



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

BDD KNIŽNICA
A BDD LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

KAROL TROŠKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ONDŘEJ LENGÁL

BRNO 2013/2014

Abstrakt

Binární rozhodovací diagram je datová struktura využívaná v mnoha oblastech informatiky. Tato práce popisuje BDD jako matematický formalismus a navrhuje možnou reprezentaci BDD v počítači. Návrh je zaměřen především na rychlost snížením počtu alokací paměti a na jednoduchost a intuitivnost využívání knihovny. V práci je několik jednoduchých příkladů užití knihovny a výstrahy, kterým by se měl programátor při používání knihovny vyvarovat. Navržená reprezentace byla implementována v jazyce C.

Abstract

Binary decision program is a data structure used in many areas of information technology. This thesis describes BDD as a mathematical formalism and proposes possible representation of BDD in a computer. The propose is focused mainly on a reduction speed of number of memory allocation and on a simplicity and an intuitive system of using a library. There are several examples of a library usage and warnings which programmer should avoid of in the thesis. Proposed representation was implemented in C language.

Klíčová slova

Binární rozhodovací diagram, logická funkce, logická proměnná, výroková logika.

Keywords

Binary decision diagram, logic function, logic variable, propositional logic.

Citace

Karol Troška: BDD knihovna, bakalářská práce, Brno, FIT VUT v Brně, 2013/2014

BDD knihnica

Prohlášení

Prehlasujem, že túto bakalársku prácu som vypracoval sám pod vedením Ing. Ondřeje Lengála. Všetky informácie som čerpal z prednášok predmetu Formální analýza a verifikace na FIT VUT v Brně a literatury uvedenej v bibliografických citáciach.

.....

Karol Troška
20. května 2014

Poděkování

Touto cestou by som rád poďakoval predovšetkým môjmu vedúcemu BP Ing. Ondřejovi Lengálovi za poskytnuté materialy, inšpiráciu a hlavne trpezlivosť so mnou. Ďalej by som rád poďakoval rodine a všetkým blízkym za to, že ma pri písaní BP psychicky podržali.

© Karol Troška, 2013/2014.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1	Úvod	3
2	Výroková logika	4
2.1	Možnosti reprezentácie funkcií	5
2.2	Minimalizácia	5
2.3	Počet funkcií	5
3	Binárne rozhodovacie diagramy	7
3.1	História	7
3.2	Matematický popis	7
3.3	Typy BDD	8
3.3.1	Binárny rozhodovací strom	8
3.3.2	Redukované BDD	9
3.3.3	Redukované usporiadané BDD	10
3.4	Procedúra <i>Apply()</i>	12
3.5	Multiterminálne BDD	13
4	Dostupné balíky	15
4.1	CUDD	15
4.1.1	Štruktúra uzlu	15
4.1.2	Manager	16
4.1.3	Zhrnutie	16
4.2	BuDDy	16
4.2.1	Štruktúra uzlu	16
4.2.2	Zhrnutie	16
4.3	CacBDD	16
4.3.1	Štruktúra uzlu	17
4.3.2	Medzivýsledky	17
4.3.3	Zhrnutie	17
5	Návrh a implementácia	18
5.1	Dátové štruktúry	18
5.1.1	BDD uzol	18
5.1.2	Garbage collector	18
5.1.3	Mená premenných	19
5.1.4	Medzivýsledky – <i>Cache</i>	19
5.1.5	BDD manager	20
5.1.6	Výčet chybových stavov	20

5.2	Algoritmy	20
5.2.1	Inicializácia	20
5.2.2	Uvolňovanie	21
5.2.3	Vytvorenie uzlu	21
5.2.4	Práca s mezdivýsledkami – <i>Cacheovanie</i>	22
5.2.5	Práca s referenciami	22
5.2.6	Procedúra <i>Apply()</i>	23
5.2.7	Práca s MTBDD	23
5.2.8	Ukážky kódov	23
6	Testy	27
6.1	Uniky pamäti	27
7	Porovnanie s knižnicou CUDD	29
7.1	Inicializácia	29
7.2	Procedúra <i>apply</i>	29
7.3	Dĺžka programu	30
8	Záver	31
A	Obsah CD	33
B	Programátorské rozhranie	34

Kapitola 1

Úvod

V mnohých oblastiach informatiky je potrebné efektívne pracovať s funkciami nad binárnymi premennými. Môže ísť napríklad o efektívnu reprezentáciu hardwarových komponent pri návrhu logických obvodov, reprezentáciu veľkých matíc v matematických knihovniach alebo o efektívnu reprezentáciu stavového priestoru vo formálnej verifikácii. Existuje niekoľko prístupov pre reprezentáciu týchto funkcií, pričom jedným z najpoužívanejších sú binárne rozhodovacie diagramy (BDD).

BDD reprezentuje funkciu nad binárnymi premennými kompaktno pomocou koreňového acyklického orientovaného spojeného grafu. Nad týmto grafom sa dajú uplatňovať rôzne algoritmy, ktoré znižujú jeho pamäťovú aj časovú náročnosť.

BDD majú niektoré vlastosti, ktoré ho robia vhodnými kandidátmi na uplatnenie v uvedených oblastiach ako to napríklad reprezentácia viacerých funkcií iba jedným BDD, operácie nad kompaktným uložením a mnoho iných.

Účelom práce je navrhnúť a implementovať vlastnú knižnicu pre prácu s BDD. Práca demonštruje základné princípy algoritmov uplatnených nad BDD, možnosti reprezentácie BDD v počítači a súčasťou je taktiež porovnanie dosiahnutých výsledkov s knižnicou CUDD.

Práca obsahuje v kapitole 2 formálny popis výrokovkej logiky, v kapitole 3 typy BDD, prevod binárneho rozhodovacieho stromu BDT na BDD, základný princíp procedúry apply a popis MTBDD. V kapitole 4 popíšeme niektoré existujúce balíky pre prácu s BDD a MTBDD. Zvyšná časť práce obsahuje popis návrhu, implementácie a testovania knižnice. Záver práce patrí zrovnaniu výsledkov s knižnicou CUDD.

Kapitola 2

Výroková logika

Výroková logika skúma spôsoby tvorby zložených výrokov z jednoduchých a pravdivosť (nepravdivosť) zloženého výroku v závislosti od pravdivosti jednoduchých, z ktorých je zložený. Celá kapitola o výrokovej logike je prevzaná z [9].

Buď P neprázdna množina prvotných formúl, ktoré hrajú úlohu jednoduchých výrokov. Z jednoduchých výrokov získaváme zložené výroky spájaním pomocou logických spojok. Poznáme nasledujúce logické spojky:

- \neg negácia
- \vee disjunkcia
- \wedge konjunkcia
- \rightarrow implikácia
- \equiv ekvivalencia

Symbolmi jazyka L_P výrokovej logiky nad množinou P sú prvky množiny P , logické spojky a zátvorky $(,)$. Skladanie zložených výrokov sa riadi týmito pravidlami:

1. Každá prvotná formula $p \in P$ je výroková formula.
2. Pokiaľ sú A, B výrokové formule, potom $(\neg A), (A \vee B), (A \wedge B), (A \rightarrow B)$ a $(A \equiv B)$ sú výrokové formule.
3. Každá výroková formula je poskladaná konečným počtom symbolov jazyka L_P , ktorá vznikne podľa predchádzajúcich pravidiel.

Pravdivostné ohodnotenie prvotných formúl je ľubovoľné zobrazenie množiny P na hodnoty *true* a *false*. Pre jednoduchosť budeme značiť *true* 1 a *false* 0. Formálny zápis je $v : P \rightarrow \{0, 1\}$. Indukciou podľa zložitosti formule definujeme rozloženie w na zobrazenie v na množinu všetkých formúl jazyka L_P .

1. $w(p) = v(p)$ pre všetky $p \in P$
2. pokiaľ A, B sú výrokové formule, potom $w(\neg A), w(A \vee B), w(A \wedge B), w(A \rightarrow B)$ a $w(A \equiv B)$ v závislosti od $w(A)$ a $w(B)$ sa definujú podľa nasledujúcej tabuľky:

$w(A)$	$w(B)$	$w(\neg A)$	$w(A \vee B)$	$w(A \wedge B)$	$w(A \rightarrow B)$	$w(A \equiv B)$
0	0	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	1	0	0	0
1	1	0	1	1	1	1

2.1 Možnosti reprezentácie funkcií

Logické funkcie sa môžu reprezentovať rôznymi spôsobmi. Najčastejšie vyjadrenie je v podobe matematického vzorca za pomoci vyššie spomínaných operátorov a funkcií. Mať však v počítači funkciu uloženú ako textový reťazec veľmi komplikuje prácu s ňou, preto sa vynuli rôzne ďalšie reprezentácie. Operátory logických funkcií sa väčšinou prevádzajú po rade z \vee, \wedge, \neg na $+, *, !$.

