistit, zda je výhodnější přesměrovávat hranu vedoucí do X spíše do uzlu, do něhož vede její bratrská hrana (*Brother*) nebo do jiného uzlu

[5] S vyšším počtem neurčených vstupních hodnot.

(*Random*). Důležité je připomenout, že u každého uzlu s hranou vedoucí do X je možné použít přesměrování typu *Brother*. Vedle toho však může existovat více možností přesměrování *Random*, ale také nemusí existovat vůbec žádná. Výsledky získané metodou *Random* jsou průměrnou hodnotou získanou z padesáti různých měření - náhodně zvolených přesměrování Experiment, jehož výsledky prezentuje tabulka 8.2, ukazuje, že je výhodnější přesměrovávat uzly metodou jedna *Brother*. Metodou *Random* nebylo ani v jednom případě dosaženo lepšího výsledku než *Brother*. V této souvislosti je nutné dodat, že při použití *Random* došlo v některých jednotlivých případech k lepším výsledkům než u metody *Brother*. Ovšem tyto výsledky nemají žádnou praktickou hodnotu, když není známa metoda (kromě *Random* a vyzkoušení všech možností), která by takový výhodný způsob přesměrování dokázala určit.

Ačkoliv není rozdíl nijak velký, je nutné brát jej na zřetel. Jednoznačné vysvětlení tohoto jevu není známé. Jednu z možných hypotéz může představovat fakt, že uzly, jejichž jedna hrana vede do terminálního uzlu, leží *většinou* v hlubších vrstvách MBD (dále od kořene). Tudíž pokud se takový uzel stane po přesměrování hrany redundatním, může následně vzniknout více redundancí, jelikož mezi daným uzlem a kořenem MBD *většinou* leží více uzlů.

V tabulce 8.2 je možné si všimnout jednoho zajímavého jevu, při kterém pouze zlomek uzlů má hranu vedoucí do terminálního uzlu s hodnotou X, ačkoliv DCset logické funkce obsahuje velké procento termů. Po přesměrování těchto hran však dochází k odstranění velkého počtu uzlů, který přibližně odpovídá procentu termů v množině DCset.

8.2.2 Celková minimalizace pomocí MBD

Schéma experimentů v tomto oddíle je velmi jednoduché. Byla testována účinnost dříve vysvětlené minimalizace pomocí MBD, která byla porovnána s výsledky, jež dosáhl samotný algoritmus ESPRESSO.

Veličinami, jimiž byla měřena kvalita minimalizace, jsou celkový počet termů a celkový počet literálů ve všech termech minimalizované funkce. Výsledky tohoto experimentu jsou zobrazeny v tabulce 8.3.

Dosažené výsledky pomocí metody založené na MBD jsou prakticky totožné s těmi, které dosáhl program implementující algoritmus ESPRESSO. To není až tolik překvapivé, protože i MBD metoda pracuje s tímto minimalizátorem. U čtyř funkcí dosáhl MBD minimalizátor lepšího výsledku než ESPRESSO - ve všech případech měl o několik literálů méně než ESPRESSO (v tabulce zvýrazněno tučným písmem).

8.2.3 Zhodnocení

V této části práce bylo dosaženo dvou velmi zajímavých výsledků. Vzhledem k tomu, že není známa žádná metoda, která by vybrala správné přesměrování hran vedoucích do terminálního uzlu s hodnotou X, je výsledek prvního experimentu zajímavý v tom, že stačí přesměrovávát hrany výhradně metodou *Brother*. Ta totiž dosahuje lepších výsledků než *Random*. Také je možné vyzkoušet několikrát prohazování metodou *Random* a jednou metodou *Brother* a vybrat z nich ten nejlepší výsledek.

Ačkoliv to není na první pohled tolik zřejmé, zajímavých výsledků bylo dosaženo i při minimalizaci logických funkcí pomocí MBD. Ano, výsledky MBD byly jen o zlomeček lepší než ESPRESSO. Nicméně už ten fakt, že nebyly ani jednou horší, je silně povzbuzující. Je nutné si uvědomit, že tato implementace MBD má ještě rezervy v dynamickém řazení, při kterém by sofistikovanější metoda, jako třeba prosévání, možná dosáhla lepších výsledků.

V neprospěch minimalizace na druhé straně hovoří velká časová náročnost (exponenciální)

Tabulka 8.3: **Minimalizace pomocí MBD.** *Normal* - počet literálů a termů v původním PLA souboru. *ESPRESSO* - výsledky, kterých dosáhl minimalizátor založený na algoritmu ESPRESSO. *MBD* - výsledky dosažené pomocí metody využívající MBD. Všechny časy jsou v *sekundách.* Symbol „<1" znamená, že byl naměřen čas kratší než 1 s.

