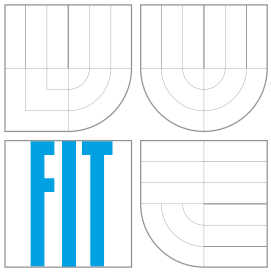
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

Binárny rozhodovací diagram BDD je dátová štruktúra využívaná v mnohých oblastiach informatiky. Táto práca popisuje BDD ako matematický formalizmus a navrhuje možnú reprezentáciu BDD v počítači. Návrh je zameraný hlavne na rýchlosť znížením počtu alokácií pamäte a na jednoduchosť a intuitívnosť používania knižnice. V práci sú aj rôzne jednoduché príklady použitia knižnice a výstrahy, ktorým by sa mal programátor pri používaní knižnice vyvarovať.

Abstract

Výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

Binárny rozhodovací diagram, implementácia, logická funkcia, logická premenná, vysoký následník, nízky následník, uzol, koreň, terminál, procedúra apply, matematický formalizmus, množina, výroková logika.

Keywords

Klíčová slova v anglickém jazyce.

Citace

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

BDD knižnica

Prohlášení

Prehlasuje, že túto bakalársku prácu som vypracoval sám pod vedením Ing. Ondřeje Lengála. Všetky informácie som čerpal z prednášok predmetu Formální analýza a verifikáce na FIT VUT v Brně, literatúry uvedenej v bibliografických citáciach a z vlastných skúseností, ktoré som nabral na FIT VUT v Brně.

.....
Karol Troška
15. 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 držali.

Obsah

1 Úvod **3**

2 Výroková logika **4**
2.1 Možnosti reprezentácie funckií 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 Procedúra *Apply()* 8
3.4 Typy BDD 8
3.4.1 BDT 9
3.4.2 Redukované BDD 10
3.4.3 Redukované usporiadané BDD 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 – *Cache* 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 20
5.2.3 Vytvorenie uzlu 21
5.2.4 Práca s mezdivýsledkami – *Cacheovanie* 21
5.2.5 Práca s referenciami 22
5.2.6 Procedúra *Apply()* 22
5.2.7 Práca s MTBDD 22
5.2.8 Ukážky kódov 22
5.3 Programátorské rozhranie 25

6 Testy **26**
6.1 Uniky pamäti 26

7 Porovnanie s knižnicou CUDD **28**

8 Závěr **29**

Kapitola 1

Úvod

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

BDD reprezentuje funkciu nad binárnimi premennými kompaktne pomocou koreňového acyklického orientovaného spojitého grafu. Nad týmto grafom sa dajú uplatňovať mnohé algoritmy, ktoré znižujú jeho pamäťovú aj časovú náročnosť.

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 niektorými knižnicami.

Práca obsahuje v kapitole 2 formálny popis výrokovej 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 literatúry[9].

Buď P neprázdna množina prvotných formúl, ktoré hrajú úlohu jednoduchých výrokov. Z jednoduchých výrokov získavame 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 riadí 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 formulí 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 formulí 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 závistosti 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 funckií

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

Pravdivostná tabuľka Najprehľadnejší a najľahšie naučiteľný typ uchovávania logickej funkcie, bohužial 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 vynikajúcí na demoštráciu pokročilých systémov zložených z logických funkcií.

BDD Formát v podobe acyklického koreňového grafu, ktorému sa budeme ďalej venovať v kapitole 3.4.

2.2 Minimalizácia

Minimalizácia logickej funkcie je proces znižovania pamäťových nárokov funkcie odstránením redundancie. V prípade matematického popisu môže ísť o uplatnenie rôznych výrokových teorémov a viet. Podrobnéj minimalizácií BDD sa budeme venovať v kapitole 3.4.

