České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Detekce tautologie pomocí BDD

Michal Navrkal

Vedoucí práce : Ing. Petr Fišer

Studijní program : Elektrotechnika a informatika strukturovaný bakalářský
Obor : Informatika a výpočetní technika

červen 2006

Poděkování

Tímto bych chtěl poděkovat vedoucímu mé bakalářské práce Ing. Petru Fišerovi za pomoc a veškerý čas, který mi věnoval.

Prohlášení

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

V Praze dne 26.06.2006 ……

Abstrakt

Cílem této bakalářské práce je naprogramovat v jazyce C++ nástroj pro zjištění, zda je logická funkce tautologie. Funkce je zadaná tabulkou ve formě součtu součinů.
Program má fungovat na bázi binárních rozhodovacích diagramů (BDD). Práce nejdříve objasňuje základní pojmy týkající se BDD a základní metody a algoritmy použité při tvorbě programu. Na závěr nechybí ani otestování programu na zkušebních úlohách a porovnání výsledků s jinými implementacemi.

Abstract

The aim of this work was to create in C++ a program package for detection tautology of boolean functions. The functions are given by truth table in sum of products form.
This program ought to operate on binary decision diagrams. This work at first explain basic notions about BDD and basic methods and algorithms used.
At the end program is tested and confront with other implementations.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Tématem mojí práce je implementace nástroje pro detekci tautologie logické funkce. Při této detekci se mají využívat binární rozhodovací diagramy (z angl. Binary Decision Diagrams – BDD). Logická funkce bude zadána jako součet součinových termů v notaci PLA Espresso. Společně s Ing. Fišerem jsme se domluvili, že při psaní téhle práce nebudeme stavět na knihovně CUDD, která je také postavena na binárních rozhodovacích diagramech, ale půjdeme vlastní cestou. Důvody byly čistě experimentální. Na konci této práce se pokusím obě řešení porovnat.
Při výběru tohoto zadání, jsem se domníval, že se jedná o jednoduchou a rychle hotovou věc. Po několika pokusech se mi podařilo skutečně naimplementovat nástroj, který byl schopný rozhodnout , zda funkce zadaná v PLA souboru je tautologií. Ovšem pro velké funkce bylo toto řešení naprosto nepoužitelné a musel jsem se ponořit do problému hlouběji a najít řešení, které je z časového hlediska přijatelnější.
Pak už zbývalo porovnat mojí implementaci s jinými dostupnými. Po dohodě s Ing. Fišerem jsme vybrali již zmíněnou knihovnu CUDD a implementaci, jejíž autorem je Radek Chromý, který detekci tautologie řeší zcela jiným způsobem.

2 Binární rozhodovací diagramy

Tato kapitola je kapitola, která se zabývá úvodem do problematiky binárních rozhodovacích diagramů.

2.1 Seznámení s BDD

V současné době se v praxi setkáváme s potřebou řešit složité problémy při návrhu nebo optimalizaci digitálních systémů a kombinační logiky, testování, verifikaci, ale třeba i otázky spojené s matematickou analýzou a prudce se rozvíjejícím odvětvím zabývajícím se umělou inteligencí. Většina takovýchto problémů se dá přetransformovat na problém nad doménou logických funkcí s použitím několika málo operací. I když se dají logické funkce reprezentovat mnoha rozdílnými kanonickými způsoby, jako například pravdivostní tabulkou, disjunktivní normální formou či konjunktivní normální formou, nejsou tyto formy příliš vhodné pro reprezentaci funkcí s velkým počtem proměnných. Prostě a jednoduše jsou příliš náročné co do velikosti paměťového místa potřebného pro reprezentaci a efektivita práce s nimi také není nejlepší. Naproti tomu binární rozhodovací diagramy(Binary Decision Diagrams - BDDs) vznikly právě jako efektivní forma pro manipulaci a reprezentaci rozsáhlých logických funkcí.

2.2 Jak vypadá BDD

Každá funkce je reprezentovaná jako orientovaný acyklický graf (z angl. DAG – directed acyclic graf), který tvoří vnitřní uzly (neterminální uzly), které korespondují vždy s určitou proměnnou funkce, kterou dané BDD reprezentuje. Graf také tvoří uzly koncové (terminální uzly, listy), které jsou označeny 1 nebo 0 a reprezentují příslušnou logickou konstantní hodnotu.

Z každého neterminálu *v* vedou dvě výstupní hrany. Jedna do tzv. low-potomka a druhá do tzv. high-potomka (dále jen low(v) a high(v)).