Soubor	Normal		ESPRESSO			MBD		
	Literály	Termy	Literály	Termy	Čas	Literály	Termy	Čas
5i-1o-20t-15dc.pla	86	20	4	1	<1	**3**	1	<1
8i-1o-40t-5dc.pla	304	40	131	19	<1	**130**	19	1
8i-1o-40t-5dc.pla	307	40	113	15	<1	113	15	1
10i-1o-50t-0dc.pla	500	50	329	34	<1	**328**	34	3
10i-1o-100t-0dc.pla	1000	100	393	40	<1	**390**	40	4
16i-1o-100t-5dc.pla	1520	100	1190	78	<1	1190	78	18
30i-1o-200t-0dc.pla	6000	200	4740	158	<1	4740	158	86
40i-1o-200t-0dc.pla	8000	200	6440	161	<1	6440	161	148
6i-1o-10t-10dc.pla	54	10	33	7	<1	33	7	<1
10i-1o-100t-0dc.pla	1000	100	691	72	<1	691	72	6
10i-1o-100t-20dc.pla	781	100	495	71	<1	495	71	9
10i-1o-100t-30dc.pla	682	100	349	62	<1	349	62	8
12i-1o-80t-30dc.pla	656	80	565	73	<1	565	73	24
12i-1o-100t-10dc.pla	1078	100	984	94	<1	984	94	16
15i-1o-100t-10dc.pla	1345	100	1344	100	<1	1344	100	27
15i-1o-200t-25dc.pla	2246	200	2171	196	<1	2171	196	120
18i-1o-200t-25dc.pla	2719	200	2687	198	<1	2687	198	236
18i-1o-200t-0dc.pla	3600	200	3543	197	<1	3543	197	40
20i-1o-200t-0dc.pla	4000	200	3979	199	<1	3979	199	48
20i-1o-200t-20dc.pla	3166	200	3166	200	<1	3166	200	233
20i-1o-400t-30dc.pla	5584	400	5556	399	1	5556	399	1784
30i-1o-300t-0dc.pla	9000	300	9000	300	<1	9000	300	189

této operace ve srovnání s ESPRESSEM (viz tabulka 8.3). Je-li čas důležitým parametrem při minimalizaci, tak samotné ESPRESSO je jedinou možnou volbou. Nezáleží-li na čase, nýbrž na každém ušetřeném literálu a termu, pak je možné použít i metodu popsanou v tomto oddílu.

Kapitola 9

Implementace

V této kapitole budou popsány detaily samotné implementace MBD. Nahlédneme přímo do zdrojových souborů. V této souvislosti je nutné připomenout, že zde uvedené výpisy zdrojového kódu mohou být v zájmu lepší čitelnosti zkráceny či jinak upraveny. Jejich kompletní verzi je možné nalézt na CD přiloženém k této publikaci.

Fyzickým výsledkem celé práce je kromě zdrojových kódů program, který v sobě mísí všechny zde popsané prvky. Jeho popis bude rovněž uveden v této kapitole.

9.1 Použité nástroje a filozofie programu

Vzhledem ke kritickým nárokům na výkon MBD, byl zvolen jazyk C++. K problému bylo přistoupeno hybridním přístupem, kdy část programu je psaná pomocí objektového postupu a část pomocí postupu strukturovaného - jde tedy o jakýsi mix C++ a C. Tento přístup si vynutil samotný problém. Zmiňovaný „mix" nejlépe vyhovuje požadavkům MBD. Jak kód psaný čistě v C, tak i striktně v C++, by byl poměrně neohrabaný.

Aplikace nepoužívá žádné externí knihovny, které by nebyly součástí C++, resp. C. Za zmínku stojí jistě mohutné použití šablon ze *Standard Template Library*, jde především o *vector*, *list* a *hash_map*[1].

9.1.1 Podporované platformy

Celý projekt byl stvořen pomocí vývojového prostředí*Visual Studio 2005* a je určen pouze pro 32bitovou platformu *Windows*. Vzhledem k tomu, že aplikace je psaná ve standardním C++ bez použití zásadních systémových závislostí, neměl by jeho přenos na jiné platformy činit žádný problém[2]. Stejný závěr lze učinit i v případě jeho přenesení na 64bitové systémy. V programu je sice jedna závislost na 32bitové systémy, na kterou bude v dalším textu upozorněno, ale její odstranění je snadné.

9.2 Struktura programu

V programu se používají čtyři základní třídy: *Node*, *MBD*, *MBDLIB* a *PLA*. Jejich význam je stručně shrnutý v tabulce 9.1, s tím že jejich detailní význam bude vyložen v pozdějších oddílech.

[1] *hash_map* Není zatím součástí standardu, a tak se podle upravené specifikace jazyka C++ firmou Microsoft nachází v prostoru jmen *stdext*

[2] Nicméně žádné pokusy s přenesením programu nebyly provedeny.