2.3 Počet funkcií

Počet možných správaní sa logickej funkcie závisí od počtu premenných, ktoré ju ovplyvňujú a počtu hodnôt, ktoré každá premenná môže nadobudnúť. Ide o jednoduchú kombinatoriku, kde zisťujeme všetky variacie vstupných hodnôt, teda $p = n^k$, kde n je počet hodnôt, ktoré môže nadobúdať vstupná premenná, k počet vstupných premenných a p je výsedny počet možných vstupov funkcie.

Výsledný počet funkcií 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, q je počet hodnôt, ktoré môže výstpná premenná nadobúdať a r je počet funkcií danej funkcie. Z tohto dostávame vzorec

$$r = q^{n^k}. \tag{2.1}$$

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

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

Kapitola 3

Binárne rozhodovacie diagramy

V tejto kapitole popíšeme históriu BDD, BDD ako matematický formalizmus a transformovanie BDD na redukované usporiadané BDD – ROBDD.

3.1 História

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

Najvýznamnejším milníkom v oblasti redukcie pamäťovej náročnosti bola štúdia R. E. Bryanta[1] z Univerzity Carnegie Mellon. Táto štúdia sa týka najmä poradia premenných a zdieľania podgrafov. Toto sa dá pokladať za vznik ROBDD popísaných v sekcií 3.4.1.

3.2 Matematický popis

Prevzaté z prednášky [8].

BDD je reprezentácia logickej funkcie $\{0,1\}^k \to \{0,1\}, k \geq 0$. Z pohľadu teorie grafov sa jedná o koreňový, orientovaný, spojitý acyklický graf. Formálna definícia BDD G nad množinou premenných M je

$$??G = (N, T, var, low, high, root, val), \tag{3.1}$$

kde

- N je konečná množina neterminálnych stavov (uzlov).
- T je konečná množina terminálnych stavov (listov), $N \cap T = \emptyset$.
- $var\colon N \to M$ je interné pomenovanie uzlov.
- $low, high\colon N \to N \cup T$ 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 $\forall n \in (N \cup T)\backslash\{root\}.root(low \cup high)^+ n$. Vstupné hrany do uzlú $root$ by porušili acyklicitu grafu.
- $val\colon T \to \{0,1\}$ je mapovacia funkcia hodnôt terminálnych uzlov

3.3 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 restrikciu určitej premennej v grafe za konkrétnu hodnotu. Nech je n-parametrová funkcia $f(v_1, \ldots, v_n)$. Vykonanie restrikcie nad funkciou f, premennej v_i za hodnotu a dostávame funkciu $f(v_1, \ldots, v_{i-1}, a, v_{i+1}, \ldots, v_n)$, skrátene písané $f|_{v_i \leftarrow a}(v_1, \ldots, v_n)$. Pokiaľ prevedieme restrikciu na všetky premenné, výsledok konrétneho dotazu nad funkciou už nie je rozhodovacia funkcia ale konkrétna hodnota. Podľa Shannonovho torému platí

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

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

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

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

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

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

3.4 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

$$f(x_1, x_2, x_3) = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3). \tag{3.2}$$

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 premennéj x_2.

Algoritmus 1: Apply

Input: $X, Y \in N \cup T$, *func* je funkcia nad terminálnymi uzlami BDD
Output: *result* je výsledok funkcie $func(X, Y)$

```
1:  xh = X, xl = X, yh = Y, yl = Y;
2:  if (X, Y ∈ T) then
3:  |  return func(X, Y) ;
4:  else
5:  |  if (X ≺ Y) then
6:  |  |  Vytvor uzol podľa var(X);
7:  |  |  xh = high(X), xl = low(X);
8:  |  else
9:  |  |  if (Y ≺ X) then
10: |  |  |  vytvor uzol podľa var(Y);
11: |  |  |  yh = high(Y), yl = low(Y)
12: |  |  else
13: |  |  |  vytvor uzol podľa var(Y);
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 (result je v cache) then
24: |  |  uvolni result;
25: |  |  result = cache zaznam;
26: |  end if
27: end if
28: return result;
```