Pravdivostná tabuľka Najprehľadnejší a najľahšie naučiteľný typ uchovávanania logickej funkcie, bohužiaľ obsahuje veľké množstvo redundancie a môže mať veľké pamäťové nároky (2^n , kde n je počet premenných funkcie).

Logický obvod Nie moc prehľadný ani ľahko naučiteľný formát, ale vynikajúci na demonštráciu pokročilých systémov zložených z logických funkcií.

BDD Formát v podobe acyklického koreňového grafu, ktorému sa budeme ďalej venovať v kapitole 3.3.

2.2 Minimalizácia

Minimalizácia logickej funkcie je proces znižovania pamäťových nárokov funkcie odstránením redundancie. V prípade matematického popisu môže ísť o uplatnenie rôznych redukčných pravidiel. Podrobnej minimalizácií BDD sa budeme venovať v kapitole 3.3.

2.3 Počet funkcií

Stavový priestor logickej funkcie závisí od počtu vstupných premenných a počtu hodnôt, ktoré každá premenná môže nadobudnúť. Ide o jednoduchú kombinatoriku, kde zisťujeme všetky variácie vstupných hodnôt, teda $p = n^k$, kde n je počet hodnôt, ktoré môže nadobúdať vstupná premenná, k počet vstupných premenných a p je výsledný počet možných vstupov funkcie.

Výsledný počet funkcií r je permutácia závisiaca od počtu možných vstupných kombinácií a počtu hodnôt, ktoré môže nadobúdať výstup funkcie, teda $r = q^p$, kde p je počet vstupných kombinácií a q je počet hodnôt, ktoré môže výstupná premenná nadobúdať. Z tohto dostávame vzorec

$$r = q^{n^k}. \quad (2.1)$$

Názornejšie vyjadrenie počtu logických funkcií je pomocou tabuľky pre dve vstupné premenné ($k=2$), dve vstupné hodnoty ($n=2$) a dve výstupné hodnoty ($q=2$) = $2^{2^2} = 16$ funkcií, viz nasledujúca tabuľka:

Vstup		Výstup																
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1

Kapitola 3

Binárne rozhodovacie diagramy

V tejto kapitole popíšeme históriu binárneho rozhodovacieho diagramu BDD, BDD ako matematický formalizmus a transformovanie BDD na redukované usporiadané BDD: ROBDD.

3.1 História

BDD vznikli z dôvodu úspornejšej reprezentácie logických funkcií. Reprezentácia je navrhnutá na základe Shanonovho expazného teorému[6]. Táto dátová štruktúra od vzniku v 1959[3] prešla mnohými úpravami na zníženie pamäťovej náročnosti a časovej náročnosti operácií.

Najvýznamnejším milníkom v oblasti redukcie pamäťovej náročnosti bola štúdia R. E. Bryanta[1] z Univerzity Carnegie Mellon. Táto štúdia sa týka najmä prevádzania operácií nad BDD v zminimalizovanej forme symbolicky. Toto sa dá pokladať za vznik ROBDD popísaných v sekcii 3.3.1.

3.2 Matematický popis

Táto časť je prevzatá z prednášky [8]. BDD je reprezentácia logickej funkcie $\{0, 1\}^k \rightarrow \{0, 1\}$, $k \geq 0$. Z pohľadu teórie grafov sa jedná o koreňový, orientovaný, spojitý acyklický graf. Formálna definícia BDD G nad množinou premenných M je

$$G = (N, T, var, low, high, root, val), \quad (3.1)$$

kde

- N je konečná množina neterminálnych uzlov,
- T je konečná množina terminálnych uzlov (listov), $N \cap T = \emptyset$,
- $var: N \rightarrow M$ je pomenovanie interných uzlov,
- $low, high: N \rightarrow N \cup T$ funkcia následníkov, definujúca vysokého a nízkeho následníka pre daný úzol $n \in N$ pre premennú $var(n)$ so vstupom 0 alebo 1,
 - graf je acyklický teda $\neg \exists n \in N. n(low \cup high)^+n$,
- $root \in N \cup T$ je koreň grafu, teda $\forall n \in (N \cup T) \setminus \{root\}. root(low \cup high)^+n$. Vstupné hrany do uzlú $root$ by porušili acyklicitu grafu,

- $Root(X)$ je funkcia vracajúca *root* grafu X ,
- $val: T \rightarrow \{0, 1\}$ je mapovacia funkcia hodnôt terminálnych uzlov.

3.3 Typy BDD

V tejto kapitole si popíšeme niektoré základné typy BDD, zdôrazníme rozdiely medzi nimi a objasníme, v čom sú niektoré efektívnejšie ako iné. Základné princípy budeme demonštrovať na funkcii

$$f(x_1, x_2, x_3) = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3). \quad (3.2)$$

V texte budeme používať nasledujúcu notáciu

- Premenné sa značia x_i , kde i je poradie premennej.
- Uzly sa značia písmenom n_{ij} , kde i je index premennej a j je poradové číslo uzlu označovaného premennou.

Príklad: n_{23} značí tretí uzol premennej x_2 .

3.3.1 Binárny rozhodovací strom

Binárny rozhodovací strom (binary decision tree, BDT) sa vytvára ako koreňový graf. Koreň tvorí jedna z premenných, väčšinou sa berie prvá premenná funkcie, teda v našom prípade x_1 . Vytvoríme teda uzol n_{11} , ktorý je mapovaný na premennú x_1 . Premenná môže nadobúdať dvoch hodnôt, pre ktoré sú definované nasledujúce uzly mapované na nasledujúcu premennú (uzly n_{21} a n_{22} mapované na premennú x_2). Takto sa postupne zanorujeme, až prejdeme cez všetky premenné a vygenerujeme všetky potrebné uzly. Hodnoty priradené za uzly mapované na poslednú premennú sú terminálne uzly. Počet generovaných uzlov je

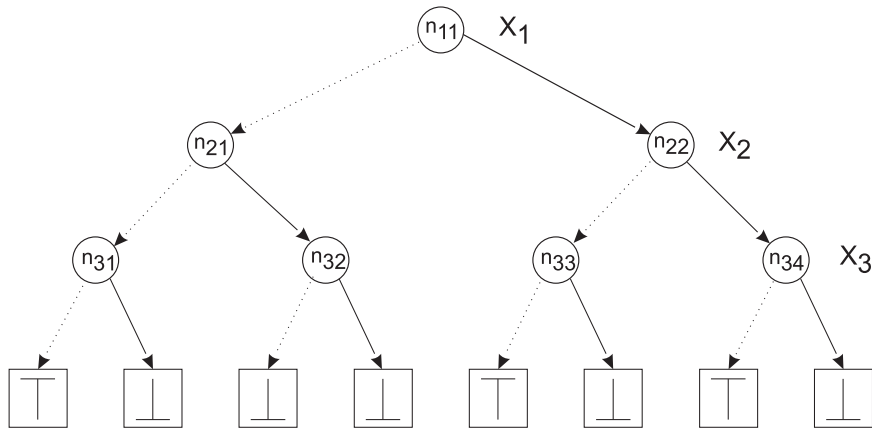
$$\sum_{i=0}^k 2^i = 2^{k+1} - 1, \quad (3.3)$$

kde k je počet premenných.

Grafické znázornenie funkcie f je na obrázku 3.1.

Notácia Pre orientáciu v grafe je použitá nasledujúca notácia

1. Neterminálne uzly sú
 - značené krúžkom,
 - hierarchicky usporiadané podľa poradia vyhodnocovania a
 - premenná ovplyvňujúca uzol je písaná vpravo.
2. Terminály sú
 - značené štvorčekom,
 - symbol \perp značí hodnotu *false* a
 - symbol \top značí hodnotu *true*.



Obrázek 3.1: Základné BDD

3. Hrany sú dvoch typov,

- plná čiara vedie k vysokému následníkovi daného uzlu $high(n)$ a
- prerušovaná čiara vedie k nízkemu následníkovi daného uzlu $low(n)$.

Ako vidíme, pre 3 premenné je počet generovaných uzlov 15, čo odpovedá $2^{3+1} - 1$. Zložitosť nájdenia daného terminálneho uzlu je lineárna vzhľadom k počtu premenných. Táto reprezentácia obsahuje mnoho redundancie, preto sa nepoužíva.

3.3.2 Redukované BDD

Počet generovaných uzlov sa dá výrazne obmedziť tým, že sa nebudú generovať zbytočné uzly. Pri pohľade na obrázok 3.1 vidíme redundanciu v terminálnych uzloch. Terminálne uzly môžeme zlúčiť do dvoch. Vzniká diagram s redukovaným počtom terminálnych uzlov, ktorý je znázornený na obrázku 3.2.

Ďalšiou redukciou je zlučovanie izomorfných podgrafov. Keď sa v BDD nachádzajú dva rôzne uzly, ktoré majú rovnaké (izomorfné) podgrafy, môžeme ich zlúčiť do jedného.