Tabulka 9.1: **Základní třídy programu.**

Třída	Popis
Node	reprezentace jednotlivých uzlů MBD
MBD	reprezentace celého MBD
MBDLIB	podpůrné a doplňující finkce k MBD
PLA	reprezentace logické funkce zadané pomocí PLA souboru

9.2.1 Node

Třída Node slouží na uložení uzlů MBD. Jde o poměrně jednoduchou spíše pasivní strukturu s velmi omezenou funkčností. Její definice je vidět na obrázku 9.1. Kde:

var_id Proměnná, do níž je uložen *identifikátor* vstupní proměnné, které uzel náleží. Identifikátory se přiřazují proměnným podle jejich pořadí v PLA souboru. Identifikátory 0, 1 a 2 jsou vyhrazeny terminálním uzlům s hodnotami 0, 1 a X.

node_id Unikátní identifikátor, který je každému uzlu přiřazen při jeho vzniku. V současné implementaci je tato proměnná nepotřebná, protože z ryze praktických důvodů se jako unikátní identifikátor používá 32bitový ukazatel na objekt uzlu. Při přechodu na 64bitové systémy je nutné brát tento fakt na zřetel.

hi a lo Odkazy na *high* a *low* podgrafy - reprezentují dvě výstupní hrany uzlu.

parents *Vector* identifikátorů všech uzlů (rodičů), které na daný uzel odkazují. MBD lze implementovat i bez této proměnné, ovšem její přítomnost velice urychluje řadu složitějších operací[3].

Node Dva konstruktory třídy. Jeden vytvoří uzel pouze s určeným identifikátorem vstupní proměnné a druhý umožňuje zadat i odkazy na podgrafy.

is_parent Test, zda uzel se zadaným unikátním identifikátorem uzlu, je rodičem uzlu.

rem_parent Funkce odstraní uzel se zadaným unikátním identifikátorem uzlu ze seznamu rodičů *parents*.

[3]Např. prohazování proměnných.

```
class Node
{
public:
    uint var_id, node_id;
    Node* hi, *lo;
    Node_Parents parents;

    Node(uint vid);
    Node(uint vid, Node* hi, Node* lo)

    bool is_parent(uint n_id);
    bool rem_parent(uint n_id);
    bool is_term();
};
```

Obrázek 9.1: **Definice třídy Node.**

is_term Test, zda uzel je terminální. Kontrola se provádí jednoduchým testem hodnoty identifikátoru, která musí být menší než 3, aby šlo o terminální uzel.

9.2.2 MBD

Zásadní třída celého programu sloužící na uložení MBD. Kromě samotného popisu dat disponuje i řadou metod, které nad těmito daty pracují. Její definice je na obrázku 9.2.

root Ukazatel na kořen MBD. Je pouze jeden - MBD nepodporuje vícehodnotové funkce.

term0 Ukazatel na terminální uzel s hodnotou 0. V každé chvíli existuje pouze jeden, který není nutné později redukovat.

term1 Ukazatel na terminální uzel s hodnotou 1.

termX Ukazatel na terminální uzel s hodnotou X.

input_num Počet vstupních proměnných logické funkce.

order2var *Vector* sloužící na převod pořadí→proměnná. Na poli s indexem i se nalézá identifikátor proměnné, jež je $i$tá v pořadí.

var2order *Vector* sloužící na převod proměnná→pořadí. Na poli s indexem i se nalézá pořadí proměnné, jejíž identifikátor je i. Jde o inverzní vektor k předchozímu.

map *Hash_map*, která slouží na ukládání uzlů podle určitého klíče.

```
class MBD
{
public:
    Node* root;
    Node* term0,*term1,*termX;
    uint input_num;
    MBD_VAR_ORDERING order2var, var2order;
    NodeMap map;

    uint load_pla_row(PLA_Row& row);

    Node* get_node(uint vid, Node* lo, Node* hi);

    uint swap_vars(uint v1, uint v2);

    void save_to_dot(char* filename);

    void get_pla(PLA& pla);
    void get_pla(PLA& pla, Node* n);

    uint cleanup(Node* d);
    uint cleanup();

    MBD(uint inputs);
    MBD(uint inputs, MBD_VAR_ORDERING);

    ~MBD();
};
```

Obrázek 9.2: **Definice třídy MBD.** „Ořezaná" verze pouze s nejdůležitějšími metodami.

load_pla_row Metoda načte do MBD jeden řádek z PLA souboru - tedy jeden term. Slouží při načítání celé logické funkce.

get_node Kriticky důležitá metoda celé třídy. Prakticky sama zajišťuje ukládání uzlů do *hash_mapy* a redukci MBD při jeho načítání. Její deklarace je na obrázku 9.3.

swap_vars Metoda sloužící na prohazování proměnných v MBD. Vstupními parametry jsou pořadí jednotlivých proměnných, které musí spolu sousedit.

save_to_dot Metoda uloží MBD ve formátu dot. Více v kapitole 10.

get_pla Metoda uloží MBD do struktury PLA. Pomocí parametru n je možné zadat kořen libovolného podgrafu, pro který se má struktura PLA uložit. Při vytváření PLA je automaticky prováděna minimalizace.

```
Node* MBD::get_node(uint vid, Node* lo, Node* hi)
{
    Node *t;
    NodeMap::const_iterator it;

    if (lo->node_id == hi->node_id) // redukce: 3. pravidlo
        return lo;

    if (map.end() != it = map.find(NodeKey(lo->node_id, hi->node_id, vid)))
    {
        // redukce: 2. pravidlo
        t = it->second.node;
    }
    else
    {
        // uzel je unikatni -> ulozeni do hash_mapy
        t = new Node(vid, hi, lo);

        lo->parents.push_back(t->node_id);
        hi->parents.push_back(t->node_id);

        map.insert( NodeMapPair(
                        NodeKey(lo->node_id, hi->node_id, vid), NodeVal(t))
                  );
    }

 return t;
}
```