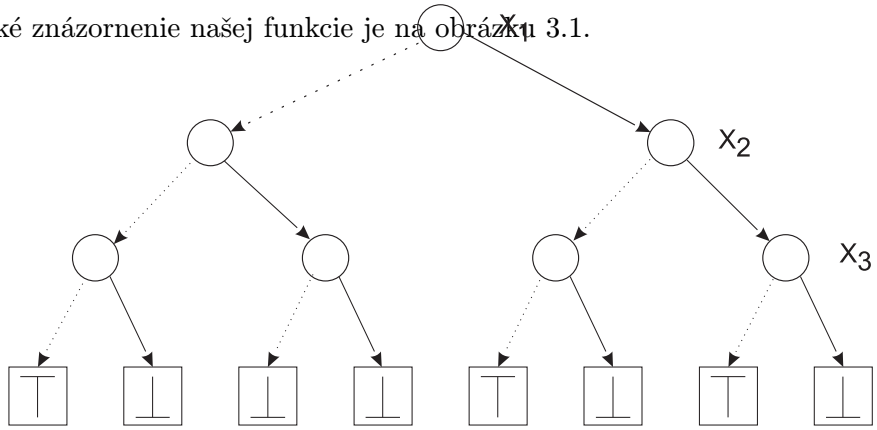
3.4.1 BDT

Binárny rozhodovací strom 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

$$\sum_{i=0}^{k} 2^i, \tag{3.3}$$

kde k je počet premenných.

Grafické znázornenie našej funkcie je na obrázku 3.1.



Obrázek 3.1: Základné BDD

Notácia Pre orientáciu v grafe je použitá nasledujúca notácia

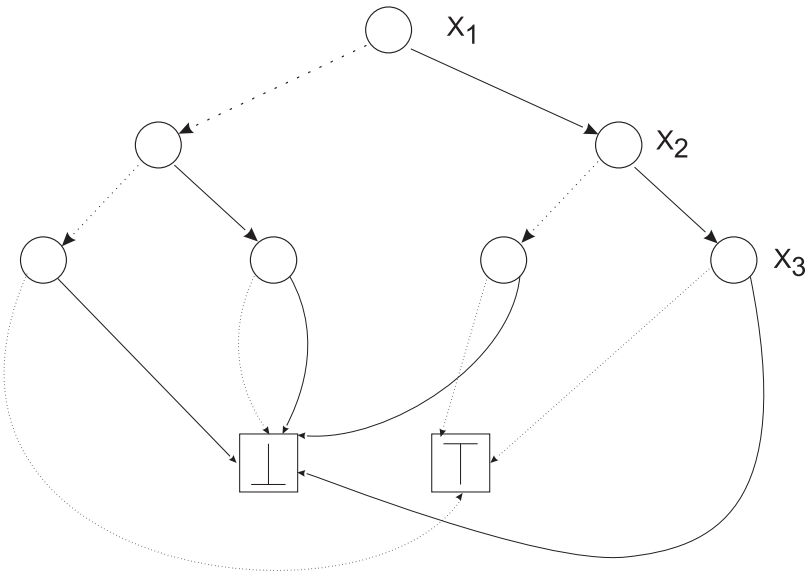
1. Uzly
   - Značené krúžkom.
   - Hierarchicky usporiadané podľa poradia vyhodnocovania.
   - Premenná ovplyvňujúca uzol je písaná vľavo.
2. Terminály
   - Značené štvorčekom.
   - Symbol $\bot$ značí hodnotu $false$.
   - Symbol $\top$ značí hodnotu $true$.
3. Hrany
   - Plná čiara vedie k vysokému následníkovi daného uzlu $high(n)$.
   - Prerušovaná čiara vedie k nízkemu následníkovi daného uzlu $low(n)$.

Ako vidíme, pre 3 premenné je počet generovaných uzlov 15, čo je exponencialna zložitosť, čím je potvrdená rovnica 3.3. Zložitosť najdenia daného terminálneho uzlu je lineárna vzhľadom k počtu premenných. Táto reprezentácia obsahuje mnoho redundancie, preto sa nepoužíva.

