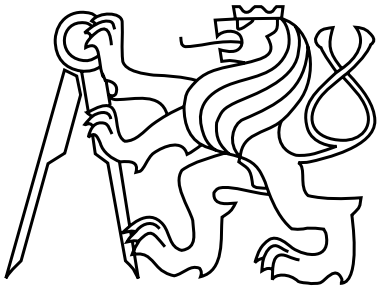
České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Skupinová minimalizace neúplně určených logických funkcí pomocí modifikovaných rozhodovacích diagramů

Pavel Černý

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný navazující magisterský

Obor: Informatika a výpočetní technika

červen 2008

Poděkování

Na tomto místě chci poděkovat Ing. Petru Fišerovi, Ph.D., který mi tuto práci nabídl a částečně mi i pomáhal s tvorbou textu. Hlavní dík pak patří mé rodině, a to především za zázemí, které mi neustále poskytuje.

Prohlášení

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

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

Ve Žďáře nad Sázavou dne 21.5. 2008 .....

Abstract

This Diploma deals with the minimization of incompletely specified multiple-output boolean functions by way of modified binary decision diagrams via CUDD package (Colorado University Decision Diagram Package). Primary task is to create software minimization tool of multiple-ouput boolean functions, verify tool's functionality and compare it experimentally with the other popular minimizators as Espresso or Boom. Second goal is to write a formal text with elementary theory of decision diagrams.

Abstrakt

Diplomová práce se zabývá skupinovou minimalizací neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů. Jako nástroj pro práci s rozhodovacími diagramy se předpokládá použití balíku CUDD (Colorado University Decision Diagram Package). Práce si za primární cíl klade vytvořit softwarový minimalizátor vícevýstupových neúplně (úplně) určených logických funkcí, ověřit funkčnost tohoto minimalizátoru a následně porovnat jeho použitelnost vzhledem k jiným používaným minimalizátorům jako Espresso či Boom. Sekundárním cílem práce je napsat text, v němž je formálně probrána elementární teorie o rozhodovacích diagramech.

Obsah

Seznam obrázků

Kapitola 1

Úvod

1.1 Úvod do problematiky

Pojmy *Booleova algebra* a *logická funkce* mají v oblasti počítačů a logických obvodů kardinální význam. Jejich uplatnění vzešlo přirozeně s potřebou zavést početní algebru v dvouhodnotovém světě nul a jedniček a na této algebře potom i funkce realizující potřebné výpočty. Hodnoty 0 a 1 chápeme často jako hodnoty *pravda* a *nepravda*, proto těmto funkcím říkáme *logické*. Logická funkce f není nic jiného, než zobrazení $f : \{0,1\}^n \to \{0,1\}$, kde množinou $\{0,1\}^n$ rozumíme kartézskou mocninu množiny $\{0,1\}$.

V matematické analýze zapisujeme často reálnou (komplexní) funkci n proměnných výrazem, který funkci definuje. Obdobně každou logickou funkci můžeme definovat tzv. booleovským výrazem. Booleovský výraz se skládá se z proměnných funkce, konstant 0 a 1 a operací Booleovy algebry. Nepříjemnou skutečností je, že ke každé logické funkci f existuje nekonečně mnoho výrazů, které určují f (všechny tyto výrazy jsou „funkčně ekvivalentní"). Z lidského pohledu jsou pro nás nejzajímavější tzv. minimální výrazy V_{min}, tj. výrazy, v nichž se vyskytuje co možná nejméně operací Booleovy algebry. Minimální výraz V_{min} získává na důležitosti především při návrhu fyzického obvodu realizujícího funkci určenou V_{min}. Počet hradel logického obvodu realizujícího funkci podle daného výrazu V přímo odpovídá počtu operací Booleovy algebry obsažených ve V. Aby hradlo pracovalo tak jak má, je nezbytné jej napájet, stejně tak každé hradlo produkuje teplo, které je při velkém množství hradel třeba odvádět. Při současném trendu co možná nejvyšší hustoty integrace obvodu hraje proto minimalizace logické funkce zásadní roli.

Složitější variantou minimalizace je pak *skupinová minimalizace* více logických funkcí. Jde o následující problém. Chceme sestrojit co do počtu hradel nejmenší fyzický obvod, který je schopen realizovat každou z n logických funkcí $f_1, f_2, \ldots, f_n$. Intuitivně je zřejmé, že potřebujeme nalézt takové výrazy $V_1, \ldots, V_n$ definující funkce $f_1, \ldots, f_n$, které jsou složené z podvýrazů, jež jsou společné pro maximální počet V_i (chceme nalézt největších redundance ve $V_1, \ldots, V_n$). Tyto redundance jsou do výsledného obvodu zahrnuty pouze jednou.

V názvu diplomové práce ovšem stojí skupinová minimalizace neúplně určených logických funkcí. Co je to tedy neúplně určená logická funkce? Předpokládejme funkci f zavedenou v prvním odstavci. Může se stát, že pro konfigurace vstupních proměnných z nějaké množiny X ($X \subset \{0,1\}^n$) je nám jedno jakých funkčních hodnot bude funkce f nabývat a obráceně bazírujeme na tom, aby f vracela dané hodnoty pro konfigurace proměnných z množiny $\{0,1\}^n \setminus X$. Takto neúplně určená funkce f tak vlastně popisuje množinu funkcí $\mathcal{F}$, kde libovolné dvě funkce z $\mathcal{F}$ se liší ve funkční(-ch) hodnotě(-ách) pro alespoň jednu konfiguraci vstupních proměnných z X a mající stejné funkční hodnoty pro všechny konfigurace proměnných z $\{0,1\}^n \setminus X$. Při minimalizaci f si proto můžeme vybrat libovolnou funkci z $\mathcal{F}$ a dosáhnout tak lepších výsledků než kdybychom například zvolili funkční hodnoty f v bodech z X náhodně. Tomuto problému se věnuje právě minimalizace neúplně určených logických funkcí.

Nabízí se otázka proč již problém minimalizace nebyl vyřešen. Odpověď je nasnadě. Už minimalizace jedné logické funkce je totiž $\mathcal{NP}$ těžký problém. Jinými slovy to znamená, že dosud nejlepší známý algoritmus nalezne minimální tvar logické funkce f v čase, který je obecně řádově exponenciální funkcí počtu proměnných f. Pokud bychom chtěli například nalézt minimální tvar funkce v řádech stovek tisíc

proměnných, pak bychom na to při současné výpočetní technice potřebovali více času než je odhadované stáří Vesmíru. Při řešení se tak musíme omezit na heuristiky a jiné algoritmy vracející pokud možno „co nejlepší“ výsledky v „rozumném“ čase.

Věnujme se na chvíli otázce reprezentace logické funkce na počítači. Pro funkci o n proměnných existuje 2^n různých vstupních konfigurací proměnných, tj. například obvyklá reprezentace funkce tabulkou dvojic: vstup, výstup nebude pro velká n jistě možná. Reprezentace funkce booleovským výrazem v případě, že s funkcí chceme dále pracovat, není také výhodná. Možností reprezentace logické funkce, zkoumané především v posledních 20-ti letech, je reprezentace *binárními rozhodovacími diagramy* (BDD), především *redukovanými uspořádanými binárnímy diagramy* (ROBDD). Významnou vlastností ROBDD je *kanonicita*. Znamená to, že pro dané uspořádání vstupních proměnných funkce existuje právě jeden diagram (přesněji právě jeden až na *izomorfismus*). Kanonicita ROBDD poskytuje možnost „rychlého“ testování dvou funkcí na rovnost nebo i třeba testování splnitelnosti funkce jak uvidíme později. BDD a ROBDD, tak jak jsou obvykle definovány, se vztahují k úplně určeným logickým funkcím. BDD (ROBDD) lze však po modifikaci stejně dobře aplikovat na neúplně určené funkce, přičemž způsobů modifikace je více. Vybereme jeden, kterému budeme říkat *modifikovaný binární rozhodovací diagram* (MBD) neúplně určené logické funkce a který budeme používat.

1.2 Motivace a cíle diplomové práce

Bylo řečeno, že práce se zabývá skupinovou minimalizací neúplně určených logických funkcí s využitím MBD. Tato práce navazuje na diplomovou práci Jana Bílka ([7]), který se zabýval minimalizací jedné logické funkce pomocí MBD. Proč právě pomocí MBD? Výhoda MBD (ROMBD) tkví právě v kanonicitě. Budeme-li pro danou funkci udržovat jedinečný jí odpovídající MBD, pak po vytvoření všech MBD skupinově minimalizovaných funkcí jsou tyto MBD pro dané uspořádání proměnných „maximálně sdílené“, tedy jsou v jisté kompaktní formě, která sama nalezne maximální redundance minimalizovaných funkcí. Při minimalizaci pak stačí tyto redundance minimalizovat externím minimalizátorem právě jednou (na rozdíl od minimalizace jednotlivých funkcí odděleně), čímž se minimalizace stává rychlejší. Samozřejmě množství a struktura redundancí je odvislá od uspořádání vstupních proměnných funkce(-í), které diagramy respektují. Otázka volby vhodného uspořádání proměnných je $\mathcal{NP}$ těžký problém. Pro funkci(-e) N proměnných existuje $N!$ různých uspořádání. Často se zde kloníme k použití heuristik. Hlavním cílem práce je naprogramovat skupinový minimalizátor neúplně určených funkcí a zhodnotit jeho použitelnost doloženými naměřenými hodnotami. Má-li být použitelnost stanovena korektně, je třeba klást důraz na co nejefektivnější implementaci MBD. Tento fakt nás vedl k použití rozšířeného balíku pro práci s diagramy CUDD. V práci Jana Bílka ([7]), který použil svojí implementaci diagramů, se totiž ukázalo, že vlastní implementace nedosahuje ani z daleka kvalit CUDDu (taková implementace by byla nesmírně pracná).

Balík CUDD je existující balík pro práci s binárními rozhodovacími diagramy (BDD, ROBDD) a s jejich variantami, které probereme později. Balík má následující charakteristiky: Je poměrně rozsáhlý a propracovaný, velmi efektivní, tj. práce s diagramy je rychlá, je zdarma a může být znovupoužitelný jako součást jakéhokoliv nekomerčního projektu. Rozhodli jsme se proto CUDD využít, bylo třeba jej však částečně modifikovat, aby byl schopen pracovat s MBD.

Sekundárním cílem práce je napsat text, v němž formálně definujeme pojmy rozhodovacích diagramů a ukážeme jejich některé důležité, již známé vlastnosti. Všechny české texty o rozhodovacích diagramech, s nimiž se autor setkal, byly totiž, ač srozimitelné, psány neformálním způsobem a v případech důležitých skutečností se odkazovaly na anglické původní texty. To chceme tímto textem změnit.

1.3 Rozdělení textu do kapitol

Práce je rozdělena do šesti kapitol. První kapitola uvádí do problematiky, shrnuje motivaci a cíle práce a opakuje potřebné, v textu používané pojmy. Druhá kapitola zavadí pojmy uspořádaných rozhodovacích diagramů (ODD), speciálně pak také diagramů BDD a MBD pro reprezentaci úplně resp. neúplně určené logické funkce. Dále jsou dokázány některé důležité vlastnosti diagramů a je ukázáno jejich použití. Třetí kapitola se zabývá minimalizací logických funkcí a to jak pomocí BDD (MBD), tak i okrajově

pomocí klasických metod. Čtvrtá kapitola popisuje balík CUDD a modifikace v něm provedené pro účely práce minimalizace. Pátá kapitola přináší naměřené výsledky minimalizace pomocí BDD (MBD) a jejich porovnání s jinými minimalizátory. Konečně šestá kapitola zhodnocuje dosažené výsledky a celkovou použitelnost uvedeného způsobu minimalizace.

1.4 Opakování potřebných pojmů

Abychom mohli pracovat s pojmy jako binární rozhodovací diagram nebo logická funkce, je třeba tyto pojmy zavést formálním způsobem. To si vyžaduje znalost elementárních pojmů z algebry, analýzy a teorie grafů. Následující sekce slouží k tomu, aby tyto v textu dále používané pojmy připomněla. Znalý čtenář může tuto část textu přeskočit, naopak neznalý najde více o analýze například v [26] nebo [18], o algebře v [13] nebo [9] a o teorii grafů v [19] nebo v [8].

1.4.1 Několik pojmů z algebry a analýzy

Definice 1.1 (**Kartézský součin a kartézská mocnina.**). Kartézským součinem množin $A_1, A_2, \ldots, A_n$ rozumíme symbolicky zapsanou množinu uspořádaných n-tic

$$A_1 \times A_2 \times \cdots \times A_n = \big\{(a_1, a_2, \ldots, a_n) \;\big|\; a_i \in A_i, 1 \leq i \leq n\big\}.$$

Prvky kartézského součinu nazýváme body nebo též vektory a značíme $\vec{a} = (a_1, a_2, \ldots, a_n)$. Pro $A_1 = A_2 = \cdots = A_n = A$ hovoříme o kartézské mocnině množiny A a místo $A \times A \times \cdots \times A$ píšeme často stručněji A^n.

Například je-li $A = \{0, 1\}$ a $B = \{0, 2\}$, pak $A \times B = \big\{\{0, 0\}, \{0, 2\}, \{1, 0\}, \{1, 2\}\big\}$. Důležitými množinami, které jsou výsledkem kartézské mocniny, jsou v analýze množiny $\mathbb{R}^n$ a $\mathbb{C}^n$.

Definice 1.2 (**N-ární relace**). O každé podmnožině $\mathcal{R}$ kartézského součinu množiny $A_1, A_2, \ldots, A_n$ ($\mathcal{R} \subset A_1 \times A_2 \times \cdots \times A_n$) mluvíme jako o *n-ární relaci* nebo jen *relaci* na $A_1, A_2, \ldots A_n$ a píšeme $\mathcal{R}(A_1, A_2, \ldots, A_n)$. Speciálně pro $n = 2$ hovoříme o *binární relaci*.

Definice 1.3 (**Zobrazení**). Zobrazením φ z množiny X do množiny Y nazýváme libovolnou binární relaci na X, Y, tj. $\varphi \subset X \times Y$, takovou, že pro každé $x \in X$ existuje právě jedno y, tak že $(x, y) \in \varphi$ a píšeme $\varphi(x) = y$.

Formálně zapisujeme zobrazení $\varphi : X \to Y$. Definičním oborem φ rozumíme množinu $\mathcal{D}_\varphi = \big\{x \in X \big|\; \exists y \in Y : \varphi(x) = y\big\} = X$. Oborem hodnot φ rozumíme množinu $\mathcal{H}_\varphi = \varphi(\mathcal{D}_\varphi) = \big\{\varphi(x) \big|\; x \in X\big\} \subset Y$.

Dále o φ říkáme, že je

- *prosté (injektivní)*, jestliže pro všechna $x_1, x_2 \in \mathcal{D}_\varphi$, $x_1 \neq x_2$ je $\varphi(x_1) \neq \varphi(x_2)$.
- *na (surjektivní)*, jestliže $\mathcal{H}_\varphi = Y$.
- *bijektivní(bijekce)*, jestliže je zároveň prosté a na.

Poznámka 1.1. Zobrazení se obvykle definuje volněji, a to tak, že $\mathcal{D}_\varphi \subset X$. Pro naše potřeby jsme mezi oběma množinami zavedli rovnost, definiční obor je tak vidět hned ze zápisu zobrazení. Naše definice odpovídá obvyklé definici pojmu **funkce**, kterou budeme záměně také používat.

Poznámka 1.2. Je-li $\mathcal{D}_\varphi \subset X_1 \times X_2 \times \cdots \times X_n$ pro nějaké množiny $X_1, X_2, \ldots, X_n$, pak pro $\vec{x} = (x_1, x_2, \ldots, x_n) \in X$ místo $\varphi\big((x_1, x_2, \ldots, x_n)\big)$ píšeme $\varphi(x_1, x_2, \ldots, x_n)$, tj. vynecháváme vnitřní pár závorek.

Jednoduchým příkladem zobrazení je například zobrazení $\varphi_1 : \mathbb{R}^2 \to \mathbb{R}$ dané výrazem $\varphi_1(x_1, x_2) = x_1^2 + x_2^2$ pro všechna $(x_1, x_2) \in \mathbb{R}^2$. φ_1 není prosté, protože například $\varphi_1(-1, -1) = \varphi_1(1, 1) = 2$. φ_1 není ani na, neboť $\mathcal{H}_{\varphi_1} = \mathbb{R}_0^+ = \big\{x \in \mathbb{R} \;\big|\; x \geq 0\big\}$. Naopak $\varphi_2 : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ definované pro všechna $x \in \mathbb{R}_0^+$ rovností $\varphi_2(x) = x^2$ je zřejmě prosté i na a tedy bijektivní.

Chceme-li na množině $\{0, 1\}^n$ zavést pojem výrazu (jednoznačně určujícího funkci), je nutné mít na $\{0, 1\}$ smysluplně definované operace a tedy algebru. Jednou z takových algeber je *Booleova algebra*, která je distributivním a komplementárním svazem. Začněme proto těmito pojmy.

Definice 1.4 (**Svaz**). *Svazem* $\mathcal{S}$ rozumíme libovolnou algebru na množině X spolu s operacemi $\vee$ a $\wedge$ (někdy též $+$ a $\cdot$) splňující pro všechna $x, y, z \in X$ následující sadu rovností:

1. Komutatitvita:	$x \vee y = y \vee x$	$x \wedge y = y \wedge x$
2. Asociativita:	$(x \vee y) \vee z = x \vee (y \vee z)$	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$
3. Idempotence:	$x \vee x = x$	$x \wedge x = x$
4. Absorbce:	$x \vee (x \wedge y) = x$	$x \wedge (x \vee y) = x$

Svaz zapisujeme jako usořádanou trojici $\mathcal{S}(X, \vee, \wedge)$, o množině X hovoříme jako o nosné množině svazu $\mathcal{S}$.

Definice 1.5 (**Distributivní svaz**). O svazu $\mathcal{S}(X, \vee, \wedge)$ řekneme, že je *distributivní*, jestliže pro všechna $x, y, z \in X$ platí:

5. Distributivita:	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

Definice 1.6 (**Booleova algebra**). *Booleovou algebrou* $\mathcal{B}$ na množině X ruzíme šestici $\mathcal{B}(X, \vee, \wedge, \top, \bot, \neg)$, kde

1. $(X, \vee, \wedge)$ je distributivní svaz.
2. $\top$ je největší, $\bot$ nejmenší prvek, tj. pro všechna $x \in X$ platí:
$$x \vee \top = \top \qquad x \wedge \bot = \bot .$$
3. $\neg : X \to X$ je unární operace nazvaná komplement nebo též negace, která pro všechna $x \in X$ splňuje rovnosti:
$$x \wedge (\neg x) = \bot \qquad x \vee (\neg x) = \top.$$

Následují tři důležité příklady Booleových algeber (ověřit, že jde o Booleovy algebry je snadné):

1. ***Booleova algebra potenční množiny*** $\mathcal{B}_{\mathcal{P}}(\mathcal{P}(A), \cup, \cap, A, \emptyset, (A \setminus \cdot))$
   (a) Nosná množina $\mathcal{P}(A)$ je potenční množina libovolné množiny A ($\mathcal{P}(A)$ je množina všech podmnožin A).
   (b) Binární operace $\cup$ a $\cap$ jsou množinové operace sjednocení a průniku.
   (c) Unární operace $(A \setminus \cdot)$ představuje doplněk do A, tj. $\neg B = (A \setminus B)$ pro všechna $B \in \mathcal{P}(A)$.
   (d) Největším prvkem je množina A, nejmenším prázdná množina $\emptyset$.
2. ***Booleova algebra dvouhodnotové logiky*** $\mathcal{B}_{log}(\{0, 1\}, +, \cdot, 1, 0, \neg)$
   (a) Binární operace $+$ (logický součet), $\cdot$ (logický součin) a unární operace $\neg$ (negace) jsou definovány tabulkou 1.1.

$+$	0	1
0	0	1
1	1	1

$.$	0	1
0	0	0
1	0	1

$\neg$	
0	1
1	0

Tabulka 1.1: Definice operací Booleovy algebry $\mathcal{B}_{log}$

3. ***Booleova algebra neúplně určené dvouhodnotové logiky*** $\mathcal{B}_{xlog}(\{0, 1, X\}, +, \cdot, 1, 0, \neg)$

$+$	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

$.$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$\neg$	
0	1
1	0
X	X

Tabulka 1.2: Definice operací Booleovy algebry $\mathcal{B}_{xlog}$

(a) Binární operace $+$ (logický součet), $\cdot$ (logický součin) a unární operace $\neg$ (negace) jsou definovány tabulkou 1.2.

Právě poslední dvě algebry budeme v textu výhradně používat. Je užitečné si všimnout, že pokud v případě algebry dvouhodnotové logiky chápeme hodnoty 0 a 1 jako prázdnou $\emptyset$ a jednoprvkovou množinu $\{a\}$, stejně tak operace $+$, $\cdot$ a $\neg$ jako množinové operace $\cup$, $\cap$ a $(\{a\} \setminus \cdot)$, pak je algebra dvouhodnotové logiky speciálním případem algebry systému potenční množiny množiny $\{a\}$, $\mathcal{P}(\{a\}) = \big\{\emptyset, \{a\}\big\}$.

1.4.2 Více o Booleových algebrách, výrazech a logických funkcích

Důležitou vlastností každé Booleovy algebry je její *dualita* ve smyslu následující věty.

Věta 1.1 (**Dualita Booleovy algebry**). *Je-li* $\mathcal{B}(X, \vee, \wedge, \top, \bot, \neg)$ *Booleova algebra, pak je i* $\mathcal{B}'(X, \wedge, \vee, \bot, \top, \neg)$ *Booleova algebra.*

Důkaz. Důkaz je triviální. Vzhledem k úplné „symetrii“ operací $\vee$ a $\wedge$ resp. prvků $\bot$ a $\top$ plynou všechny potřebné vlastnosti $\mathcal{B}'$ jakožto Booleovy algebry z vlastností algebry $\mathcal{B}$. □

Důsledkem duality jakékoliv Booleovy algebry je skutečnost, že všechny zákony platné v Booleově algebře mají „podvojný“ charakter. Podívejme se na následující tzv. *De Morganovy zákony.*

Věta 1.2 (**De Morganovy zákony**). *V každé Booleově algebře* $\mathcal{B}(X, \vee, \wedge, \top, \bot, \neg)$ *platí vztahy*

$$\neg(x \vee y) = (\neg x) \wedge (\neg y) \qquad \neg(x \wedge y) = (\neg x) \vee (\neg y),$$

pro všechna $x, y \in X$.

Důkaz. Vzhledem k dualitě $\mathcal{B}$ jistě stačí ukázat platnost pouze jedné z rovností. Dokažme tedy první rovnost. Z definice operace $\neg$ (1.4) vyplývá, že budeme hotovi ihned, jakmile ukážeme platnost vztahů

$$(x \vee y) \vee ((\neg x) \wedge (\neg y)) = \top \qquad (x \vee y) \wedge ((\neg x) \wedge (\neg y)) = \bot .$$

Následující úpravy levých stran předchozích rovností podle zákonů Booleovy algebry tyto rovnosti dokazují:

$$\begin{aligned}(x \vee y) \vee ((\neg x) \wedge (\neg y)) &= ((x \vee y) \vee (\neg x)) \wedge ((x \vee y) \vee (\neg y)) \\ &= (y \vee \top) \wedge (x \vee \top) \\ &= \top \vee \top = \top.\end{aligned}$$

$$\begin{aligned}(x \vee y) \wedge ((\neg x) \wedge (\neg y)) &= (x \wedge ((\neg x) \wedge (\neg y))) \vee (y \wedge ((\neg x) \wedge (\neg y))) \\ &= (\bot \wedge (\neg y)) \vee (\bot \wedge (\neg x)) \\ &= \bot \wedge \bot = \bot .\end{aligned}$$

□

Matematickou indukcí lze snadno rozšířit De Morganovy zákony do tvarů

$$\neg(x_1 \vee x_2 \vee \cdots \vee x_n) = (\neg x_1) \wedge (\neg x_2) \wedge \cdots \wedge (\neg x_n)$$

$$\neg(x_1 \wedge x_2 \wedge \cdots \wedge x_n) = (\neg x_1) \vee (\neg x_2) \vee \cdots \vee (\neg x_n).$$

Nyní jsme připraveni zavést pojem booleovského (logického) výrazu a logické funkce.

Definice 1.7 (**Úplně a neúplně určená logická funkce**). *Úplně* resp. *neúplně určenou logickou funkcí* φ o n proměnných rozumíme jakékoliv zobrazení $\varphi : \{0,1\}^n \to \{0,1\}$ resp. $\varphi : \{0,1\}^n \to \{0,1,X\}$. Dále zavádíme následující názvosloví:

- Nehrozí-li nebezpečí nedorozumnění nebo nezáleží-li na „určenosti“ funkce, používáme pro oba pojmy jednotný název logická funkce či funkce.
- Definičním oborem a oborem hodnot funkce f máme na mysli definiční obor a obor hodnot f jako zobrazení a značíme stejně jako u zobrazení $\mathcal{D}_f$ a $\mathcal{H}_f$.
- Prvky $\mathcal{D}_f$ nazýváme *konfigurace vstupních proměnných* nebo též *(vstupní) body.*
- Funkci f někdy popisujeme dvojicí (trojicí) disjunktních podmnožin $\mathcal{D}_f^0$, $\mathcal{D}_f^1$ (v případě neúplně určené funkce ještě $\mathcal{D}_f^X$) definičního oboru $\mathcal{D}_f$, jejichž sjednocení pokrývá $\mathcal{D}_f$. Tyto množiny jsou přirozeně definovány:
  1. $\mathcal{D}_f^0 = \{\vec{x} \in \mathcal{D}_f \mid f(\vec{x}) = 0\}$.
  2. $\mathcal{D}_f^1 = \{\vec{x} \in \mathcal{D}_f \mid f(\vec{x}) = 1\}$.
  3. $\mathcal{D}_f^X = \{\vec{x} \in \mathcal{D}_f \mid f(\vec{x}) = X\}$.

  Pro množiny $\mathcal{D}_f^0$, $\mathcal{D}_f^1$, $\mathcal{D}_f^X$ často používáme značení $\mathcal{D}_{off}$, $\mathcal{D}_{on}$, $\mathcal{D}_{dc}$ [1], píšeme symbolicky $f = (\mathcal{D}_{off}, \mathcal{D}_{on}, \mathcal{D}_{dc})$ a nazýváme je OFFset, ONset a DCset.

Příkladem neúplně určené logické funkce je funkce $g : \{0,1\}^2 \to \{0,1\}$ daná tabulkou 1.3.

x	$g(x)$
$(0,0)$	0
$(0,1)$	X
$(1,0)$	X
$(1,1)$	1

Tabulka 1.3: Příklad neúplně určené logické funkce

Funkci g můžeme popsat množinami rozkladu jejího definičního oboru jako

$$g = \Big(\{(0,0)\}, \{(1,1)\}, \{(0,1),(1,0)\}\Big).$$

Následující věta ukazuje, že počet různých logických funkcí na množině $\{0,1\}^n$ s rostoucím n rapidně roste.

Věta 1.3 (**Počet logických funkcí na** $\{0,1\}^n$). *Na* $\{0,1\}^n$ *existuje* 2^{2^n} *různých logických funkcí* $f : \{0,1\}^n \to \{0,1\}$.

Důkaz. Každý bod $x \in \{0,1\}^n$ je kódován posloupností n čísel z $\{0,1\}$. Existuje proto $|\{0,1\}|^n = 2^n$ různých bodů z $\mathcal{D}_f$ ($|\mathcal{D}_f| = 2^n$). Pro každý bod $x \in \mathcal{D}_f$ existují dvě možnosti (0 a 1) funkční hodnoty $f(x)$. Celkem tedy existuje 2^{2^n} různých funkcí na $x \in \{0,1\}^n$. □

[1] Zkratka *dc* je zkratkou anglického slova *don't-care*

Definice 1.8 (**Úplně a neúplně určený booleovský výraz**). *Úplně* případně *neúplně určený booleovský výraz* (často jen *booleovský výraz* nebo *výraz* pokud je z kontextu zřejmé o jaký typ výrazu jde) v proměnných $x_1, \ldots, x_n$ definujeme induktivně podle následujících pravidel:

1. Konstanty 0, 1 a pro neúplně určený výraz též X jsou booleovským výrazem.
2. Proměnné $x_1, x_2, \ldots, x_n$ jsou booleovské výrazy.
3. Je-li V booleovský výraz, pak je $\neg(V)$ (někdy píšeme $\overline{V}$) booleovský výraz.
4. Jsou-li V_1 a V_2 booleovské výrazy, potom jsou $(V_1) + (V_2)$ a $(V_1).(V_2)$ booleovské výrazy
5. Cokoliv, co nevzniklo konečným počtem opakování pravidel $1. - 5.$, není booleovský výraz.

Chceme-li spolu s výrazem uvést i jeho proměnné, značíme výraz V ve tvaru $V(x_1, \ldots, x_n)$. Konečně lexikografickou rovnost výrazů V_1 a V_2 zapisujeme $V_1 \equiv V_2$.

Je dobré si všimnout, že každý výraz V v proměnných $x_1, \ldots, x_n$ je zároveň výrazem v proměnných $x_1, \ldots, x_n, x_{n+1}$, kde x_{n+1} je libovolná, nově zavedená proměnná. Naopak V již ale nemusí být výrazem v proměnných $x_1, \ldots, x_{n-1}$ (obsahuje-li proměnnou x_n).

Definice 1.9 (**Restrikce booleovského výrazu**). *Restrikcí booleovského výrazu* $V(x_1, \ldots, x_n)$ podle proměnné x_i hodnotou $c \in \{0, 1\}$ nazveme výraz V', který vznikne z V substitucí všech výskytů x_i ve V (případně i žádného) hodnotou c a značíme $V' \equiv V(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n)$, $V' \equiv V[x_i = c]$ nebo $V' \equiv V_x$ resp. $V' \equiv V_{\overline{x}}$ pro $c = 1$ resp. $c = 0$. Tedy pokud x_i není proměnná V, pak $V[x_i = c] \equiv V$.

Příkladem booleovského výrazu je výraz $E \equiv (((x_1).(x_2)).(\overline{x_2})) + (x_1)$, jehož restrikcí podle proměnné x_2 hodnotou $x_2 = 0$ je výraz $E[x_2 = 0] \equiv E_{\overline{x_2}} \equiv (((x_1).(0)).(\overline{0})) + (x_1)$.

Už víme jako booleovský výraz vypadá, ale ještě jsme formálně neřekli, co je jeho hodnotou. Nyní „vtáhneme do hry“ dříve uvedené booleovy algebry $\mathcal{B}_{log}, \mathcal{B}_{xlog}$. Podřetězce $+$, . a $\neg$ výrazů budeme chápat jako stejnojmenné operace těchto algeber.

Definice 1.10 (**Hodnota booleovského výrazu**). Hodnotou $h(V)$ booleovského výrazu V bez proměnných rozumíme konstantu $c \in \{0, 1\}$ danou induktivně:

1. $h(V) = 0$ pro $V \equiv 0$
2. $h(V) = 1$ pro $V \equiv 1$
3. $h(V) = X$ pro $V \equiv X$ v případě neúplně určeného výrazu V.
4. $h(V) = h(V_1) \oplus h(V_2)$ pro $V \equiv (V_1) + (V_2)$ (V_1 a V_2 jsou výrazy).
5. $h(V) = h(V_1) \odot h(V_2)$ pro $V \equiv (V_1).(V_2)$ (V_1 a V_2 jsou výrazy).
6. $h(\neg V) = \oslash h(V)$.

Přitom operace $\oplus$, $\odot$ a $\oslash$ chápeme pro úplně (neúplně) určený výraz jako operace $+$, . a $\neg$ Booleovy algebry $\mathcal{B}_{log}$ ($\mathcal{B}_{xlog}$).

Hodnotou $h\big(V(c_1, \ldots, c_n)\big)$ výrazu $V(x_1, \ldots, x_n)$ n proměnných v bodě $(c_1, \ldots, c_n) \in \{0, 1\}^n$ rozumíme hodnotu výrazu V_n bez proměnných, který vznikne z V postupnou restrikcí podle všech proměnných:

$$V_0 \equiv V, \quad V_i \equiv V_{i-1}[x_i = c_i] \quad 1 \leq i \leq n.$$

Jako ukázku výpočtu hodnoty výrazu uveďme výpočet hodnoty uvedeného výrazu $E \equiv (((x_1).(x_2)).(\overline{x_2})) + (x_1)$ v bodě $(x_1, x_2) = (1, 0)$. Postupnými úpravami dostáváme

$$\begin{aligned} h\big(E\big) = h\Big(E[x_1=1][x_2=0]\Big) &= h\Big(\big(((1).(0)).(\overline{0})\big) + (1)\Big) \\ &= h\Big(\big((1).(0)\big).(\overline{0})\Big) + h(1) \\ &= h\Big((1).(0)\Big).h(\overline{0}) + h(1) \\ &= h(1).h(0).\big(\neg h(0)\big) + h(1) \\ &= 1.0.\overline{0} + 1 = 0 + 1 = 1 \end{aligned}$$

Zjevně při restrikci V na výraz V_n bez proměnných v definici 1.10 nezáleží na pořadí proměnných, v němž restrikci provádíme. Dále libovolný výraz $V(x_1,\ldots,x_n)$ má v každém bodě $(c_1,\ldots,c_n) \in \{0,1\}^n$ jednoznačně definovanou hodnotu. Každý výraz $V(x_1,\ldots,x_n)$ tak svými hodnotami určuje právě jednu logickou funkci $f : \{0,1\}^n \to \{0,1,X\}$.

Nyní pojmy logické funkce a logického výrazu provážeme očekávaným způsobem.

Definice 1.11 (**Funkce definovaná výrazem a funkční rovnost dvou výrazů**)**.** Nechť je dán výraz $V(x_1,\ldots,x_n)$. Funkci definovanou výrazem V nazveme funkci $f : \{0,1\}^n \to \{0,1,X\}$ definovanou

$$f(c_1,\ldots,c_n) := h\big(V(c_1,\ldots,c_n)\big) \quad \text{pro všechny body} \quad (c_1,\ldots,c_n) \in \mathcal{D}_f$$

a píšeme $f = V$. O výrazech V_1 a V_2 řekneme, že jsou funkčně ekvivalentní, jestliže $f_1 \equiv f_2$ ($f_1(\vec{x}) = f_2(\vec{x})$ pro $\forall \vec{x} \in \mathcal{D}_f$), kde f_1 resp. f_2 jsou funkce definované výrazy V_1 resp. V_2 a píšeme $V_1 = V_2$.

Konvence 1.1. V dalším textu budeme často, ač formálně nekorektně, hovořit o výrazech jako o funkcích. Víme totiž, že libovolný výraz určuje pro dané proměnné $x_1,\ldots,x_n$ na $\{0,1\}^n$ právě jednu funkci.