Obrázek 9.3: **Definice metody MBD::get_node.**Metoda provádí redukci MBD při vytvoření každého uzlu. Uzly mapy se ukládají s klíčem, který je tvořen ze tří hodnot: id proměnné; id uzlu, kam směřuje hrana *high*; id uzlu, kam směřuje hrana *low*. Je-li požadavek na vytvoření uzlu se stejným klíčem, nový uzel se nevytváří a vrací se starý.

cleanup Metoda vyčistí MBD - tj. vymaže všechny uzly.

MBD Dva konstruktory třídy. Jeden přijímá pouze počet vstupů, pomocí druhého lze zadat i pořadí vstupních proměnných.

~**MBD** Destruktor třídy. Volá jen metodu *cleanup()*.

```
class MBDLIB
{
    uint static_ordering_method;
    uint x_subst_method;

public:
    // nacitani
    uint load_from_pla(char* filename, MBD** mbd, PLA** pla);
    uint load_from_pla(PLA* pla, MBD** mbd);

    void apply(MBD** dest, MBD* s1, MBD* s2, MBD_OP_Type op);
    Node* apply_OR(MBD* dest, Node* d1, Node* d2);
    uint reduce(MBD** mbd_dst, MBD* mbd_src);

    // razeni
    MBD_VAR_ORDERING ord_stat(PLA* pla);
    MBD_VAR_ORDERING ord_force(PLA* pla, MBD_VAR_ORDERING* io=NULL);
    MBD_VAR_ORDERING ord_random(PLA* pla);

    MBD_VAR_ORDERING order_vars_static(PLA* pla,
                                       MBD_VAR_ORDERING* io=NULL);
    uint order_vars_dynamic(MBD* io);

    uint get_span(PLA* pla, MBD_VAR_ORDERING& o);

    // minimalizace
    uint minimize_X(MBD** mbd);
};
```

Obrázek 9.4: **Definice třídy MBDLIB.**

9.2.3 MBDLIB

Třída funguje jako knihovna metod na provádění složitějších operací s MBD. Jsou zde funkce jako APPLY, metody na určení pořadí vstupních proměnných a minimalizaci. Pochopitelně, že je sporné, zda ta či ona funkce má být ve třídě MBD nebo v MBDLIB. Současné rozdělení metod mezi oběma třídami vyplynulo z potřeb, které se během práce objevily, a ze zkušeností, které byly v průběhu času získány. Definice třídy je na obrázku 9.4. Popis proměnných a metod následuje.

static_ordering_method Proměnná, jež uchovává typ statické metody, která se použije před načtením logické funkce. Jsou zde čtyři možnosti:

- *NONE*: Použije se přirozené pořadí z PLA souboru.
- *Random*: Pořadí se náhodně vygeneruje.
- *FORCE*: Pořadí určí technika FORCE.
- *STAT*: Pořadí najde algoritmus STAT.

x_subst_method Uchovává typ přesměrování hran vedoucích do terminálního uzlu s hodnotou X.

load_from_pla Funkce načte MBD z PLA souboru/struktury. Je možné načítat buď přímo ze souboru nebo z PLA objektu. Pokud se nějaká funkce načítá vícekrát, pak je možné ji jednou načíst ze souboru do PLA a pro další načítání použít v paměti uloženou strukturu PLA.

apply Implementace operace APPLY. Ze dvou vstupních MBD vytvoří pomocí zadané logické funkce jeden výstupní. V současné době je podporován pouze logický součet.

apply_OR Implementace operace APPLY s logickým součtem. Jde o základní funkci potřebnou při načítání MBD. Přesnější informace o načítání MBD budou popsány v dalších oddílech této kapitoly.

reduce Metoda provádí redukci načteného MBD podle pravidel popsaných v kapitole 3.

ord_stat Metoda určí pořadí vstupních proměnných technikou STAT.

ord_force Metoda určí pořadí vstupních proměnných technikou FORCE.

ord_random Metoda vygeneruje náhodné pořadí vstupních proměnných.

order_vars_static Podle toho, jaký typ řazení je nastaven v proměnné *static_ordering_method*, provede tato metoda řazení proměnných - tj. zavolá jednu z předchozích funkcí.

order_vars_dynamic Implementace dynamické metody řazení *swap_down*. Podrobnosti jsou v kapitole 5.

get_span Metoda získá ze zadaného PLA a pořadí proměnných veličinu *span*. Používá se ve funkci *ord_force()* - více v kapitole 5.

minimize_X Metoda provádí přesměrování hran vedoucích do terminálního uzlu s hodnotou *X*. Způsob přesměrování určuje hodnota proměnné *x_subst_method*.