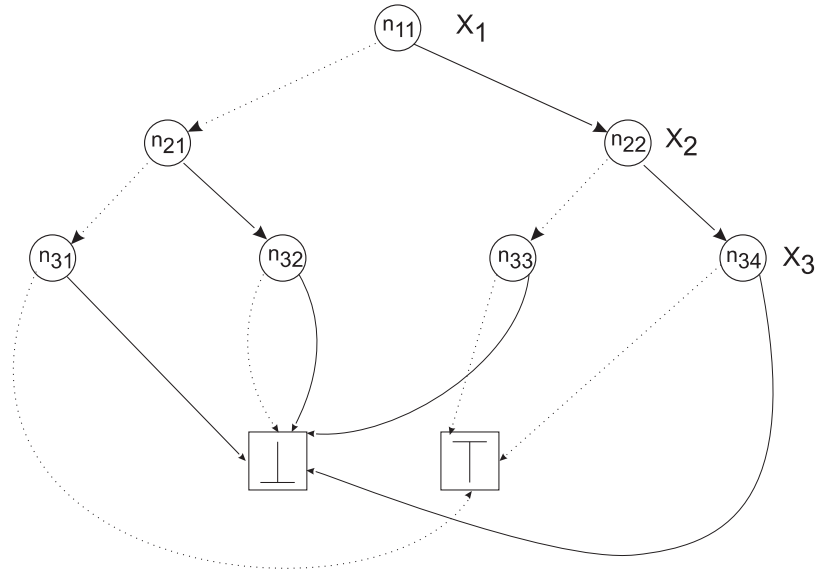
Graf je definovaný ako dvojica $G = (V, H)$, kde V je množina vrcholov a H je množina hrán tvaru (v_1, v_2) kde $v_1, v_2 \in V$, teda $H \subseteq V^2$.

Izomorfizmus dvoch grafov $G_1 = (V_1, H_1)$ a $G_2 = (V_2, H_2)$ je definovaný ako bijektívne zobrazenie $f : V_1 \rightarrow V_2$ a $g : H_1 \rightarrow H_2$ také, že ľubovoľnej hrane $h \in H_1$ sú priradené vrcholy $x, y \in V_1 \Leftrightarrow$ hrane $g(h) \in H_2$ sú priradené vrcholy $f(x), f(y) \in V_2$.

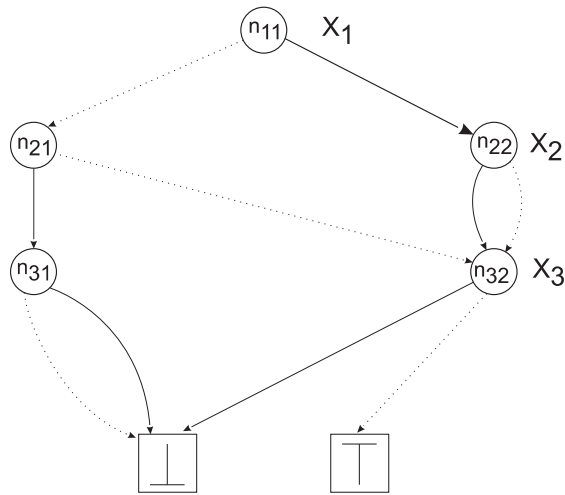
Pokiaľ grafy G_1 a G_2 sú izomorfné, teda existujú takéto funkcie f a g , potom píšeme $G_1 \cong G_2$.

Strom s odstránenými izomorfnými podstromami je na obrázku 3.3.

Ďalšiou redukciou je vynechávanie uzlov, kde $low(n) = high(n)$, pre $n \in N$. Ak sa vysoký aj nízky následník odkazujú na izomorfné podgrafy, logická funkcia v tomto bode nezávisí od danej premennej, preto sa môže tento uzol vynechať. Výsledný strom je na obrázku 3.4.



Obrázek 3.2: BDD s redukovaným počtom terminálnych uzlov



Obrázek 3.3: BDD s odstránenými izomorfnými podstromami

RBDD je klasické BDD, pre ktoré platia nasledujúce pravidlá:

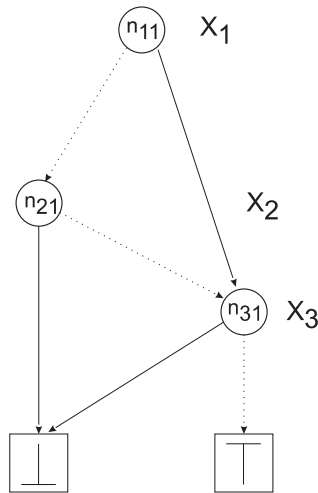
1. $\neg \exists n \in N. low(n) = high(n)$
2. $\neg \exists x_1, x_2 \in N \cup T. x_1 \neq x_2 \wedge Root(G_1) = x_1 \wedge root(G_2) = x_2 \wedge G_1 \otimes G_2$

3.3.3 Redukované usporiadané BDD

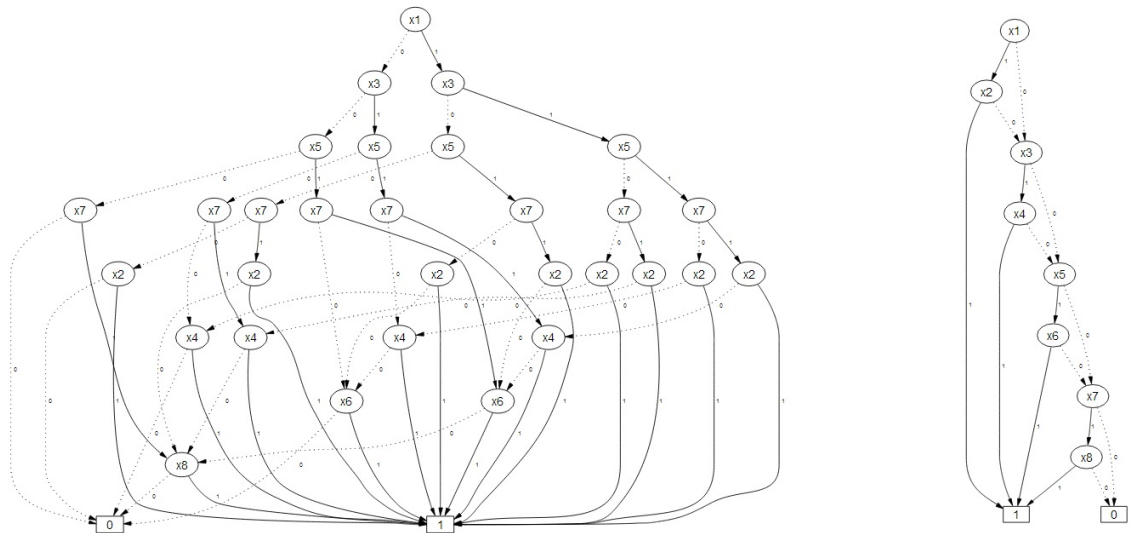
Poradia vyhodnocovania premenných, ktoré ovplyvňujú uzly, majú veľký vplyv na počet generovaných uzlov. Pre ilustráciu si to predvedme na funkcii

$$y(x_1, \dots, x_8) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6) \vee (x_7 \wedge x_8). \quad (3.4)$$

RBDD odpovedajúce danej funkcii pre dve rôzne usporiadania je na obrázku 3.5.



Obrázek 3.4: Výsledné minimalizované BDD



Obrázek 3.5: Rozdiel medzi zlým a dobrým usporiadaním[10]

Obe BDD sú minimalizované a popisujú rovnakú logickú funkciu. Z pravého obrázku však vidíme, že dobrým usporiadaním výrazne znížime počet generovaných uzlov. Na zistenia optimálneho usporiadania však neexistuje doposiaľ žiadny efektívny algoritmus pretože sa jedná o NP-náročný problém. Existujú iba rôzne heuristiky. Tie nám však môžu pomôcť, teda dostaneme výsledný strom s menším počtom uzlov, ale aj uškodiť. Tu je zoznam niektorých používaných algoritmov:

Náhodný algoritmus[4] Algoritmus sa používa v dvoch variantách a to varianta s pivotom a varianta bez pivota. Varianta bez pivota vyberá dve náhodné premenné v grafe a tie vymení postupným vymieňaním susedných uzlov. Výsledné usporiadanie premenných pre danú iteráciu je uloženie s najmenším počtom uzlov v priebehu iterácie. Varianta s pivotom pracuje na rovnakom princípe, ale pred samotným algoritmom sa vyberie pivot. To je premenná s najvyšším počtom uzlov. Dve náhodne zvolené premenné sú také, že jedna sa nachádza nad a druhá pod pivotom.

Sifting algoritmus [5] Teória je spočítať najvýhodnejšiu pozíciu v grafe pre jednu premennú s tým, že ostatné premenné zachovajú svoje poradie. Premennú posúvame najskôr smerom k terminálnym uzlom, potom smerom ku koreňu. Po príchode do koreňu vieme, v ktorej pozícii mala premenná najmenej uzlov a posunieme ju tam. Posun sa vykonáva na základe prehadzovania susedných premenných.