Už v případě uvedeného příkladu jednoduchého výrazu E je vidět, že je zápis výrazu díky závorkám nepřehledný. Abychom se použití závorek co nejvíce vyvarovali, definujeme důležitost operací algeber $\mathcal{B}_{log}$ ($\mathcal{B}_{xlog}$). Budeme dodržovat následující prioritu:

1. *operace* $\neg$ váže nejsilněji
2. *operace* . váže slaběji
3. *operace* + váže nejslaběji

Znamená to, že jakýkoliv výraz V,

1. $V \equiv (V_1) + (V_2)$ budeme v případě, že se v podvýrazech V_1 a V_2 nevyskytuje operace +, zapisovat zjednodušeně $V \equiv V_1 + V_2$.
2. $V \equiv (V_1).(V_2)$ budeme v případě, že se v podvýrazech V_1 a V_2 nevyskytují operace + a . zapisovat $V \equiv V_1.V_2$.
3. $V \equiv \neg(V_1)$ budeme v případě, že se podvýrazu V_1 nevyskytují operace +, . a $\neg$, (V_1 je konstantou nebo proměnnou), psát $V \equiv \neg V_1$.

Výraz E z příkladu se v této konvenci zjednodušší na tvar

$$E \equiv (x_1.x_2).\overline{x_2} + x_1.$$

Hledejme další zjednodušení zápisu výrazů. Nechť V' a V'' jsou výrazy, které vzniknou smysluplným uzávorkováním (tak, aby vznikl opět booleovský výraz ve smyslu definice 1.8) libovolného počtu podvýrazů $V_i + V_{i+1} + \cdots + V_{i+k}$ ($V_i.V_{i+1}.\cdots.V_{i+k}$) jednoho z výrazů

$$V \equiv V_1 + V_2 + \cdots + V_n, \qquad V \equiv V_1.V_2.\cdots.V_n, \tag{1.1}$$

kde navíc ve V dodžujeme dříve stanovenou prioritu operací $+$, . a $\neg$. Protože operace $+$ a . algeber jsou asociativní, platí pro libovolné dva takto vzniklé výrazy $V' = V''$, tedy všechny takto vzniklé výrazy indukují právě jednu funkci. Výrazy V', V'' proto budeme zapisovat bez závorek, tj. právě tvarem 1.1.

Dále abychom maximálně zjednodušili zápis výrazů, budeme ve výrazu často nahrazovat symbol logického součinu . prázdným řetězcem. Použitím všech uvedených konvencích můžeme nyní příkladový výraz V zapsat v jednoduchém tvaru

$$E \equiv x_1x_2\overline{x_3} + x_1.$$

Výraz E můžeme upravit podle zákonů $\mathcal{B}_{log}$ ($\mathcal{B}_{xlog}$) a zavedené symboliky

$$E \equiv x_1x_2\overline{x_3} + x_1 = x_1x_2\overline{x_3} + x_11 = x_1(x_2\overline{x_3} + 1) = x_1.$$

Definice 1.12. O výrazu V řekneme, že je

- **literálem**, je-li V proměnná nebo její negace
- **termem**, je-li V výraz složený výhradně z literálů a jedné logické operace.
- **součinovým termem**, je-li V term s operací logického součinu.
- **součtovým termem**, je-li V term s operací logického součtu.
- **mintermem** funkce $f : \{0,1\}^n \to \{0,1,X\}$, je-li V součinový term složený z n literálů navzájem různých proměnných.
- **maxtermem** funkce $f : \{0,1\}^n \to \{0,1,X\}$, je-li V součtový term složený z n literálů navzájem různých proměnných.
- **implikantem** funkce f, je-li V součinový term a navíc platí: je-li V v daném bodě pravdivý, pak je i funkce f v tomto bodě pravdivá ($\forall \vec{c} \in D_f : h\big(V(\vec{c})\big) = 1 \Longrightarrow f(\vec{c}) = 1$).
- **přímým implikantem** funkce f, je-li V implikantem f, který nelze dále upravit na jednodušší implikant f (V obsahuje minimální možný počet operací logického součinu).
- v **disjunktivní normální formě (DNF)**, je-li $V \equiv 1$ nebo V je součtem součinových termů, tj. $V \equiv V_1 + \cdots + V_k$ ($k \geq 1$) a všechna V_i ($1 \leq i \leq k$) jsou součinové termy.
- v **konjunktivní normální formě (CNF)**, je-li $V \equiv 0$ nebo V je součinem součtových termů, tj. $V \equiv V_1.\cdots.V_k$ ($k \geq 1$) a všechna V_i ($1 \leq i \leq k$) jsou součtové termy.

Výrazy $E_1(x) \equiv x$ a $E_2(x) \equiv \overline{x}$ jsou literály a také termy. Dalším příkladem termů jsou $E_3 \equiv x_0 + \overline{x_1} + x_2$ resp. $E_4 \equiv x_0\overline{x_1}x_2$ (E_3 je součtový resp. E_4 součinový term). E_4 je implikantem (nikoliv však přímým) funkce h dané tabulkou 1.4. Přímými implikanty $h(X)$ jsou například $E_5 = x_0\overline{x_1}$ nebo $E_6 = x_0x_1\overline{x_2}$. Funkci h lze zapsat v DNF resp. CNF jako $h = \overline{x_0}x_1 + x_0\overline{x_1} + x_0x_1\overline{x_2}$ resp. $h = (x_0 + x_1).(\overline{x_0} + \overline{x_1} + \overline{x_2})$.

Věta 1.4 (**Existence DNF a CNF funkce**). *Každou úplně určenou logickou funkci $f : \{0,1\}^n \to \{0,1\}$ lze zapsat v DNF či CNF.*

Důkaz. Uvažujme nejprve DNF. Pro každý bod $\vec{c} \in \mathcal{D}_f^1$ sestrojíme odpovídající přímý implikant $f_{\vec{c}}$ funkce f ($f_{\vec{c}}(\vec{c}) = 1$), který je navíc ve všech $\vec{x} \in \mathcal{D}_f^0$ nepravdivý ($f_{\vec{c}}(\vec{x}) = 0$). Je-li $\vec{c} = (c_1, \ldots, c_n)$, pak implikant bude mít tvar $f_{\vec{c}} = l_1l_2\cdots l_n$, kde $l_i = x_i$ pro $c_i = 1$ a $l_i = \overline{x_i}$ pro $c_i = 0$ ($1 \leq i \leq n$). Díky vlastnosti logického součtu potom

$$f = \sum_{\vec{c} \in \mathcal{D}_f^1} f_{\vec{c}}$$

je hledanou DNF. V případě CNF stačí stačí vyjádřit funkci $\overline{f}$ pomocí DNF a poté na DNF uplatnit De Morganovi zákony (viz. níže uvedený příklad). □

(x_0, x_1, x_2)	$h(x)$
$(0,0,0)$	0
$(0,0,1)$	0
$(0,1,0)$	1
$(0,1,1)$	1
$(1,0,0)$	1
$(1,0,1)$	1
$(1,1,0)$	1
$(1,1,1)$	0

Tabulka 1.4: DNF a CNF logické funkce

Vyjádření DNF resp. CNF z předešlé věty 1.4 nazýváme **úplnou disjunktivní normální formou (DNF)** resp. **úplnou konjunktivní normální formou (CNF)** funkce f.

Funkci h můžeme vyjádřit v úplné DNF jako $h = \overline{x_0}.x_1.\overline{x_2} + \overline{x_0}.x_1.x_2 + x_0.\overline{x_1}.\overline{x_2} + x_0.\overline{x_1}.x_2 + x_0.x_1.\overline{x_2}$. Úplnou CNF získáme z úplné DNF $\overline{h}$ následujícími úpravami:

$$\begin{aligned} h = \overline{\overline{h}} &= \overline{\overline{x_1}.\overline{x_2}.\overline{x_3} + \overline{x_1}.\overline{x_2}.x_3 + x_1.x_2.x_3} = \overline{(\overline{x_1}.\overline{x_2}.\overline{x_3})}.\overline{(\overline{x_1}.\overline{x_2}.x_3)}.\overline{(x_1.x_2.x_3)} \\ &= (x_1 + x_2 + x_3).(x_1 + x_2 + \overline{x_3}).(\overline{x_1} + \overline{x_2} + \overline{x_3}) \end{aligned}$$

Zajímavými tvary DNF resp. CNF jsou ty, které obsahují minimální počet mintermů resp. maxtermů. V těchto případech hovoříme o **minimální DNF** resp. **minimální CNF**.

Definice 1.13 (**Závislá proměnná a množina závislých proměnných funkce**)**.** Nechť je dána logická funkce $f : \{0,1\}^n \to \{0,1,X\}$. Proměnná x_i je *nezávislá* v f jestliže $f[x_i = 0] = f[x_i = 1]$. V opačném případě hovoříme o x_i jako o *závislé* proměnné. *Množinu všech závislých proměnných* značíme obvykle I_f.

Věta 1.5 (**Shannonův expanzní theorém**)**.** *Každou logickou funkci f lze vzhledem k logické proměnné x_i vyjádřit ve tvaru*

$$f = \overline{x_i} f[x_i = 0] + x_i f[x_i = 1]. \tag{1.2}$$

Důkaz. Označme $g = \overline{x_i} f[x_i = 0] + x_i f[x_i = 1]$. Není-li x_i proměnná f, pak $f = f[x_i = 0] = f[x_i = 1]$, a proto $g = (\overline{x_i} + x_i)f = \left| \; x_i \in \{0,1\} \right| = 1.f = f$. Je-li x_i proměnná f, pak pro $x_i = 0$ resp. $x_i = 1$ dosazením do 1.2 dostáváme $g[x_i = 0] = f[x_i = 0]$ resp. $g[x_i = 1] = f[x_i = 1]$. Protože x_i může nabývat pouze hodnot 0 nebo 1, musí nutně $f = g$. □

1.4.3 Pojmy teorie grafů

Pro potřeby definice binárního rozhodovacího diagramu zopakuje pojem grafu a některých jeho „podstruktur“.

Definice 1.14 (**Obyčejný graf**)**.** *Obyčejným neorientovaným resp. orientovaným grafem $\mathcal{G}$* rozumíme uspořádanou dvojici (V, E), kde

1. V je množina vrcholů (nebo též uzlů)
2. E, $E \subset \left\{ \{a,b\} \;\middle|\; a,b \in V \right\}$ resp. $E \subset V \times V$ množina hran

a píšeme $\mathcal{G}(V,E)$. O libovolných $a, b \in V$, takových že $\{a,b\} \in E$ ($(a,b) \in E$), říkáme že jsou spojeny nebo též incidují s hranou $\{a,b\}$ ((a,b)).

V případě obyčejného grafu může libovolnou dvojici vrcholů spojovat nejvýše jedna hrana. Tuto nevýhodu odstraňuje zobecnění obyčejného grafu, pojem multigrafu.

Definice 1.15 (**Multigraf**). *Neorientovaným resp. orientovaným multigrafem* $\mathcal{G}$ rozumíme uspořádanou trojici (V, E, σ), kde

1. V je množina vrcholů (uzlů)
2. E je množina hran splňující $V \cap E = \emptyset$
3. $\sigma : E \to \big\{\{a, b\} \mid a, b \in V\big\}$ resp. $\sigma : E \to V \times V$ je zobrazení nazývané incidence.

Zobrazení σ vrací pro libovolnou hranu z E dvojici vrcholů z V, jež hrana spojuje. Libovolné dva vrcholy z V tak může u multigrafu spojovat libovolný počet hran z E.

Konvence 1.2. Pokud to bude z kontextu jasné, budeme dále nazývat obyčejný graf a multigraf jednotně grafem.

Definice 1.16 (**Sled, cesta a kružnice grafu**). Nechť pro dvojici uzlů v_l, v_h neorientovaného (orientovaného) grafu $\mathcal{G}(V, E, \sigma)$ existuje posloupnost hran

$$\mathcal{S} = \langle e_0, e_1, e_2, \ldots, e_{n-1}, e_n \rangle$$

taková, že $e_i \in E$ pro všechna i, $0 \leq i \leq n$. Potom

1. $\mathcal{S}$ nazveme neorientovaným *sledem* (*spojením*) z vrcholu v_l do v_h, jestliže platí zároveň následující podmínky:
   (a) Každé dvě po sobě jdoucí hrany e_i, e_{i+1}z $\mathcal{S}$ jsou sousední, tzn. $\sigma(e_i) = \{v_i, v_{i+1}\}$ a $\sigma(e_{i+1}) = \{v_{i+1}, v_{i+2}\}$ ($\sigma(e_i) = (v_i, v_{i+1})$ a $\sigma(e_{i+1}) = (v_{i+1}, v_{i+2})$) pro nějakou trojici vrcholů $v_i, v_{i+1}, v_{i+2} \in V$ pro všechna přípustná e_i z $\mathcal{S}$
   (b) Vrcholy v_l a v_h jsou krajní vrcholy $\mathcal{S}$, tj. $v_l = v_0$ a $v_h = v_{n+1}$.
2. $\mathcal{S}$ nazveme neorientovanou (orientovanou) *cestou* z v_l do v_h, jestliže zároveň platí:
   (a) $\mathcal{S}$ je sled z v_l do v_h.
   (b) Všechny vrcholy hran e_i z $\mathcal{S}$ jsou navzájem různé, tj. $v_i \neq v_j$ (z definice sledu) pro $0 \leq i, j \leq n + 1$, $i \neq j$.
3. $\mathcal{S}$ nazveme *kružnicí* (*cyklem*) platí-li zároveň
   (a) $\mathcal{S}$ je sled z v_l do v_h.
   (b) Platnost různosti vrcholů z definice cesty zůstává až na krajní vrcholy v_l a v_h, tj. $v_l = v_h$ a $S' = \langle e_0, e_i, \ldots e_{n-1} \rangle$ je cestou.

Pojmy sledu (spojení) neorientované (orientované) cesty a kružnice (cyklu) zavádíme na obyčejném neorientovaném (orientovaném) grafu obdobným způsobem jako v 1.16 s tím rozdílem, že daný sled neuvádíme jako posloupnost hran, nýbrž jako posloupnost alespoň dvou vrcholů. Vzhledem k tomu, že u obyčejného grafu může dvojici vrcholů spojovat nejvýše jedna hrana, určuje takto definovaný pojem právě jednu posloupnost po sobě jdoucích hran a definice těchto pojmů u multigrafu i obyčejného grafu jsou významově ekvivalentní.

Pojem sledu, neorientované cesty a cyklu zavádíme též na orientovaném grafu. Myslíme tím odpovídající pojem na grafu $\mathcal{G}'(V, E, \sigma')$ vzniklého z $\mathcal{G}(V, E, \sigma)$ zrušením orientace hran.

Definice 1.17 (**Souvislost a silná souvislost grafu**). O grafu $\mathcal{G}(V, E, \sigma)$ (orientovaném i neorientovaném) řekneme, že je *souvislý*, jestliže pro libovolné dva různé vrcholy $x, y \in V$ existuje sled z x do y. V případě že $\mathcal{G}$ je orientovaný a pro libovolné dva různé vrcholy $x, y \in V$ existuje dokonce spojení z x do y, říkáme, že $\mathcal{G}$ je silně souvislý.

Každý silně souvislý graf je tedy souvislý.

Definice 1.18 (**Strom, acyklický, kořenový graf**). Graf $\mathcal{G}(V, E, \sigma)$ nazýváme

1. *stromem*, je-li $\mathcal{G}$ (ať už je orientovaný či neorientovaný) souvislý a v $\mathcal{G}$ neexistuje kružnice. Je-li strom orientovaný, pak vrchol do nějž nevede žádná hrana (je právě jeden) nazýváme *kořenem* stromu a vrcholy z nichž nevede žádná hrana nazýváme *listy* stromu. Délku cesty z kořene stromu do vrcholu x nazýváme *hloubkou*(*patrem*) x. N*-árním pravidelným* stromem máme na mysli strom, kde z každého vrcholu vede buď n nebo 0 hran, *úplným pravidelným stromem výšky* k pak pravidelný strom, v němž mé každý list hloubku k.

2. *acyklickým*, je-li $\mathcal{G}$ orientovaný a v $\mathcal{G}$ neexistuje cyklus.

3. *kořenovým grafem s kořenem* $v \in V$, je-li $\mathcal{G}$ orientovaný, pro libovolný vrchol $y \in V$ různý od kořene v existuje orientovaná cesta z v do y, ale neexistuje orientovaná cesta z y do v.

Z definice 1.18 plyne, že každý kořenový graf je souvislý (mezi libovolnou dvojicí vrcholů $x, y \in V$ existuje sled vzniklý napojením dvou neorientovaných cest, z x do v a z v do y nebo naopak) a není silně souvislý, neboť do kořenu v nevede žádná orientovaná hrana z E.

Poslední potřebný pojem je izomorfismus grafů.

Definice 1.19 (**Izomorfismus grafů**). O grafech $\mathcal{G}_1(V_1, E_1, \sigma_1)$ a $\mathcal{G}_2(V_2, E_2, \sigma_2)$ mluvíme jako o *izomorfních*, existuje-li bijektivní zobrazení $\varphi : V_1 \cup E_1 \to V_2 \cup E_2$ splňující

1. Zůžené zobrazení $\varphi\big|_{V_1} : V_1 \to V_2$ je bijekce.

2. Zůžené zobrazení $\varphi\big|_{E_1} : E_1 \to E_2$ je bijekce.

3. φ zachovává incidenci, tj. pro libovolné $e \in E_1$, $\sigma_1(e) = \{v_1, v_2\}$ resp. $\sigma_1(e) = (v_1, v_2)$ platí

$$\sigma_2\big(\varphi(e)\big) = \big\{\varphi(v_1), \varphi(v_2)\big\} \quad \text{resp.} \quad \sigma_2\big(\varphi(e)\big) = \big(\varphi(v_1), \varphi(v_2)\big).$$

Poznámka 1.3. Je užitečné si všimnout, že pro relaci $\cong$ a libobovolné grafy $\mathcal{G}_1(V_1, E_1, \sigma_1)$, $\mathcal{G}_2(V_2, E_2, \sigma_2)$ a $\mathcal{G}_3(V_3, E_3, \sigma_3)$ platí

1. $\cong$ je reflexivní, tj. $\mathcal{G}_1 \cong \mathcal{G}_1$ dle zobrazení identity (zobrazení, které přenáší každý vrchol a hranu sám na sebe).

2. $\cong$ je symetrická, tj. je-li $\mathcal{G}_1 \cong \mathcal{G}_2$ dle φ, pak i $\mathcal{G}_2 \cong \mathcal{G}_1$ dle φ^{-1} (φ^{-1} je také izomorfismus).

3. $\cong$ je tranzitivní, tj. je-li $\mathcal{G}_1 \cong \mathcal{G}_2$ dle φ a $\mathcal{G}_2 \cong \mathcal{G}_3$ dle ϕ, pak i $\mathcal{G}_1 \cong \mathcal{G}_3$ dle složeného zobrazení $\phi \circ \varphi$ ($\phi \circ \varphi$ je také izomorfismus).

Zobrazením jako grafový izomorfismus splňujícím reflexivitu, symetrii a tranzitivitu říkáme **ekvivalence**.

Jsou-li dva grafy izomorfní, pak jsou „až na pojmenování vrcholů a hran totožné“. Nepřesně, ale výstižně řečeno jde o „pojmenování stejných věcí jinými jmény“.

1.4.4 Asymptotické notace růstu funkcí

Nakonec ještě zopakujme asymptotické notace pro popis růstu reálné funkce jedné reálné proměnné ($f, g : \mathbb{R} \to \mathbb{R}$), které používáme pro vyjadřování složitostí.

$$\begin{aligned}
\Theta(g(x)) &= \big\{ f(x) \;\big|\; \exists c_1, c_2, x_0 \geq 0 : \forall x \geq x_0 : 0 \leq c_1.g(x) \leq f(x) \leq c_2.g(x) \;\big\} \\
\Omega(g(x)) &= \big\{ f(x) \;\big|\; \exists c, x_0 \geq 0 : \forall x \geq x_0 : 0 \leq c.g(x) \leq f(x) \big\} \\
O(g(n)) &= \big\{ f(n) \;\big|\; \exists c, x_0 \geq 0 : \forall x \geq x_0 : 0 \leq f(x) \leq c.g(x) \big\}
\end{aligned}$$

Místo „$\in$“ pak používáme nezcela přesnou notaci „$=$“, tedy například místo $f(n) \in \Theta(g(n))$ píšeme $f(n) = \Theta(g(n))$. Tedy $O(g)$, $\Omega(g)$ a $\Theta(g)$ značí roste rádově nejvýše tak rychle, řádově alespoň tak rychle a řádově stejně rychle jako g.

Kapitola 2

Rozhodovací diagramy

2.1 Uspořádaný n-ární rozhodovací diagram

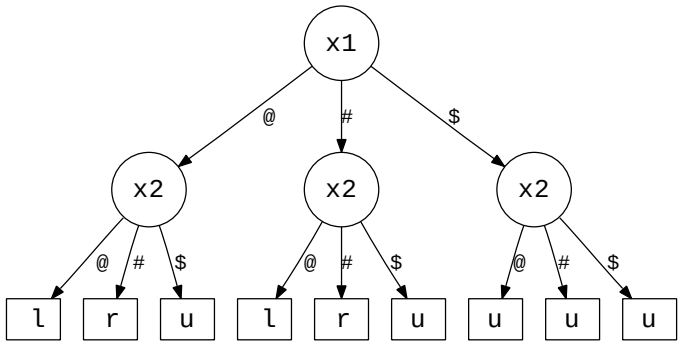
Zabývejme se reprezentací obecné funkce $f : D^k \to H$ ($|D| = n \in \mathbb{N}$). Protože je množina D konečná, jednou z možností je reprezentovat f úplným pravidelným n-árním stromem $\mathcal{G}(V, E, \sigma)$ výšky k, v němž všechny nelistové vrcholy z množiny $N \subset V$ v libovolné hloubce i odpovídají proměnné x_{i+1} funkce f a listy z $T = V \setminus N$ reprezentují funkční hodnoty f z H. Z každého vrcholu $v \in N$ reprezentujícího proměnnou x_i vede právě $n = |D|$ orientovaných hran, kde každá hrana odpovídá ohodnocení proměnné x_i jedním prvkem z D. Tyto hrany vstupují buď do vrcholů $w \in T$, které reprezentují funkční hodnoty f nebo do $w \in N$, které odpovídají proměnné x_{i+1}, v nichž dále pokračuje n-ární větvení pro x_{i+1}.

Každá cesta z kořene $\mathcal{G}$ do nějakého listu $w \in T$ tak určuje právě jedno ohodnocení všech k proměnných funkce f, tedy určuje bod $\vec{c} \in D_f$. Listu w je pak přiřazena hodnota $f(\vec{c}) \in H$. Zajímá-li nás tedy funkční hodnota v $\vec{c} \in D_f$, stačí se z kořene grafu vydat po cestě odpovídající $\vec{c}$ a na konci cesty přečíst v listu hodnotu $h \in H$ jíž funkce v $\vec{c}$ nabývá ($f(\vec{c}) = h$). Těmto grafům říkáme *úplné n-ární rozhodovací diagramy*. Příkladem budiž diagram funkce $g : \{@, \#, \$\}^3 \to \{l, r, u\}$ na obrázku 2.1.

Někdy se nám může hodit předpokládat jiné pořadí proměnných na cestách z kořene do terminálů (doteď jsme předpokládali, že větvení probíhá postupně v $x_1, x_2, \ldots, x_n$). Diagramy, v nichž je dáno pořadí proměnných permutací ρ, budeme nazývat *uspořádané*. Naopak *neuspořádaným* diagramem máme na mysli diagram, v němž se mohou posloupnosti proměnných navštívených na různých cestách z kořene do terminálů lišit. V textu se budeme věnovat výhradně uspořádaným diagramům.

Přirozeným požadavkem je, aby reprezentace f byla minimální. Často se stává, že některé podstromy úplného uspořádaného rozhodovacího diagramu jsou vzájemně izomorfní (izomorfismus zde bere v ohled i proměnné příslušející nelistovým vrcholům, ohodnocení hran prvky z D a ohodnocení listů prvky z H). V takovém případě je jeden z podstromů zcela zbytečný, neboť můžeme všechny hrany vedoucí do prvního z izomorfních podstromů přesměrovat do izomorfismem přidružených vrcholů druhého podstromu a poté první podstrom z diagramu odstranit (nevede totiž do něj již žádná hrana). Funkční hodnota f daná cestou libovolného bodu $\vec{c} \in D_f$ pak jistě zůstane zachována, tedy modifikovaný diagram určuje opět f. Na obrázku 2.2 je uveden diagram funkce g z 2.1 po odstranění všech izomorfních podstromů. Je vidět, že původní úplný diagram se tím značně redukoval.

Na obrázku 2.2 je také vidět, že u druhého vrcholu reprezentujícího x_2 vedou všechny hrany do vrcholu s hodnotou u, tj. ať ohododnotíme proměnnou x_2 jakkoliv, vždy skončíme v terminálu u. Protože pořadí proměnných (shora dolů) je v uspořádaném diagramu určeno jednoznačně, můžeme všechny hrany vedoucí do x_2 přemostit do u a vrchol x_2 z diagramu odstranit, aniž bychom ztratil nějakou informaci o f. Neprojdeme-li pak na cestě určené $\vec{c}$ žádným vrcholem odpovídající proměnné x_i, pak f v ohodnocení zbývajících proměnných (určeno $\vec{c}$) na x_i nezávisí. Diagramy, které nelze podle uvedeného postupu a izomorfismu dále zjednodušit, budeme nazývat *redukované uspořádané n-ární rozhodovací diagramy*. Vůbec, částečně či úplně zjednodušeným uspořádaným diagramům budeme říkat souhrně *uspořádané n-ární rozhodovací diagramy*. Přistupme nyní k formálním definicím.



$\vec{x}$	$g(\vec{x})$
$(@, @)$	l
$(@, \#)$	r
$(@, \$)$	u
$(\#, @)$	l
$(\#, \#)$	r
$(\#, \$)$	u
$(\$, @)$	u
$(\$, \#)$	u
$(\$, \$)$	u

Obrázek 2.1: Úplný uspořádaný rozhodovací diagram funkce $g : \{@, \#, \$\}^2 \to \{l, r, u\}$ definované tabulkou.

Definice 2.1 (**Uspořádaný n-ární rozhodovací diagram (ODD)**). *Uspořádaným n-árním rozhodovacím diagramem* $\mathcal{D}(n, D, H, N, T, E, \rho, \sigma, son, idx, val)$ na množině D ($|D| = n$) a do množiny H, daný uspořádáním proměnných ρ ($\rho = \langle i_1, i_2, \ldots, i_k \rangle$ je permutace přirozených čísel z množiny $\{0, 1, \ldots, k\}$), rozumíme libovolný souvislý kořenový acyklický graf $\mathcal{G}(V, E, \sigma)$, který díky své acykličnosti rozděluje množinu vrcholů V na dvě disjunktní podmnožiny *neterminálních* a *terminálních* vrcholů N a T ($N \cup T = V$ a $N \cap T = \emptyset$) splňujících:

1. Z libovolného neterminálního vrcholu $v \in N$ vede právě n orientovaných hran do vrcholů $v_1, v_2, \ldots, v_n \in V$, přitom vrcholy $v_1, v_2, \ldots, v_n$ nemusí být nutně různé. Na N jsou dále definovány operace:

   (a) Operace *son (potomek)*, $son : N \times D \to V$, která vrací zmíněné následníky v, tj. pro všechna $v \in N$ je
   $$son(v, D) = \big\{w \in V \;\big|\; \text{existuje } e \in E, \sigma(e) = (v, w)\big\}.$$

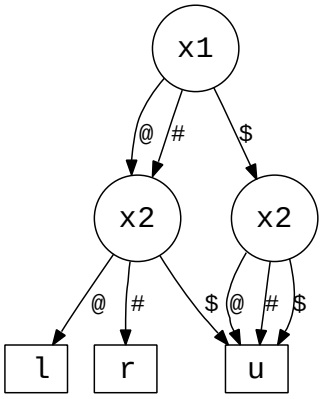
   (b) Operace *idx (index)*, $idx : N \to \{0, 1, \ldots, k\}$. Pro libovolného potomka w neterminálu v ($w = son(v, d)$, $v \in N$, $d \in D$), který je také neterminálem ($w \in N$) stojí $idx(v)$ v posloupnosti ρ před (ne nutně bezprostředně) $idx(w)$.

2. Z libovolného terminálního vrcholu $v \in T$ nevede žádná hrana, tj. pro všechny $v \in T$ platí
   $$\emptyset = \big\{w \in V \;\big|\; \text{existuje } e \in E, \sigma(e) = (v, w)\big\}.$$
   Na T je definována jediná operace:

   (a) Operace *val (value)*, $val : T \to H$.

Graf $\mathcal{G}(V, E, \sigma)$ nazýváme *grafem diagramu* $\mathcal{D}$. Je-li $\mathcal{D}(n, D, H, N, T, E, \rho, \sigma, son, idx, val)$ úplným vyváženým n-árním stromem, mluvíme o $\mathcal{D}$ jako o **úplném uspořádaném n-árním rozhodovacím diagramu** (v opačném případě o **neúplném uspořádaném n-árním rozhodovacím diagramu**).

Poznámka 2.1. V textu se budeme zabývat výhradně uspořádanými rozhodovacími diagramy. Kdykoliv dále použijeme pojem diagramu nebo rozhodovacího diagramu, budeme tím mít na mysli uspořádaný rozhodovací diagram. Místo ODD budeme často používat kratší značku **DD**.



Obrázek 2.2: Částečně redukovaný uspořádaný rozhodovací diagram funkce g (2.1).

Definice 2.2 (**Funkce určená diagramem**)**.** Nechť je dán uspořádaný rozhodovací diagram $\mathcal{D}(n, D, H, N, T, E, \rho, \sigma, son, idx, val)$, kde $\rho = \langle i_1, i_2, \ldots, i_k \rangle$. Nechť $f : D^k \to H$ je funkce proměnných $x_1, x_2, \ldots, x_k$ ($f(x_1, \ldots, x_k)$) definovaná pro všechna $\vec{c} \in D_f$

$$f(\vec{c}) = f(c_1, c_2, \ldots, c_n) := h,$$

kde $h = val(t)$, přičemž pro terminál $t \in T$ platí

$$\begin{aligned} t &= son(v_l, c_{j_l}) \\ v_l &= son(v_{l-1}, c_{j_{l-1}}) \\ v_{l-1} &= son(v_{l-2}, c_{j_{l-2}}) \\ &\ldots \\ v_2 &= son(r, c_{j_1}), \end{aligned}$$

kde r je kořen $\mathcal{D}$ ($r, v_2, \ldots, v_l \in N$). Jinými slovy $f(\vec{c})$ je určená hodnotou h terminálu t ukončujícího cestu vedoucí z kořene r diagramu $\mathcal{D}$ „podle“ vektoru $\vec{c}$. Pak o f říkáme, že je definovaná $\mathcal{D}$ a píšeme $\mathcal{D} \rhd f$ nebo $f \lhd \mathcal{D}$.

Definice 2.3 (**Izomorfismus diagramů**)**.** N a M-ární rozhodovací diagramy $\mathcal{D}_1(n, D_1, H_1, N_1, T_1, E_1, \rho_1, \sigma_1, son_1, idx_1, val_1)$ a $\mathcal{D}_2(m, D_2, H_2, N_2, T_2, E_2, \rho_2, \sigma_2, son_2, idx_2, val_2)$ jsou **izomorfní**, právě tehdy když grafy obou diagramů jsou izomorfní, tj. $\mathcal{G}_1(N_1 \cup T_1, E_1, \sigma_1) \cong \mathcal{G}_2(N_2 \cup T_2, E_2, \sigma_2)$ dle izomorfismu φ a zároveň platí:

1. $D_1 = D_2$ a pro všechna $v \in N_1$, $d \in D_1$ platí $son_1(v, d) = \varphi\big(son_2\big(\varphi(v), d\big)\big)$
2. pro všechna $v \in N : idx_1(v) = idx_2(\varphi(v))$.
3. pro všechna $v \in T : val_1(v) = val_2(\varphi(v))$.

Izomorfnost obou diagramů zapisujeme $\mathcal{D}_1 \cong \mathcal{D}_2$.

Poznámka 2.2. Z definice izomorfismu diagramů plyne, že nutnými podmínkami izomorfismu $\mathcal{D}_1$ a $\mathcal{D}_2$ z 2.3 jsou:

- $n = m$ kdykoliv $N_1 \neq \emptyset$ nebo $N_2 \neq \emptyset$.
- $D_1 = D_2$.

- $H_1 \subset H_2$ nebo $H_2 \subset H_1$.
- $|N_1| = |N_2|$, $|T_1 = T_2|$, $|E_1| = |E_2|$.
- existence grafového izomorfismu φ mezi grafy obou diagramů, který je rozšířen a respektuje i operace *son*, *index* a *value*.

Vybereme-li z jakéhokoliv diagramu $\mathcal{D}$ libovolný vrchol v, pak tento vrchol určuje právě jeden podstrom jehož je v kořenem, tedy právě jeden **subdiagram** $\mathcal{D}'$.

2.2 RODD a kanonicita

Definice 2.4 (**Zbytečný vrchol diagramu**)**.** O neterminálním vrcholu $v \in N$ diagramu $\mathcal{D}(n, D, H, N, T, E, \rho, \sigma, son, idx, val)$ řekneme, že je zbytečný, jestliže všechny hrany vycházející z v končí v právě jednom vrcholu $w \in N \cup T$, tj.

$$son(v, D) = \{w\}.$$

V úvodu kapitoly jsme nastínili případy, v nichž lze n-ární diagram $\mathcal{D}$ zjednodušit, aniž by se funkce f určená $\mathcal{D}$ změnila. Budou to tyto případy

1. Subdiagramy dvou různých vrcholů v_a a v_b jsou izomorfní.
   - *zjednodušení*: všechny hrany vedoucí do v_a přesměrujeme do v_b, poté z $\mathcal{D}$ odstraníme subdiagram určený v_a.
2. Je-li neterminál v_c zbytečný, tj. z v_c vede všech n hran do jednoho vrcholu v_d.
   - *zjednodušení*: všechny hrany vedoucí do v_c přesměrujeme do v_d, poté z $\mathcal{D}$ vrchol v_c odstraníme.

Nyní bychom měli formálně dokázat, že uvedené zjednodušující operace nemění funkci určenou původním $\mathcal{D}$ a zjednodušení jsou tak korektní. Platnost této skutečnosti je však natolik zřejmá a naopak formálně pracná, že případný důkaz necháme na čtenáři.