9.2.4 PLA

Pro úplnost je uvedena i třída PLA, která reprezentuje logickou strukturu PLA souboru. Je mezičlánkem při načítání a ukládání logických funkcí do a z MBD. Definice třídy je na obrázku 9.5.

input_num Proměnná uchovává počet vstupních proměnných.

output_num Proměnná uchovává počet výstupních hodnot.

PLA Dva konstruktory třídy. Do jednoho je možné zadat počet vstupních proměnných a výstupních hodnot.

load Metodou slouží k načítání PLA ze souboru.

```
class PLA
{
public:
    uint input_num, output_num;
    vector<PLA_Row> rows;

    PLA();
    PLA(uint i_n, uint o_n);

    uint load(char* filename);
    uint save(char* filename);
    uint get_histogram(PLA_Input_Var_Histogram& hst);
    uint get_stats(PLA_Stats&);

    void or_pla(PLA& pla);
    void and_var(uint var, uint val);
};
```

Obrázek 9.5: **Definice třídy PLA.**

save Metodou slouží k ukládání PLA do souboru.

get_histogram Metoda získá z MBD četnost výskytu literálů proměnných. Využívá se ve statické metodě řazení vstupních proměnných STAT.

get_stats Metoda získá z PLA jednoduchou statistiku: celkový počet literálů a celkový počet termů.

Kapitola 10

Vizualizace MBD

Tato část práce vznikla jako vedlejší produkt celého projektu. Nicméně by byla škoda, kdyby zde nebyla o této záležitosti žádná zmínka.

V průběhu práce vyvstala potřeba načtený MBD vizualizovat. Důvodu bylo několik, ale tím hlavním bylo ověření, že celý systém pracuje, jak má - tj. že načítání MBD a operace nad ním fungují správně. Na první pohled nejde o nijak triviální úkol, a tak možnost, že by se obrázky grafu generovaly přímo z MBD, nepřicházela v úvahu. Bylo tedy nutné najít nějaký metaformát, do kterého by se vnitřní struktura MBD uložila a tu by pak převedl do vizuální podoby[1] specializovaný program. Tím metaformátem se stal programovací jazyk *DOT* a vizualizačním programem *Graphviz* [17]. Následují řádky jsou lehkým manuálem k programu Graphviz a k jazyku DOT. Nakonec je určitě vhodné připomenout, že *všechny* obrázky MBD v této práci byly vytvořeny pomocí jazyku DOT a balíku Graphviz.

10.1 Jazyk DOT

Jazyk *DOT* je určen přímo na popis grafů, a proto se s ním velmi snadno a příjemně pracuje. Příklad jednoduchého grafu je na obrázku 10.1.

Podobně jako v jazyce C se i v DOTu používají složené závorky („{" a „}") na uzavření určitého bloku příkazů či deklarací. Z obrázku 10.1 je patrné jejich použití: celý graf MBD je uzavřen do bloku, stejně jako několik dalších položek.

V definici grafu se používají pouze dva elementy - uzly a hrany. Uzel se definuje tak, že se napíše jeho identifikátor[2] a za něj se vloží středník.

```
uzel1;
```

Jedním příkazem je možné deklarovat i více uzlů najednou. Mezi jednotlivé identifikátory se vkládá mezera.

```
uzel1 uzel2 uzel3 uzel4;
```

Hrana se definuje tak, že se napíší identifikátory uzlů a mezi ně se vloží klíčové slovo pro hranu „->" a za celou tuto sekvenci se opět vloží středník.

```
uzel1 -> uzel1;
```

[1] obrázek typu PNG či SVG

[2] musí být unikátní

Kromě uzlů a hran lze nastavit i různé vlastnosti jak jednotlivých elementů grafu, tak i grafu samotného. Jako úvod do programování v jazyce DOT těchto pár řádek jistě stačí. Pro úplné pochopení následuje seznam důležitých klíčových slov a jejich popis.

digraph Označuje blok, v němž deklarované uzly a hrany spadají do jednoho grafu. Předpona „di" říká, že jde o orientovaný (anglicky *di*rected) graf. Je možné použít i klíčové slovo **graph**, což označuje graf neorientovaný. Jelikož MBD je orientovaný graf, je pro jeho správné zobrazení třeba použít příkaz **digraph**.

size Nejde o klíčové slovo, ale o atribut bloku s grafem. Udává maximální šířku a výšku výsledného obrázku v palcích.

node Pomocí tohoto klíčového slova se přednastavují atributy všech uzlů. Atributy se vkládají do hranatých závorek hned za klíčové slovo. Všechny uzly uvedené po tomto klíčovém slovu mají atributy těchto hodnot. Atributy lze změnit novou sekvencí node nebo za každým uzlem uvést je vlastní atributy.