3.4.2 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.

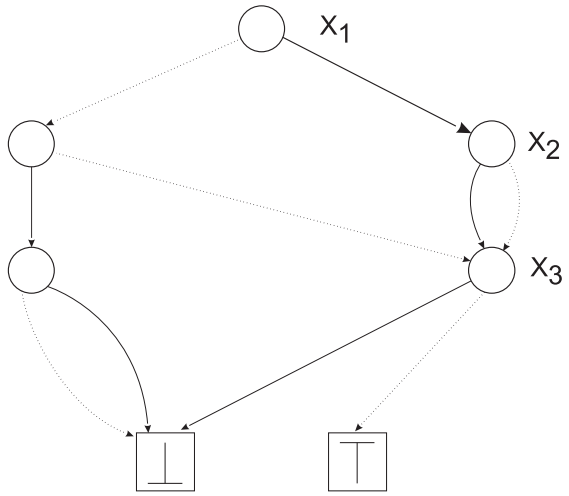


Obrázek 3.2: BDD s redukovaným počtom terminálnych uzlov

Graf je definovaný ako dvojica $G = (V, H)$, kde V je množina vrcholov a H je množina hrán typu (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 vzájomne bijektívne zobrazenie $f : V_1 \to V_2$ a $g : H_1 \to H_2$ 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 $f(x), f(x) \in V_2$.
Pokiaľ grafy G_1 a G_2 sú izomorfné, teda existujú funkcie f a g, potom píšeme $G_1 \otimes G_2$.

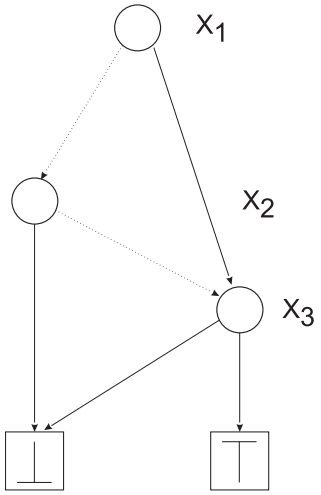
Strom s odstránenými izomorfnými podstromamy je na obrázku 3.3.



Obrázek 3.3: BDD s odstránenými izomorfnými podstromami

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 premennéj, preto sa môže tento uzol vynechať. Výsledný strom je na obrázku 3.4.



Obrázek 3.4: Výsledné minimalizované BDD

RBDD je klasické BDD, pre ktoré platia nasledujúce pravidlá:

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

3.4.3 Redukované usporiadané BDD

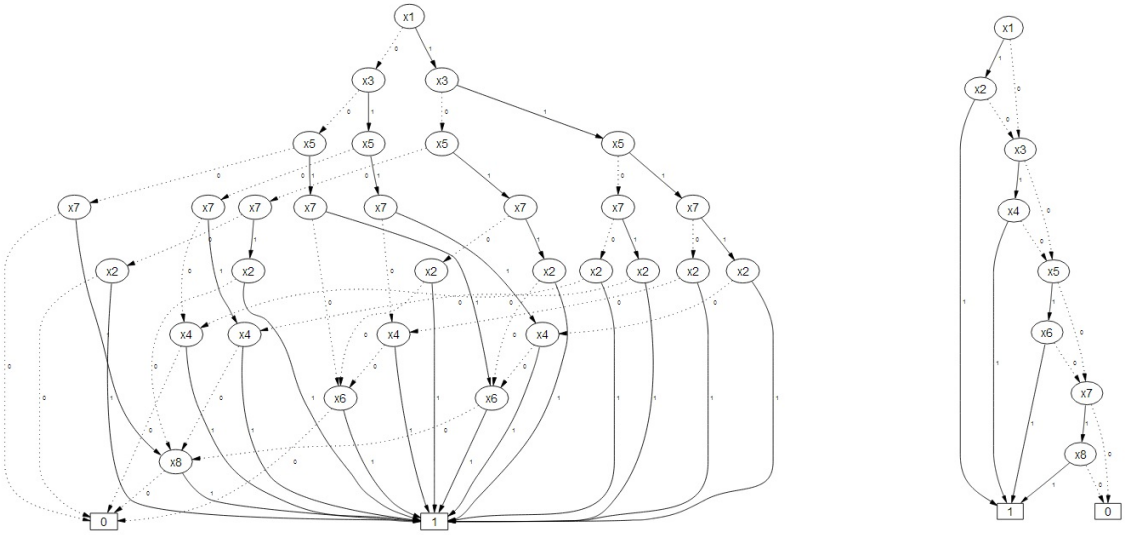
Poradia vyhodnocovania premenných, ktoré ovplyvňujú uzly, majú veľký vplyv na počet generovaných uzlov. Pre ilustráciu si to predveďme na funkcií