Definice 2.5 (**Redukovaný n-ární rozhodovací diagram (RODD)**)**.** Uspořádaný n-ární rozhodovací diagram $\mathcal{D}(n, D, H, N, T, E, \rho, \sigma, son, idx, val)$ nazveme *redukovaným rozhodovacím diagramem* nebo *RODD* tehdy, když v $\mathcal{D}$ neexistuje zbytečný vrchol a zároveň libovolné dva vrcholy $v_a, v_b \in N \cup T$ $(v_a \neq v_b)$ určují neizomorfní subdiagramy.

Na obrázku 2.3 je vidět příklad dvou redukovaných rozhodovacích diagramů pro různé uspořádání proměnných. Je vidět, že uspořádání proměnných výrazně ovlivňuje velikost výsledných RODD.

Pozorného čtenáře napadne otázka: Je výsledný RODD závislý na pořadí, v němž provádíme nalezená zjednodušení diagramu? Nebo ať aplikujeme postupná zjednodušení v jakémkoliv pořadí, vždy nakonec dospějeme k dvojici izomorfních RODD? Následující důležitá věta ukazuje, že nastává druhá možnost.

Věta 2.1 (**Kanonicita RODD**)**.** *Nechť $\mathcal{D}_1(n, D, H, N_1, T_1, E_1, \rho, \sigma_1, son_1, idx_1, val_1)$ resp. $\mathcal{D}_2(n, D, H, N_2, T_2, E_2, \rho, \sigma_2, son_2, idx_2, val_2)$ jsou RODD vzniklé zjednodušením diagramu $\mathcal{D}(n, D, H, N, T, E, \rho, \sigma, son, idx, val)$. Potom $\mathcal{D}_1 \cong \mathcal{D}_2$.*

Důkaz. Víme, že $\mathcal{D}_1$, $\mathcal{D}_2$ i $\mathcal{D}$ určují stejnou funkci ($\mathcal{D}_1 \rhd f$, $\mathcal{D}_2 \rhd f$, $\mathcal{D} \rhd f$). Větu dokážeme matematickou indukcí podle velikosti množiny I_f (množina proměnných, na níž funkce závisí, která je analogií definice 1.13).

1. *Indukční předpoklad:* Je-li $|I_f| = 0$, pak je f konstantní, tj. $f = h \in H$. Pak RODD funkce f obsahuje jediný terminální vrchol t ($T = \{t\}$, $N = \emptyset$) s hodnotou $val(t) = h$. Skutečně, předpokládejme opak, že $N \neq \emptyset$. Protože $f = h$, musí pro všechna $w \in T$ platit $val(w) = h$ (jinak by f nebyla konstantní). Pak, jsou ale všechny $w \in T$ vzájemně izomorfní a lze je nahradit jedinným terminálem $t \in T$. V tom případě se stává neterminál $v \in N$, z nějž vystupuje n hran do t zbytečným a proto diagram nemůže být redukovaný.

2. *Indukční krok:* Dokážeme, že platí-li tvrzení pro $0 \leq |I_f| \leq k$, potom platí i pro $|I_f| = k+1$ ($k \geq 0$). Označme x_i nejvýše postavenou proměnnou f (proměnnou jejíž index i leží v uspořádání proměnných ρ co nejblíže začátku), na níž f závisí ($x_i \in I_f$). Protože $|I_f| > 0$, taková proměnná musí existovat. Teoreticky mohou nastat tři případy:

   (a) V RODD $\mathcal{D}_1$ nebo $\mathcal{D}_2$ neexistuje neterminál v odpovídající proměnné x_i ($idx(v) = i$). Bez újmy na obecnosti předpokládejme, že v $\mathcal{D}_1$. Protože $\mathcal{D}_1 \rhd f$, musí pak platit

   $$f[x_i = d_1] = f[x_i = d_2] = \cdots = f[x_i = d_n] = f \quad d_i \in D,$$

   odkud plyne $x_i \notin I_f$. Spor.

   (b) V RODD $\mathcal{D}_1$ a $\mathcal{D}_2$ existuje právě jeden vrchol $r_1 \in N_1$ a $r_2 \in N_2$, který reprezentuje x_i ($idx_1(r_1) = i$ a $idx_2(r_2) = i$), přičemž r_1 resp. r_2 je kořenem $\mathcal{D}_1$ resp. $\mathcal{D}_2$. Označme $\mathcal{D}_{1,1}, \ldots \mathcal{D}_{1,n}$ resp. $\mathcal{D}_{2,1}, \ldots \mathcal{D}_{2,n}$ subdiagramy určené jednotlivými potomky kořenu r_1 resp. r_2

   $$\begin{array}{lll} \mathcal{D}_{1,j} & \text{je subdiagram určený} & son_1(r_1, d_j) \quad 1 \leq j \leq n \quad d_j \in D \\ \mathcal{D}_{2,j} & \text{je subdiagram určený} & son_2(r_2, d_j) \quad 1 \leq j \leq n \quad d_j \in D, \end{array}$$

   ($\mathcal{D}_{1,1}, \ldots \mathcal{D}_{1,n}$ resp. $\mathcal{D}_{2,1}, \ldots \mathcal{D}_{2,n}$ nemusí být nutně navzájem různé). Funkce indukované $\mathcal{D}_{1,j}$ a $\mathcal{D}_{2,j}$ ($0 \leq j \leq n$) jsou totožné, neboť vznikly v obou případech restrikcí f podle stejné proměnné x_i s hodnotou d_j. Protože $|I_{f[x_i=d_j]}| = k+1-1 = k$ a $\mathcal{D}_{1,j}$ i $\mathcal{D}_{2,j}$ jsou také RODD (jinak by $\mathcal{D}_1, \mathcal{D}_2$ nebyli RODD), platí dle indukčního předpokladu $\mathcal{D}_{1,j} \cong \mathcal{D}_{2,j}$ pro všechna přípustná j (dle izomorfismů φ_j). Hledaný izomorfismus φ (pro nějž $\mathcal{D}_1 \cong \mathcal{D}_2$) sestrojíme následovně

   $$\varphi(v) = \begin{cases} r_2 & v = r_1 \\ \varphi_1(v) & v \in \mathcal{D}_{1,1} \\ \varphi_2(v) & v \in \mathcal{D}_{1,2} \\ \cdots & \\ \varphi_n(v) & v \in \mathcal{D}_{1,n} \end{cases} \tag{2.1}$$

   Platí, že $(\mathcal{D}_{1,j} = \mathcal{D}_{1,l}) \iff (\mathcal{D}_{2,j} = \mathcal{D}_{2,l})$, $0 \leq j, l \leq n$. Pokud by totiž například $\mathcal{D}_{1,j} \neq \mathcal{D}_{1,l}$, ale $\mathcal{D}_{2,j} = \mathcal{D}_{2,l}$, potom $\mathcal{D}_{1,j} \cong \mathcal{D}_{2,j} \cong \mathcal{D}_{2,l} \cong \mathcal{D}_{1,l}$, tedy $\mathcal{D}_{1,j} \cong \mathcal{D}_{1,l}$, proto $\mathcal{D}_1$ nemůže být redukovaný. Případ $\mathcal{D}_{1,j} = \mathcal{D}_{1,l}$, ale $\mathcal{D}_{2,j} \neq \mathcal{D}_{2,l}$ se ověří analogicky s využitím skutečnosti, že φ_j^{-1} a φ_l^{-1} jsou také izomorfismy. Zkontrolovat nyní, že složené zobrazení φ je diagramový izomorfismus je triviální.

   (c) Kořen $r_1 \in N_1$ RODD $\mathcal{D}_1$ nebo kořen $r_2 \in N_2$ RODD $\mathcal{D}_2$ neodpovídá proměnné x_i ($val(r_1) \neq i$ nebo $val(r_2) \neq i$). Bez újmy na obecnosti předpokládejme, že jde o r_1. Pak $val(r_1)$ musí být index výše postavené proměnné (stojí v posloupnosti ρ před i). V opačném případě by nastal případ (a), který jsme vyloučili. Odtud plyne, že $x_{val(r_1)} \notin I_f$ (jinak bychom se dostali do sporu s výběrem proměnné x_i), Označme $\mathcal{D}_{1,1}, \ldots, \mathcal{D}_{1,n}$ subdiagramy určené potomky $v_1, \ldots, v_n$ kořene r_1 ($v_j = son(r_1, d_j)$, $d_j \in D$). Potom dle $x_{val(r_1)} \notin I_f$

   $$\mathcal{D}_{1,1} \rhd f, \quad \mathcal{D}_{1,2} \rhd f, \quad \ldots \quad , \mathcal{D}_{1,n} \rhd f.$$

   Na $\mathcal{D}_{1,1}, \ldots \mathcal{D}_{1,n}$ můžeme rekurzivně opakovat právě uvedenou myšlenkum dokud v každé větvi nedorazíme do nějakého $w \in N$ takového, že $idx(w) = i$ (to se nám díky neplatnosti (a) musí podařit). Jinými slovy v $\mathcal{D}_1$ musí existovat $u \in N$, který má potomky $u_1, \ldots, u_n$, kde každý u_j odpovídá proměnné x_i a určuje subdiagram $\mathcal{D}_{1,\ldots,j}$. Protože

   $$\mathcal{D}_{1,\ldots,1} \rhd f, \quad \mathcal{D}_{1,\ldots,2} \rhd f, \quad \ldots \quad , \mathcal{D}_{1,\ldots,n} \rhd f,$$

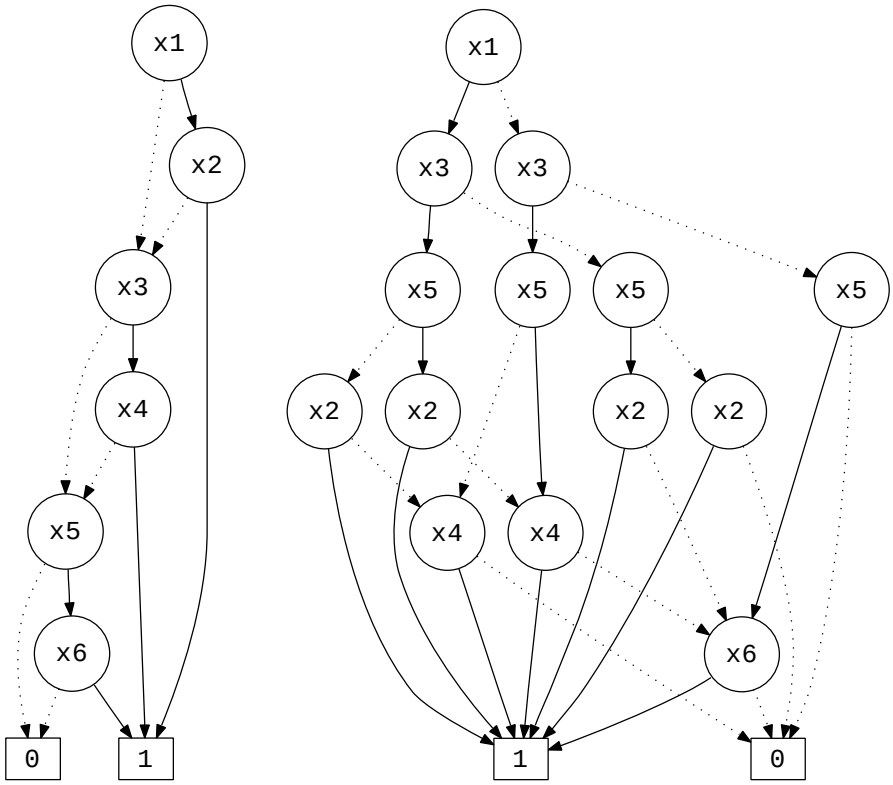
   jsou všechny $\mathcal{D}_{1,\ldots,j}$ dle případu (b) izomorfní a lze je nahradit jediným subdiagramem, např. $\mathcal{D}_{1,\ldots,1}$. Tím se stává u zbytečný. Proto $\mathcal{D}_1$ není redukovaný. Spor.

□

Důsledek 2.1 (**Funkce ku RODD jako** $1:1$)**.** *Libovolné dva RODD funkcí g, h jsou izomorfní, právě tehdy když $f \equiv h$ ($f = h$ na celém $\mathcal{D}_f = \mathcal{D}_h$).*

Důsledek 2.2 (**Charakteristika RODD**)**.** *Z uvedeného důkazu věty 2.1 je vidět, že kořen r každého RODD $\mathcal{D}$ určujícího funkci f odpovídá nejvýše postavené závislé proměnné x_i ($x_i \in I_f$, $idx(r) = i$). Dokonce protože každý subdiagram $\mathcal{D}'$ diagramu $\mathcal{D}$ je také redukovaný, musí $\mathcal{D}$ obsahovat pouze vrcholy příslušející proměnným z množiny závislých proměnných I_f. Celkově s přihlédnutím k 2.(a) v 2.1*

$$idx(N) = \left\{ i \in \mathbb{N} \middle|\ 0 \leq i \leq n = |D| \quad and \quad x_i \in I_f \right\}.$$



Obrázek 2.3: BDD logické funkce $g = x_1x_2 + x_3x_4 + x_5x_6$ pro dvě různá uspořádání proměnných dané posloupnostmi $\rho_1 = \langle 1, 2, 3, 4, 5, 6 \rangle$ a $\rho_2 = \langle 1, 3, 5, 2, 4, 6 \rangle$.

2.3 Uspořádané binární rozhodovací diagramy jako speciální případ ODD

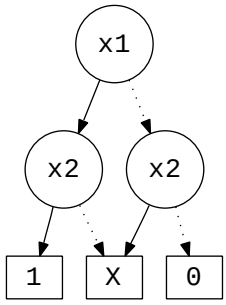
V minulých dvou podkapitolách jsme se zbývaly obecnnými n-árními DD. Zaměřme nyní pozornost na binární rozhodovací diagramy, které budeme dále výhradně používat.

Definice 2.6 (**Uspořádaný binární rozhodovací diagram (OBDD)**)**.** *Uspořádaným binárním rozhodovacím diagramem* máme na mysli každý DD $\mathcal{D}(2, \{0,1\}, \{0,1\}, N, T, E, \rho, \sigma, son, idx, val)$ a pro stručnost značíme $\mathcal{D}(N, T, E,\ \rho, \sigma, son, idx, val)$ nebo jen $\mathcal{D}(N, T, E, \rho)$.

Poznámka 2.3. Definice OBDD vyvstala z potřeby práce s logickými funkcemi. Je tedy striktnější, než bychom čekali, tj. že pouze $|D| = 2$.

Na obrázku 2.3 jsou vidět příklady BDD (plná resp. přerušovaná čára značí ohodnocení proměnné hodnotami 1 resp. 0). Každý BDD definuje jednu úplně logicky určenou funkci. Stejně tak můžeme definovat diagram neúplně určené logické funkce.

Definice 2.7 (**Uspořádaný modifikovaný binární rozhodovací diagram (OMBD)**). *Uspořádaným modifikovaným binárním rozhodovacím diagramem* rozumíme DD $\mathcal{D}(2, \{0,1\}, \{0,1,X\}, N, T, E, \rho, \sigma, son, idx, val)$ a značíme $\mathcal{D}(N, T, E, \rho, \sigma, son, idx, val)$ nebo $\mathcal{D}(N, T, E, \rho)$.



Obrázek 2.4: MBD neúplně určené logické funkce 1.3.

Poznámka 2.4. Obdobně jako u obecných uspořádaných digramů, budeme uspořádané binární a uspořádané modifikované binární diagramy nazývat bez slova uspořádaný a značit pouze **BDD** resp. **MBD** (formálně nekorektně).

Obrázek 2.4 zachycuje MBD funkce 1.3. Vlastnosti redukovaných BDD a MBD se nijak neliší od obecných RODD, **redukované binární rozhodovací diagramy** resp. **redukované modifikované binární rozhodovací diagramy** značíme **ROBDD** resp. **ROMBD**. Protože budeme později pracovat téměř výhradně s ROBDD a ROMBD, budeme pojmy BDD a ROBDD resp. MBD a ROMBD libovolně zaměňovat (pokud nebude hrozit vznik nedorozumění). V analogii s úplnými DD zavádíme také **úplné BDD** a **úplné MBD**.

2.4 Paměťová složitost ROBDD, ROMBD

Nabízí se otázka, nakolik ovlivňuje uspořádání proměnných ρ velikost BDD (MBD) $\mathcal{D}(N, T, E, \rho)$ funkce $f : \{0,1\}^k \to \{0,1,X\}$, tzn. počet vrcholů v $N \cup T$ (samotné hrany přímo minimalizovat nelze, z každého neterminálu vedou právě dvě). Počet vrcholů BDD, MBD nemůže převýšit počet vrcholů úplného BDD, MBD, pro nějž platí

$$|N \cup T| = 1 + |D| + |D|^2 + \cdots + |D|^k + |D|^{k+1} = \frac{|D|^{k+2} - 1}{|D| - 1} = \big||D| = 2\big| = 4.2^k - 1 = O(2^k).$$

Pro úplný n-ární diagram platí $|N \cup T| = O(n^k)$. V nejhorším případě proto BDD (MBD) obsahuje řádově exponenciální počet vrcholů. Jak je tomu ale u ROBDD (ROMBD)?

Uvažujme funkci $g : \{0,1\}^6 \to \{0,1\}$ danou rovností $g(x_1, \ldots, x_6) = x_1x_4 + x_2x_5 + x_3x_6$. Na obrázku 2.3 jsou ROBDD g pro dvě různá uspořádání proměnných $\rho_1 = \langle 1,2,3,4,5,6 \rangle$ a $\rho_2 = \langle 1,3,5,2,4,6 \rangle$. Zatímco pro ρ_1 má ROBDD 8 vrcholů, ROBDD pro ρ_2 má vrcholů 16. Ukážeme, že pro každou zobecněnou funkci $g(x_1, x_2, \ldots, x_{2n}) = x_1x_2 + x_3.x_4 + \cdots + x_{2n-1}x_{2n}$ o $2n$ proměnných, dosáhne ROBDD g při nevhodné volbě uspořádání proměnných paměťové složitosti $\Omega(2^n)$.

Věta 2.2 (**Horní mez paměťové složitosti ROBDD (ROMBD)**). *Existuje logická funkce g : $\{0,1\}^n \to \{0,1\}$ jejíž ROBDD (ROMBD) $\mathcal{D}(N, T, E, \rho)$ při nevhodné volbě uspořádání proměnných ρ má paměťovou složitost $\Omega(2^{\frac{n}{2}})$.*

Důkaz. Uvažujme logickou funkci o $2n$ argumentech definovanou rovností $g(\vec{x}) = x_1x_{n+1} + x_2x_{n+2} + \cdots + x_{n-1}x_{2n-1} + x_nx_{2n}$. Nechť $\rho = \langle 1, 2, 3, \ldots, n\rangle$ a dále B je množina funkcí n proměnných

$$B = \left\{ g_{\vec{b}} \;\middle|\; g_{\vec{b}}(x_{n+1}, x_{n+2}, \ldots, x_{2n}) := b_1x_{n+1} + b_2x_{n+2} + \cdots + b_nx_{2n} \quad \vec{b} = (b_1, \ldots, b_n) \in \{0,1\}^n \right\}.$$

Potom $|B| = 2^n$. Ekvivalentně řečeno pro $\vec{b} \neq \vec{c}$ je $g_{\vec{b}} \neq g_{\vec{c}}$. Skutečně, pokud $\vec{b} \neq \vec{c}$, pak existuje souřadnice i $(1 \leq i \leq n)$, tak že $b_i \neq c_i$. Bez újmy na obecnosti předpokládejme $b_i = 1$ a $c_i = 0$. Potom pro bod

$$\vec{x} = (\underbrace{0, \ldots, 0, 1}_{i}, 0, \ldots, 0) \quad \Longrightarrow \quad 1 = g_{\vec{b}}(\vec{x}) \neq g_{\vec{c}}(\vec{x}) = 0,$$

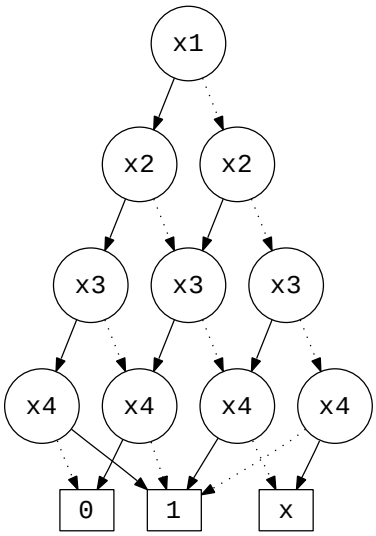
tj. $g_{\vec{b}} \neq g_{\vec{c}}$. Přitom diagram každé funkce $g_{\vec{b}} \in B$ se v $\mathcal{D}$ vyskytuje jako subdiagram $\mathcal{D}_{\vec{b}}$ (do kořene $\mathcal{D}_{\vec{b}}$ se dostaneme cestou určenou $(x_1, x_2, \ldots, x_n) = (b_1, b_2, \ldots, b_n)$). Dle důsledku 2.1 tak v $\mathcal{D}$ existuje $\Omega(2^n)$ vrcholů. □

2.4.1 Symetrické funkce a jejich ROBDD(ROMBD)

Zajímavou třídou funkcí jsou funkce symetrické v proměnných. Jsou to funkce $f : D^k \to H$, pro něž platí

$$f(d_{i_1}, d_{i_2}, \ldots, d_{i_k}) = f(d_{j_1}, d_{j_2}, \ldots, d_{j_k}),$$

nezávisle na volbě všech $d_l \in D$ ($(i_1, i_2, \ldots, i_k)$ a $(j_1, j_2, \ldots, j_k)$ jsou libovolné permutace čísel $1, 2, \ldots, k$). Na obrázku 2.5 je příklad ROMBD neúplně určené symetrické logické funkce. Následující věta ukazuje, že počet vrcholů ROBDD (ROMBD) každé symetrické logické funkce je $O(n^2)$.



Obrázek 2.5: Příklad ROMBD neúplně určené symetrické logické funkce

Lemma 2.1. *Pro libovolnou symetrickou úplně (neúplně) určené logickou funkci f proměnných $x_1, x_2, \ldots, x_n$ na $\{0,1\}^n$ a $\vec{c} = (c_1, c_2, \ldots, c_k) \in \{0,1\}^k$, $\vec{d} = (d_1, d_2, \ldots, d_k) \in \{0,1\}^k$ $(0 \leq k \leq n)$, kde $\vec{c}$ a $\vec{d}$ obsahují stejný počet 0 a 1, platí*

$$f_{\vec{c}} := f[x_{i_1} = c_1][x_{i_2} = c_2] \ldots [x_{i_k} = c_k] \quad \equiv \quad f_{\vec{d}} := f[x_{i_1} = d_1][x_{i_2} = d_2] \ldots [x_{i_k} = d_k],$$

ať už jsou navzájem různá $i_1, i_2, \ldots, i_k \in \{1, 2, \ldots, n\}$ jakákoliv.

Důkaz. Protože f je symetrická, závisí pro $\forall\vec{b} \in \{0,1\}^n$ funkční hodnota $f(\vec{b})$ pouze na počtu 0 a 1 v $\vec{b}$, nikoliv na jejich uspořádání. Pro $\forall\vec{e} \in \{0,1\}^{n-k}$ mají $\vec{c}, \vec{d} \in \{0,1\}^n$ vzniklé „rozšířením“ $\vec{e}$ o hodnoty vektorů $\vec{c}, \vec{d}$ na pozicích $i_1, \ldots, i_k$ stejný počet hodnot 0 a 1. Dle zmíněného $f(\vec{c}) = f(\vec{d})$. Protože $f(\vec{c}) = f_{\vec{c}}(\vec{e})$ a $f(\vec{d}) = f_{\vec{d}}(\vec{e})$, platí $f_{\vec{c}}(\vec{e}) = f_{\vec{d}}(\vec{e})$. Nezávislost volby $\vec{e} \in \{0,1\}^{n-k}$ ukončuje důkaz. □

Věta 2.3 (Paměťová složitost ROBDD (ROMBD) symetrické logické funkce). *Počet vrcholů ROBDD (ROMBD) $\mathcal{D}(N,T,E,\rho)$ symetrické úplně (neúplně) určené logické funkce f na $\{0,1\}^n$ je $O(n^2)$.*

Důkaz. Ukážeme, že v i-tém patře $(0 \leq i \leq n-1)$ $\mathcal{D}$ se může vyskytovat nejvýše $i+1$ vrcholů z N. Potom totiž

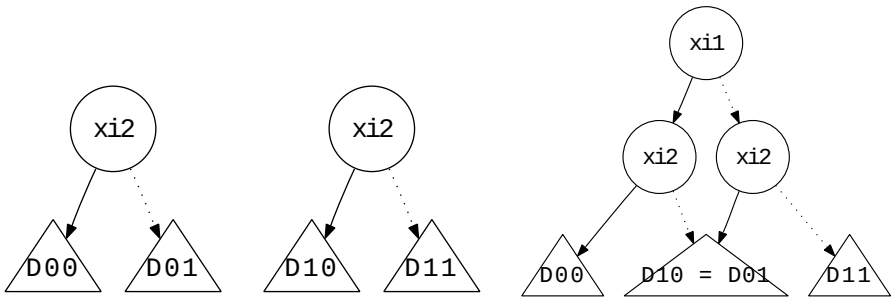
$$|N| \leq 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \quad \Longrightarrow \quad |N \cup T| = O(n^2).$$

Postupujme matematickou indukcí podle počtu proměnných (n).

1. *Indukční předpoklad.* Pro $n = 1$ mohou nastat dva případy:
   (a) $f[x=0] = f[x=1]$. Pak $\mathcal{D}$ obsahuje pouze jeden terminál $(N = \emptyset)$, tj. předpoklad je triviálně splněn.
   (b) $f[x=0] \neq f[x=1]$. Pak $\mathcal{D}$ obsahuje jediný neterminál $(N = \{v\})$, z nějž vedou hrany do dvou terminálů. Neterminál v je 0-tém patře, takže předpoklad je opět splněn.
2. *Indukční krok.* Ukážeme, že platí-li tvrzení pro funkci o $n \geq 1$ proměnných, potom platí i pro funkci o $n+1$ proměnných. Nechť tedy f je libovolná symetrická logická funkce o $n+1$ proměnných a $\mathcal{D}(N,T,E,\rho)$ $(\rho = \langle i_1, \ldots, i_{n+1}\rangle)$ její ROBDD (ROMBD). Dle 2.1 platí

$$f_{01} := f[x_{i_1} = 0][x_{i_2} = 1] \qquad \equiv \qquad f_{10} := f[x_{i_1} = 1][x_{i_2} = 0]$$

a odpovídající subdiagramy $\mathcal{D}_{01}$ a $\mathcal{D}_{10}$ jsou izomorfní (obrázek 2.6). Poznamenejme, že subdiagramy $\mathcal{D}_{00}$ a $\mathcal{D}_{01}$ resp. $\mathcal{D}_{10}$ a $\mathcal{D}_{11}$ nemusí být nutně disjunktní.



Obrázek 2.6: Izomorfismus subdiagramů symetrické logické funkce

Označme

$$f_0 := f[x_{i_1} = 0] \qquad f_1 := f[x_{i_1} = 1]$$

a $\mathcal{D}_0$ resp. $\mathcal{D}_1$ RODD určené f_0 resp. f_1 $(\mathcal{D}_0 \triangleright f_0, \mathcal{D}_1 \triangleright f_1)$. f_0 a f_1 jsou funkce n proměnných, splňují proto předpoklad věty, tedy v $\mathcal{D}_0$ i $\mathcal{D}_1$ existuje nejvýše $1, 2, \ldots, n$ vrcholů reprezentující proměnné $x_2, x_3, \ldots, x_{n+1}$. Hledejme nyní vrcholy v_j $(j \geq 2)$ RODD $\mathcal{D}$ reprezentující proměnné x_{j+1} (vrcholy j-tého patra $\mathcal{D}$), obsažené v diagramu $\mathcal{D}_{00}$ resp. $\mathcal{D}_{11}$, které nejsou v $\mathcal{D}_{10} = \mathcal{D}_{01}$. V j-tém patře $\mathcal{D}$ existuje nejvýše jeden takový v_j^{00} z $\mathcal{D}_{00}$ a nejvýše jeden v_j^{11} z $\mathcal{D}_{11}$, kde v_j^{00} resp. v_j^{11} je určen cestou $(\underbrace{0,\ldots,0}_{j})$ resp. $(\underbrace{1,\ldots,1}_{j})$ z kořenu RODD $\mathcal{D}$. Cesty do ostatních vrcholů z $\mathcal{D}_{00}$ a $\mathcal{D}_{11}$ jsou

permutacemi cest do vrcholů v $\mathcal{D}_{01} = \mathcal{D}_{10}$ a proto dle lematu 2.1 leží v $\mathcal{D}_{01} = \mathcal{D}_{10}$. Znamená to, že v j-tém patře $\mathcal{D}$ se nachází maximálně o 1 vrchol více než v odpovídajícím patře $\mathcal{D}_0$ nebo $\mathcal{D}_1$. Navíc v 0-tém resp. 1-ním patře $\mathcal{D}$ se vyskytují jeden resp. nejvýše 2 vrcholy.

□

2.4.2 Funkce celočíselného násobení

Ve větě 2.2 jsme ukázali, že při nevhodně zvoleném uspořádání proměnných ρ může velikost diagramu $\mathcal{D}(N, T, E, \rho)$ logické funkce f dosáhnout exponenciální složitosti. Funkce f v důkazu 2.2 byla však zvolena poněkud uměle, proto se může čtenář ptát: Existuje nějaká v praxi často používaná funkce, jejíž ROBDD má velmi vysokou paměťovou složitost? Ano, taková funkce existuje. Jde o funkci realizující celočíselné násobení.

Věta 2.4. *Uvažujme funkci $f : \{0,1\}^{2n} \to \{0,1\}^{2n}$ realizující násobení celých čísel $a, b \in \mathbb{Z}$ kódovaných binárně jako vektory $\vec{a} = (a_1, a_2, \ldots, a_n) \in \{0,1\}^n$, $\vec{b} = (b_1, b_2, \ldots, b_n) \in \{0,1\}^n$ (výsledkem je $a.b = c \in \mathbb{Z}$ kódovaný binárně $\vec{c} = (c_1, c_2, \ldots, c_{2n})$). Potom pro celkovou sumu $\sum v$ všech vrcholů ve všech sdílených ROBDD $\mathcal{D}_i$ ($0 \leq i \leq 2n$) reprezentující funkce $f_i : \{0,1\}^{2n} \to \{0,1\}$ i-tého bitu výsledku násobení ($f_i = c_i$) platí*

$$\sum v = \Omega(2^{\frac{n}{8}}).$$

Důkaz. Důkaz uvedené skutečnosti je rozsáhlejšího chrakteru. Protože s tématem práce přímo nesouvisí, nebudeme jej uvádět. Čtenář jej nalezne v [20]. □

2.5 Manipulace s ROBDD, ROMBD

Většinu symbolických operací s logickými funkcemi f lze přenést do roviny ROBDD (ROMBD) a provádět tak tyto symolické operace pomocí grafových algoritmů na ROBDD a ROMBD. Výsledkem takového algoritmu je RODD indukující logickou funkci, která je výsledkem dané symbolické operace nad f.

Mezi nejdůležitější základní operace nad ROBDD (ROMBD) $\mathcal{D}_1$ a $\mathcal{D}_2$ ($\mathcal{D}_1 \rhd f_1$ a $\mathcal{D}_2 \rhd f_2$), které uvádí [20] patří:

OPERACE	VÝZNAM	ČASOVÁ SLOŽITOST
Reduce	redukuj BDD (MBD) $\mathcal{D}_1$ na ROBDD (ROMBD)	$O(\lvert\mathcal{D}_1\rvert log_2(\lvert\mathcal{D}_1\rvert))$
Apply	vrať RODD výsledku f_1<op>f_2	$O(\lvert\mathcal{D}_1\rvert.\lvert\mathcal{D}_2\rvert)$
Restrict	vrať RODD výsledku $f_1[x_i = b]$, $b \in \{0,1\}$	$O(\lvert\mathcal{D}_1\rvert)$

Tabulka 2.1: Základní operace nad ROBDD (ROMBD)

$|\mathcal{D}|$ máme na mysli počet vrcholů $\mathcal{D}$, operace <op> je z množiny <op> $\in$ {AND,OR,XOR,NAND, NOR,NXOR, ...}. Protože budeme v naší implementaci pracovat výhradně s ROBDD a ROMBD, operace *Reduce* nás nebude příliš zajímat. Operace *Apply* a *Restrict* probereme později. Věnujme nyní pozornost datové reprezentaci diagramů.

2.5.1 Datová reprezentace

Všechny algoritmy a jiný zdrojový kód budeme uvádět v pseudokódu podobnému jazyku *C* [21] resp. *C++* [1].

Každý vrchol ROBDD (ROMBD) $\mathcal{D}(N, T, E, \rho)$ reprezentujeme strukturou (2.7) `DdNode` (význam jednotlivých položek je uveden v tabulce 2.2 a bude často ozřejměn později). Struktura `DdNode` odpovídá přesně stuktuře vrcholu v balíku CUDD.

```
struct DdNode                       struct DdChildren
{                                   {
    unsigned short index;               DdNode * T;
    unsigned short ref;                 DdNode * E;
    DdNode *       next;            };
    union
    {
        double     value;
        DdChildren kids;
    }
    type;
};
```

Zdrojový kód 2.7: Struktura vrcholu diagramu (v CUDDu)

POLOŽKA	NETERMINÁL ($v \in N$)	TERMINÁL ($v \in T$)
ref	počet referencí na vrchol (viz. dále)	
next	ukazatel na další vrchol v *unique table* (tabulka unikátních vrcholů, viz. dále)	
type	struktura typu union (proměnné struktury se kladou přes sebe $\Longrightarrow$ úspora místa)	
index	$idx(v)$	null
value	null	$val(v)$
T	ukazatel na funkci vrcholu $son(v, 1)$	null
E	ukazatel na funkci vrcholu $son(v, 0)$	null

Tabulka 2.2: Význam položek struktury vrcholu diagramu

2.5.2 Sdílené ROBDD (ROMBD) a unique table

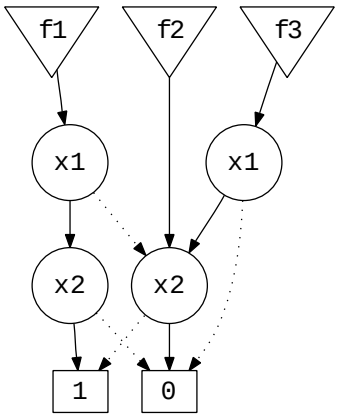
Chceme-li pro více funkcí $f_1, \ldots, f_k$ vytvořit odpovídajícími RODD $\mathcal{D}_1, \ldots \mathcal{D}_k$, pak je z hlediska úspory paměti nevýhodné, aby tyto RODD byli navzájem disjunktní. Snadno se totiž stane, že nějaké dva vrcholy z různých dvou $\mathcal{D}_i$ a $\mathcal{D}_j$ určují stejnou funkci a proto odpovídající subdiagramy jsou izomorfní. Pro každou funkci f si místo toho stačí v daném prostředí udržovat jedinečný vrchol v_f. Množina funkcí $f_1, \ldots, f_n$ je pak reprezentována jediným acyklickým grafem s více kořeny (viz. příklad 2.8).