Atributy uzlů určují jeho velikost, tvar, barvu nebo nadpis[3]. Z obrázku 10.1 je patrné, že každému uzlu je přiřazen jeho vlastní název (label), a zároveň je možné si všimnout, jak se pomocí druhého příkazu mění atributy pro vnitřní uzly MBD [4].

rank Jedná se o atribut bloku, ve kterém je vložen. Určuje způsob, jakým budou zobrazeny uzly v daném bloku. Hodnota „same" říká, že všechny uzly v bloku mají být zobrazeny ve stejné vertikální hladině (stejně vysoko). Jen díky tomuto příkazu je možné nakreslit MBD tak, že uzly patřící stejným proměnným jsou na stejné úrovni. Místo „same" je možné například použít „min" - uzly v bloku by pak byly umístěny na co nejnižší úroveň.

edge Tento příkaz je obdobou příkazu node pro hrany. Pro všechny hrany deklarované po něm přednastavuje jejich parametry. Ty lze opět změnit nebo doplnit pomocí atributů vložených za deklaraci každé hrany. Jediný atribut v příkazu na obrázku 10.1 je *arrowsize*, který určuje velikost šipky na konci každé hrany. Zároveň je možné u každé hrany explicitně určit, jaký má styl - *tečkované* (dotted) pro záporné hrany a *plné* (solid) pro kladné hrany. Určitě je vhodné zde zmínit, že i hrany mohou mít svůj název, čehož na vizualizaci MBD využito nebylo.

10.2 Graphviz

Graphviz je program nebo lépe řečeno sada programů, jejichž vývoj podporují laboratoře firmy AT&T[5]. Jde o bezplatný software, který je možno dostat jak v podobě binárních souborů pro platformu Windows, Linux a MacOS, tak i jako zdrojové soubory jazyka C++.

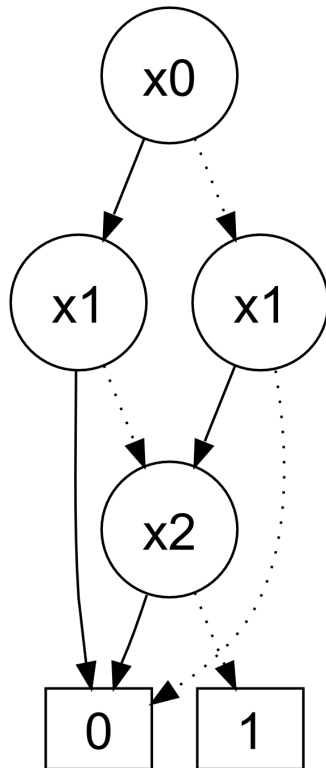
Použití toho programu je veskrze jednoduché. Na příkazový řádek se zadá program, který se má spustit, název vstupního a výstupního souboru a typ výstupního souboru. Nejsložitějším úkonem je zde výběr programu, který má vstupní DOT soubor zpracovat. Graphviz nabízí celkem čtyři různé aplikace, z nichž každá má jiný účel (všechny však mění grafy specifikované ve vstupních souborech na grafické formáty souborů výstupních).

---

[3] všech atributů je samozřejmě mnohem více

[4] První příkaz **node** stanovuje atributy terminálních uzlů.

[5] velká telekomunikační firma z USA




```
digraph BDD
{
size="7,11"

node [fontname="Helvetica",
     fontsize="20", height="0.3",
     width="0.6", color=black,
     style=unfilled , shape=box];

    { rank=same;
        uzel1 [label="1"];
        uzel2 [label="0"]; }

node [fontname="Helvetica",
     fontsize="20", height="0.3",
     width="0.5", style=unfilled ,
     color=black, shape=circle];

    { rank=same;
        uzel3 [label="x0"]; }

    { rank=same;
        uzel4 [label="x1"];
        uzel5 [label="x1"]; }

    { rank=same;
        uzel6 [label="x2"]; }

edge [arrowsize="1.0"];
    uzel3 -> uzel5 [style=dotted];
    uzel3 -> uzel4 [style=solid];
    uzel4 -> uzel6 [style=dotted];
    uzel4 -> uzel2 [style=solid];
    uzel5 -> uzel2 [style=dotted];
    uzel5 -> uzel6 [style=solid];
    uzel6 -> uzel1 [style=dotted];
    uzel6 -> uzel2 [style=solid];
}
```

Obrázek 10.1: **MBD a jeho popis pomocí jazyka DOT.**

Programy Graphvizu jsou:

- **dot**: Vytváří hierarchické orientované grafy.
- **neato**: Vytváří neorientované grafy.
- **twopi**: Vytváří radiální grafy (se středem).
- **circo**: Vytváří cyklické grafy.

Pro potřeby vizualizace MBD je vhodný jedině program **dot**. Proto by příkazový řádek vypadal takto:

```
c:\dot -Tsvg -ooutputfile.svg inputfile.dot
```

Parametrem „-T" se specifikuje formát výstupního souboru[6]. Graphviz podporuje okolo třiceti různých formátů, mezi nimiž nechybí důležité bitmapové formáty jako *jpg* nebo *png* nebo vektorové formáty jako *svg* či *ps*.

10.3 Nic není dokonalé

Graphviz je bezpochyby vynikají praktická pomůcka, ovšem ani těm nejlepším se nevyhnou menší problémy či nedotaženosti. Graphviz má poměrně velké problémy s vytvářením bitmapových formátů. Soubory PNG či JPG jdou jedním slovem popsat jako „ošklivé". Naštěstí program podporuje i vektorové formáty jako SVG, ze kterých je možné pěkné bitmapové soubory vytvořit pomocí některého z nespočetných vektorových editorů jako *Inscape* či *Adobe Illustrator* nebo pomocí grafických konvertorů jako *Batik*[7].