Iné používané algoritmy sú napríklad

- Symmetric sifting algoritmus
- Group sifting algoritmus
- Window permutation algoritmus
- Algoritmus využívajúci genetické algoritmy

Samotné spúšťanie preusporiadania premenných môže byť rôzne, napríklad dosiahnutie určitého počtu výpočetných krokov, alebo doposiaľ generovaný počet uzlov.

3.4 Procedúra *Apply()*

Procedúra *Apply()* aplikuje booleovskú funkciu na BDD. Obecne môže byť k -árna. V takom prípade dostáva procedúra k diagramov a funkciu, ktorú má nad diagramami vykonať. Výsledkom procedúry je jedno BDD.

Základný princíp funkcie je odvodený z Shannonovho expandného teorému, ktorý je popísaný v literatúre[6]. Ide o restrikciiu určitej premennej v grafe za konkrétnu hodnotu. Nech je n -parametrová funkcia $f(v_1, \dots, v_n)$. Vykonanie restrikciiu nad funkciou f , premennej v_i za hodnotu a dostávame funkciu $f(v_1, \dots, v_{i-1}, a, v_{i+1}, \dots, v_n)$, skrátene písané $f|_{v_i \leftarrow a}(v_1, \dots, v_n)$. Pokiaľ prevedieme restrikciiu na všetky premenné, výsledok konkrétneho dotazu nad funkciou už nie je rozhodovacia funkcia, ale konkrétna hodnota. Podľa Shannonovho teorému platí

$$f(v_1, \dots, v_n) = (\neg v_1 \wedge f|_{v_1 \leftarrow 0}(v_1, \dots, v_n)) \vee (v_1 \wedge f|_{v_1 \leftarrow 1}(v_1, \dots, v_n)).$$

Buď f_1, \dots, f_k k funkcií nad rovnakou doménou premenných a $op()$ k -árny operátor nad BDD. Operátor $op()$ vykonáva restrikciiu nad všetkými funkciami nasledovne:

$$op(f_1, \dots, f_k) = (\neg v \wedge op(f_1|_{v \leftarrow 0}, \dots, f_k|_{v \leftarrow 0})) \vee (v \wedge op(f_1|_{v \leftarrow 1}, \dots, f_k|_{v \leftarrow 1})).$$

Restrickia sa pri tvorbe BDD pomocou funkcie f využíva pre každý uzol n nasledovne:

- $low(n) = \neg var(n) \wedge f|_{n \leftarrow 0}$
- $high(n) = var(n) \wedge f|_{n \leftarrow 1}$
- $f(n) = high(n) \vee low(n)$

Postupným zanorovaním restrikciiu vzniká rekurzívny algoritmus popísaný v algoritme 1 pre binárne *apply*. Operátor $a \prec b$ značí, že premenná a sa nachádza bližšie ku koreňu ako premenná b .

Algoritmus 1: Apply

Input: $X, Y \in N \cup T, func: \{0, 1\}^n \rightarrow \{0, 1\}$

Output: *result* je výsledok funkcie $func(X, Y)$

```
1:  $xh = X, xl = X, yh = Y, yl = Y;$ 
2: if  $(X, Y \in T)$  then
3:   | return  $func(X, Y)$  ;
4: else
5:   | if  $(X \prec Y)$  then
6:     |  $result := var(X);$ 
7:     |  $xh = high.X, xl = low.X;$ 
8:   | else
9:     | if  $(Y \prec X)$  then
10:    |  $result := var(Y);$ 
11:    |  $yh = high.Y, yl = low.Y;$ 
12:    | else
13:    |  $result := var(X);$ 
14:    |  $xh = high.X, xl = low.X, yh = high.Y, yl = low.Y;$ 
15:    | end if
16:  | end if
17: end if
18:  $result.high = Apply(xh, yh, func);$ 
19:  $result.low = Apply(xl, yl, func);$ 
20: if  $(low.result = high.result)$  then
21:   |  $result = low.result;$ 
22: else
23:   | if  $(tmp := cache[result])$  then
24:     |  $uvolni\ result;$ 
25:     |  $result := tmp;$ 
26:   | end if
27: end if
28: return  $result;$ 
```

3.5 Multiterminálne BDD

Formálna definícia MTBDD je rovnaká ako definícia 3.1 pre BDD s rozdielom, že v MT-BDD množinu *val* definujeme takto:

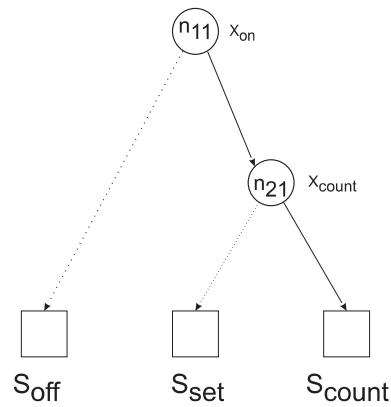
- $val: T \rightarrow \mathbb{D}$, kde \mathbb{D} je potenciálne nekonečná množina.

Multiterminálne BDD sú vo svojej podstate obyčajné ROBDD, ktoré majú väčší, obecné neobmedzený počet terminálnych uzlov. Ako príklad si môžeme uviesť kombinačný systém¹, ktorý sa môže nachádzať v stavoch S_{off} , S_{set} a S_{count} . Vstupné premenné sú X_{on} a X_{count} . Tabuľka systému je

¹Stav závisí iba na kombinácii vstupu

X_{on}	X_{count}	$S(X_{on}, X_{count})$
0	0	S_{off}
0	1	S_{off}
1	0	S_{set}
1	1	S_{count}

Na minimalizáciu MTBDD sa dajú použiť opäť použiť obe techniky, teda odstránenie izomorfných podstromov a uzlov, kde $high(x) = low(x)$. Podľa tabuľky jednoducho zostrojíme a zminimalizujeme BDD, ktoré však nemá iba 2 terminálne hodnoty ale 3.



Obrázek 3.6: Príklad MTBDD

Kapitola 4

Dostupné balíky

V tejto kapitole si popíšeme niektoré knižnice pracujúce s BDD. Bude popísaná štruktúra uzlu a spôsob práce s diagramami.

4.1 CUDD

Balíček bol vyvíjaný na univerzite v Coloradu. Už od roku 1996 stálym vývojom dosiahol vynikajúce pamäťové a výkonnostné výsledky. Podporuje prácu s BDD, diagramami s potlačenou nulou (ZDD) a aritmetickými rozhodovacími diagramami (ADD). ADD sú MTBDD, kde doménu \mathbb{D} tvoria čísla s plávajúcou rádovou čiarkou. V knižnici je taktiež implementovaná sada algoritmov na zlepšenie usporiadania premenných. Balíček sa môže používať 3 spôsobmi:

1. Black box – Umožňuje používať iba exportované funkcie popísané v nápovede.
2. Clean box – Umožňuje písanie zložitejších projektov, kde je vzhľadom k efektívnosti nutné pridať do balíka vlastné funkcie.
3. Interface – Balík disponuje kvalitne spracovaným rozhraním pre prácu s BDD ako objektami.

Podrobná definícia celého balíka CUDD je popísaná v literatúre[7].

4.1.1 Štruktúra uzlu

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

Štruktúra reprezentuje terminálne aj neterminálne uzly. Na zistenie, či konkrétny uzol je terminálny slúži makro `Cudd_IsConstant()`. Makro sa pozerá na index uzlu. Najvyšší index je rezervovaný pre terminálne uzly. V premennej `index` je uložený názov premennej, podľa

ktorej je pomenovaný uzol. Odráža poradie vzniku uzlov. V premennej `ref` je počítadlo referencií na uzol. Premenná `next` ukazuje na ďalší uzol v `unique table`. Ak ide o terminálny uzol, obsahuje premennú `value`, ktorá značí hodnotu uzlu, ináč obsahuje štruktúru `kids`, kde sú uložené ukazatele na vysokého a nízkeho následníka daného uzlu.

4.1.2 Manager

Všetky použité uzly sú uložené v špecialnej hashovacej tabuľke (`unique table`). BDD a ADD zdieľajú rovnakú tabuľku, ZDD má svoju vlastnú. Manager sa pred použitím inicializuje funkciou `Cudd.Init()` a po použití ruší funkciou `Cudd.Quit()`.

4.1.3 Zhrnutie

CUDD je univerzálny balíček pre prácu s BDD. Má kvalitne spracovanú dokumentáciu včetně užívateľského a programátorského manuálu. Výhodou je disponovanie širokou škálou funkcií pre radenie poradia premenných.

4.2 BuDDy

Táto knižnica bola vytvorená ako súčasť dizertačnej práce. Spočiatku bola určená len na demonštráciu princípov BDD. Vo svojej poslednej verzii je už však plnohodnotným balíkom so všetkými štandardnými operáciami, radením poradia premenných a dokumentáciou.