Uvedená reprezentace si vynucuje ukládat záznamy o existujících vrcholech v_f (funkcích) v daném **prostředí** (2.5). V opačném případě bychom nebyli při vytváření nového vrcholu v_f schopni zjistit, zda v prostředí neexistuje již jiný vrchol v'_f odpovídající f. Vrcholy v_f jsou proto po vytvoření ukládány podle klíče, který je trojicí

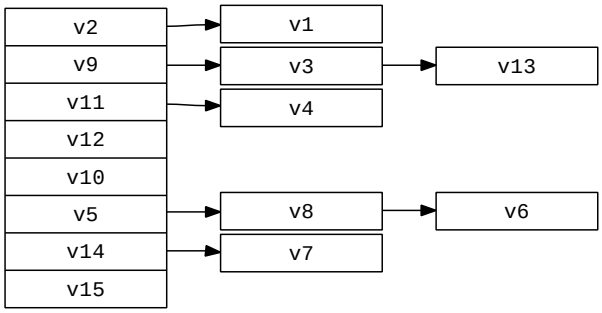
$$key(v_f) := \Big(idx(v_f),\ son(v_f, 1),\ son(v_f, 0) \Big),$$

do tabulky unikátních vrcholů, tzv. **unique table**. Unique table tedy obsahuje všechny existující vrcholy prostředí. Protože se RODD vytváří vždy „odspodu“, musí v okamžiku vzniku v_f existovat vrcholy $son(v_f, 1)$ a $son(v_f, 0)$, tj. uvedený klíč $key(v_f)$ je relevantní. Každý vrchol v_f je reprezentován ukazatelem (adresou v paměti) na jeho strukturu `DdNode`.

Poznámka 2.5 (**Prostředí**). Pojem prostředí představuje jeden separátní prostor diagramů a budeme jej reprezentovat později specifikovanou strukturou `DdManager *`. Výhoda prostředí tkví právě v možnosti pracovat izolovaně s více prostory RODD.



Obrázek 2.8: Sdílené ROBDD (ROMBD)



Obrázek 2.9: Ukázka jednoduché unique table

Tabulka unikátních vrcholů je obvykle řešena jako *hashovací tabulka* s použitím *zřetězeného hashovaní* (viz. 2.9). Důvod je jednoduchý, počet vrcholů prostředí dopředu neznáme, proto volíme velikost tabulky odhadem a v případě kolizí vrcholů používáme zřetězení. Právě parametr **next** v 2.2 obsahuje ukazatel na následující vrchol v daném řetězu unique table.

Poznámka 2.6. Ve skutečnosti je často unique table implementována jako množina tabulek typu 2.9, kde pro každou proměnnou x_i prostředí existuje právě jedna subtabulka $\mathcal{T}_i$, tedy pro všechna $v \in \mathcal{T}_i$ je $idx(v) = i$. Navíc existuje zvlášť subtabulka s terminálními vrcholy. Uvedenou implementaci používá právě balík CUDD [4].

2.5.3 Správa paměti

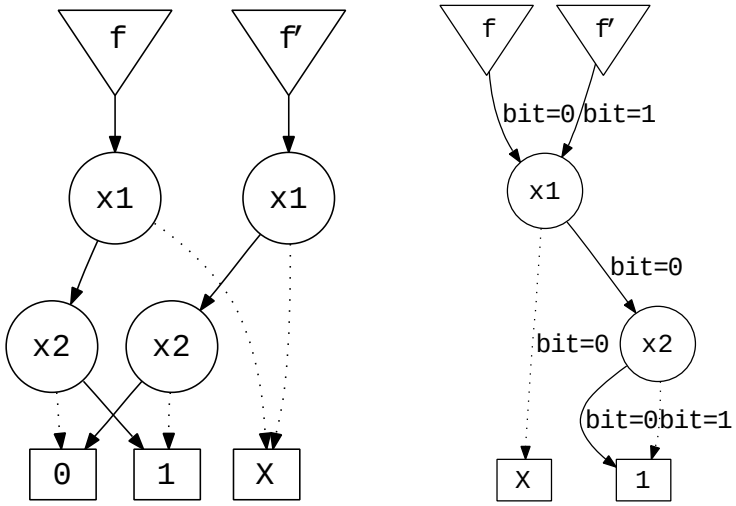
Po odstranění některých RODD (vrcholů) z prostředí se může stát, že dříve používaný vrchol v_f není obsažen v žádném RODD v prostředí. Potom je v_f v prostředí k ničemu a měl by být odstraněn (aby se nám nepoužívané vrcholy nehromadily a nezabíraly zbytečně paměť). K odstranění v_f musíme však mít jistotu, že žádný jiný vrchol se na v_f neodkazuje. Proto si u každého vrcholu udržujeme počet do něj vedoucích hran (*referencí*, parametr **ref** v 2.2). Kdykoliv je vytvořen resp. odstraněn vrchol referující na vrchol v_f, je třeba inkrementovat resp. dekrementovat hodnotu ref vrcholu v_f. V okamžiku $ref = 0$ je vrchol prohlášen za **mrtvý** (*dead*) a může být odstraněn.

Ukazuje se, že není dobré provádět odstranění mrtvých vrcholů ihned, nýbrž počkat, než počet mrtvých vrcholů dosáhne určité meze *deadLimit* dané prostředím (poté jsou všechny mrtvé vrcholy odstraněny najednou). Existuje totiž nenulová pravděpodobnost, že uživatel vytvoří opět diagram funkce f. V tako-

vém případě může být v_f prohlášen zpět za **živý** a diagram f není nutné konstruovat. Platí, že kdykoliv je v_f zpětně oživen, jeho potomci $son(v_f, 0)$ a $son(v_f, 1)$ musí existovat a obnovení v_f je proto korektní (pokud by $son(v_f, 0)$ a $son(v_f, 1)$ neexistovali, museli být odstraněny v minulém nebo dřívějším jednorázovém uvolňování paměti mrtvých vrcholů, což by znamenalo, že v okamžiku umrtvení $son(v_f, i)$ nevedla do $son(v_f, i)$ žádná hrana, to ale vzhledem k existenci v_f není pravda). Uvolňování paměti vrcholu v závislosti na počtu refrencí na vrchol říkáme **garbage collection** (*sbírání odpadu*).

2.5.4 Negované hrany

Významné paměťové úspory lze dosáhnout použitím tzv. *negovaných hran*. Tato technika je založena na faktu, že redukované ROBDD (ROMBD) funkce f se liší od ROBDD (ROMBD) funkce $\overline{f}$ pouze tím, že všechny hrany f vedoucí do terminálu s hodnotou $val(t) \in \{0, 1, X\}$ vedou v $\overline{f}$ do terminálu t' s hodnotou $val(t') = \overline{val(t)}$. Přidáním nového jednobitové příznaku každé hraně RODD docílíme značné úspory. Pokud bit není nastaven, odpovídá subdiagram funkci f, je-li bit nastaven, odpovídá subdiagram funkci $\overline{f}$. Obě funkce f a $\overline{f}$ mohou pak mohou být reprezentovány jediným RODD (obrázek 2.10).



Obrázek 2.10: ROMBD bez a s použitím techniky negovaných hran

Problémem použití techniky negace hran je, že ztrácíme kanonickou reprezentaci ROBDD a ROMBD. Platí totiž $\overline{\overline{f}} = f$, a proto se pro každou funkci f nabízejí dvě možnosti:

1. $bit = 0$ pro f a $bit = 1$ pro $\overline{f}$.
2. $bit = 0$ pro $\overline{f}$ a $bit = 1$ pro f.

Jinými slovy potřebujeme najít způsob, jakým jednoznačně prohlásit jednu z funkcí f a $\overline{f}$ za **regulární** (tu, která odpovídá $bit = 0$) a druhou za **komplementovanou** (tu, která odpovídá $bit = 1$). Na obrázku 2.11 jsou čtyři možné páry kombinací, které mohou nastat. Funkce každého páru jsou ekvivalentní, naopak každý pár určuje jinou funkci. (tučná hrana do x_i zahrnuje případy obou ohodnocení předešlé proměnné). Kanonickou reprezentaci zajistíme například tak, že hrana **then** (**T**) pro $x_i = 1$ bude u regulární funkce nenegovaná (bude určovat vždy regulární subfunkci). Na obrázku 2.11 jsou to vždy části diagramů ležící v páru vlevo. U terminálních vrcholů t budeme předpokládat, že vrcholy s hodnotou $val(t) = 1$ a $val(t) = X$ jsou regulární a vrcholy s hodnotou $val(t) = 0$ komplementované.

U neúplně určených funkcí si musíme dát pozor ještě na jednu věc. Pro neúplně určenou hodnotu X platí $\overline{X} = X$. Tzn. v případě, že hrana **else** (**E**) vede do terminálu s hodnotou X, pak ať už je hrana negována nebo ne, v obou případech určuje stejnou subfunkci $f[x_i = 0] \equiv X$ a reprezentace není kanonická

(u hrany then tento případ nastat nemůže, neboť jsme řekli, že then nebude nikdy negovaná). Kanonickou reprezentaci napravíme tak, že zaručíme, že každá hrana else vedoucí do terminálu t s hodnotou $val(t) = X$ nebude nikdy negována. Nyní již bude každá úplně i neúplně určená logická funkce reprezentována jedinečným diagramem.

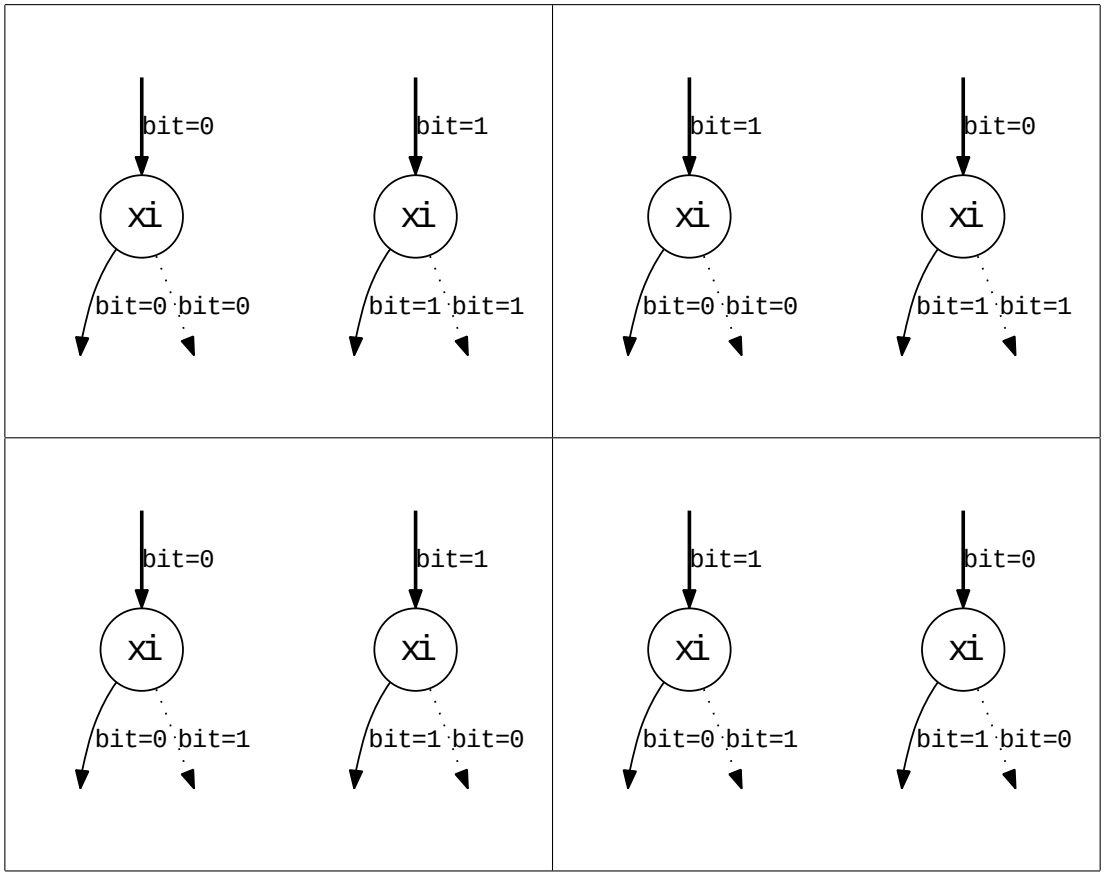
Poznámka 2.7 (**Realizace negovaných hran v C,C++**). Čtenář si jistě všiml, že ve struktuře `DdChildren` v 2.5.1 chybí příznaky *bit* pro informaci o negaci hran vedoucích do potomků **T** a **E**. V naší implementaci v C (C++) bude mít ukazatel `DdNode *` velikost v bytech, která je dělitelná dvěma a leží proto na adrese dělitelné dvěma. Jeho hodnota je tedy vždy číslo v binárním tvaru $0xb_1b_2 \dots b_{n-1}b_n$, kde $b_n = 0$ ($b_i \in \{0, 1\}$). Příznak *bit* lze tak ukrýt do hodnoty b_n ukazatele `DdNode *`, tzn. položit $bit = b_n$.

Konvence 2.1. Od této chvíle budeme v tichosti předpokládat, že negované hrany jsou v ROBDD a ROMBD použity, prostože diagramy na obrázcích budou pro lepší přehlednost bez negovaných hran. Vzniklé diagramy budeme (formálně nekorektně) nazývat stejně, tj. ROBDD a ROMBD.

2.5.5 Operace Apply a computed table

Datovou reprezentaci ROBDD (ROMBD) jsme uvedli, ale dosud nemáme prostředky pro práci s diagramy. Neumíme například vytvořit RODD logické funkce. Pro tyto a jiné potřeby zavádíme na ROBDD (ROMBD) $\mathcal{D}_1$ ($\mathcal{D}_1 \rhd f$) a $\mathcal{D}_2$ ($\mathcal{D}_2 \rhd g$) operaci $Apply(\mathcal{D}_1, \mathcal{D}_2, \texttt{<op>})$ vracející RODD $\mathcal{D}_3$ ($\mathcal{D}_3 \rhd h$) ($\mathcal{D}_1$, $\mathcal{D}_2$ i $\mathcal{D}_3$ jsou sdílené) jako výsledek jedné z operací

- logický součin: $h = f$ AND $g = fg$.



Obrázek 2.11: Případy negovaných hran

- logický součet: $h = f \text{ OR } g = f + g$.
- exkluzivní logický součet: $h = f \text{ XOR } g = f\overline{g} + \overline{f}g$.
- negaci logického součinu: $h = f \text{ NAND } g = \overline{fg}$.
- negaci logického součtu: $h = f \text{ NOR } g = \overline{f + g}$.
- negaci exkluzivního logického součtu: $h = f \text{ NXOR } g = \overline{f\overline{g} + \overline{f}g}$.

Protože používáme techniku negovaných hran a naše reprezentace je kanonická, stačí vzhledem k poznámce 2.7 nalézt algoritmus konstrukce $\mathcal{D}_3$ pro operace AND,OR a XOR pro libovolné $\mathcal{D}_1$ a $\mathcal{D}_2$. U zbylých operací dostaneme výsledek komplementováním ukazatele `DdNode *` diagramu $\mathcal{D}_3$, který vznikne korespondující nenegovanou operací. Dokonce vzhledem k platnosti De-Morganových zákonů (1.2) a 2.7 bude stačit nalézt algoritmus konstrukce $\mathcal{D}_3$ pro operace AND a XOR [1].

Vyjádříme-li f a g vzhledem k libovolné proměnné x pomocí Shanonnova expanzního teorému 1.2

$$\begin{aligned} f = \overline{x}f[x=0] + xf[x=1] &= \overline{x}f_0 + xf_1 \\ g = \overline{x}g[x=0] + xg[x=1] &= \overline{x}g_0 + xg_1, \end{aligned} \tag{2.2}$$

můžeme použitím zákonů Booleovy algery vyjádřit funkci h pro <op>=AND resp. <op>=XOR

$$\begin{aligned} h = f \text{ AND } g &= \Big(\overline{x}f_0 + xf_1\Big).\Big(\overline{x}g_0 + xg_1\Big) = \\ &= \overline{x}.\overline{x}.f_0.g_0 + \overline{x}.x.f_0.g_1 + x.\overline{x}.f_1.g_0 + x.x.f_1.g_1 = \big|\ x \in \{0,1\}\big| = \\ &= \overline{x}\Big(f_0 \text{ AND } g_0\Big) + x\Big(f_1 \text{ AND } g_1\Big). \end{aligned} \tag{2.3}$$

$$\begin{aligned} h = f \text{ XOR } g &= \Big(\overline{x}f_0 + xf_1\Big).\overline{\Big(\overline{x}g_0 + xg_1\Big)} + \overline{\Big(\overline{x}f_0 + xf_1\Big)}.\Big(\overline{x}g_0 + xg_1\Big) = \big|\ 1.2\big| = \\ &= \Big(\overline{x}f_0 + xf_1\Big).\Big(x + \overline{g_0}\Big).\Big(\overline{x} + \overline{g_1}\Big) + \Big(x + \overline{f_0}\Big).\Big(\overline{x} + \overline{f_1}\Big).\Big(\overline{x}g_0 + xg_1\Big) = \big|\ x \in \{0,1\}\big| = \\ &= \Big(\overline{x}f_0 + xf_1\Big).\Big(x.\overline{g_1} + \overline{x}.\overline{g_0} + \overline{g_0}.\overline{g_1}\Big) + \Big(\overline{x}g_0 + xg_1\Big).\Big(x.\overline{f_1} + \overline{x}.\overline{f_0} + \overline{f_0}.\overline{f_1}\Big) = \\ &= \overline{x}.f_0.\overline{g_0} + \overline{x}.f_0.\overline{g_0}.\overline{g_1} + x.f_1.\overline{g_1} + x.f_1.\overline{g_0}.\overline{g_1} + \overline{x}.g_0.\overline{f_0} + \overline{x}.g_0.\overline{f_0}.\overline{f_1} + x.g_1.\overline{f_1} + x.g_1.\overline{f_0}.\overline{f_1} = \\ &= \overline{x}\Big(f_0\overline{g_0}(1+\overline{g_1}) + \overline{f_0}g_0(1+\overline{f_1})\Big) + x\Big(f_1\overline{g_1}(1+\overline{g_0}) + \overline{f_1}g_1(1+\overline{f_0})\Big) = \\ &= \overline{x}\Big(f_0\overline{g_0} + \overline{f_0}g_0\Big) + x\Big(f_1\overline{g_1} + \overline{f_1}g_1\Big) = \\ &= \overline{x}\Big(f_0 \text{ XOR } g_0\Big) + x\Big(f_1 \text{ XOR } g_1\Big). \end{aligned} \tag{2.4}$$

Pokud x není proměnná f resp. g, pak $f_0 = f_1 = f$ resp. $g_0 = g_1 = g$. Rovnosti 2.3 platí pro libovolné úplně i neúplně určené logické funkce f, g a nabízí rekurzivní řešení problému operace *Apply* prohledáváním do hloubky. Při volání $\mathcal{D}_3 := Apply(\mathcal{D}_1, \mathcal{D}_2, \texttt{<op>})$ stačí vyřešit rekurzivně

$$\mathcal{D}_{30} := Apply(\mathcal{D}_{10}, \mathcal{D}_{20}, \texttt{<op>}) \qquad \mathcal{D}_{31} := Apply(\mathcal{D}_{11}, \mathcal{D}_{21}, \texttt{<op>})$$

pro subdiagramy $\mathcal{D}_{1i}$ a $\mathcal{D}_{2i}$ ($i \in \{0,1\}$) určených buď potomky $son_i(r_i, 0)$ a $son_i(r_i, 1)$ kořenů r_i diagramů $\mathcal{D}_i$ (reprezentuje-li r_i „větvící“ proměnnou x) nebo určených přímo diagramem $\mathcal{D}_i$. ($\mathcal{D}_{1i} := \mathcal{D}_{2i} := \mathcal{D}_i$ reprezentuje-li r_i jinou níže postavenou proměnnou). Diagramy $\mathcal{D}_{30}$ a $\mathcal{D}_{31}$ pak představují potomky $son_3(r_3, 0)$ a $son_3(r_3, 1)$ kořene r_3 výsledného diagramu $\mathcal{D}_3$. Rekurzi lze terminovat v případech:

[1] Obecně postačí pouze algoritmus pro AND, operace XOR je totiž složením dvou AND a jedné OR. Uvidíme ale, že existuje rychlejší způsob výpočetu XOR

1. <op> = AND
   (a) $f \equiv 0 \Longrightarrow h \equiv 0, \quad f \equiv 1 \Longrightarrow h = g, \quad g \equiv 0 \Longrightarrow h \equiv 0, \quad g \equiv 1 \Longrightarrow h = f$ (ROBDD i ROMBD).
   (b) $f \equiv X$ a zároveň $g \equiv X \Longrightarrow h \equiv X$ (ROMBD).
   (c) $f = g \Longrightarrow h = f$ (ROBDD i ROMBD).
   (d) $f = \overline{g} \Longrightarrow h \equiv 0$ (ROBDD).
2. <op> = XOR
   (a) $f \equiv 0 \Longrightarrow h = g, \quad f \equiv 1 \Longrightarrow h = \overline{g}, \quad g \equiv 0 \Longrightarrow h = f, \quad g \equiv 1 \Longrightarrow h = \overline{f}$ (ROBDD i ROMBD).
   (b) $f \equiv X \Longrightarrow h = X(g + \overline{g}) \equiv X, \quad g \equiv X \Longrightarrow h = X(f + \overline{f}) \equiv X$ (ROMBD).
   (c) $f = g \Longrightarrow h \equiv 0$ (ROBDD).
   (d) $f = \overline{g} \Longrightarrow h = f + \overline{f} \equiv 1$ (ROBDD).

Je vidět, že jsou-li $\mathcal{D}_1$ a $\mathcal{D}_2$ redukované, pak všechny terminální případy vrací opět RODD, tj. $\mathcal{D}_3$ bude opět redukovaný. Pro neterminální případy pokud náhodou $\mathcal{D}_{30} \cong \mathcal{D}_{31}$ (oba ukazatele `DdNode *` mají stejnou hodnotu), pak by měl vzniklý $\mathcal{D}_3$ zbytečný kořen, proto položíme $\mathcal{D}_3 := \mathcal{D}_{30}$. Matematickou indukcí nyní snadno vyplývá, že každý $\mathcal{D}_3$ bude vždy redukovaný.

Na obrázku 2.12 je zobrazen graf výpočtu *Apply* s operací <op>=AND dvou ROMBD. Obecně se často stává (ne v našem případě), že pro dané subdiagramy funkcí f a g se volá neterminující *Apply* vícekrát (nějakaký vrchol $A_i.B_j$ je ve stromu výpočtu <op> vícekrát). Je proto výhodné udržovat *hashovací tabulku výpočtu* **computed table**, která uchovává dříve spočtené vrcholy. Funguje na principu *cache-hash* tabulky, tzn. že uchovává pouze podmnožinu výsledků (např. podle pravidla LRU - Least recently used, málo používané výsledky jsou při nedostatku paměti přepsány nejdříve). Klíče do tabulky jsou trojice, kterou tvoří operace a její dva argumenty (subdiagramy)

$$key(v_h) = (\texttt{<op>}, v_f, v_g).$$

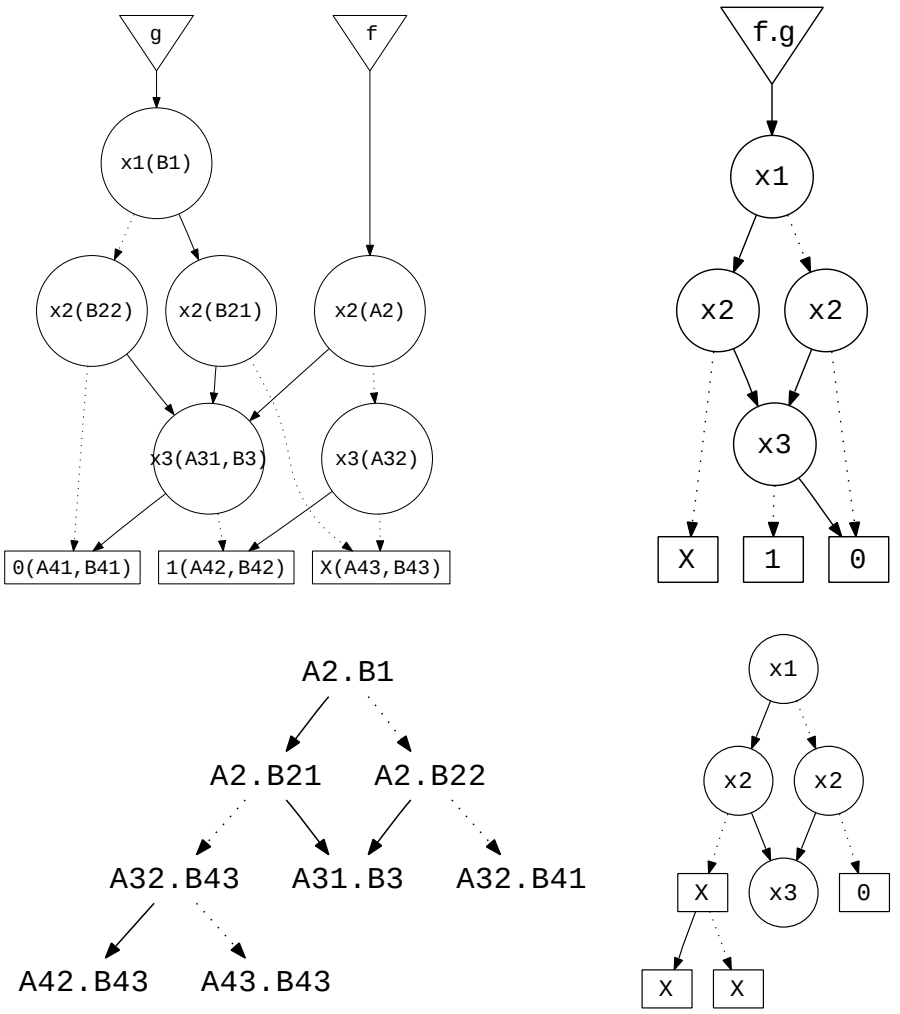
Na pozici dané klíčem $key(v_h)$ uchováváme do computed table vzniklý výsledek *Apply*, vrchol v_h. Kdykoliv tedy než se do *Apply* rekurzivně zanoříme, prozkoumáme, zda se výsledek volaného *Apply* nenachází v computed table. Pokud ano, vrátíme jej bez volání *Apply*. V ideálním případě dostatečně veliké computed table počítáme *Apply* dvou subdiagramů pouze jednou (poprvé, později vždy najdeme výsledek v computed table), tzn. že složitost operace $Apply(\mathcal{D}_1, \mathcal{D}_2, \texttt{<op>})$ je $O(|\mathcal{D}_1|.|\mathcal{D}_2|)$.

V A.1 je uvedena funkce `ApplyAnd()` implementující *Apply* pro operaci <op>=AND. `ApplyXor()` implementující *Apply* pro <op>=XOR je analogií `ApplyAnd()` (rozdíl je pouze v terminálních případech rekurze), proto ji uvádět nebudeme.

Poznámka 2.8 (**Vytváření ROBDD a ROMBD**). Pomocí operace *Apply* můžeme konstruovat ROBDD (ROMBD) $\mathcal{D}$ libovolné logické funkce f. Je-li f zadána bool. výrazem V, pak strom operací *Apply* vytvářejících $\mathcal{D}$ bude přesně kopírovat strom vyhodnocení V, s tím rozdílem, že místo hodnot podvýrazů budeme předávat subdiagramy RODD, v případě proměnných x_i budeme předávat *Apply* diagramy $\mathcal{D}_{x_i}$ proměnných x_i funkce f ($\mathcal{D}_{x_i}$ je tvořen jedním neterminálem odpovídajícímu x_i, z nějž vede then resp. else hrana do terminálů s hodnotami 1 resp. 0).

Poznámka 2.9 (**Jednoznačnost klíče do computed table operace Apply**). Z komutativity logického součinu a podkapitoly o negovaných hranách víme, že pro výpočet $Apply(\texttt{<op>}, v_f, v_g)$ funkce $h = f.g$ existuje více možností, jak se zeptat computed table na v_h, a to:

- $key(v_h) = (\texttt{<op>}, v_f, v_g)$.
- $key(v_h) = (\texttt{<op>}, v_g, v_f)$.
- $key(\overline{v_h}) = (\texttt{<op>}, \overline{v_g}, \overline{v_f})$ a následné komplementace získananého $\overline{v_h}$.



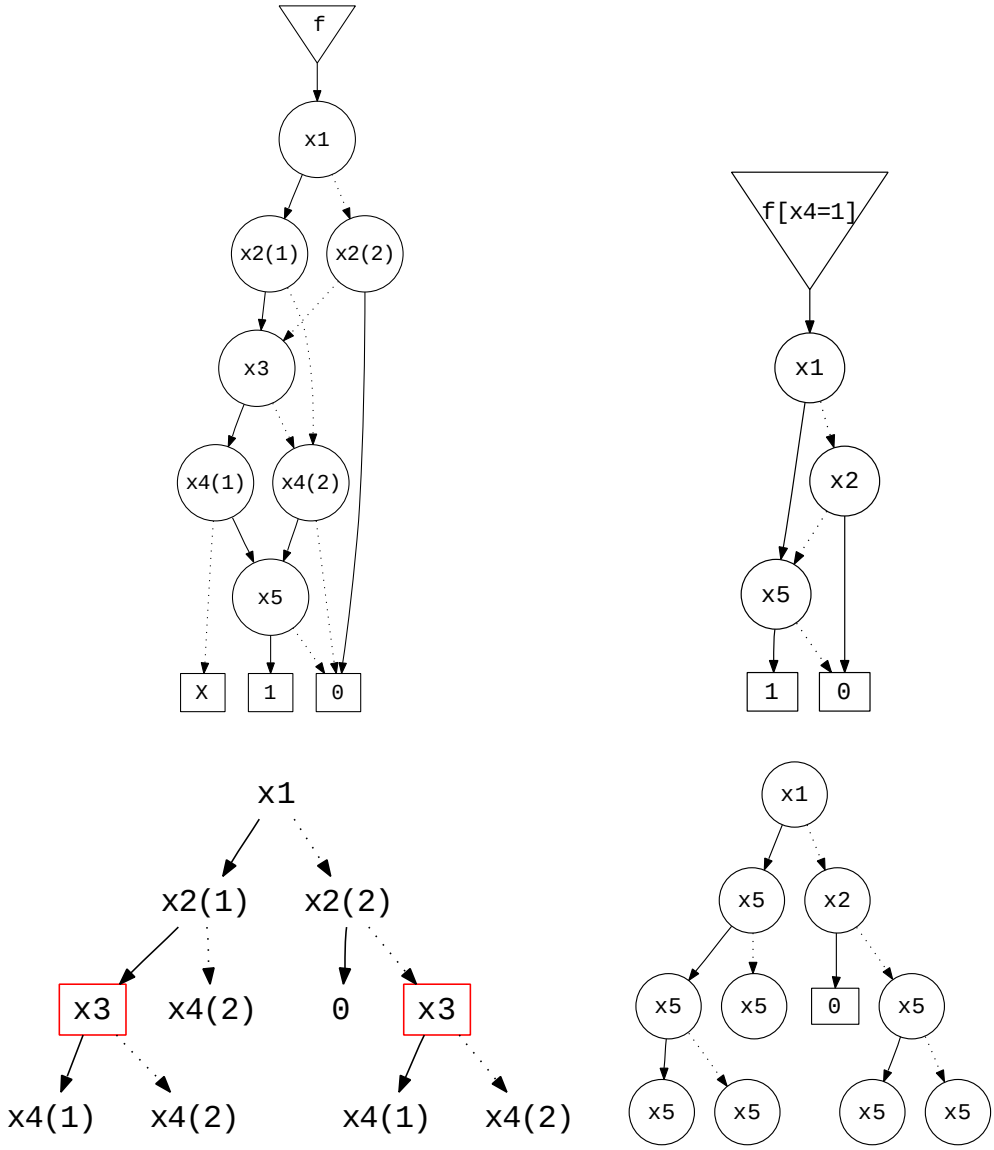
Obrázek 2.12: Výpočtet *Apply* operace `<op>`= AND ROMBD funkcí f a g

- $key(\overline{v_h}) = (\texttt{<op>}, \overline{v_f}, \overline{v_g})$ a následné komplementace získananého $\overline{v_h}$.

Pokud bychom tedy nedbali na to, jakou z trojic do computed table uložíme, museli bychom při následném hledání v_h provádět test na existenci všech čtyřech trojic v computed table. Výhodné proto bude zavést kanonickou reprezentaci trojice, takovou, že do computed table uložíme tu trojici, pro níž je v_f regulární a hodnota `DdNode *` $v_f <$ `DdNode *` v_g (pokud $f \equiv X$ nebo $g \equiv X$, pak uvažujeme do trojice vždy regulární tvar v_f či v_g). Uvědomte si, že místo uvedených čtyř trojic pak stačí provádět test na existenci pouze jedné trojice v computed table.