$$y(x_1, ..., x_8) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6) \vee (x_7 \wedge x_8). \tag{3.4}$$

RBDD odpovedajúce danej funkcie je na obrázku 3.5.

Obe BDD sú minimalizované a popisujú rovnakú logickú funckiu. Z pravého obrázku však vidíme, že dobrým usporiadaním výrazne znížime počet generovaných uzlov. Na zistenia optimálneho usporiadania však neexistuje doposiaľ žiadný efektívny algoritmus. Jedná sa o NP-náročný problém. Existujú iba rôzne heuristiky. Tie nám však môžu pomôcť, teda dostaneme výsledný strom s menším počtom uzlom, 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 itérácie.
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.



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

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 pozícií 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

3.5 Multiterminálne BDD

Formálna definácia MTBDD je rovnaká ako definiícia **??** pre BDD s rozdielom, že v MTBDD množinu *val* definujeme takto:

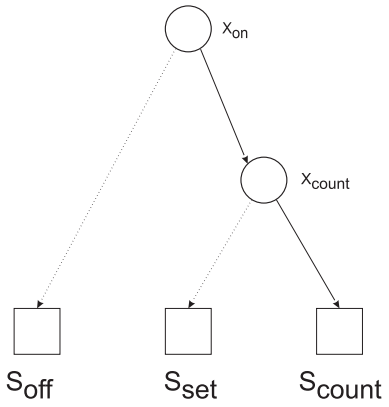
- $val\colon T \to \mathbb{D}$, kde $\mathbb{D}$ je potenciálne nekonečná množina.

Multiterminálne BDD sú vo svojej podstate obyčajné ROBDD, ktoré majú väčší, obecne neobmedzený počet terminálnych uzlov. Ako príklad si môžeme uviesť kombinačný systém[1], ktorý sa môže nachádzať v stavoch S_{off}, S_{set} a S_{count}. Vstupné premenné sú X_{on} a X_{count}. Tabuľka systému je

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}

[1]Stav závisí iba na kombinácii vstupu

Na miminalizáciu 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, Aritmetickými rozhodovacími diagramami ADD a diagramami s potlačenou nulou ZDD. 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 – Umožnuje písanie zložitejších projektov, kde je vzhľadom k efektivite nutné pridať do balíka vlastné funkcie.
3. Interface – Balík disponuje kvalitne spracovaním 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()`. 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 hashovaciej 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 funckiou `Cudd_Quit()`.

4.1.3 Zhrnutie

CUDD je univerzálny balíček pre prácu s BDD. Má kvalitne spracovanú dokumentáciu včetne uživateľského 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 demoštráciu princípov BDD. Vo svojej poslednej verzií je už však plnohodnotným balíkom so všetkými štandartnými 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 poľom 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 súčastnom 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 vyvýjanú v jazyku C++, ktorá je podobne ako BuDDy založená na indexovaní v poli. Podporuje obvyklé operátory využívané v modelcheckingu. CacBDD má kvalitne spracované ukľadanie 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.kailesu.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 následujúcej 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. hand. `hit-rate` sa však 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 implementácií dynamického ukladania medzivýsledkov. V literatúre[2] je možné nájsť aj rýchlostné zrovnanie s balíkom CUDD.