4.2.1 Štruktúra uzlu

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

Je potrebné uviesť, že BuDDy nepracuje s managerom ale s počtom uzlov, preto všetky premenné, ktoré odkazujú nejaký uzol sú reprezentované ako index do tohto poľa.

Premenná `refcou` obsahuje počet referencií vedúcich na uzol. Premenná `level` ukazuje pozíciu premennéj pri súčasťnom uložení. Premenné `high` a `low` reprezentujú hrany, `next` je ukazateľ na nasledujúci uzol. Premenná `hash` je ukazateľ na koreň stromu.

4.2.2 Zhrnutie

BuDDy je kompletný balík na prácu s BDD. Podporuje usporiadavanie premenných. BuDDy má veľmi kvalitne spracované C++ rozhranie, ktoré zefektívňuje prácu s týmto balíkom.

4.3 CacBDD

Jedná sa o knižnicu vyvíjanú v jazyku C++, ktorá je podobne ako BuDDy založená na indexovaní v poli. Podporuje obvyklé operátory využívané v modelcheckingu. CacBDD

má kvalitne spracované ukladanie medzivýsledkov, vďaka čomu dosahuje výborné časové výsledky.

Knižnica je veľmi jednoducho popísaná v literatúre[2]. Ja som potrebné informácie nachádzal priamo v zdrojových súboroch dostupných z <http://www.kaillesu.net/CacBDD/CacBDD.zip>.

4.3.1 Štruktúra uzlu

```
class DdNode{
    public:
        int var;
        int Then;
        int Else;
        int Next;

        // metody
}
```

Jeden uzol je uložený v triede `DdNode`. Táto štruktúra je uložená v poli, preto sú odkazy typu `int` a reprezentujú index do tohto poľa. `var` značí index premennej, `Then` a `Else` sú indexy následníkov a `Next` obsahuje index nasledujúcej voľnej premennej.

Súčasťou triedy sú aj metody na vytvorenie objektu, jeho zrušenie a nastavenie hodnôt.

4.3.2 Medzivýsledky

V `CacBDD` slúži na ukladanie medzivýsledkov štruktúra `XCTable`, ktorá sa dynamicky mení v závislosti od tzv. `hit-rate`, čož je pomer zásahov do tabuľky k celkovému počtu dotazov nad tabuľkou. `hit-rate` sa dynamicky so zmenami v tabuľke mení, čím dochádza k zmene veľkosti tabuľky.

4.3.3 Zhrnutie

Balík `CacBDD` je ďalší kvalitne spracovaný balík pre prácu s BDD. Využíva indexový prístup k uzlom. Návrhári balíku sa zamerali hlavne na jeho rýchlosť, ktorá je dosť vysoká vzhľadom k implementácii dynamického ukladania medzivýsledkov. V literatúre[2] je možné nájsť aj rýchlostné zrovnanie s balíkom `CUDD`.

Kapitola 5

Návrh a implementácia

V tejto kapitole popíšem svoj návrh implementácie jednotlivých dátových štruktúr a algoritmov. Knižnica je implementovaná v jazyku C a prekladaná pomocou prekladača `gcc` s prepínačmi `-std=c99`, `-Wall`, `-Wextra` a `-pedantic`.

5.1 Dátové štruktúry

V tejto časti sa zoznámime s návrhom jednotlivých dátových štruktúr, ako sú navzájom previazané a ktorá položka štruktúry na čo slúži.

5.1.1 BDD uzol

Základnou jednotkou každého grafu je uzol. Môj návrh uzlu obsahuje všetky dôležité premenné pre úplnú prácu s ním.

```
typedef struct BddNode{
    struct BddNode *high, *low;
    unsigned int var;
    unsigned int ref;
    struct BddNode *nextFree;
}tBddNode;
```

Premenné `high` a `low` sú ukazatele na vysokého a nízkeho následníka uzlu. V prípade, že ide o terminálny uzol, `high` a `low` sú nastavené na `NULL`. Na zistenie, či je daný uzol `t` terminálny sa používa makro `isTerminal(t)`. Premenná `var` je index do tabuľky mien premenných, ktorú si detailnejšie popíšeme v sekcii 5.1.3. Premenné `ref` a `nextFree` sú premenné, ktoré využíva garbage collector. `ref` je počítadlo referencií, ktoré je dôležité pri zdieľaných uzloch a `nextFree` je ukazateľ, ktorý previazáva zoznam voľných premenných ktoré má k dispozícii manager. Manageru sa budeme viac venovať v časti 5.1.5.

5.1.2 Garbage collector

Garbage collector má jednoduchú štruktúru.

```
typedef struct Garbage{
    tBddNode *nodes;
    struct Garbage *next;
}tGarbage;
```

Premenná `nodes` ukazuje na pole všetkých uzlov a premenná `next` na pola ďalšieho kusu pamäti, ktorý sa alokuje v prípade, že manager nemá voľné uzly.

5.1.3 Mená premenných

Je indexované pole reťazcov. Štruktúra je taktiež veľmi jednoduchá.

```
typedef struct Labels {
    char **lab;
    unsigned int count;
} tLabels;
```

Premenná `lab` je odkaz na pole reťazcov¹ a `count` je premenná označujúca počet reťazcov, používaná hlavne pri realokácií prvej úrovne `lab`.

5.1.4 Medzivýsledky – *Cache*

Medzivýsledky znižujú pamäťové nároky na tvorbu BDD a zjednodušujú zdielanie podgrafov. Štruktúra je nasledujúca. Implementácia medzivýsledkov je ako hashovacia tabuľka, ktorej kľúč pre daný uzol sa spočíta z trojice [*high*, *low*, *var*] daného uzlu. Dátová štruktúra nesúca informáciu tj. trojicu [*high*, *low*, *var*] a ukazateľ na uzol obsahujúci túto trojicu je nasledovná.

```
typedef struct CacheItem {
    unsigned int var;
    tBddNode *high, *low, *address;
    struct CacheItem * next;
} tCacheItem;
```

Premenné `high`, `low`, `var` a `address` nesú informáciu o uzle. Premenná `next` preväzuje uzly. V prípade že je uzol voľný tvorí zoznam so všetkými voľnými uzlami, v prípade že je uzol používaný, tvorí zoznam v hashovacej tabuľke. Táto štruktúra, podobne ako štruktúra `tNodes` sa alokuje ako pole, ktoré je uložené v bloku `tCacheGar`. Štruktúra je nasledujúca.

```
typedef struct CacheGar {
    tCacheItem * nodes;
    struct CacheGar * next;
} tCacheGar;
```

Premenná `nodes` ukazuje na pole odkiaľ sa priradzujú cache záznamy. `next` zabezpečuje previazanie týchto štruktúr do zoznamu. Ďalší blok štruktúry sa alokuje v prípade, že počet voľných cache záznamov je 0. Toto zistíme zo zastrešujúcej štruktúry `tCache`.

```
typedef struct Cache {
    tCacheGar * gar;
    tCacheItem * free;
    tCacheItem ** hash;
} tCache;
```

Nosná štruktúra, ktorá má nastarosť celý managment medzivýsledkov. `gar` je zoznam všetkých záznamov, ktoré môžeme použiť. `free` sú previazané zatiaľ nepoužité uzly. `hash` je odkaz do hashovacej tabuľky, do ktorej sa ukadajú záznamy medzivýsledkov.

¹Reťazcom je myslený typ `char *`.

5.1.5 BDD manager

Zastrešujúca štruktúra, riadiaca celú tvorbu a úpravy BDD.

```
typedef struct Manager{
    tGarbage *nodes;
    tBddNode *free;
    tLabels *variables;
    tLabels *terminals;
    tCache *cache;
}tManager;
```

`nodes` je ukazateľ na štruktúru garbage collectoru, ktorá ma nastarosti uvoľňovanie a vytváranie nových zhukov uzlov. `free` je ukazateľ na prvý voľný uzol. Ide opäť o zoznamovú implementáciu, kde sú všetky uzly prepojené pomocou `nextFree`, popísaného v časti 5.1.1. `cache` je štruktúra starajúca sa o vytváranie medzivýsledkov. `variables` a `terminals` sú tabuľky mien uzlov.

5.1.6 Výčet chybových stavov

Pri vykonávaní programu môže pri niektorých operáciách dôjsť k výskytu chyby ako je napríklad nedostatok pamäti. Tieto chybové stavy sú zapúzdrené vo výčtovom type `tError`. Balík disponuje makrom

```
bddThrowError(
    tError    e,
    tManager * mgr),
```

ktoré v prípade vzniknutej chyby ukončuje program s odpovedajúcou správou pre užívateľa. Makro vytlačí odpovedajúcu chybovú hlášku k chybe predanej parametrom `e`, uvoľní prostriedky manažera predaného parametrom `mgr` a ukončuje program chybovým kódom `e`. Knížnica nepoužíva makro sama, ale chybu propaguje vyššie. Ošetrenie týchto chýb má rieši užívateľ.