2.5.6 Operace Restrict

Známe-li hodnoty některých vstupních proměnných logické funkce f ROBDD (ROMBD) $\mathcal{D}$, můžeme jejich hodnoty fixovat a získat tak diagram restrikce f. Cílem operace *Restrict*$(\mathcal{D}, i, b_i)$ je pro ROBDD (ROMBD) $\mathcal{D}$ logické funkce f ($\mathcal{D} \rhd f$) vrátit RODD $\mathcal{D}'$ funkce $f' := f[x_i = b_i]$ ($b_i \in \{0, 1\}$) ($\mathcal{D}' \rhd f'$). Vícenásobným opakováním operace *Restrict* pak můžeme získat restrikci f podle více proměnných. Operace *Restrict* lze implementovat rekurzí (prohledáváním do hloubky) podobně jako *Apply*. Algoritmus řešení A.2 (se složitostí $|\mathcal{D}|$) uvedený v příloze textu je založen na platnosti následujících identit:



Obrázek 2.13: Výpočtet *Restrict* ROMBD funkce f podle $x_4 = 1$

$$
\begin{aligned}
f[x_i = b_i] &= f \quad \text{pokud } f \text{ nezávisí na } x_i. \\
f[x_i = b_i] &= \left(\overline{x_j} f_{\overline{x_j}} + x_j f_{x_j}\right)[x_i = b_i] = \\
&= \left(\overline{x_j} f_{\overline{x_j}}\right)[x_i = b_i] + \left(x_j f_{x_j}\right)[x_i = b_i] = \\
&= \left(\overline{x_j}[x_i = b_i]\right).\left(f_{\overline{x_j}}[x_i = b_i]\right) + \left(x_j[x_i = b_i]\right).\left(f_{x_j}[x_i = b_i]\right) = \\
&= \overline{x_j}\left(f_{\overline{x_j}}[x_i = b_i]\right) + x_j.\left(f_{x_j}[x_i = b_i]\right) \quad \text{pro } i \neq j. \qquad (2.5)
\end{aligned}
$$

Na obrázku 2.13 je ukázka výpočtu *Restrict* ROMBD $\mathcal{D}$ neúplně určené funkce f. V levém dolním podobrázku, který představuje strom výpočtu, je vidět, že pro červeně orámovaný vrchol se volá *Restrict* 2x. Obecně se *Restrict* volá vícekrát pro všechny vrcholy v RODD $\mathcal{D}$, které odpovídají libovolné proměnné x_j, která je v prostředí výše postavena než proměnná x_i (proměnná restrikce) a do nichž vede více hran z $\mathcal{D}$ (*Restrict* se volá tolikrát, kolik hran z $\mathcal{D}$ do v vede). Je tedy stejně jako u *Apply* výhodné využít opět **computed table** částečných výpočtů (tentokráte operace *Restrict*).

2.5.7 Operátor ITE

Aby se mohlo se všemi nulárními, unárními a binárními operacemi nad množinou logických funkcí mohlo pracovat nějakým sjednoceným způsobem, vymysleli Brace, Rudell a Bryant ternární operátor ITE definovaný pro logické funkce f, g a h:

$$ITE(f,g,h) = f.g + \overline{f}.h. \tag{2.6}$$

V tabulce 2.3 je naznačeno, jak lze možné operace pomocí ITE realizovat. Operátor ITE byl navržen pro úplně určené funkce, pro neúplně určené funkce existují operace, které nemají ITE ekvivalent (značeny -). Je-li x_i nejvýše postavená proměnná RODD funkcí f, g, h, pak v analogii s 2.5 můžeme pro operátor ITE psát:

$$\begin{aligned}
ITE(f,g,h) &= f.g + \overline{f}.h = \overline{x_i}\Big(f.g + \overline{f}.h\Big)_{\overline{x_i}} + x_i\Big(f.g + \overline{f}.h\Big)_{x_i} = \\
&= \overline{x_i}\Big((f.g)_{\overline{x_i}} + (\overline{f}.h)_{\overline{x_i}}\Big) + x_i\Big((f.g)_{x_i} + (\overline{f}.h)_{x_i}\Big) = \\
&= \overline{x_i}\Big(f_{\overline{x_i}}.g_{\overline{x_i}} + \overline{f_{\overline{x_i}}}.h_{\overline{x_i}}\Big) + x_i\Big(f_{x_i}.g_{x_i} + \overline{f_{x_i}}.h_{x_i}\Big) = \\
&= ITE\Big(x_i,\ ITE(f_{x_i}, g_{x_i}, h_{x_i}),\ ITE(f_{\overline{x_i}}, g_{\overline{x_i}}, h_{\overline{x_i}})\Big).
\end{aligned} \tag{2.7}$$

KÓD OP.	VÝZNAM	OPERACE	ITE v $\mathcal{B}_{log}$	ITE v $\mathcal{B}_{xlog}$
0000	0	0	0	0
0001	f AND g	$f.g$	$ITE(f,g,0)$	$ITE(f,g,0)$
0010	$f > g$	$f.\overline{g}$	$ITE(f,\overline{g},0)$	$ITE(f,\overline{g},0)$
0011	f	f	f	f
0100	$f < g$	$\overline{f}.g$	$ITE(f,0,g)$	$ITE(f,0,g)$
0101	g	g	g	g
0110	f XOR g	$f \oplus g$	$ITE(f,0,g)$	$ITE(f,0,g)$
0111	f OR g	$\overline{f+g}$	$ITE(f,1,g)$	-
1000	f NOR g	$\overline{f+g}$	$ITE(f,0,g)$	$ITE(f,0,g)$
1001	f XNOR g	$\overline{f \oplus g}$	$ITE(f,\overline{g},g)$	$ITE(f,\overline{g},g)$
1010	$\overline{g}$	$\overline{g}$	$ITE(g,0,1)$	$ITE(g,0,1)$
1011	$f \geq g$	$f + \overline{g}$	$ITE(f,1,\overline{g})$	-
1100	$\overline{f}$	$\overline{f}$	$ITE(f,0,1)$	$ITE(f,0,1)$
1101	$f \leq g$	$\overline{f} + g$	$ITE(f,g,1)$	-
1110	f NAND g	$\overline{f.g}$	$ITE(f,\overline{g},1)$	-
1111	1	1	1	1

Tabulka 2.3: Vyjádření operací nad logickými funkcemi pomocí ITE operátoru

Úpravami jsme získali návod na rekurzivní výpočet hodnoty ITE. Výsledkem $ITE(f,g,h)$ vznikne RODD, jehož kořen odpovídá proměnné x_i a potomci rekurzivnímu řešení ITE na odpovídajících subdiagramech, tj. potomek then $ITE(f_{x_i}, g_{x_i}, h_{x_i})$ a potomek else $ITE(f_{\overline{x_i}}, g_{\overline{x_i}}, h_{\overline{x_i}})$. Časovou složitost ITE můžeme při použití ideální computed table triviálně odhadnout jako $O(|\mathcal{D}_1|.|\mathcal{D}_2|.|\mathcal{D}_3|)$, kde RODD $\mathcal{D}_1 \rhd f$, $\mathcal{D}_2 \rhd g$ a $\mathcal{D}_3 \rhd h$. Experimentálně bylo změřeno, že časová složitost výpočtu ITE je průměrně nepatrně vyšší než složitost výpočtu *Apply*. Obecně je výhodnější provést jednou *Apply* než jednou ITE a jednou ITE než dvakrát *Apply*. Rekurzi výpočtu $ITE(f,g,h)$ lze terminovat nebo zjednodušit na jednu operaci `ApplyAnd()` v případech (test na ukončení rekurze se provádí v pořadí v jakém jsou zapsány, některé případy zahrnují totiž případy jiné neuvedené):

1. $f \equiv 0 \Longrightarrow ITE(f,g,h) = h$ (ROBDD, ROMBD).
2. $f \equiv 1 \Longrightarrow ITE(f,g,h) = g$ (ROBDD, ROMBD).
3. $f \equiv X$ a zároveň $g = h \Longrightarrow ITE(f,g,h) = X(g + \overline{g}) = X$ (ROMBD).
4. $f \equiv X$ a zároveň $g = \overline{h} \Longrightarrow ITE(f,g,h) = X(g + g) = X.g$ (ROMBD).
5. ($g \equiv 1$ nebo $f = g$) a zároveň $h \equiv 1 \Longrightarrow ITE(f,g,h) = f + \overline{f} \equiv 1$ (ROBDD).
6. ($g \equiv 1$ nebo $f = g$) a zároveň $h \equiv 1 \Longrightarrow ITE(f,g,h) = f + \overline{f} = \overline{\overline{f}.f}$ (ROMBD).
7. ($g \equiv 1$ nebo $f = g$) a zároveň $h \equiv 0 \Longrightarrow ITE(f,g,h) = f$ (ROBDD, ROMBD).
8. $g \equiv 1$ nebo $f = g \Longrightarrow ITE(f,g,h) = f + h = \overline{\overline{f}.\overline{h}}$ (ROBDD).
9. ($g \equiv 0$ nebo $f = \overline{g}$) a zároveň $h \equiv 1 \Longrightarrow ITE(f,g,h) = \overline{f}$ (ROBDD).
10. $g \equiv 0$ a zároveň $h \equiv 1 \Longrightarrow ITE(f,g,h) = \overline{f}$ (ROMBD).
11. $g \equiv 0$ nebo $f = \overline{g} \Longrightarrow ITE(f,g,h) = \overline{f}.h$ (ROBDD).
12. $g \equiv 0 \Longrightarrow ITE(f,g,h) = \overline{f}.h$ (ROBDD, ROMBD).
13. $h \equiv 0$ nebo $f = h \Longrightarrow ITE(f,g,h) = f.g$ (ROBDD).
14. $h \equiv 0 \Longrightarrow ITE(f,g,h) = f.g$ (ROMBD).
15. $h \equiv 1$ nebo $f = \overline{h} \Longrightarrow ITE(f,g,h) = f.g + \overline{f} = \overline{f} + g = \overline{f.\overline{g}}$ (ROBDD).
16. $g = h \Longrightarrow ITE(f,g,h) = (f + \overline{f})g = g$ (ROBDD).
17. $g = h$ a zároveň $g \equiv X \Longrightarrow ITE(f,g,h) = (f + \overline{f})X = X$ (ROMBD).
18. $g = \overline{h}$ a zároveň $g = X \Longrightarrow ITE(f,g,h) = (f + \overline{f})X = X$ (ROMBD).
19. $g = \overline{h} \Longrightarrow ITE(f,g,h) = f.\overline{h} + \overline{f}.h = f \oplus h$ (ROBDD).

Operace ITE je jinak analogiií operace `ApplyAnd()` (s jediným rozdílem, že nejvýše postavená proměnná se dle 2.7 hledá ve všech třech funkcích f, g a h), a proto zdrojový kód ITE uvádět nebudeme (implementace je např. v souboru `Cudd\cuddBddIte.c` námi modifikovaného balíku CUDD [4]).

Poznámka 2.10 (**Jednoznačnost klíče do computed table operace ITE**). Stejně jako u operace *Apply* i pro ITE existuje několik různých, výsledkově ekvivalentních ITE trojic a je proto z důvodu hledání v computed table (v neterminujících případech pro urychlení výpočtu) nutné nalézt jejich pokud možno co nejjednoznačnější reprezentaci, kterou budeme při ukládání a výberu v computed table respektovat. Platí

1. $ITE(f,g,h) = f.g + \overline{f}.h = ITE(\overline{f},h,g)$ (ROBDD,ROMBD).

2. $\overline{ITE(f,g,\overline{h})} = \overline{f.g + \overline{f}.\overline{h}} = \big(\overline{f.g}\big).\big(\overline{\overline{f}.\overline{h}}\big) = \big(\overline{f} + \overline{g}\big).\big(f + h\big) = \overline{f}.f + \overline{f}.h + \overline{g}.f + \overline{g}.h = \big|\ \mathcal{B}_{log}\big| =$
$= f.\overline{g} + \overline{g}.h = ITE(f,\overline{g},h)$ (ROBDD).

Můžeme proto pro ROBDD i ROMBD vždy zaručit, aby vrchol v_f funkce f byl v regulárním tvaru. V případě ROBDD může být zároveň i vrchol v_g předán v regulárním tvaru. Takto bude vypadat naše kanonická reprezentace trojice klíče do computed table. Obdobně pro ROMBD pokud $f \equiv X$, $g \equiv X$ nebo $h \equiv X$, pak předáváme v_f, v_g nebo v_h v regulárním tvaru.

2.5.8 Reduce a další operace

V případě operací *Apply*, *Restrict* i ITE vznikne vždy z jednoho nebo více ROBDD (ROMBD) opět redukovaný diagram (ROBDD či ROMBD). Vytváření diagramů provádíme taktéž operací *Apply* z RODD funkcí proměnných x_i prostředí nebo RODD konstant $0, 1, X$. Nikdy proto při používání *Apply*, *Restrict* a ITE nedostaneme neredukovaný diagram. V jiné implementaci může být operace *Reduce*, která z BDD (MBD) vytvoří funkčně ekvivalentní ROBDD (ROMBD) potřebná. Algoritmus *Reduce* uvádět nebudeme, čtenář jej nalezne v [20]. Podotkněme jen, že algoritmus vyžaduje pro svůj chod na $\mathcal{D}$ $O\big(|\mathcal{D}|\big)$ paměti a má časovou složitost $O\big(|\mathcal{D}|.log_2(|\mathcal{D}|)\big)$.

Na ROBDD a ROMBD lze zavést a pracovat s dalšími, někdy potřebnými operacemi, které se této práce netýkají (více v [20]). Jsou to například:

1. **Satisfy-One**. Vrací pro logickou funkci f (definovanou na $\{0,1\}^n$) hodnotu true resp. false (případně i implikant) pokud f je resp. není *splnitelná* na svém definičním oboru (zjišťuje, zda existuje $\vec{x} \in \{0,1\}^n$, kde $f(\vec{x}) = 1$). Pro ROBDD (ROMBD) je f splnitelná, právě tehdy když ROBDD (ROMBD) f neopdovídá konstantě 0.

2. **Satisfy-All**. Vrací množinu S_f všech bodů definičního oboru f, v nichž je f splněná, tj. $S_f = \big\{\vec{x} \in \{0,1\}^n \big|\ f(\vec{x}) = 1\big\}$.

3. **Satisfy-Count**. Vrací velikost $|S_f|$ právě popasné množiny S_f.

4. **Largest-Cube**. Vrací „největší“ přímý implikant funkce f (implikant obsahující nejmenší počet termů).

2.5.9 Reprezentace neúplně určené logické funkce pomocí dvou ROBDD

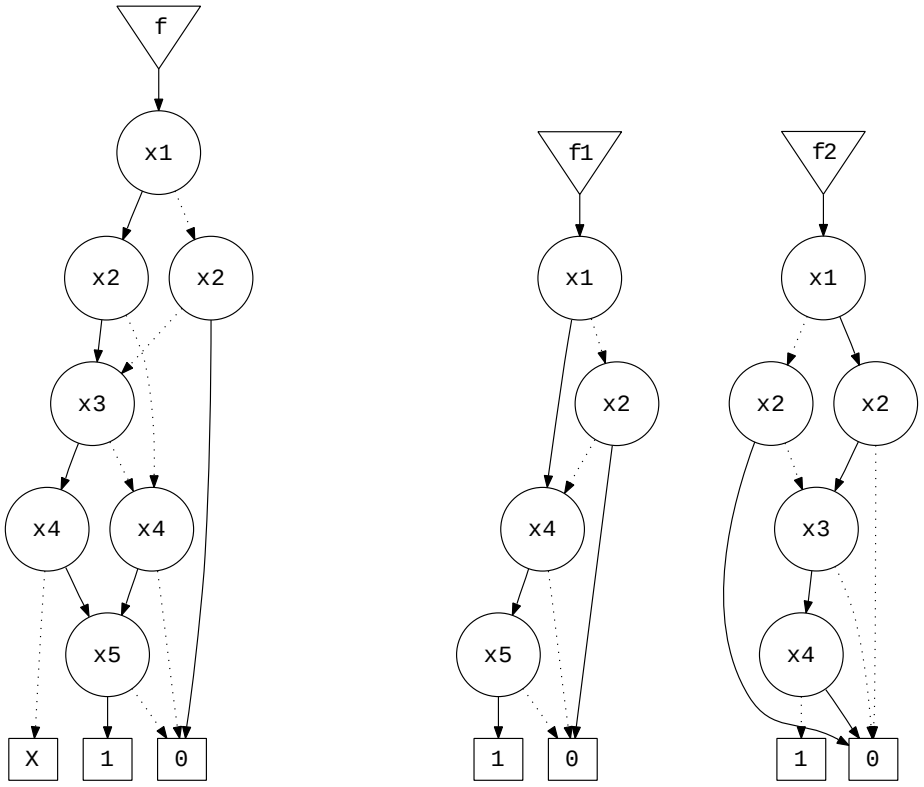
V 2.7 jsme zavedli pojem MBD a ROMBD jako prostředek pro diagramové vyjádření neúplně určených logických funkcí. Jinou možností je vyjádřit neúplně určenou logickou funkci $f : \{0,1\}^n \to \{0,1,X\}$ pomocí dvojice úplně určených logických funkcí $f_1 : \{0,1\}^n \to \{0,1\}$, $f_2 : \{0,1\}^n \to \{0,1\}$ určujících BDD (ROBDD) $\mathcal{D}_1$ a $\mathcal{D}_2$. Funkce f_1 a f_2 můžeme popsat pomocí dvojice bitových vektorů

$$\left\langle \vec{b_1}, \vec{b_2} \right\rangle = \left\langle (b_{10}, b_{11}, b_{1X}), (b_{20}, b_{21}, b_{2X}) \right\rangle,$$

kde pro $\vec{b_1} \in \{0,1\}^3$ resp. $\vec{b_2} \in \{0,1\}^3$ a libovolný $\vec{x} \in \{0,1\}^n$

$$f_1(\vec{x}) = \begin{cases} b_{10} & \text{pro } f(\vec{x}) = 0 \\ b_{11} & \text{pro } f(\vec{x}) = 1 \\ b_{1X} & \text{pro } f(\vec{x}) = X \end{cases} \qquad f_2(\vec{x}) = \begin{cases} b_{20} & \text{pro } f(\vec{x}) = 0 \\ b_{21} & \text{pro } f(\vec{x}) = 1 \\ b_{2X} & \text{pro } f(\vec{x}) = X. \end{cases}$$

Na obrázku 2.14 je příklad reprezentace neúplně určené funkce f pomocí dvojice úplně určených funkcí f_1 a f_2, které jsou specifikovány vektory $b_1 = (0,1,0)$ a $b_2 = (0,0,1)$ (ROBDD f_1 resp. f_2 získáme



Obrázek 2.14: Reprezentace ROMBD funkce f dvěma ROBDD funkcí f_1, f_2 určených $b_1 = (0,1,0)$ a $b_2 = (0,0,1)$

nahrazením terminálu X terminálem 0 resp. terminálu 1 terminálem 0 a terminálu X terminálem 1 a následnou redukcí vzniklých BDD).

Zkoumejme nyní, kolik reprezentativních dvojic $\langle \vec{b_1}, \vec{b_2} \rangle$ z hlediska různé paměťové složitosti uložení obou ROBDD $\mathcal{D}_1$, $\mathcal{D}_2$ funkcí f_1, f_2 existuje a jaké to jsou. Každý z vektorů $b_1, b_2 \in \{0,1\}^3$ můžeme reprezentovat číslem z množiny $\{0,1,2,\ldots,7\}$, kde binární hodnota čísla odpovídá právě tomuto vektoru. Tabulka 2.4 ukazuje všechny možné dvojice vektorů b_1, b_2, které mohou nastat. V prvním přiblížení jsou možnými kandidáty pouze zarámované dvojice, neboť

$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 0,2\rangle$	$\langle 0,3\rangle$	$\langle 0,4\rangle$	$\langle 0,5\rangle$	$\langle 0,6\rangle$	$\langle 0,7\rangle$
$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,2\rangle$	$\langle 1,3\rangle$	$\langle 1,4\rangle$	$\langle 1,5\rangle$	$\langle 1,6\rangle$	$\langle 1,7\rangle$
$\langle 2,0\rangle$	$\langle 2,1\rangle$	$\langle 2,2\rangle$	$\langle 2,3\rangle$	$\langle 2,4\rangle$	$\langle 2,5\rangle$	$\langle 2,6\rangle$	$\langle 2,7\rangle$
$\langle 3,0\rangle$	$\langle 3,1\rangle$	$\langle 3,2\rangle$	$\langle 3,3\rangle$	$\langle 3,4\rangle$	$\langle 3,5\rangle$	$\langle 3,6\rangle$	$\langle 3,7\rangle$
$\langle 4,0\rangle$	$\langle 4,1\rangle$	$\langle 4,2\rangle$	$\langle 4,3\rangle$	$\langle 4,4\rangle$	$\langle 4,5\rangle$	$\langle 4,6\rangle$	$\langle 4,7\rangle$
$\langle 5,0\rangle$	$\langle 5,1\rangle$	$\langle 5,2\rangle$	$\langle 5,3\rangle$	$\langle 5,4\rangle$	$\langle 5,5\rangle$	$\langle 5,6\rangle$	$\langle 5,7\rangle$
$\langle 6,0\rangle$	$\langle 6,1\rangle$	$\langle 6,2\rangle$	$\langle 6,3\rangle$	$\langle 6,4\rangle$	$\langle 6,5\rangle$	$\langle 6,6\rangle$	$\langle 6,7\rangle$
$\langle 7,0\rangle$	$\langle 7,1\rangle$	$\langle 7,2\rangle$	$\langle 7,3\rangle$	$\langle 7,4\rangle$	$\langle 7,5\rangle$	$\langle 7,6\rangle$	$\langle 7,7\rangle$

Tabulka 2.4: Číselné reprezentace dvojic úplně učených logických funkcí kódující neúplně určenou logickou funkci

- Kandidáti v horním řádku resp. pravém sloupci 2.4 odpadají, protože pro ně je $\vec{b_1} = (0,0,0)$ resp.

$\vec{b_2} = (0,0,0)$, tj. $f_1 \equiv 0$ resp. $f_2 \equiv 0$, tedy f_1 resp. f_2 neposkytuje o f žádnou informaci a f tak není jednoznačně určena.

- Kandidáti v dolním řádku resp. levém sloupci 2.4 odpadají, protože pro ně je $\vec{b_1} = (1,1,1)$ resp. $\vec{b_2} = (1,1,1)$, tj. $f_1 \equiv 1$ resp. $f_2 \equiv 1$, tedy f_1 resp. f_2 neposkytuje o f žádnou informaci a f tak není jednoznačně určena.
- Kandidáti na hlavní diagonále 2.4 odpadají, protože pro ně je $\vec{b_1} = \vec{b_2}$, tj. $f_1 = f_2$, tedy f_1 a f_2 poskytuje o f stejnou informaci a f tak není jednoznačně určena.
- Pro každého zbývajícího nezarámovaného kandidáta 2.4 existuje zarámovaný kandidát, který se liší od nezarámovaného pouze pořadím čísel, tj. f_1 a f_2 se u obou kandidátů liší pouze pořadím.

V druhém přiblížení můžeme ze zbylých (zarámovaných) kandidátu opustit kandidáty $\langle 1,6\rangle$, $\langle 2,5\rangle$ a $\langle 3,4\rangle$, pro něž platí $\vec{b_1} = (b_{10}, b_{11}, b_{1X}) = (\overline{b_{20}}, \overline{b_{21}}, \overline{b_{2X}}) = \overline{\vec{b_2}}$, tj. $f_1 = \overline{f_2}$, a f tak není jednoznačně určena.

Vlastností techniky negovaných hran (používáme v naší implementaci) je, že každé dvě úplně i neúplně určené funkce g a $\overline{g}$, jsou reprezentovány dvěma RODD danými stejným kořenem v_g prostředí. Oba RODD tak představují totožné (izomorfní) grafy. Proto ve smyslu nároků na paměť jsou následující zbývající kandidáti vždy po čtyřech ekvivalentní (liší se pouze pořadím f_1 a f_2 a případně negovaností):

1. $\langle 1,2\rangle$, $\langle 1,5\rangle$, $\langle 5,6\rangle$, $\langle 2,6\rangle$ protože
   (a) $\vec{b_1}^{\langle 1,2\rangle} = \vec{b_1}^{\langle 1,5\rangle} = \overline{\vec{b_2}^{\langle 5,6\rangle}} = \overline{\vec{b_2}^{\langle 2,6\rangle}}$, tedy $f_1^{\langle 1,2\rangle} = f_1^{\langle 1,5\rangle} = \overline{f_2^{\langle 5,6\rangle}} = \overline{f_2^{\langle 2,6\rangle}}$.
   (b) $\vec{b_2}^{\langle 1,2\rangle} = \overline{\vec{b_2}^{\langle 1,5\rangle}} = \overline{\vec{b_1}^{\langle 5,6\rangle}} = \vec{b_1}^{\langle 2,6\rangle}$, tedy $f_2^{\langle 1,2\rangle} = \overline{f_2^{\langle 1,5\rangle}} = \overline{f_1^{\langle 5,6\rangle}} = f_1^{\langle 2,6\rangle}$.
2. $\langle 1,3\rangle$, $\langle 1,4\rangle$, $\langle 4,6\rangle$, $\langle 3,6\rangle$ protože
   (a) $\vec{b_1}^{\langle 1,3\rangle} = \vec{b_1}^{\langle 1,4\rangle} = \overline{\vec{b_2}^{\langle 4,6\rangle}} = \overline{\vec{b_2}^{\langle 3,6\rangle}}$, tedy $f_1^{\langle 1,3\rangle} = f_1^{\langle 1,4\rangle} = \overline{f_2^{\langle 4,6\rangle}} = \overline{f_2^{\langle 3,6\rangle}}$.
   (b) $\vec{b_2}^{\langle 1,3\rangle} = \overline{\vec{b_2}^{\langle 1,4\rangle}} = \overline{\vec{b_1}^{\langle 4,6\rangle}} = \vec{b_1}^{\langle 3,6\rangle}$, tedy $f_2^{\langle 1,3\rangle} = \overline{f_2^{\langle 1,4\rangle}} = \overline{f_1^{\langle 4,6\rangle}} = f_1^{\langle 3,6\rangle}$.
3. $\langle 2,3\rangle$, $\langle 2,4\rangle$, $\langle 4,5\rangle$, $\langle 3,5\rangle$ protože
   (a) $\vec{b_1}^{\langle 2,3\rangle} = \vec{b_1}^{\langle 2,4\rangle} = \overline{\vec{b_2}^{\langle 4,5\rangle}} = \overline{\vec{b_2}^{\langle 3,5\rangle}}$, tedy $f_1^{\langle 2,3\rangle} = f_1^{\langle 2,4\rangle} = \overline{f_2^{\langle 4,5\rangle}} = \overline{f_2^{\langle 3,5\rangle}}$.
   (b) $\vec{b_2}^{\langle 2,3\rangle} = \overline{\vec{b_2}^{\langle 2,4\rangle}} = \overline{\vec{b_1}^{\langle 4,5\rangle}} = \vec{b_1}^{\langle 3,5\rangle}$, tedy $f_2^{\langle 2,3\rangle} = \overline{f_2^{\langle 2,4\rangle}} = \overline{f_1^{\langle 4,5\rangle}} = f_1^{\langle 3,5\rangle}$.

Díky De Morganovým zákonům 1.2 musím i výpočet operací NOT f, f AND g, f OR g, f XOR g, ... (f a g jsou neúplně určené funkce, specifikované dvojicemi úplně určených funkcí $f = \langle f_1, f_2\rangle$ a $g = \langle g_1, g_2\rangle$) být pro různá dvě určení f_1, g_1, f_2, g_2 dvěma ekvivalentními kandidáty z jedné čtveřice (viz. výše) stejně časově náročný. Opravdu, pokud například pro $\langle 1,2\rangle \simeq \left\langle (0,0,1),(0,1,0)\right\rangle$ je f AND $g = \langle f_1.g_1, f_2.g_2\rangle$, pak pro $\langle 1,5\rangle \simeq \left\langle (0,0,1),(1,0,1)\right\rangle$ (tj. $\vec{b_1}^{\langle 1,2\rangle} = \vec{b_1}^{\langle 1,5\rangle}$ a $\vec{b_2}^{\langle 1,2\rangle} = \overline{\vec{b_2}^{\langle 1,5\rangle}}$) je f AND $g = \langle f_1.g_1, \overline{\overline{f_2}.\overline{g_2}}\rangle$ a podobně. Z hlediska paměťové složitosti uložení f a časové složitosti výpočtu provádění operací na f tak existují pouze tři různé reprezentace určující dvojici úplně určených f_1 a f_2 (vybereme vždy ty kandidáty z každé čtveřice, které mají nejvíce nul). Jsou to:

1. $\langle 1,2\rangle \simeq \left\langle (0,0,1),(0,1,0)\right\rangle$.
2. $\langle 1,4\rangle \simeq \left\langle (0,0,1),(1,0,0)\right\rangle$.

3. $\langle 2,4\rangle \simeq \left\langle (0,1,0),(1,0,0) \right\rangle$.

Každá jiná reprezentace f_1 a f_2 je ekvivalentní nějaké z tří uvedených. Zkoumejme dále jak vypočítat operace NOT, AND,OR, NAND, NOR, XOR, XNOR,... funkcí f a g pomocí dvojic $\langle f_1, f_2\rangle$ a $\langle g_1, g_2\rangle$. Protože množina {NOT,AND} tvoří v $\mathcal{B}_{xlog}$ úplný logický systém (všechny ostatní binární operace lze dle 1.2 pomocí těchto dvou vyjádřit), stačí se zaměřit právě na výpočet těchto dvou operací (viz. tabulka 2.5).

$\langle 1,2\rangle \simeq \left\langle (0,0,1),(0,1,0) \right\rangle$

f	g	$\langle f_1, f_2\rangle$	$\langle g_1, g_2\rangle$	f AND g	NOT f
0	0	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$
0	1	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$
0	X	$\langle 0,0\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$
1	0	$\langle 0,1\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$
1	1	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$
1	X	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$
X	0	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 1,0\rangle$
X	1	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$
X	X	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$

$\langle 1,4\rangle \simeq \left\langle (0,0,1),(1,0,0) \right\rangle$

f	g	$\langle f_1, f_2\rangle$	$\langle g_1, g_2\rangle$	f AND g	NOT f
0	0	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$
0	1	$\langle 0,1\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$
0	X	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$
1	0	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$
1	1	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$
1	X	$\langle 0,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 0,1\rangle$
X	0	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$
X	1	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$
X	X	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$

$\langle 2,4\rangle \simeq \left\langle (0,1,0),(1,0,0) \right\rangle$

f	g	$\langle f_1, f_2\rangle$	$\langle g_1, g_2\rangle$	f AND g	NOT f
0	0	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$
0	1	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$
0	X	$\langle 0,1\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$
1	0	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$
1	1	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 1,0\rangle$	$\langle 0,1\rangle$
1	X	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,1\rangle$
X	0	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,0\rangle$
X	1	$\langle 0,0\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$
X	X	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$

Tabulka 2.5: Hodnoty AND a NOT pro neúplně určené funkce dané dvojicemi úplně určených funkcí

Z tabulky lze vydedukovat, že platí:

1. Pro $\langle 1,2\rangle \simeq \left\langle (0,0,1),(0,1,0) \right\rangle$ je f AND $g = \langle (f_1 \oplus g_1).(f_2 \oplus g_2), f_2.g_2\rangle$ a NOT $f = \langle f_1, \overline{f_1 + f_2}\rangle$.

2. Pro $\langle 1,4\rangle \simeq \left\langle (0,0,1),(1,0,0) \right\rangle$ je f AND $g = \langle (f_1+g_1).(\overline{f_2 + g_2}), f_2+g_2\rangle$ a NOT $f = \langle f_1, \overline{f_1 + f_2}\rangle$.

3. Pro $\langle 2,4\rangle \simeq \left\langle (0,1,0),(1,0,0) \right\rangle$ je f AND $g = \langle f_1.g_1, f_2 + g_2\rangle$ a NOT $f = \langle f_2, f_1\rangle$.

Uvedené vyjádření operací je nejjednodušší možné (vzhledem k počtu vnitřních operací). Co do složitosti výpočtu operací se tak zdá nejvýhodnější použít reprezentaci f kandidátem $\langle 1,4\rangle$. Opravdu speciálně pro $\langle 1,4\rangle$ dále platí:

$$\begin{aligned} f \text{ NAND } g &= \overline{f \text{ AND } g} = \overline{\langle f_1.g_1, f_2+g_2\rangle} = \langle f_2+g_2, f_1.g_1\rangle. \\ f \text{ OR } g &= \overline{\overline{f} \text{ AND } \overline{g}} = \overline{\langle f_2, f_1\rangle.\langle g_2, g_1\rangle} = \overline{\langle f_2.g_2, f_1+g_1\rangle} = \langle f_1+g_1, f_2.g_2\rangle. \\ f \text{ NOR } g &= \overline{f \text{ OR } g} = \overline{\langle f_1+g_1, f_2.g_2\rangle} = \langle f_2.g_2, f_1+g_1\rangle. \end{aligned} \tag{2.8}$$

Tedy pro kandidáta $\langle 1,4\rangle$ lze všechny binární operace s výjimkou XOR a NXOR lze provést na dvě vnitřní operace (z komutativity a idempotence $\mathcal{B}_{xlog}$ lze snadno ukázat, že je to opravdu teoretické minimum). XOR a NXOR je možné provést složením dvou AND a jednoho OR na 6 operací. Vyjádřením ostatních operací pro zbylé kandidáty $\langle 1,2\rangle$, $\langle 1,4\rangle$ lze ukázat, že kandidát $\langle 2,4\rangle$ vyžaduje v průměru na operaci skutečně nejmenší počet vnitřních operací.

Zastavme se ještě u paměťové složitosti. ROBDD $\mathcal{D}_1$ funkce f_1 resp. ROBDD $\mathcal{D}_2$ funkce f_2 můžeme vytvořit tak, že v ROMBD $\mathcal{D}$ funkce f přepíšeme hodnoty terminálů 0, 1, X na hodnoty b_{10}, b_{11}, b_{1X} vektoru $\vec{b_1}$ resp. na hodnoty b_{20}, b_{21}, b_{2X} vektoru $\vec{b_2}$. Protože v $\mathcal{D}$ tím vždy dojde ke spojení dvou terminálů do jednoho (X se ztrácí), musí každý ROBDD obsahovat nejvýše tolik vrcholů jako původní ROMBD, tj. $|\mathcal{D}_1| \leq |\mathcal{D}|$ a $|\mathcal{D}_1| \leq |\mathcal{D}|$ a celkově oba diagramy neobsahují více než $2|\mathcal{D}|$ vrcholů.

2.5.10 Problém nalezení vhodné permutace vstupních proměnných

Pro logickou funkci f na množině existuje $\{0,1\}^n$ existuje $n!$ různých permutací proměnných f. Přitom pro každou permutaci má odpovídající ROBDD (ROMBD) paměťovou složitost $O(2^n)$. Triviální algoritmus nalezení nejlepší permutace ρ_{min} proměnných, která minimalizuje počet vrcholů všech RODD funkce f, tj.

$$\rho_{min} = \left\{ \rho_{min} \in \Phi \;\middle|\; \mathcal{D}_i(N,T,E,\rho_i) \rhd f \;\text{ a zároveň }\; |\mathcal{D}_{min}| = min\{|\mathcal{D}_i|\} \right\},$$

kde Φ je množina všech permutací čísel z $\{1,2,\ldots,n\}$, má tak složitost $O(n!2^n)$. Bohužel i dosud nejlepší známý algoritmus řešící uvedený problém ([25]), který je založen na dynamickém programování, má složitost $O(n^2 3^n)$. Při praktickém řešení se tak musíme uchýlit k použití heuristik či jiných algoritmů, které dělíme z hlediska přístupu k problému do dvou tříd:

1. **statické**. Vychází ze zadané funkce f, tj. pracují nezávisle na aktuálním minimalizovaném RODD. Patří sem:
   (a) Exaktní algoritmus ([25]).
   (b) Heuristika MINCE ([14]).
   (c) Heuristika FORCE ([15]).
2. **dynamické**. Vychází z aktuálního RODD a snaží se najít další vhodné kroky, které RODD dále minimalizují. Patří sem například:
   (a) Hladový algoritmus ([23]).
   (b) Algoritmy simulovaného ochlazování ([23]).
   (c) Genetické algoritmy (obecně více o genetických algoritmech např. v [12]).