Určitě závažnější komplikací je špatná podpora formátování textu jazykem DOT. Jak je možné si všimnout ze všech obrázků MBD v této práci, všechna jména uzlů jsou tvořena znakem „x" a číselným dolním indexem (např. x_4). Dolní indexy však DOT nepodporuje [8]. MBD na obrázku 10.1 je přímým výstupem z formátu DOT (DOT→SVG→PNG). Do ostatních MBD bylo složitější formátování textu doplněno až díky speciálnímu filtru.

10.4 Jiné způsoby vizualizace

Pochopitelně, že ukládání MBD do souboru typu DOT a jeho převod pomocí Graphviz není jediným způsobem, jak vytvořit grafické znázornění jeho vnitřní struktury. Nicméně je to cesta nejschůdnější a nejsnáze aplikovatelná.

Jiným přístupem je generování obrázků grafů přímo z programu. Například samotný Graphviz nabízí C++ knihovnu, kterou je možné použít ve vlastním programu. Vedle Graphvizu má podobné schopnosti i knihovna BGL [18] (Boost Graphic Library), vyvíjená skupinou nadšenců.

Pro úplnost je nutné připomenout, že vizualizovat MBD je možné i manuálně pomocí nějakého softwaru na kreslení diagramů jako je Visio, SmartDraw a řady dalších. Ovšem nakreslit s těmito nástroji graf s více jak dvaceti uzly je poměrně pracné.

[6] v případě příkladu je použit formát SVG - Scalable Vector Graphics
[7] možná nejlepší konvertor z SVG na PNG
[8] DOT nepodporuje ani jiné druhy formátování textu.

Kapitola 11

Závěr

Na tomto místě je nutné potvrdit, že všechny úkoly, které byly vytýčeny v zadání práce, byly splněny. To znamená, že byla úspěšně implementována struktura MBD včetně řady doprovodných funkcí na řazení vstupních proměnných a minimalizaci neúplně určených logických funkcí. Provedené experimenty potvrdily nejen funkčnost implementace, ale odhalily i řadu zajímavých skutečností o MBD a jejich použití pro minimalizaci.

Experimenty ukázaly, že rozložení četností výskytu počtu uzlů v MBD má přibližně tvar Gaussovy křivky, jejíž rozptyl a průměrná hodnota počtu uzlů je závislá na počtu neurčených vstupních proměnných. Čím je neurčených vstupních proměnných více, tím je průměrná hodnota počtu uzlů a rozptyl Gaussovy křivky větší.

Z řadících technik se nejvíce prosadilo dynamické prohazování vstupních proměnných a statická metoda STAT, která je jedním z vedlejších produktů této práce. Jednoznačně se ukázalo, že nejlepší variantou jak přistoupit k řazení proměnných, je nejprve použít jednoduchou statickou metodu řazení a po načtení použít dynamické prohazování. Ačkoliv v práci implementovaná dynamická metoda měla svá omezení (uvíznutí v nejbližším lokálním minimu), přesto ukázala cestu, kterou by se případní pokračovatelé této práce měli vydat. Doplněním současné metody prohazování proměnných tak, aby se zabránilo uvíznutí v lokálním minimu, by se celá práce přiblížila *konkurenčním* balíkům jako CUDD nebo BuDDy. Možným řešením je použít například metody jako je simulované ochlazování nebo genetické algoritmy, které už byly na problém dynamického prohazování vstupních proměnných s úspěchem použity. Velmi snadno implementovatelná metoda STAT je naproti tomu velmi vhodným doplňkem, který dokáže razantně snížit počet uzlů dříve, než dojde na použití dynamických metod.

Ohledně minimalizace logických funkcí pomocí MBD je nutné si položit otázku, zda se MBD k tomuto účelu hodí a zda by tento postup mohl konkurovat známým a úspěšným minimalizatorům (především ESPRESSU). Odpovědí je jednoznačné ANO. Ano, MBD lze využít na minimalizaci logických funkcí a dokonce lze dosáhnout lehce lepších výsledků než zmiňované ESPRESSO. *ALE!* Vše je vykoupeno velkou časovou složitosti MBD. Načítání a booleovské (i algebraické) procházení MBD implementované v této práci má totiž složitost $O(2^n)$. Ačkoliv se reálné MBD jen výjimečně přiblíží této složitosti, i tak je čas velkým minusem metody a nedostatkem oproti heuristice ESPRESSO.

Při minimalizaci neúplně určených logických funkcí se naplno projevila výhoda MBD (oproti klasickému BDD), která spočívá v tom, že jak ONset, tak i DCset jsou v jednom společném grafu. Díky tomu bylo možné přesměrováním hran vedoucích do terminálního uzlu s hodnotou X razantně snížit celkovou velikost MBD. V této souvislosti bylo zjištěno, že hrany vedoucí do terminálu s hodnotou X stačí přesměrovávat do uzlu, kam vede její sesterská hrana.