5.2 Algoritmy

Hlavné užívateľské rozhranie tvoria funkcie `bddInit()`, ktorá inicializuje všetky komponenty pre manager, `bddDestroy()`, ktorá uvoľňuje všetky prostriedky, ktoré daný manager alokoval, `bddCreateNode()` a `bddCreateTerminal()`, ktoré vytvoria uzol pre danú premennú alebo terminál a funkcia `bddApply()`.

5.2.1 Inicializácia

Žiadnu z operácií nie je možné prevádzať nad neinicializovaným managerom. Na inicializáciu slúži funkcia

```
bddInit(
    tManager    * mgr,
    unsigned int size),
```

ktorá postupne inicializuje všetky prvky premennej `mgr`. Najdôležitejšiu časťou je inicializácia premennej `nodes`, ktorá vytvorí pole uzlov podľa zadanej veľkosti. Všetky tieto uzly sú prístupné zo zoznamu v premennej `free`. Automaticky sa generujú dva uzly `bddFalse` a `bddTrue` prístupné na globálnej úrovni. Funkcia zapúzdruje inicializáciu všetkých vlastných premenných.

Parameter `size` je veľkosť na akú sa manager inicializuje. V balíku sú 3 preddefinované konštanty a to `BDD_SMALL`, `BDD_MEDIUM` a `BDD_LARGE`.

5.2.2 Uvolňovanie

Na uvolnenie slúži funkcia `bddDestroy(tManager *mgr)`, ktorá postupne zruší všetky alokované štruktúry. Je neprípustné ukončiť program pred zavolaním tejto funkcie, preto že by dochádzalo k úniku pamäti.

Postupne sa uvoľňujú prostriedky garbage collector, tj. zrušia sa uzly a potom sa dealokuje štruktúra. Je potreba dávať pozor na to, že garbage collector je zoznam, teda je potrebné ho uvoľniť korektne celý a nie iba prvý blok pamäti. Cache tvorí taktiež zoznam, preto sa musí uvoľňovať postupne. Uvolnenie štruktúry návestí je iba volanie funkcie `free()` nad premennou štruktúry `lab`. Ďalej sa už uvoľňuje iba samotná štruktúra.

Premenná `free` je iba pomocný ukazateľ, teda sa neuvoľňuje. Miesto ktoré odkazoval bolo uvoľnené pri uvoľňovaní premennej `nodes`.

5.2.3 Vytvorenie uzlu

Na vytváranie uzlov sa používa funkcia

```
bddCreateNode(  
    tManager * mgr,  
    char      * label,  
    tBddNode * high,  
    tBddNode * low,  
    tBddNode ** res).
```

Parametry funkcie sú `mgr` čož je manager nad ktorým pracujeme, `label` je meno vytváraného uzlu, ktoré sa ukladá do premennej manageru `variables`, `high` a `low` sú ukazatele na potomkov a `res` je výstupný parameter, cez ktorý vraciame ukazateľ na uzol. Návrátová hodnota funkcie je chybový kód v prípade, že nám došla pamäť. Funkcia nastavuje uzlu všetky jeho informácie a to takto:

1. Nastavíme premennú `var` na index v tabuľke `variables`.
2. Prehľadáme medzivýsledky, či takýto uzol už neexistuje.
 - Ak existuje, vraciame ukazateľ z medzivýsledkov.
 - Ak neexistuje, pokračujeme.
3. Počítadlo referencií `ref` nastavíme na 1.
4. Označíme uzol ako používaný nastavením `nextFree` na `NULL`.
5. Vysokému následníkovi uzlu `high` priradíme hodnotu parametru `high`.
6. Nízkeho následníkovi uzlu `low` priradíme hodnotu parametru `low`.

V prípade nového terminálneho uzla sa používa funkcia

```
bddCreateTerminal(  
    tManager * mgr,  
    char      * label,  
    tBddNode ** res).
```

Parametre reprezentujú to isté ako vo funkcií `bddCreateNode()`. Sú tu iba nepatrné zmeny v implementácií:

- Návestie sa vkladá do tabuľky `terminals`.
- Referencie sa na terminálnych uzloch nepočítajú.
- Do `high` a `low` priradíme `NULL`.

5.2.4 Práca s mezdivýsledkami – *Cacheovanie*

Každý vytvorený uzol (s výnimkou terminálnych) sa ukladá do štruktúry popísanej v časti 5.1.4. Ukladajú sa ako trojica vysokého následníka, nízkeho následníka a návestia premennej. Ďalej sa ukladá adresa tohto uzlu. Pri vytváraní uzlu najskôr skontrolujeme mezdivýsledky pomocou funkcie

```
cacheCheck(  
    tCache      * c,  
    tBddNode    * high,  
    tBddNode    * low,  
    unsigned int var).
```

Ak daný uzol už existuje, funkcia vracia ukazateľ na uzol. V prípade že uzol ešte neexistuje, vracia sa `NULL`. V tomto prípade sa uzol vkladá do cache pomocou funkcie `cacheInsert()`.

5.2.5 Práca s referenciami

Pre prácu s referenciami sa používajú dve funkcie a to

```
nodeIncRef(  
    tBddNode * node)
```

a

```
nodeDecRef(  
    tManager * mgr,  
    tBddNode * node).
```

Funkcia `nodeIncRef()` zvyšuje počítadlo referencií na uzol. Volá sa implicitne pri vytváraní uzlu za predpokladu, že uzol už existuje a bol najdený v mezdivýsledkoch.

Funkcia `nodeDecRef()` naopak znižuje počítadlo referencií. Programátor ju musí volať explicitne nad každým ukazateľom na graf, ktorý už nemieni používať. V prípade, že počítadlo referencií daného uzlu klesne na 0, uzol je automaticky uvoľnený z používaných uzlov aj z mezdivýsledkov. Funkcia sa rekurzívne volá aj na potomkov. Takýmto spôsobom sa dajú rušiť celé zhluky uzlov.

5.2.6 Procedúra *Apply()*

Základný princíp procedúry bol už popísaný v časti 3.4. Implementáciu v knižnici tvoria dve funkcie. Prvá je

```
bddApply(  
    tManager * mgr,  
    tBddNode * x,  
    tBddNode * y,  
    tBddNode * (*func) (tBddNode *, tBddNode *)),
```

ktorá sa volá z užívateľského programu. Ako parameter dostáva dva ukazatele na grafy (*x* a *y*), manager, v ktorom sa má procedúra vykonať (*mgr*) a odkaz na funkciu, ktorá sa má vykonať na terminálnych uzloch (*func*). Funkcia spúšťa rekurzívnu funkciu `_apply()` s rovnakými parametrami. `_apply()` sa rekurzívne zanočuje podľa algoritmu 1 popísaného v časti 3.4.

Pri vytváraní nových uzlov v procedúre sa používa funkcia

```
bddNewNode(  
    tManager * mgr,  
    unsigned int var,  
    tBddNode * high,  
    tBddNode * low,  
    tBddNode ** res),
```

ktorá vytvára nový uzol s tým rozdielom, že nekontroluje medzivýsledky. Funkcia `_apply` kontroluje medzivýsledky až pri rekurzívnom vynorovaní, keď sa pridajú danému uzlu korektní potomkovia. Taktiež sa pri vynorovaní kontroluje, či potomkovia nie sú rovnakí. V takom prípade sa uzol ruší a vracia sa ukazateľ na potomka.

Knižnica disponuje základnými logickými funkciami, ktoré môžu byť ako parameter funkcie `bddApply()`. Tieto funkcie sú `bddOr()`, `bddNor()`, `bddAnd()`, `bddNand()`, `bddXor()`, `bddImp()` a `bddNeg()`. V prípade použitia `bddNeg()`, funkcia `bddApply()` predáva ako parameter funkcie `_apply()` dva rovnaké ukazatele na grafy a funkciu `bddNand()`.

5.2.7 Práca s MTBDD

Ako bolo už vyššie spomínané, balík pracuje na globálnej úrovni kvôli premenným `bddTrue` a `bddFalse`. Z tohto dôvodu pri práci s MTBDD musí užívateľ vytvárať uzly na globálnej úrovni. To značne obmedzuje prácu s nekonečnými množinami. Inak sa s MTBDD pracuje rovnakým spôsobom ako s BDD.

5.2.8 Ukážky kódov

Predvedieme si niektoré konštrukcie, ktoré sa využívajú na tvorbu rozhodovacích diagramov.