V balíku CUDD ([4]) je implementován exaktní i všechny uvedené dynamické algoritmy. Pro lepší pochopení věci uveďme, jak funguje hladový algoritmus. Všechny dynamické algoritmy jsou postaveny na operaci `Sift()`, která z ROBDD (ROMBD) $\mathcal{D}_1$ daného ρ_1 vyrobí ROBDD (ROMBD) $\mathcal{D}_2$ určeného ρ_2, kde ρ_2 vznikne z ρ_1 prohozením nějakých dvou sousedů. Obě permutace se tedy liší pořadím dvou po sobě jdoucích proměnných (snadno implementovatelné). Hladový algoritmus vyjde vždy z počátečního RODD $\mathcal{D}_0$ určeného ρ_0 (dán uživatelem) a pracuje v iteracích, kdy v i-té iteraci najde vhodnou dvojici sousedních proměnných ρ_i, která po aplikaci `Sift()` na tuto dvojici dá za vznik RODD $\mathcal{D}_{i+1}$ určeného ρ_{i+1}. Dvojice prohazovaných sousedních vrcholů je vybrána tak, aby RODD $\mathcal{D}_{i+1}$ co nejvíce minimalizoval $\mathcal{D}_i$. Hladový algoritmus iteruje do té doby, dokud k minimalizaci dochází ($|\mathcal{D}_{i+1}| < |\mathcal{D}_i|$), jinak vrací $\mathcal{D}_i$. Nevýhodou hladového přístupu je snadné uváznutí v lokálním minimu stavového prostoru problému.

Kapitola 3

Minimalizace logických funkcí

3.1 Formát PLA

Dříve než se budeme věnovat minimalizaci logických funkcí, věnujme pozornost formátu PLA. PLA představuje formát, pomocí něhož je možné reprezentovat libovolný dvouvrstvý logický výraz (signál projde v logickém obvodu realizujícího výraz právě dvěma logickými členy - při předpolkadu, že máme logické členy o libovolném počtu vstupů). Jde o jednoduchý a přímočarý způsob ukládání logické funkce (výrazu). Každý PLA můžeme myšlenkově rozdělit na tři části:

1. **Hlavička PLA** - V hlavičce se nachází obecné informace popisující logickou(-é) funkce. Mezi obvyklá klíčová slova, která zde mohou být, patří klíčová slova (symbol `[d]` značí číslo, symboly `[s1],...,[sN]` řetězce):
   - `.i [d]` Číslo udává počet vstupních proměnných logické funkce(-í) (povinný parametr).
   - `.o [d]` Číslo udává počet výstupů logické funkce (jedno- či vícevýstupová logická funkce) (povinný parametr).
   - `.p [d]` Číslo udává počet součinových termů uvedených v PLA dále (nepovinný parametr).
   - `.ilb [s1]...[sN]` Řetězce specifikují jména vstupních proměnných (nepovinný parametr).
   - `.ob [s1]...[sN]` Řetězce specifikují jména výstupních funkcí (nepovinný parametr).
   - `.type [typ]` Typ udává způsob specifikace logické funkce (viz. dále), povolenými hodnotami typu jsou typy: `f, r, fd, fr, dr, fdr`. Není-li typ uveden, předpokládá se typ `fd` (nepovinný parametr).
2. **Matice hodnot** - Matice hodnot je rozdělena do dvou částí:
   - **Vstupní část** Levá strana každého řádku matice odpovídá jednomu součinovému termu, který je pro n vstupních proměnných funkce zadán ve tvaru $b_1 b_2 b_3 \dots b_n$ ($b_i \in \{0, 1, -\}$ pro $1 \leq i \leq n$). Hodnota $b_i = 0$ značí negaci i-té proměnné, hodnota $b_i = 1$ značí i-tou proměnnou a hodnota $b_i = -$ značí, že se proměnná v součinovém termu nevyskytuje. Mezi jakýmikoliv b_i a b_{i+1} se navíc může vyskytovat libovolný počet bílých znaků (mezer, nových řádků, tabulátorů, ...).
   - **Výstupní část** - Pravá část každého řádku matice odpovídá výstupní hodnotě resp. výstupnímu vektoru hodnot definované jedno- resp. vícevýstupové funkce v bodě(-ech), v nichž je součinový term na levé straně řádku pravdivý. Mezi jakýmikoliv dvěma prvky vektoru se teoreticky může opět vyskytovat libovolný počet bílých znaků. Řádek matice hodnot může například vypadat

     `0-110-0 101.`

     Řádek určuje term $T = \overline{x_1}.x_3.x_4\overline{x_5}.\overline{x_7}$ a vektor výstupních hodnot $(1, 0, 1)$ (předokládáme zde, že proměnné $x_1, \dots, x_7$ jdou po sobě a mezera odděluje vstupní a výstupní část řádku matice).

3. **Příznak konce matice hodnot** - Příznak `.e` značí konec matice hodnot PLA, tj. konec souboru.

```
 # toto je komentar
.i 6
.o 3
.ilb x1 x2 x3 x4 x5 x6
.ob f1 f2 f3
.type fd
.p 4
10-01- 101
0----1 001
1101-- 010
011--0 100
.e
```

Obrázek 3.1: Příklad PLA souboru

V PLA je možné uvádět komentáře, začínají znakem `#` a končí znakem nového řádku. Každá logická funkce může být v PLA reprezentována jedním z následujících způsobů (viz. 1.7):

1. `.type f` Pomocí ONsetu. OFFset funkce se získá jako doplněk ONsetu, DCset se položí roven prázdné množině. Termy určující případně body OFFsetu a DCsetu jsou ignorovány.

2. `.type r` Pomocí OFFsetu. ONset funkce se získá jako doplněk ONsetu, DCset se položí roven prázdné množině. Termy určující případně body ONsetu a DCsetu jsou ignorovány.

3. `.type fd` Pomocí ONsetu a DCsetu. OFFset funkce se získá jako doplněk sjednocení ONsetu a DCsetu. Termy určující případně body OFFsetu jsou ignorovány. Je-li nějaký bod definičního oboru popisované funkce přiřazen v PLA zároveň do ONsetu a DCsetu, pak se předpokládá, že bod patří do DCsetu.

4. `.type fr` Pomocí ONsetu a OFFsetu. DCset funkce se získá jako doplněk sjednocení ONsetu a OFFsetu. Termy určující případně body DCsetu jsou ignorovány. Je-li nějaký bod definičního oboru popisované funkce přiřazen v PLA zároveň do ONsetu i OFFsetu, pak není primárně stanoveno, do které z množin bod patří (např. minimalizátor ESPRESSO to považuje za chybu).

5. `.type dr` Pomocí DCsetu a OFFsetu. ONset funkce se získá jako doplněk sjednocení DCsetu a OFFsetu. Termy určující případně body ONsetu jsou ignorovány. Je-li nějaký bod definičního oboru popisované funkce přiřazen v PLA zároveň do DCsetu a OFFsetu, pak se předpokládá, že bod patří do DCsetu.

6. `.type fdr` Pomocí ONsetu a OFFsetu a DCsetu. Je-li nějaký bod definičního oboru popisované funkce přiřazen v PLA zároveň do ONsetu a DCsetu resp. OFFsetu a DC setu, pak se předpokládá, že bod patří do DCsetu. Je-li nějaký bod přiřazen v PLA zároveň do ONsetu i OFFsetu, pak opět není primárně stanoveno, do které z množin bod patří (ESPRESSO to považuje za chybu). Dále některé minimalizátory v případě `fdr` požadují, aby termy v matici hodnot popisovaly celý definiční obor funkce.

Na obrázku 3.1 je pro příklad uveden PLA funkce: $f : \{0,1\}^6 \rightarrow \{0,1,X\}^3$ definované

$$f = (f_1, f_2, f_3) = \big(x_1.\overline{x_2}.\overline{x_4}.x_5 + \overline{x_1}.x_2.x_3.\overline{x_6},\ x_1.x_2.\overline{x_3}.x_4,\ x_1.\overline{x_2}.\overline{x_4}.x_5 + \overline{x_1}.x_6\big).$$

Více se o formátu PLA lze dočíst na webových stránkách [6].

3.2 Pojmy minimalizace a skupinová minimalizace

V úvodu práce bylo řečeno, že **minimalizace logických funkcí** (přesněji výrazů) představuje základní krok při návrhu každého logického obvodu. Účelem minimalizace je minimalizovat logický výraz jako model struktury logického obvodu s cílem zíkat co nejjednodušší výraz pro původní logickou funkci a tím i tvar logické funkce pro minimální realizační strukturu.

Kritérii minimalizace nejčastěji jsou:

1. Výraz má obsahovat minimální počet logických operací (logických členů).
2. Výraz má obsahovat minimální počet literálů (počet vstupů logických členů).
3. Kombinace obou předešlých kritérií pro jistý soubor realizačních prvků obvodu.
4. Výraz má obsahovat minimální počet vrstev logických členů (minimalizace zpoždění signálu).

V [2] je například dokázáno, že existují funkce, které při použití dvouvstupových logický členů, obsahují $\Omega(\frac{2^n}{2n})$ logických členů, tzn. mají exponenciální složitost. Obecně patří problém minimalizace mezi $\mathcal{NP}$ těžké problémy. My se v dalším textu budeme zabývat výhradně „dvouvrstvou“ minimalizací, a to že minimální výraz bude vždy definován pomocí PLA (3.1).

Složitější variantou minimalizace je **skupinová minimalizace**. Předpokládejme, že máme vícevýstupovou logickou funkci f (určující jednohodnotové logické funkce $f_1, \ldots, f_n$) definovanou PLA. Skupinovou minimalizací funkce f (funkcí $f_1, \ldots, f_n$) budeme mít na mysli nalezení takového PLA odpovídajícího typu (`f,fd,fr,fdr, ...`), který obsahuje v matici hodnot *minimální počet součinových termů (řádků)*. Někdy budeme též brát v potaz kritérium *minimálního počtu literálů* obsažených v minimalizovaném PLA.

```
.i 3
.o 2
.type fd
.p 6
100 10
010 01
000 1-
101 10
101 01
011 -0
.e
```

```
.i 3
.o 2
.type fd
.p 3
-00 10
0-0 01
101 11
.e
```

Obrázek 3.2: Ukázka skupinově minimalizované funkce v PLA

Pro lepší pochopení je na obrázku 3.2 ukázka PLA původní (vlevo) a skupinově minimalizované vícevýstupové funkce (vpravo).

3.3 Způsoby minimalizace

Absolutní většina minimalizačních algoritmů pracuje ve dvou krocích:

1. Nalezení množiny přímých implikantů (1.12)
2. Výběr minimálního počtu přímých implikantů, které minimalizovanou funkci f definují. Sjednocením tohoto minimálního počtu přímých implikantů dostaneme funkci, která je pravdivá v množině S_f, kde pro S_f platí: $S_f \subset \mathcal{D}_f$, $\mathcal{D}_{on} \subset S_f$ a $S_f \subset \mathcal{D}_{on} \cup \mathcal{D}_{dc}$ (viz. 1.7).

3.3.1 Tradiční způsoby minimalizace

Mezi základní metody minimalizace řadíme:

1. Minimalizace pomocí **zákonů Booleovy algebry** (nesystematická).

   Jde o intuitivní metodu, kdy pro funkci f zadanou výrazem V se snažíme V použitím zákonů Booleovy algebry maximálně zjednodušit. Například následující výraz určující funkci g můžeme postupně upravit:

$$\begin{aligned} g &= x_1.x_2.\overline{x_3} + x_1.x_3 + \overline{x_2}.x_3 + x_2.x_3 = \\ &= x_1.x_2.\overline{x_3} + x_1.x_3 + (\overline{x_2} + x_2).x_3 = \\ &= x_1.x_2.\overline{x_3} + x_1.x_3 + x_3 = \\ &= x_1.x_2.\overline{x_3} + (x_1 + 1).x_3 = \\ &= x_1.x_2.\overline{x_3} + x_3 = \\ &= x_1.x_2.\overline{x_3} + (x_1.x_2 + 1).x_3 = \\ &= x_1.x_2.(\overline{x_3} + x_3) + x_3 = \underline{x_1.x_2 + x_3} \end{aligned}$$

   Metoda je použitelná pouze pro jednoduché funkce a není vhodná pro skupinovou minimalizaci.

2. Minimalizace pomocí **Karnaughových map** (nesystematická)

   Opět intuitivní metoda, kdy funkci zapíšeme do Karnaughovy mapy a poté hledáním jistých maximálních čtvercových či obdelníkových oblastí pokrytých stejnými čísly (případně obsahující don't care $-$) nalezneme množinu přímých implikantů, z nichž posléze vybereme nejmenší možný počet, který pokryje právě minimalizovanou funkci. Vlastností Karnaughovy mapy je, že každé dva sousední (nebo krajní) řádky či sloupce se pro daný term liší v ohodnocení právě jedné proměnné, viz. 3.3 (předpokládáme, že kraje mapy jsou „slepené“, tj. ve skutečnosti mapa tvoří toroid). Odtud plyne, že jakékoliv implikanty jsou obdelníky nebo čtverce se stranami délek mocniny dvou. Na obrázku 3.3 je příklad Karnaughovy mapy funkce v minimální DNF $f = \overline{x_1}.x_2 + \overline{x_2}.x_3$. Přitom f má čtyři přímé implikanty: $\overline{x_1}.x_2$ (a), $\overline{x_1}.x_3$ (b), $\overline{x_2}.x_3$ (c), $x_3.\overline{x_4}$ (d).

		(x_1, x_2)			
		00	01	11	10
(x_3, x_4)	00	0	$1_{(a)}$	0	0
	01	0	$1_{(a)}$	$-$	0
	11	$1_{(b)(c)}$	$1_{(a)(b)}$	0	$1_{(c)}$
	10	$1_{(b)(c)(d)}$	$1_{(a)(b)(d)}$	$-_{(d)}$	$1_{(c)(d)}$

Obrázek 3.3: Příklad Karnaugovy mapy

   Metoda je použitelná pouze pro jednoduché funkce a lze rozšířit i pro skupinovou minimalizaci, více se lze dočíst v [9].

3. Minimalizace metodou **Quine-McCluskey** (systematická).

   Jde o systematickou metodu, kterou lze přirovnat k minimalizaci pomocí Karnaughovy mapy. Vyjde se součinových 0-termů (termy obsahující všechny proměnné minimalizované funkce) a konstruuje postupně 1-termy, 2-termy, ..., k-termy (termy, v nichž chybí k proměnných), dokud k roste. Dva k-termy se dají sloučit do jednoho $k+1$-termu, pokud se oba k-termy liší v negovanosti právě jedné proměnné (jinak jsou totožné). Tímto postupem se získá množina přímých implikantů. Poté se zkonstruuje tabulka pokrytí obsahující tyto přímé implikanty a vybere se minimální počet přímých implikantů pokrývající $\mathcal{D}_{on}$ funkce f. Metoda v nejhorším případě vyžaduje exponenciální množství paměti i času, lze ji však dobře rozšířit i pro skupinovou minimalizaci. Výborný manuál ke Quine-McCluskey ja na stránkách [3]. Metoda byla poprvé uveřejněna v práci [10].

3.3.2 Heuristické způsoby minimalizace

Je-li exaktní metoda neúnosná, musí přijít na řadu heuristický přístup řešení problému. Patrně nejčastěji používanou heuristikou pro „obyčejnou“ i skupinovou minimalizaci je algoritmus **ESPRESSO**. ESPRESSO bylo vyvinuto v 80.letech minulého století ve spolupráci firmy IBM a univerzitou v Berkley. Heuristika je přímo určená pro počítačové zpracování a dosahuje často „velmi dobrých výsledků“ v „únosném čase“. Uveďme stručně základní myšlenku algoritmu ESPRESSO. Klíčem jsou podle původního článku [22]) tři metody:

1. **Reduce**. Redukce pokrytí ONsetu. „Rozmělní“ množinu přímých implikantů pokrytí ONsetu na množinu implikantů M (ne nutně přímých).

2. **Expand**. Expanze pokrytí ONsetu. Nahradí každý implikant K v pokrytí M ONsetu přímým implikantem, který pokrývá implikant K.

3. **Irredundant**. Extrakce minimálního pokrytí. Z pokrytí ONsetu je extrahováno minimální pokrytí (výběr minimální množiny přímých implikantů).

Algoritmus pak pracuje ve dvou cyklech, ve vnitřním cyklu probíhají po sobě oberace Reduce, Expand a Irredundant, dokud dochází k minimalizaci pokrytí ONsetu. Po vyskočení z vnitřního cyklu následuje druhý krok ve vnějším cyklu, krok **LastGasp**, který je alternativou k Reduce, Expand a Irredundant (liší ze ve způsobu jak dosáhnot lepšího řešení). LastGasp perturbuje získané řešení. Podle výsledku LastGasp (zda je pokrytí ONsetu po LastGasp menší než po posledním dřívějším volání LastGasp) se rozhoduje, zda dále pokračovat ve vnějším cyklu. Pokud se má pokračovat, program se uchýlí zpět do vnitřního cyklu. Aby mohl být vnější cyklus vůbec spuštěn, je na začátku algoritmu nutné vygenerovat nějaké pokrytí ONsetu. Na konci algoritmu je pak spuštěna metoda **Verify**, která ověří, zda obě funkce (minimalizovaná a původní) jsou „ve smyslu minimalizace“ ekvivalentní.

Heuristika ESPRESSO bude standardně použita i v našem způsobu minimalizace (viz. dále), proto jsme se o ní zmínili. Dalšími minimalizátory jsou například **BOOM** ([5]) či **BOOM-II** ([11]).

3.4 Minimalizace pomocí ROBDD a ROMBD

Minimimalizace logické funkce určené BDD (či MBD) $\mathcal{D}(N, T, E, \rho, \sigma, son, idx, val)$ je přímočaře založena na následující rekurzivní myšlence: Pro každý neterminální vrchol $v_f \in N$ diagramu nalezni nejprve minimimální výrazy V_0 a V_1 pro logické funkce potomků vrcholu v_f ($son(v_f, 0)$ a $son(v_f, 1)$). Výrazy V_0 a V_1 pak „spoj“ podle Shannonova expanzního theorému

$$f = \overline{x_i} f[x_i = 0] + x_i f[x_i = 1]$$

(předpokládáme, že $idx(v_f) = i$) do jednoho výrazu V', tak aby V' definoval právě funkci f vrcholu v_f. Nakonec V' minimalizuj na výraz V a případně předej rodičovskému vrcholu jako výsledek rekurzivního volání. Vytvoření minimálních výrazů V_0, V_1 a V je závislé na použití libovolného *externího* minimalizátoru. Nalezení minimálních výrazů pro terminální vrcholy $t \in T$ je triviální úloha.

Uvedený postup popisuje stručně algoritmus 3.4. Jedná o jeden z možných způsobů, jak minimalizaci na BDD či MBD provádět (algoritmickým procházením diagramu do hloubky). Druhý způsob minimalizace na BDD či MBD spočívá v procházení diagramu do šířky, tj. po patrech (od kořene k terminálům). Tento způsob je paměťově náročnější, časová složitost se nemění My se budeme zabývat výhradně prvním způsobem minimalizace (procházením od kořene k terminálům do hloubky). Diagram bychom mohli procházet i obráceně, tj. od terminálů ke kořeni (ať už do hloubky nebo do šířky), avšak to si vyžaduje, aby každý vrchol obsahoval ve své struktuře `DdNode` reference na všechny své předky. Pokud je $\mathcal{D}$ veliký, zvyšují tyto zpětné reference nároky na paměť (někdy neúnosně).

Algoritmus 3.4 má minimální paměťovou složitost, na druhou stranu ale vždy exponenciální časovou složitost, projde totiž dohromady $1 + 2 + 4 + \cdots + n = O(2^n)$ vrcholů. Pokud pracujeme s ROBDD (ROMBD), snadno se stane, že se funkce jistého subdigram (toho, do nějž vede dvě nebo více hran z $\mathcal{D}$)

```
DdString * Minimize( DdNode * f )              // predokladame: f neni terminal
{
    DdNode   * then = Regular(f)->type.kids.T; // potomek vetve then
    DdNode   * else = Regular(f)->type.kids.E; // potomek vetve else
    DdString * V0, *V1, *V, *Vmin;             // minimalni vyrazy potomku
                                               // a vyrazy pro f
    if ( IsNotRegular(f) ) {                   // neni-li f regularni
        then = Not(then); else = Not(else);    // negace potomku (neg. hrany)
    }
    if ( IsConstant(then) ) {                  // je-li then terminalni
        V1 = TrivialMinimizationCase(then);    // trivialni pripad
    } else {
        V1 = Minimize(then);
    }
    if ( IsConstant(else) ) {                  // je-li else terminalni
        V0 = TrivialMinimizationCase(else);    // trivialni pripad
    } else {
        V0 = Minimize(else);
    }
    V = JoinMinimalSons(V1, V0);               // sjednoceni V0 a V1 do V
                                               // dle Shannonova e.t.
    Vmin = ExetrnalMinimizationCall(V);        // externi minimalizace
    free(V0); free(V1); free(V);               // uvolneni pameti
    return Vmin;                               // navrat minimalniho vyrazu
}
```

Zdrojový kód 3.4: Zjednodušená verze algoritmu minimalizace logické funkce pomocí BDD, MBD

diagramu $\mathcal{D}$ počítá vícekrát a algoritmus 3.4 je silně neefektivní. Výhodnější by proto bylo počítat minimální funkci g každého vrcholu v_g v $\mathcal{D}$ právě jednou. To je opravdu díky kanonicitě ROBBD (ROMBD) po zavedení dvou prostředků možné:

1. Je-li funkce `Minimize()` na $v_g \in N$ zavolána, zjistíme nejprve, zda funkce g již nebyla dříve minimalizována. Pokud do v_g vede pouze jedna hrana v prostředí `DdManager` (parametr `ref` struktury `DdNode`), pak g minimalizována nebyla (touto jedinou hranou jsme do v_g právě vstoupili). Vede-li do v_g více hran, potřebujeme najít způsob jak tento fakt zjistit. Zavedeme proto strukturu `DdMinimizationData` (3.5) obsahující počet již prošlých hran v průběhu minimalizace. Každému vícenásobně referencovanému subdiagramu (vrcholu v_g) diagramu $\mathcal{D}$ přiřadíme jednu instanci této struktury.

   Struktura `DdMinimizationData` pokrývá jak regulární, tak komplementovaný vrchol, tj. jeden `DdNode` v prostředí `DdManager`, proto maji všechny proměnné v `DdMinimizationData` podvojný charakter. Nabízí se dvě možnosti, jak přistupovat k `DdMinimizationData` vrcholu `DdNode`, buď umístit dovnitř struktury `DdNode` ukazatel na `DdMinimizationData` (ten bude pro jednonásobně referencované vrcholy roven `NULL`) nebo použít hashovací tabulku s klíčem adresy `DdNode *` a výsledkem ukazatele `DdMinimizationData *`. První varianta nabízí rychlejší přístup k `DdMinimizationData` vrcholu, druhá zase umožňuje ukládat pouze podmnožinu všech `DdMinimizationData`, tzn. pokud se pro vrchol v_g nenajde v tabulce jeho `DdMinimizationData`, minimalizace v_f se provede, třeba že již byl v_f dříve minimalizován (úspora času na úkor paměti a naopak). Ve své implementaci jsem se rozhodl použít první způsob, který je rychlejší, ale žádá obecně více paměti (při minimalizaci nebude s paměťí problém, pokud ano, pak i čas výpočtu minimalizace bude neúnosný). Struktura

```
struct DdMinimizationData
{
    unsigned int refsR;    // kolikrat byl regularni vrchol navstiven
    unsigned int refsC;    // kolikrat byl komplementovany vrchol navstiven
    unsigned int refsCntR; // celkovy pocet hran do regularniho vrcholu ve funkci
    unsigned int refsCntC; // celkovy pocet hran do komplementovaneho vrcholu ve funkci
};
```

Zdrojový kód 3.5: Zdrojový kód struktury `DdMinimizationData`

`DdNode` se tak rozšíří (3.6):

```
struct DdNode
{
    ...
#if DD_MINIMIZATION == 1 //podmineny preklad, je-li pouzita minimalizace
    DdMinimizationData * data;
#endif
};
```

Zdrojový kód 3.6: Rozšíření struktury `DdNode`

Na začátku minimalizace se nejprve projde $\mathcal{D}$ do hloubky a u vícenásobně referencovaných vrcholů se alokuje pro každý `DdNode` instance `DdMinimizationData`, nastaví hodnoty `refsCntR` a `refsCntC` (podle počtu návštěv regulárního či komplemntovaného vrcholu) a položí se `refsR` = `refsC` = 0. Kód inicializace `MinimizationInit()` je uvedena v 3.7.

Dynamicky alokovaná instance `DdMinimizationData` v `DdNode` je při minimalizaci odstraněna ihned po poslední návštěvě `DdNode` (aby instance `DdMinimizationData` zbytečně nezabírala paměť) a ukazatel `data` je nastaven zpět na `NULL`.

2. Minimalizované výrazy vícenásobně referencovaných vrcholů v_g potřebujeme někam ukládat (jednonásobně referencované vrcholy nepotřebujeme ukládat, jejich minimalizované výrazy se vrací ihned jako výsledek rekurzivního volání `Minimize()`). Pokud to bude do paměti, stačí do `DdMinimizationData` přidat ukazatele `DdString * VR` a `DdString * VC` na minimální výrazy regulárního resp. komplementovaného vrcholu. Budeme-li ukládat výrazy na pevný disk, můžeme je ukládat do souborů s názvy odpovídající adrese ukazatele `DdNode *` (adresa je pro každou logickou funkci jedinečná). Protože většina externích minimalizátorů (ESPRESSO,...) načítá výrazy ze souborů, zvolil jsem druhý způsob uložení na disku.

Zdrojový kód sofistikovanější verze algoritmu minimalizace `MinimizeSmart()` se tedy skládá z inicializace `MinimizationInit()` (3.7) a rekurzivní minimalizace `MinimizeRecur()` (pro rozsáhlost v příloze A.3) a je uveden v 3.8.

3.5 Skupinová minimalizace pomocí ROBDD a ROMBD

Doteď jsme se zabývali minimalizací jedné logické funkce. Nyní před námi stojí úkol rozšířit minimalizaci pomocí ROBDD (ROMBD) na skupinovou minimalizaci. Napadnou nás dvě možnosti jako to udělat:

```
void MinimizationInit( DdNode * f )            // predpoklad: f je neterminal
{
    DdNode   * then = Regular(f)->type.kids.T; // potomek vetve then
    DdNode   * else = Regular(f)->type.kids.E; // potomek vetve else

    if ( IsNotRegular(f) ) {                   // neni-li f regularni
        then = Not(then); else = Not(else);    // negace potomku (neg. hrany)
    }
    if ( IsNotConstant(then) ) {               // neni-li then terminalni
        data = Regular(then)->data;
        if ( data != NULL ) {                  // kdyz inicializace jiz probehla
            if ( IsRegular(then) )             // je-li then regularni
                data->refsCntR++;
            else                               // then je komplementovany
                data->refsCntC++;
        } else {
            MinimizationInit(then);            // rekurze na then
        }
    }
    if ( IsNotConstant(else) ) {               // neni-li then terminalni
        data = Regular(else)->data;
        if ( data != NULL ) {                  // kdyz inicializace jiz probehla
            if ( IsRegular(else) )             // je-li else regularni
                data->refsCntR++;
            else                               // else je komplementovany
                data->refsCntC++;
        } else {
            MinimizationInit(else);            // rekurze na else
        }
    }
    if ( Regular(f)->ref > 1 ) {               // je-li f vicenasobne referencovany
        data = malloc(sizeof(DdMinimizationData)); // alokuj strukturu
        data->refsR = data->refsC = 0;         // minimalizace jeste nezapocala
        if ( IsRegular(f) ) {                  // nastav prvni referenci podle typu f
             data->refsCntR = 1; data->refsCntC = 0;
        } else {
             data->refsCntC = 1; data->refsCntR = 0;
        }
    }
}
```

Zdrojový kód 3.7: Inicializace vícenásobně referencovaných vrcholů před minimalizací pomocí ROBDD, ROMBD

1. **Sériové zpracování** - Rozšíříme `MinimizeSmart()` tak, že `MinimizationInit()` a `MinimizeRecur()` obalíme do cyklů (každou do jiného), v nichž postupně iterujeme přes kořeny diagramů $\mathcal{D}_1, \ldots, \mathcal{D}_n$ všech n skupinově minimalizovaných funkcí $f_1, \ldots, f_n$. Voláními `MinimizeRecur()` získáme minimální výrazy $V_1, \ldots, V_n$ funkcí $f_1, \ldots, f_n$. Poté V_i každé f_i rozšíříme na vektor výrazů $\vec{v_i} = (\underbrace{0, 0, \ldots, f_i}_{i}, 0, \ldots, 0)$ a všechny $\vec{v_i}$ sečteme jako $\vec{v} = \sum_{i=1}^{n} \vec{v_i}$ (sloučíme do jednoho). Nakonec

```
DdString * MinimizeSmart( DdNode * root )
{
    MinimizationInit(root);      // inicializace vicenasobne ref. vrcholu
    return MinimizeRecur(root);  // samotna minimalizace
}
```

Zdrojový kód 3.8: Vylepšený algoritmus minimalizace logické funkce pomocí ROBDD, ROMBD

```
DdString * MinimizeMultipleOutputSerial( DdNode ** roots, int count )
{
    unsigned int i;                                          // promenna cyklu
    DdString *   V, *Vmin;                                   // pomocny a skup. min. vyraz
    Ddstring **  vArray = malloc(sizeof(DdString*)*count); // min. vyrazy korenu

    for ( i = 0; i < count; i++ )
        MinimizationInit(roots[i]);             // inicializace vrcholu
    for ( i = 0; i < count; i++ )
        vArray[i] = MinimizeRecur(roots[i]);    // samotna minimalizace
    V = JoinAllFormulas(vArray, count);         // slouceni vyrazu do jednoho
    for ( i = 0; i < count; i++ )
        free(vArray[i]);                        // uvolneni pameti vyrazu
    Vmin = ExternalMinimizationCall(V);         // vrat vysledek externi minimalizace
    free(V);                                    // uvolni pamet vyrazu V
    return Vmin;                                // vrat vysledek
}
```

Zdrojový kód 3.9: Algoritmus „sériové“ skupinové minimalizace logických funkcí pomocí ROBDD, ROMBD

zavoláme na $\vec{v}$ externí minimalizátor, čímž získáme skupinově minimální výraz vektoru funkcí $(f_1, \ldots, f_n)$. Jsou-li výrazy uloženy v PLA formátu (viz. naše implementace), je třeba provést rozšíření podle typu PLA souboru (sloučení proběhne prostým sloučením všech rozšířených PLA). Na pozice nul ve v_i dáme:

(a) `.type f:` 0, neboť se v tomoto případě ignorují body OFFsetu.

(b) `.type r:` 1, neboť se v tomoto případě ignorují body ONsetu.

(c) `.type fd:` 0, neboť se v tomoto případě ignorují body OFFsetu.

(d) `.type fr:` $-$, neboť se v tomoto případě ignorují body DCsetu.

(e) `.type dr:` 1, neboť se v tomoto případě ignorují body ONsetu.

(f) `.type fdr:` $\sim$, speciální znak, který říká, že součinový term v řádku vlevo nemá na hodnotu funkce žádný vliv.

Algoritmus „sériové“ skupinové minimalizace `MinimizeMultipleOutputSerial()` je uveden v 3.9.

2. **Paralelní zpracování**

Paralelní zpracování se od sériového liší tím, že skupinovou minimalizaci provádíme v průběhu rekurze. Není tedy nutné na konci minimalizace provádět `JoinAllFormulas()` a závěrečnou sku-

pinovou minimalizaci. Funkci `MinimizeRecur()` upravíme na funkci `MinimizeRecurParalell()` následujícím způsobem:

(a) Vstupním parametrem `MinimizeRecurParalell()` nebude pouze jeden vrchol právě minimalizované funkce, nýbrž celkem n vrcholů, pro každou skupinově minimalizovanou subfunkci právě jeden. `MinimizeRecurParalell()` tedy skupinově minimalizuje vektor $(g_1, \ldots, g_n)$ subfunkcí, určených jistou krychlí (podrostorem) definičního oboru funkcí (podprostor je určen součinovým termem cesty z nejvýše položeného kořene diagramů minimalizovaných funkcí do nejníže položeného kořenu diagramu subfunkce g_i, $1 \leq i \leq n$).

(b) Protože se při volání `MinimizeRecurParalell()` snadno může stát, že dva různé vektory subfunkcí $(g_1, \ldots, g_n)$ a $(h_1, \ldots, h_n)$, kde $g_i = h_j$ (pro nějaké indexy i, j), budou předávány `MinimizeRecurParalell()`, je v průběhu minimalizace v `MinimizeRecurParalell()` nutné vytvářet minimální výrazy funkcí $g_1 \ldots, g_n$ také extra každý zvlášť, poté je v `MinimizeRecurParalell()` spojit do jednoho vektoru výrazů, skupinově minimalizovat a vrátit rekurzivní výsledek nadřazenému volání `MinimizeRecurParalell()`.

(c) Protože se v `MinimizeRecurParalell()` změnil způsob procházení diagramů, je nutné odpovídajícím způsobem změnit i inicializaci vícenásobně referencovaných vrcholů `MinimizationInit()` na paralelní verzi `MinimizationInitParalell()`, která prochází diagramy stejným způsobem jako `MinimizeRecurParalell()`.