Kapitola 5

Návrh a implementácia

V tejto kapitole popíšem svôj návrh implementácie jednotlivých dátových štruktúr a algoritmov.

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 úplnu 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 terminálny sa používa makro `isTerminal(t)`. Premenná `var` je index do tabuľky mien premenných, ktorú si detailnejšie popíšeme v sekcií 5.1.3. Premenné `ref` a `nextFree` sú premenné, ktoré využíva garbage collector. `ref` je počítadlo referencií, ktoré je dôležité pri zdielaných uzloch a `nextFree` je ukazateľ, ktorý previazáva zoznam voľných premenných ktoré má k dispozícií 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 reťazcov[1] a `count` je premenná označujúca počet reťazcov, používana 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 klúč 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.

[1] Reťazcom 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 uvoľnovanie a vytváranie nových zhlukov 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

Balíček sa môže pri používaní nachádzať v nejakom chybovom stave. 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 uživateľa. Makro vytlačí odpovedajúcu chybovú hlašku k chybe predanej parametrom `e`, uvoľní prostriedky managera predaného parametrom `mgr` a ukončuje program chybovým kódom `e`.

5.2 Algoritmy

Hlavné uživateľské rozhranie tvoria funkcie `bddInit()`, ktorá inicializuje všetky komponenty pre manager, `bddDestroy()`, ktorá uvoľnuje 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. Funckia 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 únikom pamäti.

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

Premenná `free` je iba pomocný ukazateľ, teda sa neuvoľnuje. Miesto ktoré odkazoval bolo uvoľnené pri uvoľnovaní 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. Návratová 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. Nízkemu 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 uvedenej 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 medzivýsledky pomocou funckie `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(tBddNode *node)` zvyšuje počítadlo referencií na uzol. Volá sa implicitne pri vytváraní uzlu za predpokladu, že uzol už existuje a bol najdený v medzivýsledkoch.

Funkcia `nodeDecRef(tManager * mgr, tBddNode *node)` 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 medzivýsledkov. 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 3.3. Implementáciu v knižnici tvoria dve funkcie. Prvá je `bddApply(tManager *mgr,tBddNode *x, tBddNode *y, tBddNode* (*func) (tBddNode *, tBddNode *))`, ktorá sa volá z uživateľského programu. Ako parameter dostáva dva ukazatele na grafy (`x` a `y`), manager, v ktorom sa má procedúra vykonať (`mgr`) a odkaz na funkciu vracajúcu nový diagram (`func`). Funkcia spúšťa rekurzívnu funkciu `_apply()` s rovnakými parametrami. `_apply()` sa rekurzívne zanoruje podľa algoritmu 1 popísaného v časti 3.3.

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 ukazateľe 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 uživateľ 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 inicializácií 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 uplatenia 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 funckiu 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 odtranil 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.

5.3 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)`

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 uvolňovania 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/test-init.c` a spúšťa sa pomocou príkazu `make t-init` v koreňovom adresári.

```
make t-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/test-simplyf.c` a spúšťa sa príkazom `make t-simplyf` v koreňovom adresári.

```
make t-simplyf
==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
```

[1] `<http://valgrind.org/>`.

```
==32244== All heap blocks were freed -- no leaks are possible
```

Zložitá funkcia Program inicializuje managera a vykonáva nad ním nasledujúcu funkciu

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

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

```
make t-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 uvoľnovaním zdrojov, nespôsobuje úniky pamäte a nespôsobuje neoprávnené prístupy do pamäti.

Kapitola 7

Porovnanie s knižnicou CUDD

Kapitola 8

Závěr

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 Managment.* 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, iSBN: 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>.