Jednoduchá binárna funkcia

Nasledujúci kód vytvorí funkciu

$$f(a, b, c) = (a \wedge \neg b) \vee (\neg c \wedge d)$$

```

#include "bdd.h"

int main() {
    tError e;
    tManager manager;
    tBddNode *a, *b, *c, *d, *tmp;

    e = bddInit(&manager, BDD_SMALL);
    if(e) bddThrowError(e, &manager);

    e = bddCreateNode(&manager, "a", bddTrue, bddFalse, &a);
    if(e) bddThrowError(e, &manager);
    e = bddCreateNode(&manager, "b", bddFalse, bddTrue, &b);
    if(e) bddThrowError(e, &manager);
    e = bddCreateNode(&manager, "c", bddTrue, bddFalse, &tmp);
    if(e) bddThrowError(e, &manager);
    c = bddApply(&manager, tmp, NULL, bddNeg);
    nodeDecRef(&manager, tmp);

    e = bddCreateNode(&manager, "d", bddTrue, bddFalse, &d);
    if(e) bddThrowError(e, &manager);

    tmp = bddApply(&manager, a, b, bddAnd);
    nodeDecRef(&manager, a);
    nodeDecRef(&manager, b);

    a~ = bddApply(&manager, c, d, bddAnd);
    nodeDecRef(&manager, c);
    nodeDecRef(&manager, d);

    b = bddApply(&manager, a, tmp, bddOr);
    nodeDecRef(&manager, a);
    nodeDecRef(&manager, tmp);

    printTree(&manager, b);

    bddDestroy(&manager);
    return 0;
}

```

Teraz si trošku detailnejšie rozoberieme kód. Na začiatku inicializujeme manager. Postupne vytvárame uzly pre premenné a, b, c a d . Sú tu ukázané dva rôzne druhy vytvorenia uzlu popisujúceho negáciu. Prvý je pri inicializácii premennej b pomocou vymenených parametrov $high$ a low . Druhý je pomocou funkcie `bddApply()` a funkcie negácie.

Po každom volaní funkcie `bddApply()` je volaná funkcia `nodeDecRef()` nad grafmi predanými ako parameter `bddApply()`. Je to z dôvodu, že `bddApply()` nijako nemodifikuje pôvodné grafy a vytvára úplne nové. Pokiaľ nechceme už s ukazateľmi pracovať, je potrebné znížiť počítadlo referencií uzla na ktorý ukazujú. Potom je možné ukazateľ používať už ľubovoľne.

Pred skončením programu je nutné volať funkciu `bddDestroy()`, ktorá uvoľní všetky alokované prostriedky.

Práca s MTBDD

Práca s MTBDD je analogická ako práca s BDD. V podstate sa líši iba v tom, že úplne na začiatku práce sa vytvárajú vlastné terminálne uzly pomocou funkcie `bddCreateTerminal()`.

Ako ukážku som zvolil funkciu zo obrázka 3.6. Systém sa dá popísať tabuľkou výsledných stavov uplatnenia funkcie na jednotlivé terminály. tabuľka je nasledujúca:

$func(x, y)$	S_{off}	S_{on}	S_{set}	S_{count}
S_{off}	S_{off}	S_{off}	S_{off}	S_{off}
S_{on}	S_{off}	S_{on}	S_{set}	S_{count}
S_{set}	S_{off}	S_{set}	S_{set}	S_{count}
S_{count}	S_{off}	S_{count}	S_{count}	S_{count}

Tabuľka znaží prechodovú funkciu nášho systému. Implementácia pomocou knižnice je nasledujúca

```
#include "bdd.h"

tBddNode * Son, *Soff, *Sset, *Scount;

tBddNode * func(tBddNode *x, tBddNode *y){
    if(x == y) return x;
    if(x == Soff || y == Soff) return Soff;
    if(x == Son) return y;
    if(y == Son) return x;
    return Scount;
}

int main(){
    tManager mgr;
    tBddNode *xOnOff, *xSetCount, *tmp;

    bddInit(&mgr, BDD_SMALL);

    bddCreateTerminal(&mgr, "S_on", &Son);
    bddCreateTerminal(&mgr, "S_off", &Soff);
    bddCreateTerminal(&mgr, "S_set", &Sset);
    bddCreateTerminal(&mgr, "S_count", &Scount);

    bddCreateNode(&mgr, "X_on_off", Son, Soff, &xOnOff);
    bddCreateNode(&mgr, "X_count_set", Scount, Sset, &xSetCount);

    tmp = bddApply(&mgr, xOnOff, xSetCount, func);
    nodeDecRef(&mgr, xOnOff);
    nodeDecRef(&mgr, xSetCount);

    printTree(&mgr, tmp);
}
```

```
    bddDestroy(&mgr);  
}
```

Z priestorových dôvodov som odstránil ošetrovanie chybových stavov funkcií `bddInit()` a `bddCreateTerminal()`. Ako vidíme, práca s MTBDD je veľmi podobná práci s BDD, s tým rozdielom, že sami vytvárame terminálne uzly a funkcie nad nimi.

Kapitola 6

Testy

V tejto kapitole popíšem ako bol balík testovaný.

6.1 Uniky pamäti

Dealokáciu všetkých dátových štruktúr súvisiacich s BDD má nastarosti procedúra

```
bddDestroy(  
    tManager *mgr).
```

Na testovanie pamäti som použil nástroj `valgrind`¹. Testy sú zamerané hlavne na správnosť uvoľňovania všetkých využívaných zdrojov. Samozrejme `valgrind` automaticky kontroluje, či nepracujeme s neinicializovanými premennými alebo či nesiahame do pamäti, kam prístup nemáme. Pokiaľ nebude povedané inak, všetky testy sa vykonávajú nad managerom inicializovaným na veľkosť `BDD_SMALL`.

Inicializácia V programe sa volá inicializácia managera pomocou funkcie `bddInit()` a následne sa volá funkcia `bddDestroy()`, ktorá by mala korektne uvoľniť všetky alokované časti pamäte. Zdrojový kód sa nachádza v `./tests/m-init.c` a spúšťa sa pomocou príkazu `make m-init` v koreňovom adresári.

```
$ make m-init  
==24458== HEAP SUMMARY:  
==24458==    in use at exit: 0 bytes in 0 blocks  
==24458==   total heap usage: 10 allocs, 10 frees, 8,856 bytes allocated  
==32244== All heap blocks were freed -- no leaks are possible
```

Jednoduchá funkcia Program inicializuje managera, a vytvorí jednoduchú logickú funkciu o 3 logických premenných. Funkcia je nasledujúca

$$f(a, b, c) = (\neg a \wedge b) \vee (a \wedge \neg b \wedge \neg c).$$

Kód je uložený v `./tests/m-simplify.c` a spúšťa sa príkazom `make m-simplify` v koreňovom adresári.

¹<http://valgrind.org/>.

```
$ make m-simplify
==24458== HEAP SUMMARY:
==24458==      in use at exit: 0 bytes in 0 blocks
==32244== total heap usage: 13 allocs, 13 frees, 8,904 bytes allocated
==32244== All heap blocks were freed -- no leaks are possible
```

Zložitá funkcia Program inicializuje managera a vykonáva nad ním nasledujúcu funkciu

$$f(x_1 \dots, x_{33}) = (x_1 \wedge \dots \wedge x_{33}),$$

Manager sa inicializuje na veľkosť 16. Tento test sleduje prácu s novými blokmi pamäte a s cache. Kód je uložený v `./tests/m-bigf.c` a spúšťa sa príkazom `make m-bigf`.

```
$ make m-bigf
==11659== HEAP SUMMARY:
==11659==      in use at exit: 0 bytes in 0 blocks
==11659== total heap usage: 65 allocs, 65 frees, 30,544 bytes allocated
==11659== All heap blocks were freed -- no leaks are possible
```

Zhrnutie testov zameraných na pamäť Pri korektom používaní, knižnica nemá problém s alokáciou a uvoľňovaním zdrojov, nespôsobuje úniky pamäte a nespôsobuje neoprávnené prístupy do pamäti.

Kapitola 7

Porovnanie s knižnicou CUDD

V kapitole dochádza k porovnaniu programov s mnou implementovanou knižnicou pre prácu s BDD a programov s knižnicou CUDD. Testy sú zamerané hlavne na porovnanie časových náročností základných operácií s knižnicami. Všetky testy boli vykonané nad školskom serveri `merlin`. Všetky jednotky sú uvádzané v sekundách.

7.1 Inicializácia

Jedná sa o inicializáciu manažera. V CUDD na to slúži funkcia `Cudd_Init()`, v našej knižnici funkcia `bddInit()`. Dosiahnuté výsledky popisuje nasledujúca tabuľka:

Knižnica	Meranie					Priemer
	1	2	3	4	5	
CUDD	0.023532	0.021655	0.022248	0.022790	0.018378	0.021721
BDD knižnica	0.000499	0.000473	0.000449	0.000265	0.000431	0.000413

Zhrnutie V testoch inicializácie je výrazne rýchlejšia moja knižnica. To je spôsobené hlavne jednoduchosťou mojej knižnice, teda nie je potreba alokovať rozsiahle štruktúry.