Výsledná funkce `MinimizeMultipleOutputParalell()` je uvedena v 3.10. Tato varianta minimalizace nebude implementována. Jádro algoritmu, funkce `MinimizationInitParalell()` a `MinimizeRecurParalell()` zde uvádět nebudeme. Nevýhodou tohoto přístupu je, že parametry rekurze zabírají oproti `MinimizeMultipleOutputSerial()` n-krát více paměti. Výhodou je, že skupinová minimalizace probíhá postupně, nikoliv nakonec. Časová složitost je bez ohledu na načítání ze souboru a volání externího minimalizátoru stejná jako u `MinimizeMultipleOutputSerial()`.

```
DdString * MinimizeMultipleOutputParalell( DdNode ** roots, int count )
{
    MinimizationInitParalell(roots,count);    // inicializace vicenasobne ref. vrcholu
    return MinimizeRecurParalell(roots,count);// samotna minimalizace
}
```

Zdrojový kód 3.10: Algoritmus „paralelní“ skupinové minimalizace logických funkcí pomocí ROBDD, ROMBD

Kapitola 4

CUDD a jeho úprava pro potřeby minimalizace

4.1 O CUDDu obecně

Balík CUDD ([4]) patří v současné době mezi jeden z nejpoužívanějších (né-li nejpoužívanější, je použit v systému VIS - Veryfication Interacting with Synthesis, více o VIS např. v [16]) softwarových nástrojů pro práci s rozhodovacími diagramy. Balík vyvýjí(-el) Fabio Somenzi a jeho pracovní skupina na University of Colorado. Postupným vylepšováním a rozšiřováním algoritmů dospěl do až do poslední verze 2.4.1 (rok 2001). Balík je velmi komplexní, zdrojové kódy obsahují odhadem několik desítek tisíc řádku.

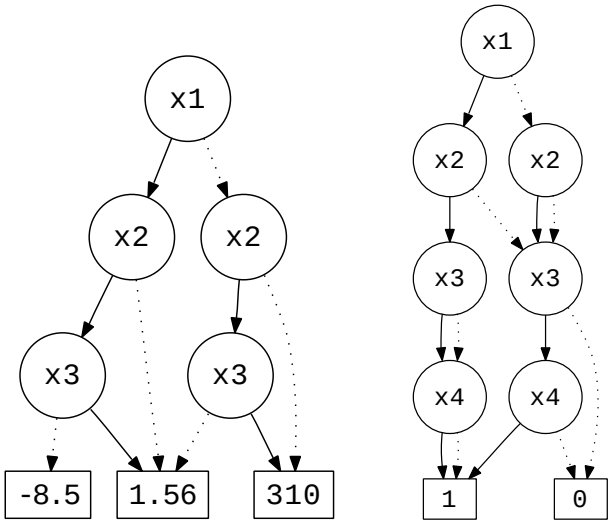
V CUDDu je standardně možné pracovat se třemi druhy diagramů:

1. **BDD** *Binární rozhodovací diagramy*, v naší terminilogii jde o ROBDD.
2. **ADD** *Algebraické rozodovací diagramy*, tj. diagramy $\mathcal{D}(2, D, H, N, T, E, \rho, \sigma, son, idx, val)$, kde $D = \{0,1\}^n$ a $H \subset \mathbb{R}$. Pracuje se vždy s redukovanými ADD (obrázek 4.1).
3. **ZDD** *Diagramy s potlačenou nulou.* Jedná se o uspořádané binární diagramy, které se redukují jiným způsobem. Za zbytečné vrcholy (ty, které jsou z úplného ZDD odstraněny) se u ZDD považují vrcholy jejichž hrana ohodnocená 1 (*then*) vede do terminálu s hodnotou 0. Tato úprava je výhodná pro reprezentaci řídkých množin. Pracuje se vždy s redukovanými ZDD, které mají stejně jako BDD a ADD vlastnost kanonicity (příklad na obrázku 4.1). Původcem myšlenky diagramů s potlačenou nulou je Minato, více o ZDD se lze dočíst v [24] nebo [17].

Poznámka 4.1. Obdobně jako u ROBDD, lze na ADD zavést operace *Apply*, *Restrict* a jiné. V případě ADD jsou pro *Apply* možnými operacemi `<op>` $\in \{+, -, *, /, \dots\}$, kde výsledkem operace $\mathcal{D}_1$ `<op>` $\mathcal{D}_2$ na dvou ADD je redukovaný ADD $\mathcal{D}$, pro nějž platí $f = f_1$`<op>`f_2 ($\mathcal{D} \rhd f$, $\mathcal{D}_1 \rhd f_1$ $\mathcal{D}_2 \rhd f_2$). Přitom operaci `<op>` chápeme jako stejnojmennou operaci nad tělesem reálných čísel $\mathbb{R}$. U ZDD je množina přípustných `<op>` operace *Apply* stejná jako u BDD, neboť ZDD reprezentuje také logické funkce.

CUDD disponuje širokou nabídkou algoritmů pro hledání vhodné permutace proměnných (obsahuje všechny alogritmy uvedené v 2.5.10), dále nabízí možnost exportu diagramů do různých formátů jako jsou DOT, DaVinci a další a v neposlední řadě poskytuje širokou třídu operací nad diagramy (operace *Apply* a *Restrict* jsou samozřejmě součástí). Celý balík může být používán třemi způsoby:

1. Jako černá skříňka (*black box*) - uživatel používá dostupné metody a funkce a nestará se o to, co se uvnitř těchto funkcí děje (co se děje v pozadí)
2. Jako průhledná skříňka (*clear box*) - nevyhovují-li uživateli dostupné metody, může vytvořit novou nebo upravit existující funkci a přidat ji do balíku



Obrázek 4.1: Ukázka ADD, dále pak ZDD pro $f = x_1.x_2 + x_3.x_4$

3. Jako rozhraní - použitím objektově orientované nadstavby je uživatel osvobozen od správy paměti. C++ rozhraní je přímo dostupné v poslední distribuci CUDDu, rozhraní pro Perl15 existuje samostatně

4.2 Ukázka vytvoření ROBDD v CUDDu

Na stránkách [4] je dostupná dokumentace CUDDu. Pro lepší orientaci uveďme ukázku kódu vytvoření jednoduchého ROBDD funkce $f(x_1, x_2, x_3) = x_1.x_2.x_3 + \overline{x_1}.\overline{x_2}.\overline{x_3}$ (4.2

V každém CUDD programu je nutné na začátku zavolat funkci `Cudd_Init()`, která vrátí dříve popsaný pojem nového prostředí (manažeru). Prostředí zastřešuje skupinu diagramů (BDD, ADD a ZDD), které v něm uživatel vytvoří a má na starosti správu paměti. Funkce `Cudd_ReadOne()` vrací diagram konstanty 1. V horním resp. dolním cyklu se pak postupně vytvoří oba termy $x_1.x_2.x_3$ resp. $\overline{x_1}.\overline{x_2}.\overline{x_3}$. Operace `Cudd_bddIthVar()` vrací ROBDD i-té proměnné, operace `Cudd_bddAnd()` realizuje operaci $Apply(And, \cdot, \cdot)$. Pod oběma cykly dojde konečně ke spojení obou ROBDD pomocí funkce `Cudd_bddOr()` realizující $Apply(Or, \cdot, \cdot)$. Všimněme si, že po každém vytvoření nové částečné nebo i konečné funkce je třeba zavolat metody `Cudd_Ref()` resp. `Cudd_RecursiveDeref()` inkrementující resp. dekrementující počet referencí na nový resp. starý vrchol diagramu částečné funkce. Každý program musí nakonec volat metodu `Cudd_Quit()`, která uvolní veškerou paměť použitou prostředím (vrcholy diagramů a další struktury).

4.3 Struktury DdNode a DdManager

Hlavními strukturami v CUDD jsou struktury vrcholu diagramu a prostředí (manažeru). Struktura vrcholu odpovídá přesně námi dříve uvedené struktuře `DdNode` v 2.5.1 (zde se osvětluje proč jsme v 2.5.1 použili jméno `DdNode` a daná jména proměnných struktury), a proto se jí nebudeme dále zabývat. Struktura prostředí `DdManager` je mnohem složitější (obsahuje více jak 120 proměnných), uveďme alespoň ty nejdůležitější:

1. `DdNode *one, *zero, *plusinfinity, *minusinfinity` - diagramy konstant $1, 0, +\infty, -\infty$, vrchol `one` je společný pro ADD, BDD i ZDD, vrchol `zero` je určen pro ADD a ZDD, u BDD se získá diagram nuly negací ukazatele `one` (technika negace hran).

```
#include "include/cudd.h"
int main()
{
    DdManager * manager;
    DdNode *    f1, *f2,*f, *tmp, *var;
    int         i;
    double      k;
    //inicializace prostredi (manazeru)
    manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS, 0);
    f1      = Cudd_ReadOne(manager); //vytvoreni ROBDD termu x1*x2*x3
    Cudd_Ref(f1);
    for ( i = 2; i >= 0; i-- ) {
        var = Cudd_bddIthVar(manager, i);
        tmp = Cudd_bddAnd(manager, var, f1);
        Cudd_Ref(tmp);
        Cudd_RecursiveDeref(manager,f1);
        f1  = tmp;
    }
    f2 = Cudd_ReadOne(manager);      //vytvoreni ROBDD termu x1'*x2'*x3'
    Cudd_Ref(f2);
    for( i = 2; i >= 0; i-- ) {
        var = Cudd_bddIthVar(manager, i);
        tmp = Cudd_bddAnd(manager, Cudd_Not(var), f2);
        Cudd_Ref(tmp);
        Cudd_RecursiveDeref(manager,f2);
        f2  = tmp;
    }
    f = Cudd_bddOr(manager, f1, f2); //sjednoceni obou ROBDD pomoci Or
    Cudd_Ref(f);
    Cudd_RecursiveDeref(manager, f1);
    Cudd_RecursiveDeref(manager, f2);
    //
    // Zde je mozne s f dale pracovat
    //
    Cudd_RecursiveDeref(manager, f);
    Cudd_Quit(manager);
}
```

Zdrojový kód 4.2: Vytváření ROBDD v CUDDu

2. `int *perm, *permZ, *invperm, *invpermZ` - permutace proměnných určující jejich aktuální uspořádání pro BDD a ADD (`perm`) resp. ZDD (`permZ`) a permutace k těmto inverzní (pro zpětné hledání co je na daném místě permutace za proměnnou).

3. `DdNode **memoryList, DdNode *nextFree` - ukazatel do posledního pole ukazatelů na vrcholy (`memoryList`) a ukazatel do paměti na místo v poli pro možného uložení nového vrcholu (`nextFree`). Vrcholy jsou ukládány do paměti jako jednoduše zřetězený seznam polí ukazatelů `DdNode *`, kde každé pole má konstantní délku. První čtyři byty každého pole vždy ukazují na pole předchozí (předchozí prvek seznamu). Ukazatel `DdNode *next` ve struktuře `DdNode` 2.5.1 ukazuje standardně na sousední místo v poli (pokuď je `DdNode` na konci pole, pak ukazuje `next` na místo za adresou do předchozího pole v následujícím poli). Dále kvůli použití negovaných hran je nutné zajistit, aby

adresa každého `DdNode` ležela na místě dělitelném dvěma (viz. poznámka 2.7). Protože je velikost struktury `DdNode` dělitelná dvěma, stačí zajistit uložení prvního `DdNode` v poli na adrese dělitelné dvěma (CUDD dokonce ukládá `DdNode` na adresy dělitelné velikostí `DdNode` - kvůli exportu do výstupních formátů, v nichž každý `DdNode` je reprezentován právě adresou dělenou jeho velikostí). Význam seznamu tkví v rychlém prohledávání, alokaci nebo dealokaci vrcholů `DdNode`.

4. `DdSubtable *subtables, *subtablesZ, constants` - ukazatele na pole unique tabulek pro BDD a ADD resp. ZDD a unique tabulka terminálů (konstant). Jak bylo řečeno v 2.6, pro každou proměnnou existuje právě jedna tabulka unikátních vrcholů, každá tabulka `DdSubtable` obsahuje ukazatel `DdNode **nodelist` někam do pole, které je prvkem seznamu polí ukazatelů (viz. předchozí bod - `DdNode **memoryList`), dále pak pomocné informace jako proměnné `unsigned int slots, keys, maxKeys, dead,...` uchovávající velikost, počet vrcholů, maximální počet vrcholů, počet mrtvých vrcholů (2.5.3),... v tabulce `nodelist`. Aby bylo hledání v unique tabulkách co nejrychlejší, jsou na unique tabulkách zavedeny relace $<$, $>$ a $==$, definované pro libovolné neterminální vrcholy `DdNode *u, *v`:

   (a) `u < v` jestliže `Cudd_Regular(u)->type.kids.T` $<$ `Cudd_Regular(v)->type.kids.T`

   (b) `u < v` jestliže `Cudd_Regular(u)->type.kids.T` $==$ `Cudd_Regular(v)->type.kids.T` a zároveň `Cudd_Regular(u)->type.kids.E` $<$ `Cudd_Regular(v)->type.kids.E`

   (c) `u == v` jestliže `Cudd_Regular(u)->type.kids.T` $==$ `Cudd_Regular(v)->type.kids.T` a zároveň `Cudd_Regular(u)->type.kids.E` $==$ `Cudd_Regular(v)->type.kids.E`

   (d) `u > v` pokud nenastává ani jeden z předchozích případů

   Vrcholy `DdNode * v` jsou do unique table ukládány tak, aby respektovali relaci $<$. Dochází tedy k přepojování ukazatelů `DdNode * next` ve struktuře `DdNode *` třeba i doprostřed jiných polí v seznamu polí (viz. předchozí bod - `DdNode **memoryList`), na nejmenší vrchol, který je větší ($>$) než `v`).

5. `unsigned int size, sizeZ, maxSize, maxSizeZ, slots, keys, keysZ, dead, deadZ, maxLive, minDead,...` - proměnné vztažené k zmíněným unique tabulkám. Uchovávají informace o počtu unique tabulek pro ADD a BDD resp. ZDD (počtu proměnných v prostředí) a jejich horní meze, celkovému počtu hash slotů, celkovému počtu BDD a ADD resp. ZDD vrcholů, dále počtu mrtvých vrcholů v BDD a ADD resp. ZDD, horní mezi počtu živých vrcholů a dolní mezi počtu mrtvých vrcholů pro spuštění garbage collection (2.6).

6. `DdCache *acache, *cache` - ukazatel na adresu resp. první položku pole prvků `DdCache` realizující computed table (2.9). Computed table představuje jedno pole struktur `DdCache`. `DdCache` obsahuje podle 2.9 položky klíče, tzn. operaci `<op>`, operandy `DdNode *f, *g` operace a výsledek operace `DdNode * data`. Položky `DdCache` jsou v poli `acache` uloženy tak, aby respektovali hashovací funkci.

7. `unsigned int cacheSlots; double cacheMisses, cacheHits, minHit` - proměnné vztažené ke computed table, obsahují postupně: počet slotů v computed table `DdCache *cache`, procentuální počet neúspěšných a úspěšných hledání v cache a dolní mez procentuálního množství úspěšných hledání v computed table, při němž dojde k úpravě velikosti (počtu slotů) computed table.

8. `Cudd_ReorderingType autoMethod, autoMethodZ,` a další globální proměnné algoritmů pro dynamické hledání vhodné permutace proměnných diagramů (2.5.10). `autoMethod` resp. `autoMethodZ` značí defaultní algoritmus pr hledání vhodné permutace.

9. `unsigned long memused, maxmem, maxmemhard; int garbageCollections; double totCacheHits, totCacheMiss; long GCTime, reordTime,...` - statistické proměnné uchovávající množství použité paměti, aktuálně stanovené maximální množství použité paměti, horní mez maximálního množství použité paměti, počet provedených garbage collections, celkové procentuální množství úspěšných a neúspěšných hledání v computed table, celkový čas strávený garbage collections, celkový čas strávený přeuspořádáváním všech diagramů, které je důsledkem změny permutace proměnných diagramů.

10. `FILE *out, *err` - souborové nebo konzolové proudy pro zápis procesních nebo chybových hlášení jednotlivých operací a algoritmů volaných v prostředí.

4.4 Tvorba minimalizátoru v CUDDu

Při vytváření minimalizátoru logických funkcí v CUDDu jsem vyšel z upravené verze CUDDu, která byla výsledkem Bakalářské práce Ondřeje Kolоše [17] (jde o port CUDDu ve verzi 2.4.1 pod OS Windows). Veškeré úpravy mají ve výsledném kódu formát (4.3):

```
// <BEG> Pavel Cerny modification
//
// Zde se nachází provedené změny
//
// <END> Pavel Cerny modification
```

Zdrojový kód 4.3: Značení provedených modifikací v CUDDu

Výjimky tvoří samostatné soubory `Cudd\cuddLoadPla.c` a `Cudd\cuddMini.c`, které v původním CUDDu 2.4.1 vůbec nebyli. Pro celý balík jsem vytvořil jednotný projekt (*solution*) ve Visual Studiu 2008 (studenti ČVUT FEL mají dostupnou MSDN-AA licenci tohoto vývojového nástroje). Projekt jsem rozdělil do čtyřech podprojektů (tak jako Ondřej Kolоš v [17]):

1. **Cudd** - Základní balík, ve vlastnostech `Properties\Configuration Properties\C++\Advanced\ Compile As` jsem nastavil kopilaci kódu pro jazyk C.
2. **Dddmp** - Funkce pro ukládání a načítání BDD, ADD, ZDD a formulí v CNF do a ze souborů v úsporném formátu. Opět jsem nastavil kompilaci kódu pro jazyk C.
3. **Nanotrav** - Funkce pro načítání booleovské síťe a následnou práci se sítí (tvorba tranzitivního uzávěru, hledání silně souvislých komponent, hledání maximální $0-1$ toku v síti, hledání nejkratších cest - více o těchto úlohách v [8]). Nastavena kompilace kódu pro jazyk C.
4. **CuddObj** - Objektově orientované rozhraní (C++) zastřešující tři uvedené podprojekty. Objektové rozhraní nabízí uživatelsky příjemnější práci s diagramy. Nastavena kompilace pro jazyk C++.

Zároveň ve všech čtyřech podprojektech byla ve vlastnosti `Properties\Configuration Properties \C++\Advanced\Disable Specific Warning` zrušena zobrazování některých varování při kompilaci (4311; 4312; 4273; 4996 - varování při přetypování či použití nebezpečných metod, při jejichž nesprávné manipulaci hrozí přepis nepříslušející paměti). Dále jsem v projektu (solution) přidal závislosti jednotlivých podprojektů `solution\Properties\Common Properties\Project Dependicies`, kde každému podprojektu jsem nastavil závislost (i-tý projekt je vždy závislý na prvním až $i-1$-ním podprojektu - v námi uvedeném pořadí). Poté již kompilace celého projektu proběhla až na pár varování v pořádku. Otestováním programu v `CuddObj.cpp` jsem ověřil, že portování proběhlo bez problému.

Při modifikacích podprojektů jsem pak postupoval v následujících čtyřech krocích:

4.4.1 Krok 1: Přidání diagramů MBD (rozšíření BDD)

Ve struktuře prostředí `DdManager` jsem zavedl terminální vrchol `DdNode * incomplete` neúplně určeného symbolu X algebry $\mathcal{B}_{xlog}$ a smysluplně jsem mu přiřadil hodnotu 0.5 (vrcholy `DdNode * zero` a `DdNode * one` mají hodnoty 0.0 resp. 1.0). V metodě `Cudd_Init()` inicializující nové prostředí (soubor `Cudd\cuddInit.c`) jsem pak přidal kód, který vrchol `incomplete` vytvoří (4.4):

```
// <BEG> Pavel Cerny modification
// unique je instance DdManager*, cuddUniqueConst() vraci novy konstantni vrchol
unique->incomplete = cuddUniqueConst(manager, 0.5);
if ( unique->incomplete == NULL )
return(0);
cuddRef(unique->incomplete);
// <END> Pavel Cerny modification
```

Zdrojový kód 4.4: Inicializace neúplně určeného terminálu X v `DdManager`u

Dále bylo nutné přidat metody pro *Apply*(`<op>`, $\cdot, \cdot$) pro různé operace `<op>` (abychom mohli načítat MBD a dále s nimi pracovat). Do souboru `Cudd\cuddBddIte.c`, který obsahuje operace *Apply* a *ITE* na BDD, jsem tedy přidal odpovídající operace nad MBD (přesněji ROMBD). Jde především o tři základní funkce:

1. `DdNode * cuddBddAndRecurIncomp(DdManager *manager, DdNode *f, DdNode *g)`
   Realizuje operaci *Apply*(*And*, $\cdot, \cdot$) přesně tak jak je uvedeno v A.1.
2. `DdNode * cuddBddXorRecurIncomp(DdManager *manager, DdNode *f, DdNode *g)`
   Realizuje operaci exkluzivivního logického součtu podle pravidel v 2.3 (analogie *Apply*(*And*, $\cdot, \cdot$)).
3. `DdNode * cuddBddIteRecurIncomp(DdManager *manager, DdNode *f, DdNode *g, DdNode *h)`
   Realizuje operátor ITE podle pravidel v 2.7 (opět částečná analogie s *Apply*(*And*, $\cdot, \cdot$)).

Z těchto tří funkcí jsou pak odvozeny další implementované funkce (dle DeMorganvých zákonů 1.2):

1. `Cudd_bddAndIncomp(DdManager *manager, DdNode *f, DdNode *g)`
2. `Cudd_bddOrIncomp(DdManager *manager, DdNode *f, DdNode *g)`
3. `Cudd_bddNandIncomp(DdManager *manager, DdNode *f, DdNode *g)`
4. `Cudd_bddNorIncomp(DdManager *manager, DdNode *f, DdNode *g)`
5. `Cudd_bddXorIncomp(DdManager *manager, DdNode *f, DdNode *g)`
6. `Cudd_bddXnorIncomp(DdManager *manager, DdNode *f, DdNode *g)`
7. `Cudd_bddIteIncomp(DdManager *manager, DdNode *f, DdNode *g, DdNode *h)`
8. `Cudd_bddAndLimitIncomp(DdManager *manager, DdNode *f, unsigned int limit)`

Význam všech funkcí je zřejmý z jejich názvu, funkce `Cudd_bddAndLimitIncomp` provádí *Apply*(*And*, $\cdot, \cdot$), kde v průběhu chodu kontroluje, aby počet alokovaných vrcholů operací *Apply* nepřekročil daný limit. Je-li limit překročen, vrací funkce ukazatel `DdNode *` s hodnotou NULL. Vzhledem k neplatnosti vztahu č.2 pro ROMBD v poznámce 2.10 bylo ještě nutné zavést statickou metodu `static void bddVarToCanonicalSimple-Incomp(DdManager *dd, DdNode **fp, DdNode **gp, DdNode **hp,...)` pro převod trojice (f, g, h) do kanonického tvaru pro hledání v computed table.

Poznámka 4.2 (**Deklarace nových funkcí a metod**]). Deklarace nově uvedených funkcí a metod se přidává do jenoho ze souborů `include\cudd.h` (jde-li o z vnějšku přístupnou funkci - začíná prefixem `Cudd_`) nebo `include\cuddInt.h` (jde-li o interní funkci, z vnějšku nepřístupnou - začíná prefixem `cudd`). Dále budeme v tichosti přepokládat, že jakékoliv nově definované funkce jsou podle těchto pravidel deklarovány.

4.4.2 Krok 2: Úprava metody načítání diagramů ze souborů formátu PLA

Ondřej Kološ ve své modifikaci CUDDu rozšířil CUDD o funkci `BddArray * Cudd_plaLoad()` (soubor `Cudd\cuddLoadPla.c`) načítání diagramů ze souborů formátu PLA (3.1). Funkce vrací ukazatel na strukturu `BddArray`, jejíž definice vypadala dříve následovně (4.5):

```
struct BddArray
{
    DdNode ** roots;   // ukazatel na koreny diagramu (vystupne vicehodnotova funkce)
    char **   inames;  // ukazatel na retezce jmen promennych
    char **   onames;  // ukazatel na retezce jmen funkci
                       // (jmen slozek vystupne vicehodnotove funkce)
    int       inputs;  // pocet vstupnich promennych funkce
    int       outputs; // pocet výstupnich slozek výstupne vicehodnotove funkce
};
```

Zdrojový kód 4.5: Zdrojový kód struktury `BddArray`

Význam jednotlivých proměnných v 4.5 je uveden v komentářích. O původní funkci `Cudd_plaLoad()` se lze více dočíst v [17]. Nevýhodou původní funkce byly dvě skutečnosti: funkce načítala pouze PLA typu `.type fd`, kde navíc v případě neúplně určené funkce v PLA sestrojila pouze ROBDD jejího ONsetu. Navíc funkce `Cudd_plaLoad()` obsahovala chybu, u PLA typu `.type fd` platí, že pro libovolný $\vec{x} \in \{0,1\}^{\texttt{PlaData->inputs}}$ který je v PLA pro funkci f popsán dvakrát, jednou $f(\vec{x}) = 1$ a podruhé $f(\vec{x}) = X$, se položí $f(\vec{x}) = X$, tedy bod $\vec{x}$ je zahrnut do DCsetu funkce f. Původní implementace však díky vytváření ROBDD funkce f pomocí `Cudd_bddOr()` položila vždy $f(\vec{x}) = 1$, tj. zahrnula $\vec{x}$ do ONsetu f.

Ve smyslu co největší univerzality funkce `Cudd_plaLoad()` jsem se rozhodl tuto funkci modifikovat následovně:

1. Aby bylo možné načítat neúplně určenou funkce buď jako jeden ROMBD nebo dvojicí ROBDD, přidal jsem funkci `Cudd_plaLoad()` dva vstupní parametry, a to vektory `int *specVec1` a `specVec2` zastupující vektory b_1 a b_2 v 2.5.9. Oba tříprvkové vektory mohou však mimo čísel 0 a 1 obsahovat i čísla 2, které značí, že pokud f nabývá v libovolném bodě $\vec{x}$ hodnotu $f(\vec{x})$ určující ve vektoru b_i pozici, na níž v b_i stojí číslo 2, pak v diagramu vektoru b_i je hodnota $f(\vec{x})$ reprezentována hodnotou neúplně určené konstanty X. Je na uživateli, aby vektory `specVec1` a `specVec2` specifikoval. Pokud tedy například chce uživatel načíst f v PLA jako jeden ROMBD, stačí položit `specVec1 = {0,1,2}` a `specVec2 = {0,0,0}`. V `Cudd_plaLoad()` je zavedena konvence, že pokud všechny tři složky `specVec1` mají stejné hodnoty, pak se diagramy vektoru nevytváří (to samé platí pro `specVec2`). Funkce `Cudd_plaLoad()` má deklaraci:

   ```
   BddArray * Cudd_plaLoad(DdManager *manager, FILE *fp, const int *specVec1,
                           const int *specVec2);
   ```

2. Vzhledem k předešlému bodu bylo nutné rozšířit strukturu `BddArray` o pole `DdNode ** roots2` případných kořenů druhých diagramů (je-li každá neúplně určená funkce v PLA reprezentována dvojicí ROBDD).

3. Při načítání se postupuje v krocích:

   (a) Nejprve se zavolá interní funkce `PlaData * cuddPlaFileLoad(FILE *fp)`, která načte soubor PLA do vnitřní struktury `PlaData` (4.6).

   (b) Po načtení PLA souboru do instance `PlaData` se v cyklu projdou všechny součinové termy a pro každý term T se v poli `DdNode ** terms` vytvoří diagram funkce g_T, pro niž platí: $g_T = T$ (diagram termu).

```
struct PlaData
{
    struct  PlaArray * iplaArray; // radek matice vstup. hodnot
    struct  PlaArray * oplaArray; // radek matice vystup. hodnot
    char **            inames;    // jmena promennych
    char **            onames;    // jmena funkci
    int                terms;     // pocet termu PLA
    int                inputs;    // pocet vstupnich promennych
    int                outputs;   // pocet funkci v PLA
    char               plaType;   // typ PLA (f,fd,fr,fdr)
};

struct PlaArray
{
    char *            line; //radek matice
    struct PlaArray * next; //ukazatel na strukturu dalsiho radku matice
};
```

Zdrojový kód 4.6: Zdrojový kód struktury `PlaData`

(c) Pro každou funkci f_i z PLA ($i \in \{1, 2, \ldots,$ `PlaData->outputs`$\}$) se v poli `DdNode ** vec[3]` na pozicích `vec[0][i]`, `vec[1][i]` a `vec[2][i]` vytvoří diagramy funkce f_i^{off}, f_i^{on} a f_i^{dc} OFF,ON a DCsetu funkce f_i (f_i^{off}, f_i^{on}, f_i^{dc} je pravdivá právě pro $\forall \vec{x} \in \{0,1\}^{\texttt{PlaData->inputs}}$ z $\mathcal{D}_{off}$, $\mathcal{D}_{on}$, $\mathcal{D}_{dc}$, všude jinde je f_i^{off}, f_i^{on}, f_i^{dc} nepravdivá). Diagramy OFF, ON a DCsetu f_i se vytváří v závislosti na typu PLA z diagramů v `DdNode ** terms` takto:

i. `.type f` - Uvažují se pouze termy T v `DdNode ** terms`, pro něž je $f_i(\vec{x}) = 1$, kdykoliv $T(\vec{x}) = 1$. Pomocí funkce `Cudd_bddOr()` se vytvoří diagram funkce ONsetu f_i^{on}, diagram f_i^{off} se získá $f_i^{off} = \overline{f_i^{on}}$, f_i^{dc} se položí identicky rovná nule ($f_i^{dc} \equiv 0$).

ii. `.type r` - Uvažují se pouze termy T, pro něž je $f_i(\vec{x}) = 0$, kdykoliv $T(\vec{x}) = 1$. Pomocí funkce `Cudd_bddOr()` se vytvoří diagram funkce OFFsetu f_i^{off}, diagram f_i^{on} se získá $f_i^{on} = \overline{f_i^{off}}$, f_i^{dc} se položí identicky rovnen nule ($f_i^{dc} \equiv 0$).

iii. `.type fd` - Uvažují se jednak termy T, pro něž je $f_i(\vec{x}) = X$, kdykoliv $T(\vec{x}) = 1$ (z těchto termů se operací `Cudd_bddOr()` vytvoří diagram funkce f_i^{dc}) a dále pak termy T, pro něž je $f_i(\vec{x}) = X$ nebo $f_i(\vec{x}) = 1$, kdykoliv $T(\vec{x}) = 1$ (z těchto termů se operací `Cudd_bddOr()` vytvoří diagram funkce $\overline{f_i^{off}}$ a následnou negací získá diagram funkce f_i^{off}). Diagram funkce f_i^{on} se pak získá jako negace diagramu funkce $f_i^{dc} + f_i^{off}$ (provedeme `Cudd_bddOr()` na diagramech f_i^{off} a f_i^{dc} a negujeme výsledek). Není obtížné si uvědomit, že takto vzniklé funkce f_i^{off}, f_i^{on} a f_i^{dc} splňují „privilegovanost DCsetu před ONsetem“, tedy při uvedení $f_i(\vec{x}) = 1$ a $f_i(\vec{x}) = X$ v PLA bude platit $f_i^{on}(\vec{x}) = 0$ a $f_i^{dc}(\vec{x}) = 1$.

iv. `.type fr` - Uvažují se jednak termy T, pro něž je $f_i(\vec{x}) = 0$, kdykoliv $T(\vec{x}) = 1$ (z těchto termů se operací `Cudd_bddOr()` vytvoří diagram funkce f_i^{off}) a dále pak termy T, pro něž je $f_i(\vec{x}) = 1$, kdykoliv $T(\vec{x}) = 1$ (z těchto termů se operací `Cudd_bddOr()` vytvoří diagram funkce f_i^{on}). Konečně diagram funkce f_i^{dc} se získá jako $f_i^{dc} = \overline{f_i^{off} + f_i^{on}}$ (operací `Cudd_bddOr()` a následnou negací). Odtud vyplývá, že je nutné, aby pro libovolný bod $\vec{x}$ nenastala v PLA dvojí definice $f_i(\vec{x}) = 0$ i $f_i(\vec{x}) = 1$ (v tomto případě je průnik ON a OFFsetu f_i neprázdný, což později může vést k nesprávné reprezentaci f_i). V průběhu vytváření nejsme tuto nekorektnost schopni podchytit (museli pro každý $\vec{x}$, kde $f_i(\vec{x}) = 1$ resp. $f_i(\vec{x}) = 0$ dohledávat, zda náhodou již dříve nebylo řečeno, že $f_i(\vec{x}) = 0$ resp.

$f_i(\vec{x}) = 1$).

v. `.type dr` - Uvažují se jednak termy T, pro něž je $f_i(\vec{x}) = X$, kdykoliv $T(\vec{x}) = 1$ (z těchto termů se operací `Cudd_bddOr()` vytvoří diagram funkce f_i^{dc}) a dále pak termy T, pro něž je $f_i(\vec{x}) = X$ nebo $f_i(\vec{x}) = 0$, kdykoliv $T(\vec{x}) = 1$ (z těchto termů se operací `Cudd_bddOr()` vytvoří diagram funkce $\overline{f_i^{on}}$ a následnou negací získá diagram funkce f_i^{on}). Diagram funkce f_i^{off} se pak získá jako negace diagramu funkce $f_i^{dc} + f_i^{on}$ (provedeme `Cudd_bddOr()` na diagramech f_i^{on} a f_i^{dc} a negujeme výsledek). Není obtížné si uvědomit, že takto vzniklé funkce f_i^{off}, f_i^{on} a f_i^{dc} splňují „privilegovanost DCsetu před OFFsetem“, tedy při uvedení $f_i(\vec{x}) = 0$ a $f_i(\vec{x}) = X$ v PLA bude platit $f_i^{off}(\vec{x}) = 0$ a $f_i^{dc}(\vec{x}) = 1$.

vi. `.type fdr` - V tomto případě probíhá konstrukce f_i^{on} , f_i^{off} a f_i^{dc} úplně stejně jako pro PLA typu `.type fd`. Stejně jako u `.type fd` platí i zde „privilegovanost DCsetu před ONsetem“ (viz. výše), navíc by měli součinové termy v PLA pokrýt celý definiční obor (tato podmínka také není při načítání kontrolována).

(d) Jakmile máme pro každou funkci f_i v PLA sestrojeny diagramy funkcí f_i^{off}, f_i^{on} a f_i^{dc}, je vytváření výsledných ROBDD (ROMBD) jednoduché. Pro obě pole `specVec1` a `specVec2` stačí brát v ohled hodnoty v `specVec1` a `specVec2` na jednotlivých pozicích. Pokud například `specVec1={0,1,2}`, pak jednoduše $f_{i1} = f_i^{on} + X.f_i^{dc}$. Pokud `specVec2={1,0,1}`, pak $f_{i2} = f_i^{on} + f_i^{dc}$ a podobně.

4. Nakonec je uvolněna veškerá pomocně použitá paměť (pole `DdNode ** terms, **vec[3],...`) a je vrácen výsledek prostřednictvím ukazatele na strukturu `BddArray`.