Nedostatky této implementace MBD jsou jmenovány na mnoha místech tohoto textu, ale určitě je vhodné je ještě připomenout.

Zásadním problémem je zde rychlost načítání MBD. Ačkoliv zde použitá metoda není nijak naivní, přesto by potřebovala značně vylepšit nebo vytvořit znovu na zcela jiném přístupu. Dynamické řadící metodě - prohazování proměnných by zase prospělo vylepšení v podobě techniky, jež by dokázala uniknout z lokálního minima.

Méně náročnými vylepšeními pak jsou podpora vícehodnotových funkcí a podpora jiných platforem než Windows.

Příloha 1

Program implementující MBD

Zde je popis ovládání programu, který je výsledkem této práce. Program je určen pro platformu Windows a spouští se z příkazového řádku.

Program se spouští souborem mbd.exe, který je možné nalézt na přiloženém CD. Příkazový řádek vypadá takto:

```
mbd.exe [options]
```

Možné parametry programu uvádí tabulka 11.1.

Příklad Program načte logickou funkci, provede minimalizaci a uloží výsledek do formátu DOT:

```
mbd –i infile.pla –o outfile.dot –ot dot –min yes –so stat –do no
```

Tabulka 11.1: **Přepínače MBD programu.**

Parametr	Popis
-i filename	*filename* udává jméno vstupního PLA souboru
-o filename	*filename* udává jméno výstupního souboru
-ot type	*type* nastavuje typ výstupního souboru - možnosti jsou *dot* nebo *pla* (implicitní)
-so type	*type* nastavuje statickou metodu - možnosti jsou *none*(implicitní), *random*, *stat*, *force*
-do flag	*flag* určuje, zda se má použít dynamické řazení či ne - možnosti jsou *yes* (implicitní) a *no*
-xm method	*method* určuje jaká metoda přesměrování hran vedoucích do X se má použít - *none*, *brother*(implicitní) a *random*
-min flag	*flag* určuje, zda se má funkce minimalizovat či ne - možnosti jsou *yes* a *no* (implicitní)
-help	vytiskne informace o programu

Příloha 2

Obsah CD

CD přiložené k tomuto výtisku obsahuje zdrojové soubory MBD knihovny, spustitelný soubor *mbd.exe*, zdrojové soubory toho textu i jeho elektronickou verzi a benchmarky použité v testech. Adresářová struktura je následující:

- **kořenový adresář**: V tomto adresáři jsou všechny adresář, spustitelný soubor *mbd.exe* a elektronická podoba této publikace - *dp-mbd.pdf*.
- **source**: Adresář obsahuje zdrojové soubory MBD knihovny.
- **text**: Adresář obsahuje zdrojové soubory tohoto textu.
- **benchmarks**: Adresář obsahuje benchmarky použité při experimentech.

Literatura

[1] R. P. Jacobi: „A Study of the Aplication of Binary Decision Diagrams to Multi-level Logic Synthesis," Doktorandská práce, Prosinec 1993.

[2] R. E. Bryant: „Graph based algorithms for Boolean function manipulation," IEEE Transactions on Computers, 1986.

[3] S. J. Friedman, K. J. Supowit: „Finding optimal variable ordering for binary decision diagrams," IEEE Transactions on Computers, 1990.

[4] F. A. Aloul, I. L. Markov, K. A. Sakallah: „MINCE: A Static Global Variable-Ordering for SAT and BDD," IWLS, 2001.

[5] F. A. Aloul, I. L. Markov, K. A. Sakallah: „FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic," GLSVLSI, 2003.

[6] F. Somenzi: „Colorado University Decision Diagram package (CUDD)," http://vlsi.colorado.edu/ ~fabio/CUDD, 2006.

[7] H. Cohen: „BuDDy: A Binary Decision Diagram library," http://sourceforge.net/projects/buddy/, 2006.

[8] V. Jáneš, J. Douša: „Logické systémy," skriptum FEL ČVUT, 2001.

[9] R. Rudell: „Dynamic variable ordering for ordered binary decision diagrams," Proc. of the International Conference on Computer-Aided Design, 1993.

[10] K. Cho, R. E. Bryant: „Test pattern generation for sequential MOS circuits by symbolic fault simulation," Design Automation Conference, 1989.

[11] S. Chang, D. Cheng, M. Marek-Sadowska: „Minimizing ROBDD size of incompletely specified multiple output functions," In European Design & Test Conf., 1994.

[12] S. Minato, N. Ishiura, and S. Yajima, „Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," Design Automation Conference, 1990.

[13] T. Sasao and M. Matsuura, „BDD Representation for Incompletely Specified Multiple-Output Logic Functions and Its Applications to Functional Decomposition," Design Automation Conference, 2005.

[14] O. Kološ: „Port programového balíku CUDD pod platformu Windows," Bakalářská práce na FEL ČVUT, 2006.

[15] http://cs.wikipedia.org/wiki/Hladový_algoritmus

[16] http://en.wikipedia.org/wiki/QuineMcCluskey_algorithm

[17] http://www.graphviz.org

[18] http://www.boost.org/libs/graph/doc

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