7.2 Procedúra apply

Testy v tejto časti sú zamerané na rýchlosť jednorázoveho prevedenia procedúry `apply` nad dvoma funkciami reprezentovanými pomocou BDD. Nasledujúca tabuľka popisuje rýchlosť vykonania nasledujúcej funkcie

$$f = f_1 \vee f_2,$$

kde $f_1 = (x_1 \wedge \dots \wedge x_{200})$ a $f_2 = (\neg x_1 \wedge \dots \wedge \neg x_{200})$.

Knižnica	Meranie					Priemer
	1	2	3	4	5	
CUDD	0.000470	0.000509	0.000259	0.000450	0.000246	0.000387
BDD knižnica	0.000519	0.000232	0.000510	0.000548	0.000437	0.000449

Výsledky majú len nepatrné rozdiely, preto som sa rozhodol rozšíriť funkcie f_1 a f_2 tvorí 1000 uzlov, teda $f_1 = (x_1 \wedge \dots \wedge x_{1000})$ a $f_2 = (\neg x_1 \wedge \dots \wedge \neg x_{1000})$. Namerané výsledky popisuje nasledujúca tabuľka:

Knižnica	Meranie					Priemer
	1	2	3	4	5	
CUDD	0.000650	0.000603	0.000618	0.000713	0.000619	0.000641
BDD knižnica	0.000678	0.000710	0.000703	0.000757	0.000698	0.000709

Po tomto teste je možné povedať, že knižnica CUDD je rýchlejšia. Výpočet bol však v oboch prípadoch veľmi rýchly, preto nasledujúce testy merajú dobu trvania celého programu bez inicializácie a uvoľnenia.

7.3 Dĺžka programu

V tejto časti sú testy zamerané na celkovú dĺžku programu pri vytváraní zložitejších funkcií. Testy merajú celú dobu behu programu s výnimkou inicializácie a uvoľňovania manažera, lebo v knižnici CUDD sú tieto operácie veľmi náročné a mohli by skreslovať porovnanie rýchlostí procedúry apply.

Funkcia ktorú v programe vytvoríme je

$$f = (x_1 \wedge \dots \wedge x_4) \oplus (x_2 \wedge \dots \wedge x_5) \oplus \dots \oplus (x_{22} \wedge \dots \wedge x_{25}),$$

kde \oplus značí logickú operáciu xor. Namerané výsledky sú v nasledujúcej tabuľke:

Knižnica	Meranie					Priemer
	1	2	3	4	5	
CUDD	0.001229	0.001163	0.001143	0.001175	0.001187	0.001179
BDD knižnica	2.230097	1.785477	1.778962	1.938497	1.962328	1.939072

Je vidieť, že mnou implementovaná knižnica nestíha pri vyšších výpočtoch. Zrejme to bude zapríčinené slabou optimalizáciou výpočtov a nevhodnou štruktúrou ukladania medzivýsledkov.

Kapitola 8

Záver

Práca sa venuje návrhu a implementácií knižnice pracujúcej s binárnymi rozhodovacími diagramami BDD. Hlavným prínosom práce je poskytnutie jednej z možných reprezentácií efektívneho uloženia BDD v počítači. Návrh sa zameriava hlavne na zníženie počtu alokácií. Práca je implementovaná v jazyku C.

V práci sa nachádza matematický popis výrokovej logiky, ktorý je neodeliteľnou súčasťou popisu BDD. Ďalej je popísané BDD pomocou formálneho matematického zápisu ako acyklický spojený orientovaný koreňový graf. V práci sú popísané taktiež multiterminálne BDD: MTBDD. Práca obsahuje algoritmy na zníženie pamäťovej náročnosti BDD v počítači. Ďalej je tu popísaný algoritmus funkcie *apply*. V práci sú taktiež popísané niektoré existujúce balíky pre prácu s BDD.

Implementácia prešla sadou testov, ktoré mali odhaliť prípadné úniky pamäte a neoprávnené prístupy do pamäti. Knižnica vyhovela všetkým testom. Ďalej bola knižnica testovaná výkonnosťnými testami a porovnávaná s knižnicou CUDD. V teste inicializácie a uvoľňovania managera moja knižnica výrazne predbiehala knižnicu CUDD, čo môže byť spôsobené jednoduchšou štruktúrou návrhu. V testoch ohľadom porovnania funkcie *apply* moja knižnica výrazne zaostávala za CUDD. Programom `gprof`¹ som zistil, že až pätinu času program strávil prehľadávaním medzivýsledkov.

Návrhy rozšírenia práce do budúcnosti sú:

- podpora MTBDD aby bolo možné pracovať aj s nekonečnými doménami,
- prerobenie štruktúry medzivýsledkov na zníženie časovej náročnosti programov,
- vstupný prekladač pre načítanie BDD zo súboru,
- výstupný prekladač pre zápis BDD do súboru v nejakom použiteľnom formáte.

Dosiahnuté výsledky určujú knižnicu skôr na ďalší vývoj a prípadnú demonštráciu práce s BDD a nie na praktické nasadenie.

¹http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

Literatura

- [1] Bryant, R. E.: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, 1986, iSSN: 0018-9340, s. 677, dostupné z <http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.ps>.
- [2] Guanfeng Lv, Y. X., Kaile Su: *CacBDD: A BDD Package with Dynamic Cache Management*. Dostupné z <http://www.kailesu.net/CacBDD/CacBDD.pdf>.
- [3] Lee, C. Y.: *Representation of Switching Circuits by Binary-Decision Programs, Vol. 38*. Bell System Technical Journal, 1959, s. 985.
- [4] Rucký, P.: *Převod víceúrovňové logické sítě na dvouúrovňovou pomocí BDD*. České vysoké učení technické v Praze, Diplomová práce, 2007.
- [5] Rudell, R. L.: *Dynamic variable ordering for ordered binary decision diagrams*. IEEE Computer Society Press Los Alamitos, CA, USA, 1993, iISBN: 0-8186-4490-7, s. 42.
- [6] Shannon, C. E.: *A Symbolic Analysis of Relay and Switching Circuits*. Transactions of the AIEE, Vol. 57, 1938, s. 713.
- [7] Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.5.0. Dostupné z <http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html>.
- [8] Vojnar, T.: Binary Decision Diagrams BDDs. Vysoké učení technické v Brně, Fakulta informačních technologií, Prezentace k předmětu Formální analýza a verifikace, 2013.
- [9] Šlapal, J.: *Úvod do predikátového počtu 1. řádu*. Vysoké učení technické v Brně, Fakulta informačních technologií, Skripta k předmětu Matematické struktury v informatice, 2014, dostupné z http://www.math.fme.vutbr.cz/download.aspx?id_file=3670, s. 3.
- [10] Wikipedia: Binary decision diagram. http://en.wikipedia.org/wiki/Binary_decision_diagram#Variable_ordering.

Dodatek A

Obsah CD

Na priloženom CD sa nachádzajú nasledujúce súbory a adresáre:

- src/ priečinko so zdrojovými kódmi,
 - tests/ priečinko s testami,
 - *.h, *.c zdrojové kódy jednotlivých modulov,
 - main.c priestor pre Váš program,
 - makefile súbor pre prácu s priečinkom src/,
 - README popis pravidiel v makefile,
- bp_tex/ zdrojové súbory k tomuto textu,
 - zdrojové súbory a obrázky,
 - makefile súbor pre prácu s priečinkom,
 - README popis pravidiel makefile,
- bp-xtrosk00-desky.pdf Vysádzanie desiek BP,
- bp-xtrosk00-print.pdf tento text,
- bp-xtrosk00-wis.pdf tento text s hypertextovými odkazmi,
- README obsah CD

Dodatek B

Programátorské rozhranie

Programátor by mal využívať iba funkcie určené pre vonkajšiu prácu s balíkom, používanie vnútorných funkcií balíka môže spôsobiť nepredvídateľné správanie. Funkcie určené pre programátora sú:

- `tError bddInit(tManager *mgr, unsigned int size),`
- `void bddDestroy(tManager *mgr),`
- `tBddNode *bddApply(tManager *mgr, tBddNode *x, tBddNode *y, tBddNode *(*func)(tBddNode *, tBddNode *)),`
- `tBddNode *bddCreateNode(tManager *mgr, char *label, tBddNode *high, tBddNode *low, tBddNode **result),`
- `tBddNode *bddCreateTerminal(tManager *mgr, char *label, tBddNode **result),`
- `void nodeDecRef(tManager *mgr, tBddNode *node),`
- makro `bddThrowError(tError e, tManager *mgr),`
- množina booleovských funkcií
 - `tBddNode *bddOr(tBddNode *x, tBddNode *y),`
 - `tBddNode *bddNor(tBddNode *x, tBddNode *y),`
 - `tBddNode *bddAnd(tBddNode *x, tBddNode *y),`
 - `tBddNode *bddNand(tBddNode *x, tBddNode *y),`
 - `tBddNode *bddXor(tBddNode *x, tBddNode *y),`
 - `tBddNode *bddImp(tBddNode *x, tBddNode *y),`
 - `tBddNode *bddNeg(tBddNode *x, tBddNode *y).`