4.4.3 Krok 3: Tvorba minimalizátoru

V souboru `Cudd\cuddMini.c` je implementován „sériový“ skupinový minimalizátor popsaný v 3.9. Algoritmus při chodu ukládá minimalizované výrazy ve formátu PLA typu `type .fd` (tzn. i výsledek je typu `fd`, prvotní načítání pro konstrukci diagramů minimalizovaných funkcí probíha pomocí `Cudd_plaLoad()`, tzn. z PLA libovolného typu). Skutečná implementace se od 3.9 liší tím, že veškerá rekurze je simulována zásobníkem. Struktura položky zásobníku vypadá (4.7):

```
struct DdMinimizStackEntry
{
    DdNode * node;    //ukazatel na minimalizovany vrchol
    int      son[2];  //pole s klici pro potomka then a else
};
```

Zdrojový kód 4.7: Zdrojový kód struktury `DdMinimizPlaCommand`

Pole `int son[2]` uchovává kód potomka then nebo else. Kód může nabývat hodnot (deklarovány v `Cudd\cuddInt.h`):

1. `MINI_STACK_SON_NINIT` - Neinicializovaný potomek (větví jsme se dosud rekurzivně nepustili)
2. `MINI_STACK_SON_NCONST` - Potomek není terminál (konstantní)
3. `MINI_STACK_SON_1` - Potomek je terminál 1.
4. `MINI_STACK_SON_0` - Potomek je terminál 0.
5. `MINI_STACK_SON_X` - Potomek je terminál X.

Vzhledem k pojmům kapitoly 3.5 se v `Cudd\cuddMini.c` nachází funkce a metody:

1. `int Cudd_ddMinimization(DdManager *dd, const DdNode **roots, const unsigned int inputs, const unsigned int outputs, const DdMinimizPlaCommand * command, const char *output)`

   Realizace funkce `MinimizeMultipleOutputSerial()`(3.9), Parametry znamenají:

   - `DdNode *dd` - ukazatel na prostředí diagramů minimalizovaných funkcí (diagramy musí pocházet z jednoho prostředí).
   - `DdNode **roots` - ukazatel na pole kořenů diagramů (ROBDD či ROMBD) minimalizovaných funkcí.
   - `unsigned int *inputs` - počet proměnných minimalizovaných funkcí.
   - `unsigned int *outputs` - počet minimalizovaných funkcí.
   - `DdMinimizPlaCommand * command` - Ukazatel strukturu příkazu minimalizace, která byla dodefinována v `include\cudd.h` a vypadá (4.8):

```
struct DdMinimizPlaCommand
{
    char prefix[48];  //prefix minimalizacniho prikazu
    char middle[48];  //stred minimalizacniho prikazu
    char suffix[48];  //suffix minimalizacniho prikazu
};
```

   Zdrojový kód 4.8: Zdrojový kód struktury `DdMinimizPlaCommand`

   Příkaz minimalizace se totiž očekává ve formátu: `<prefix> <vstupni_soubor> <middle> <vystupni_soubor> <suffix>`. Je na uživateli, aby tento příkaz nastavil. V případě standardního použití ESPRESSA bude: `<prefix> = "espresso.exe"`, `<middle> = ">"`,`<suffix> = ""`, tj. příkaz vypadá: `"espresso.exe <vstupni_soubor> > <vystupni soubor>"`.

   Funkce vrací `TRUE`, pokud během minimalizace nenastala chyba (např. nedošla paměť nebo nebyl nalezen soubor s PLA funkce vrcholu), jinak vrací `FALSE`

   - `DdNode *output` - ukazatel na řetězec se jménem výstupního minimalizovaného PLA.

2. `int cuddDdMinimizitionInit(DdManager *dd,DdNode *roots, const unsigned int inputs, const unsigned int outputs, DdMinimizStackEntry *stack)`

   Realizace metody `MinimizationInit()` v 3.7. Význam parametrů (včetně návratové hodnoty) je stejný jako u `Cudd_ddMinimization()`, `DdMinimizStackEntry *stack` je zásobník rekurze.

3. `int cuddDdMiniGroupJoin(DdManager *dd, DdNode *roots, const unsigned int inputs, const unsigned int outputs, char *output)`

   Realizace funkce `JoinAllFormulas()` v 3.9. Význam parametrů (včetně návratové hodnoty) je stejný jako u `int Cudd_ddMinimization()`.

4. `int cuddDdMinimizationReadPla()`

   Funkce načtení PLA ze souboru (viz. `ReadFormulaFromFile()` v A.3), vstupní parametry zde neuvádíme, návratová hodnota značí, zda operace proběhla v pořádku.

5. `int cuddDdMinimizationWritePla()`

   Funkce uložení PLA do souboru (viz. `StoreFormulaToFile()` v A.3), vstupní parametry zde neuvádíme, návratová hodnota značí, zda operace proběhla v pořádku.

```
struct DdString
{
    int           length; //delka retezce
    DdSubstring * first;  //ukazatel na prvni strukturu podretezce
};

struct DdSubstring
{
    char *        data;  //ukazatel do pole znaku podreteze konstantni delky
    DdSubstring * next;  //ukazatel na strukturu dalsiho podretezce
};
```

Zdrojový kód 4.9: Zdrojový kód struktury `DdString`

Před načítáním PLA ze souboru předem nevíme, jako veliký PLA bude a protože v jazyku C neexistuje snadná manipulace s řetězci jako v C++, deklaroval jsem v `include\cuddInt.h` svoji strukturu řetězce variabilní délky `DdString`, ta má strukturu 4.9.

Pro práci s podřetězci jsou pak v `Cudd\cuddMini.c` definovány funkce: `cuddStringAddCharString()` (přičtení řetězce znaků k řetězci `DdString`), `cuddStringAddChar()` (n-násobné přičtení jednoho znaku k řetězci `DdString`), `cuddStringInit()` (inicializace `DdString`), `cuddStringFree()` (uvolnění paměti `DdString`). Podrobněji zde tyto funkce uvádět nebudu.

4.4.4 Krok 4: Rozšíření objektového rozhraní (C++) o MBD a metodu minimalizace

V oběktovém rozhraní v `obj\cuddObj.cpp` a `include\cuddObj.h` existuje následující hierarchie tříd:

1. `class DD`: základní třída pro všechny typy rozhodovacích diagramů
   (a) `class ABDD`: třída pro ADD a BDD, podtřída `class DD`
      i. `class ADD`: třída pro ADD, podtřída `class ABDD`
      ii. `class BDD`: třída pro BDD, podtřída `class ABDD`
      iii. `class MBD`: nově přidaná třída pro MBD, podtřída `class ABDD`
   (b) `class ZDD`: třída pro ZDD, podtřída `class DD`
2. `Cudd`: třída zapouzdřující strukturu `DdManager`
3. `ADDvector`: vektor ADD
4. `BDDvector`: vektor BDD
5. `ZDDvector`: vektor ZDD

Soubory `obj\cuddObj.cpp` a `include\cuddObj.h` jsem tedy rozšířil o třídu `class MBD` reprezentující libovolný ROMBD. Do této třídy jsem přidal některé funkce (obdobně jak je tomu u třídy `class BDD`) pro práci s ROMBD. Jde o funkce zapouzdřující nově přidané funkce v `Cudd\cuddBddIte.c` (viz. 4.4.1). Uvádět je zde nebudu. Třídu vektoru MBD verb—MBDvector— jsem již nezaváděl (uživatel si ji může dodatečně dodefinovat).

Kapitola 5

Testování a naměřené hodnoty

Veškeré testy jsem prováděl na notebooku Hewlett Packard s procesorem: Intel Core2Duo 1.8 GHz, operační pamětí 1024MB a pevným diskem s počtem 5400ot/min, v operačním systému Microsoft Windows XP with SP2. Testovací programy jsem kompiloval v MS Visual Studiu 2008, v konfiguraci Release (rychlostně optimalizovaný kód).

5.1 Načítání PLA ze souboru

Následující test sloužil k otestování rychlosti načítání diagramů funkcí z PLA souborů. Pro tento účel jsem si napsal testovací prográmek `testPlaLoad.exe`, který je na CD v adresáři `\exe`. Zdrojový kód programu je v `Cudd\Cudd_New.zip\src`. Syntaxe volání programu vypadá:

```
testPlaLoad.exe [cesta k PLA]
```

Program ve svém těle volá funkci `Cudd_plaLoad()`. Tabulka 5.1 uvádí naměřené výsledky:

soubor	**.i**	**.o**	**.p**	**čas načítání [ms]**
`10i-1o-100t-30dc.pla`	10	1	100	< 1
`15i-1o-200t-25dc.pla`	15	1	200	< 1
`20i-1o-200t-20dc.pla`	20	1	200	< 1
`100i-1o-400t-20dc.pla`	100	1	16671	39
`ibm.pla`	48	17	173	8
`soar.pla`	83	94	529	32
`ex4.pla`	128	28	620	40
`test2.pla`	11	35	2048	578
`test3.pla`	10	35	1024	258
`pdc.pla`	16	40	2810	240
`bb_250x5x200_20%_9.pla`	250	5	200	109
`bb_250x5x250_20%_9.pla`	250	5	250	133
`bb_250x5x300_20%_9.pla`	250	5	300	156

Zdrojový kód 5.1: Test načítání ROBDD a ROMBD z PLA

První čtyři soubory v tabulce jsem převzal z diplomové práce Jana Bílka ([7]), soubory `ibm.pla`, `soar.pla`, `ex4.pla`, `test2.pla`, `test3.pla`, `pdc.pla` z bakalářské práce Ondřeje Kološe ([17]) a zbylé tři instance z benchmarků pro minimalizátor BOOM-II na stránkách [6].

Z naměřených hodnot nelze triviálně vyvodit funkci závislosti času načítání PLA na počtu vstupů, výstupů a termů, s určitostí lze pouze říci, že doba načítání PLA závisí na všech těchto parametrech. Doba načítání PLA se oproti původní verzi načítání PLA ([17]) zvýšila $0 - 9x$ (v průměru z uvedených

instancí $4x$). Platíme tím daň za univerzalitu načítání (možnost načítat PLA v libovolném formátu. viz. 3c v 4.4.2).

5.2 Testování minimalizace

Dále jsem napsal jednoduchý prográmek `testMinimizer.exe` pro otestovaní minimalizace pomocí MBD (umístěn na CD v adresáři `\exe`, zdrojový kód v `Cudd\Cudd_New.zip\src`). Program funguje tak, že nejprve pomocí `Cudd_plaLoad()` načte z PLA diagramy mininimalizovaných funkcí a poté spustí minimalizační funkci `Cudd_ddMinimization()` v 4.4.3. Výsledek minimalizace je zapsán do souboru `res.pla`. Program se volá jednoduše:

```
testMinimizer.exe [cesta k PLA minimalizovaných funkcí]
```

Tabulka 5.2 uvádí naměřené hodnoty.

vstup minimalizace				**ESPRESSO**			**MBD**		
soubor	**.i**	**.o**	**.p**	**.p**	**čas[s]**	**ver.**	**.p**	**čas[s]**	**ver.**
`5i-1o-20t-15dc-20odc.pla`	5	1	20	1	0.041	y	1	0.005	y
`8i-1o-40t-5dc-50odc.pla`	8	1	40	15	0.047	y	15	0.5	y
`10i-1o-50t-0dc-20odc.pla`	10	1	50	34	0.047	y	34	1.3	y
`10i-1o-100t-30dc.pla`	10	1	100	62	0.078	y	62	3.7	y
`15i-1o-200t-25dc.pla`	15	1	200	196	0.094	y	196	40.5	y
`20i-1o-200t-20dc.pla`	20	1	200	200	0.125	y	200	48.5	y
`30i-1o-300t-0dc-20odc.pla`	30	1	300	246	0.156	y	246	9.8	y
`40i-1o-200t-0dc-50odc.pla`	40	1	200	44	0.063	y	44	1.8	y
`50i-1o-500t-10dc.pla`	50	1	500	1	1.9	n	500	406.2	y
`100i-1o-100t-20dc-alg.pla`	100	1	734	100	0.641	y	100	23.8	y
`ibm.pla`	48	17	173	173	0.078	y	173	18.4	y
`soar.pla`	83	94	529	353	0.656	y	353	16.8	y
`ex4.pla`	128	28	620	279	0.360	y	279	27.8	y
`test2.pla`	11	35	2048	1103	7.5	n	2019	206.7	n
`test3.pla`	10	35	1024	541	4.8	n	1008	106.3	n
`pdc.pla`	16	40	2810	145	0.922	n	227	18.2	y
`bb_50x5x50_20%_9.pla`	50	5	50	27	0.871	n	49	72.2	y
`bb_50x5x300_20%_9.pla`	50	5	300	154	160.1	n	295	485.2	y
`bb_100x5x50_20%_9.pla`	100	5	50	23	4.156	n	50	152.6	y
`bb_150x5x150_20%_9.pla`	150	5	150	56	164.2	n	148	664.7	y
`bb_250x5x200_20%_9.pla`	250	5	200	65	1165.8	n	193	1458.5	y

Zdrojový kód 5.2: Naměřené hodnoty „obyčejné“ a skupinové minimalizace

Položka **ver.** v 5.2 značí, zda proběhla v pořádku verifikace ESPRESSEM dosaženého výsledku. Ta se v ESPRESSU pouští příkazem:

```
espresso.exe -Dverify [cesta k minimalizovanemu PLA] [cesta k puvodnimu PLA]
```

Je zvláštní, že pro mnoho instancí neproběhla verifikace výsledků ESPRESSA v pořádku. Obecně k tomu došlo u PLA jiných typů než `type .fd`. Naopak verifikace proběhla v pořádku, pokud jsem ukládal výsledek do PLA stejného typu jako byl typ minimalizovaného PLA. Vzhledem k tomu ale, že výstup našeho minimalizátoru je vždy typu `type .fd`, nebyly dosažené výsledky porovnatelné. V manuálu ESPRESSA jsem o tomto problému nebo omezení nenašel žádnou zmínku. Přitom test na disjunktnost ON, OFF a DCsetu volaný

```
espresso.exe -Dcheck [cesta k PLA]
```

proběhl u všech problémových instancí v pořádku.

Porovnávejme tedy ty PLA, u nichž proběhla verifikace bez problému. U těchto instancí dosáhl náš minimalizátor stejných výsledků (co do počtu termů minimalizovaného PLA) jako ESPRESSO. V časové rovině počítalo ESPRESSO zhruba $25 - 388x$ rychleji a potvrdilo svůj primát. Naopak v porovnání s implementací Jana Bílka [7] je náš minimalizátor na shodně testovaných instancích $3 - 5x$ rychlejší.

vstup minimalizace				**Minimalizace pomocí MBD**		
soubor	**.i**	**.o**	**.p**	**inicializace[ms]**	**ext.minimalizace[s]**	**celkový čas[s]**
`5i-1o-20t-15dc-20odc.pla`	5	1	20	< 1	< 0.001	0.005
`8i-1o-40t-5dc-50odc.pla`	8	1	40	< 1	4.6	0.5
`10i-1o-50t-0dc-20odc.pla`	10	1	50	< 1	13.0	1.3
`10i-1o-100t-30dc.pla`	10	1	100	< 1	3.6	3.7
`15i-1o-200t-25dc.pla`	15	1	200	< 1	39.1	40.5
`20i-1o-200t-20dc.pla`	20	1	200	< 1	46.8	48.5
`30i-1o-300t-0dc-20odc.pla`	30	1	300	< 1	8.3	9.8
`40i-1o-200t-0dc-50odc.pla`	40	1	200	< 1	1.3	1.8
`50i-1o-500t-10dc.pla`	50	1	500	15	391.4	406.2
`100i-1o-100t-20dc-alg.pla`	100	1	734	< 1	20.8	23.8
`ibm.pla`	48	17	173	< 1	17.7	18.4
`soar.pla`	83	94	529	< 1	16.1	16.8
`ex4.pla`	128	28	620	< 1	26.8	27.8
`test2.pla`	11	35	2048	< 1	198.1	206.7
`test3.pla`	10	35	1024	< 1	102.1	106.3
`pdc.pla`	16	40	2810	< 1	17.4	18.2
`bb_50x5x50_20%_9.pla`	50	5	50	< 1	68.7	72.2
`bb_50x5x300_20%_9.pla`	50	5	300	< 1	465.2	485.2
`bb_100x5x50_20%_9.pla`	100	5	50	< 1	146.7	152.6
`bb_150x5x150_20%_9.pla`	150	5	150	< 1	635.7	664.7
`bb_250x5x200_20%_9.pla`	250	5	200	16	1400.0	1458.5

Zdrojový kód 5.3: Časové složitosti dílčích úkonů minimalizace pomocí MBD

V tabulce 5.3 je vidět, jakou část naší minimalizace zabírá inicializace vícenásobně referencovaných vrcholů (`cuddDdMinimizitionInit()` v 4.4.3) a dále pak čas strávený externí minimalizací. Ukazuje se, že inicializace (inteligentní procházení diagramů do hloubky) má bezvýznamnou časovou složitost. Naopak externí minimalizace pokrývá absolutní většinu času minimalizace. Pracuje tedy ESPRESSO celkově tak dlouho nebo je to způsobené neustálým načítáním a ukládáním PLA do souboru? První možnost je správná. Totiž v čase t_{dif} =**celkový čas** − **ext.minimalizace** je zahrnut čas načítání a spojování minimalizovaných PLA potomků vrcholu do jednoho PLA a následné ukládání PLA. Čas t_{dif} defacto odpovídá složitosti načítaní, spojování a ukládání (samotné procházení stromu má bezvýznamnou složitost a žádné další kroky se při minimalizaci nečiní). Těchto načtení, spojení a uložení je při minimalizaci alespoň polovina z celkového počtu volání externího minimalizátoru. Z 5.3 je však vidět, že $2.t_{dif}$ je $7 - 15x$ menší než čas externí minimalizace. Tímto můžeme odhadnout, že pouze $\frac{1}{16}$ až $\frac{1}{8}$ z celkového času externí minimalizace přísluší načítání a ukládání PLA ESPRESSEM (předpokládáme-li, že načítání a ukládání PLA v ESPRESSU probíhá stejně rychle jako v našem minimalizátoru).

Jistě by se tedy vyplatilo vyzkoušet místo ESPRESSA jiný, rychlejší externí minimalizátor (např. BOOM [5]). Z důvodu nedostatku času jsem byl bohužel nucen tento test vynechat (stejně tak i provést srovnání našeho minimalizátoru s BOOMem). Dále by bylo dobré provést test, který by zdokumentoval, kolik maximálně zabírají v průběhu naší minimalizace na disku dočasné PLA, tj. minimalizované PLA vícenásobně referencovaných vrcholů (v mých pozorováních to bylo nejvíce zhruba 4MB).

Kapitola 6

Závěr

Hlavní cíl práce byl splněn. Balík CUDD byl nejprve rozšířen pro práci s neúplně určenými logickými funkcemi (zavedením nového terminálu X, přidáním funkcí pro manipulaci s MBD a rozšířením objektového rozhraní o třídu MBD). Zároveň s tím byla upravena funkce načítání MBD (BDD) z PLA (funkce pochází z [17]), tak aby byla maximálně univerzální, tj. umožňovala načítat jak úplně, tak neúplně určené funkce do jednoho MBD či dvou BDD. Poté byl v CUDDu implementován skupinový minimalizátor neúplně určených logických funkcí pracující nad MBD. Minimalizátor byl úspěšně otestován na některých zkušebních obvodech a porovnán s pravděpodobně nejrozšířenějším minimalizátorem ESPRESSO. Oba minimalizátory dosáhly stejných výsledků (co do minimalizace), samotné ESPRESSO však ukázalo, že je několikanásobně rychlejší. Je otázkou, nakolik by se naměřené výsledky změnily, pokud bychom v našem minimalizátoru používali místo ESPRESSA, jakožto externě volaného minimalizátoru, jiný, rychlejší minimalizátor (např. BOOM) nebo dokonce minimalizátor „šitý minimalizaci nad MBD na míru“.

Algoritmus nově implementovaného minimalizátoru byl teoreticky popsán v textu práce (stejně jako některé klasické minimalizační metody). V textu byla dále formálně probrána elementární teorie o uspořádaných rozhodovacích diagramech (kanonicita RODD, horní mez paměťové složitosti RODD,...), speciálně pak o uspořádaných binárních a uspořádaných modifikovaných binárních rozhodovacích diagramech (složitosti ROBDD a ROMBD symetrických funkcí, funkcí celočísleného násobení,...). Bylo popsáno jak lze s ROBDD a ROMBD pracovat a jakým způsobem je implementovat v počítači. Stručně byla probrána teorie o reprezentaci neúplně určené funkce pomocí dvou ROBDD. Okrajově jsem se zmínil o problému řazení proměnných diagramu. Dále byla popsána struktura balíku CUDD, detailně pak provedené změny v CUDDu (pro potřeby minimalizace). Nakonec jsou v textu obsaženy výsledky provedených testů.

Případné budoucí vylepšení minimalizátoru nad MBD vidím v zakomponování externího minimalizátoru dovnitř algoritmu minimalizace nad MBD (nikoliv spouštět externí minimalizátor jako program). Při vhodné implementaci správy paměti bychom si tím ušetřili neustálé načítání a zápis „částečných“ PLA do souboru (pracovali bychom s PLA v operační paměti, pouze při nedostatku operační paměti bychom ukládali PLA na pevný disk - sofistikovaně například tak, že bychom ukládali na pevný disk PLA funkcí vrcholů, které budou čteny ve zbylém průběhu minimalizace nejméněkrát). Jak bylo řečeno, také by se určitě vyplatilo napsat externí minimalizátor „šitý minimalizaci nad MBD na míru“.

Všechny věci týkající se práce (programy, projekty a text) jsou na přiloženém CD, jehož struktura je popsána v příloze B.

Literatura

[1] Stroustrup B. *The C++ Programming Language.* Adison-Wesley, 1st edition, 1997.

[2] Papadimitriou Ch.H. *Computational Complexity.* Adison-Wesley, 1st edition, 1994.

[3] csg.csail.mit.edu/6.884/handouts/other/qm.pdf.

[4] Somenzi F. *CUDD: CU Decision Diagram Package, Release 2.4.1.* http://vlsi.colorado.edu/ fabio/CUDD/, 2001.

[5] Fišer P. Hlavička J. Boom - a boolean minimizer. *Research Report DC-2001*, 2001.

[6] http://service.felk.cvut.cz/vlsi/prj/BoomBench/.

[7] Bilek J. *Minimalizace neúplně určených logických funkcí pomocí modifikovaných binárních rozhodovacích diagramů.* Diplomová práce, ČVUT FEL, 2007.

[8] Kolář J. *Teoretická Informatika.* Vydavatelství ČVUT, 2nd edition, 2000.

[9] Velebil J. Diskrétní matematika a logika, 2006.

[10] McCluskey J.E. *Algebraic Minimization of Two-Terminal Contact Networks.* PhD thesis, 1956.

[11] Fišer P. Kubtov H. Two-level boolean minimizer boom-ii. *Proc. 6th Int. Workshop on Boolean Problems, Freiberg, (Germany)*, 2004.

[12] Mitchell M. *An Introduction to Genetic Algorithms.* MIT Press, McGraw-Hill, 2nd edition, 2001.

[13] Birkhoff G. Mac Lane S. *A Survey of Modern Algebra.* Macmillan Publishing, 4th edition, 1977.

[14] Aloul F.A. Markov I.L., Sakallah K.A. Mince: A static global variable-ordering for sat and bdd. *IWLS*, 2001.

[15] Aloul F.A. Markov I.L., Sakallah K.A. Force: A fast and easy to implement variable-ordering heuristic. *GLSVLSI*, 2003.

[16] Theobald T. Meinel C. Algorithms and data structures in vlsi design, 1998.

[17] Kološ O. *Port Programového Balíku CUDD Pod Windows.* Bakalářská práce, ČVUT FEL, 2006.

[18] Jankovský Z. Průcha L. *Diferenciální Počet I.* ČVUT, 1st edition, 2000.

[19] Diestel R. *Graph theory.* Springer-Verlag, 2nd edition, 2000.

[20] Bryant R.E. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions On Computers, C-35-8*, pages 677–691, 1986.

[21] Kernighan B.W Ritchie D.M. *The C Programming Language.* Prentice Hall, 1st edition, 1978.

[22] Rudell R.L. *Multiple-Valued Logic Minimization for PLA Synthesis.* PhD thesis, Berkley, University of California, 1986.

[23] Jacobi R.P. *A Study Of The Application Of Binary Decision Diagrams In Multilevel Logic Synthesis.* PhD thesis, Universitas Catholique De Louvain, 1993.

[24] Minato S. Binary decision diagrams and applications for vlsi cad, 1996.

[25] Friedman J.M. Supowit K.J. Finding optimal variable ordering for binary decision diagrams. *IEEE Transactions On Computers*, 1990.

[26] Rudin W. *Principles of Mathematical Analysis.* McGraw-Hill, 3rd edition, 1977.

Příloha A

Zdrojové kódy

A.1 Operace Apply(AND,f,g)

```
DdNode * ApplyAnd(
    DdManager * dd,  // prostredi ( muzeme pracovat ve vice prostredich )
    DdNode *    f,   // prvni argument AND
    DdNode *    g )  // druhy argument AND
{
    DdNode * t, *e, *r, *f0, *g0, *f1, *g1;
    DdNode * one  = Constant(dd, 1);  // one == 1
    DdNode * x    = Constant(dd, X);  // X == X
    DdNode * F    = Regular(f);       // pro pristup k polozkam f
    DdNode * G    = Regular(g);       // pro pristup k polozkam g
    int      fVarPos, gVarPos;        // pozice promenne korene f,g v prostredi

    // terminalni případy
    if ( F == G ) {     // f == g nebo f == g'
        if ( F == x ) return x;
        if ( f == g ) return f;
    }
    if ( F == one ) {                       // f == 1 nebo f == 0
        if ( f == one ) {
            return ( G == x ) ? x : g;      // predejdeme vraceni Not(x)
        }
        return f;
    }
    if ( G == one ) {   // g == 1 nebo g == 0
        if ( g == one ) {
         return ( F == x ) ? x : f;        // predejdeme vraceni Not(x)
        }
        return g;
    }
    ConvertToCanonicalForm(&f, &g); //(f,g) do kanon. tvaru pro hledani v comp. table
    r = ComputedTableLookup(dd, OP_AND, f, g); // zkus sesti v computed table
    if ( r != NULL )  return r;               // vysledek nalezen, vrat jej
```

```
    fVarPos = GetVariablePosition(dd, f);   // f == X -> umele fVarPos == infinity
    gVarPos = GetVariablePosition(dd, f);   // f == X -> umele fVarPos == infinity

    if ( fVarPos <= gVarPos ) {  // promenna korene f je vyse jak promenna korene g
        f1 = F->type.kids.T; f0 = F->type.kids.E;
        if ( IsNotRegular(f) ) {
           f1 = Not(f1); f0 = Not(f0);      // f == F', neguj potomky
        }
    } else {
        f1 = f0 = f;
    }
    if ( gVarPos <= fVarPos ) {  // promenna korene f je vyse jak promenna korene g
        g1 = G->type.kids.T; f0 = G->type.kids.E;
        if ( IsNotRegular(g) ) {
            g1 = Not(g1); g0 = Not(g0);     // g == G', neguj potomky
        }
    } else {
      g1 = g0 = g;
    }
    t = ApplyAnd(dd,f1,g1);  e = ApplyAnd(dd,f2,g2);  // rekurzivni reseni podproblemu
    if ( t == e ) // novy vrchol by byl zbytecny
      return t;
    else {
        if ( IsNotRegular(t) ) {            // t musi byt kvuli kanonicite regularni
          if ( e == x ) e = Not(e);         // e == X => e' = X (kanonicita)
          r = FindInOrAddToUniqueTable(dd, Not(t), Not(e));
        } else {
          r = FindInOrAddToUniqueTable(dd, t, e);   // e == Not(X) nenastava => kanonicita
        }
    }
    Regular(t)->ref++; Regular(e)->ref++;   // zvys reference na then a else vrcholy
    AddToComputedTable(OP_AND, f, g, r);    // uloz vysledek do computed table
    return r;                               // vrat r;
}
```

Zdrojový kód A.1: *Apply*(AND, f, g) pro ROMBD i ROBDD

A.2 Operace Restrict(f)

```
DdNode * Restrict(
    DdManager * dd,      // prostredi ( muzeme pracovat ve vice prostredich )
    DdNode *    f,       // funkce, na niz je Restrict aplikovan
    int         varPos,  // pozice promenne restrikce v prostredi
    double      val )    // hodnota promenne restrikce ( value = 0,1 )
{
    DdNode * t, *e, *r, *f0, *f1;
    DdNode * one = Constant(dd,1);   // one == 1
    DdNode * X   = Constant(dd,X);   // X == 0
```

```
    DdNode * F   = Regular(f);       // regularni tvar f
    int      fVarPos                 // pozice promenne korene f v prostredi

    if ( IsConstant(F) )             // f = 0,1,X -> terminujici pripad
        return ( F == one ) ? f : X; // kanonicka reprezentace pro f == Complement(X)
    fVarPos = GetVariablePosition(dd, f);
    if ( fVarPos > varPos ) // f nezavisi na promenne restrikce -> terminujici pripad
        return f;

    f1 = F->type.kids.T;  f0 = F->type.kids.E; // nastav potomky f
    if ( IsNotRegular(f) ) {
      f1 = Not(f1);  f1 = Not(f0);
    }
    if ( fVarPos == varPos ) { // koren f odpovida promenne restrikce
        return ( val == 1 ) ? f1 : f0;
    }
    // odted fVarPos < varPos
    r = ComputedTableLookup(dd, OP_RESTRICT, f ); // zkus sesti v computed table
    if ( r != NULL )  return r;                   // vysledek nalezen, vrat jej

    t = Restrict(dd, f1, varPos, val);  // rekurzivni reseni then vetve
    e = Restrict(dd, f0, varPos, val);  // rekurzivni reseni else vetve
    if ( t == e ) // novy vrchol by byl zbytecny
      return t;
    else {
        if ( IsNotRegular(t) ) {           // t musi byt kvuli kanonicite regularni
          if ( e == x ) e = Not(e);        // e == X => e' = X (kanonicita)
          r = FindInOrAddToUniqueTable(dd, Not(t), Not(e));
        } else {
          r = FindInOrAddToUniqueTable(dd, t, e);   // e == Not(X) nenastava => kanonicita
        }
    }
    Regular(t)->ref++; Regular(e)->ref++;   // zvys reference na then a else vrcholy
    return r;                               // vrat r;
}
```

Zdrojový kód A.2: Operace *Restrict* pro ROBDD (ROMBD)

A.3 Rekurzivní krok minimalizace pomocí ROBDD, ROMBD

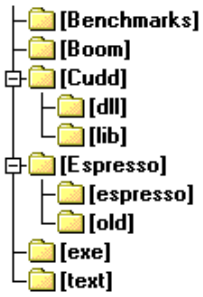
```
void MinimizeRecur( DdNode * f )
{
    DdNode   * then = Regular(f)->type.kids.T; // potomek vetve then
    DdNode   * else = Regular(f)->type.kids.E; // potomek vetve else
    DdNode   * act;                            // aktualni vrchol cyklu
    DdString * V0, *V1, *V, *Vmin;             // minimalni vyrazy potomku
```

```
    if ( IsNotRegular(f) ) {                    // neni-li f regularni
        then = Not(then); else = Not(else);     // negace potomku (neg. hrany)
    }
    for ( i = 0; i < 2; i++ ) {                     // cykli dvakrat pro then a else
        act = i==0 ? then : else;                   // aktualni jednou then, podruhe else
        V   = i==0 ? V1   : V0;                     // aktualni vyraz
        if ( IsConstant(act) ) {                    // je-li act terminalni
            V = TrivialMinimizationCase(act);       // trivialni pripad
        } else if ( Regular(act)->data != NULL ) { // neni-li data NULL
              if ( IsRegular(act) ) {               // act je regularni?
                  data->refsR++;                    // inkrementuj pocet navstev act
                  if( data->refsR > 0 ) {           // kdyz byl act jiz minimalizovan
                      V = ReadFormulaFromFile(act);// nacti min. vyraz act ze souboru
                      if ( data->refsR >= data->refsCntR ) // budeme jeste V potrebovat ?
                          DeleteFileFormula(act);  // nebudeme, smaz soubor s min. formuli
                  }
                  else {                            // act nebyl dosud minimalizovan
                      V = MinimizeRecur(act);       // minimalizuj act
                      if ( data->refsCntR > 1 )     // vyraz V budeme pozdeji potrebovat?
                          StoreFormulaToFile(act,V);//uloz V do souboru
                  }
              } else {                              // IsRegular(act) == FALSE
                  data->refsC++;                    // inkrementuj pocet navstev act
                  if( data->refsC > 0 ) {           // kdyz byl act jiz minimalizovan
                      V = ReadFormulaFromFile(act);// nacti min. vyraz act ze souboru
                      if ( data->refsC >= data->refsCntC ) // budeme jeste V potrebovat ?
                          DeleteFileFormula(act);  // nebudeme, smaz soubor s min. formuli
                  }
                  else {                            // act nebyl dosud minimalizovan
                      V = MinimizeRecur(act);       // minimalizuj act
                      if ( data->refsCntC > 1 )     // vyraz V budeme pozdeji potrebovat?
                          StoreFormulaToFile(act,V);//uloz V do souboru
                  }
              }
        } else {
            V = MinimizeRecur(then);                // jednonasobne referovany vrchol
        }
    }                                               // konec for cyklu
    V    = JoinMinimalSons(V1, V0);                 // sjednoceni V0 a V1 do V
    Vmin = ExetrnalMinimizationCall(V);             // externi minimalizace
    free(V0); free(V1); free(V);                    // uvolneni pameti
    return Vmin;                                    // navrat minimalniho vyrazu
}
```

Zdrojový kód A.3: Rekurzivní krok vylepšeného algoritmus minimalizace pomocí ROBDD, ROMBD

Příloha B

Obsah přiloženého CD



Zdrojový kód B.1: Struktura přiloženého CD

- `BENCHMARKS`
  - `benchmarks.zip` - Instance PLA (z práce Jana Bílka [7], Ondřeje Kološe [17] a pro BOOM [6])
- `BOOM`
  - `boom27.zip` - Minimalizátor Boom
- `CUDD`
  - `DLL` - Přeložené dynamické knihovny upravené verze CUDDu
  - `LIB` - Přeložené statické knihovny upravené verze CUDDu
  - `cudd-2.4.1.tar.gz` - Poslední dostupná oficiální verze CUDDu
  - `Cudd_New.zip` - Kompilovatelný projekt upravené verze CUDDu v MSVS 2008
- `ESPRESSO`
  - `ESPRESSO` - Zdrojové kódy, manuál a příklady použití poslední verze ESPRESSA
  - `OLD` - Zdrojové kódy starší verze ESPRESSA
  - `EspressoTest.zip` - Zdrojové kódy testovacího programu `testEspresso` (viz. níže)
  - `Espresso_New.zip` - Kompilovatelný projekt ESPRESSA v MSVS 2008
- `EXE` - Testovací programy (načítání diagramů - `testPlaLoad.exe`, minimalizaci pomocí MBD - `testMinimizator`, - minimalizaci ESPRESSEM s měřením času `testEspresso`
- `TEXT` - Zdrojový kód textu práce a výstup textu v pdf.