Jako příklad Obrázek 1 ilustrující reprezentaci funkce $f(x_1,x_2,x_3)$ definovanou pravdivostní tabulkou nalevo. Každý neterminální uzel v je označen proměnnou var(v) a má dvě hrany směřující ke dvěma potomkům :

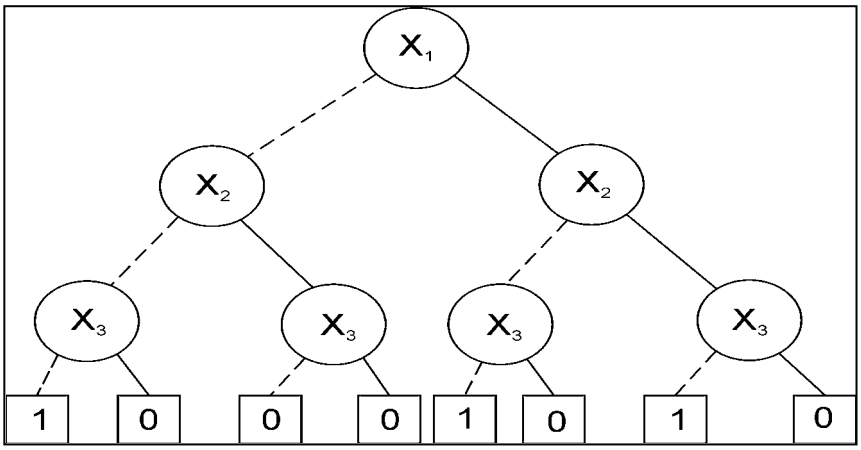
low(v) označen čárkovanou čarou (případ kdy proměnná je nulová)

high(v) označen plnou čarou (případ kdy proměnná je jedničková)

Každý terminální uzel je označen 0 nebo 1.

Pro dané přiřazení hodnot proměnných je hodnota funkce určena procházením grafu z kořene k terminálu, kde rozhodnutí o větvení je prováděno na základě hodnoty dané proměnné v přiřazení. Jako hodnotu funkce dostaneme hodnotu terminálu na konci cesty.

X_1	X_2	X_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



Obrázek 1 – BDD vzniklé z pravdivostní tabulky nalevo

Když už víme jak reprezentovat logickou funkci pomocí grafu, můžeme operace nad booleovskými funkcemi implementovat jako grafové algoritmy. Ačkoli BDD representace logické funkce může mít obecně velikost exponenciálně závislou na počtu proměnných, v praxi dospějeme k mnohem kompaktnější reprezentaci.

2.3 (R)OBDD

V této kapitole ozřejmím rozdíl mezi OBDD, potažmo ROBDD, a BDD. Tyto rozdíly se často opomíjejí a spousta lidí mluvících o BDD má vlastně na mysli ROBDD.

2.3.1 Uspořádání

BDD byly objeveny jako abstraktní reprezentace booleovských funkcí před mnoha lety. Pod názvem „branching programs“ byly studovány teoretiky zabývajícími se složitostí. BDD, jak byly popsány v kapitole 2.2, nejsou kanonickou reprezentací. Může existovat mnoho rozdílných BDD reprezentující jednu a tu samou funkci. Proto se zavádí pojem OBDD (ordered BDD).

Pro OBDD využijeme uspořádání nad množinou proměnných funkce. Aby se BDD stalo OBDD musíme vyžadovat určité uspořádání proměnných. Pro každý uzel *u* a jeho potomka *v* musí platit následující nerovnost:

$$\mathbf{var(u) < var(v)}$$

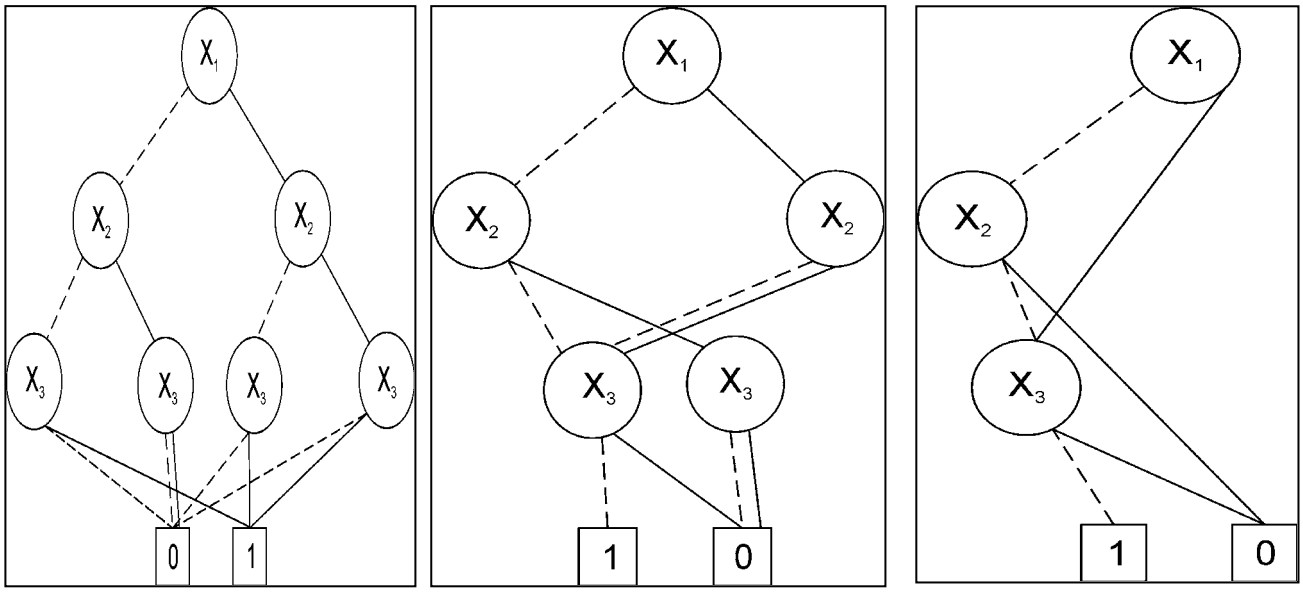
Např. na Obrázku 1 jsou proměnné seřazeny $x_1 < x_2 < x_3$. To však není podmínkou, v principu může pořadí proměnných být zvoleno libovolně, protože algoritmy pracují korektně pro jakékoli pořadí. Nalezení nejvhodnějšího uspořádání proměnných tak, aby požadavek na velikost paměťového prostoru nutného k reprezentaci funkce byl co nejmenší, je v praxi dosti náročný problém.

2.3.2 Redukce

Takto popsaný OBDD má však ještě jeden neduh a tím je možná redundance dat. Abychom ji odstranili, nadefinujeme tři transformační pravidla pro takovýto graf:

1) **odstranění duplicitních terminálů:** Eliminujeme všechny terminály se stejnou hodnotou až na jeden
2) **odstranění duplicitních neterminálů:** Když neterminální uzly *u* a *v* mají var(u)=var(v), low(u)=low(v) a high(u)=high(v), potom jeden z uzlů *u* a *v* odstraníme a všechny hrany vedoucí do odstraněného uzlu převedeme do druhého uzlu.
3) **Odstranění redundantních uzlů:** Když neterminální uzel *v* má low(v)=high(v), potom ho odstraníme a všechny vstupní hrany převedeme do low(v).

Aplikací těchto pravidel na obecný OBDD dostaneme ROBDD (reduced OBDD), tedy minimalizovaný nebo chcete-li redukovaný uspořádaný binární rozhodovací diagram.



a) Duplicitní terminály b) Duplicitní neterminály c) Redundantní uzly

Obrázek 2 - Použití redukčních pravidel na BDD z Obrázku 1

Jak je vidět z Obrázku 2 první pravidlo zredukovalo počet terminálních uzlů z 8 na 2. Druhé pravidlo odstranilo dva uzly označené proměnnou x_3 a nakonec třetí pravidlo odstranilo dva redundantní uzly, jejichž vstupní hrany jsme „přemostili“ do jednoho z potomků.

Takováto reprezentace funkce pomocí ROBDD je už kanonická – odpovídá lemmatu kanonicity.

Lemma kanonicity :

Pro každou funkci f: $B^n \rightarrow B$ je přesně jedno ROBDD *u* s pořadím proměnných $x_1 < x_2 < ... < x_n$ tak ,že $\mathbf{f^u = f(x_1,\ldots,x_n)}$

To znamená, že pro dané pořadí proměnných musí být dvě ROBDD pro danou funkci identické (pro OBDD jsou izomorfní).
Kanonicita má několik důležitých důsledků. Například je možno snadno testovat ekvivalenci funkcí. V konstantním čase také zjistíme, zda je funkce tautologie. Pokud ano, pak její graf má podobu jedničkového terminálu. Můžeme také jednoduše testovat splnitelnost funkce – v grafu musí existovat alespoň jedna cesta vedoucí z kořene do

terminálu 1, nebo-li grafem není samotný terminál 0. Když je funkce nezávislá na proměnné x, neobsahuje graf žádný uzel označený proměnnou x.

Jak ilustruje Obrázek 1 a Obrázek 2, můžeme vytvářet OBDD reprezentaci funkce z její pravdivostní tabulky a poté provést redukci. Tento přístup je jednoduchý a někdy i praktický, ale musíme si uvědomit, že velikost (R)OBDD exponenciálně závisí na počtu proměnných funkce. Proto se používá při konstrukci jiný přístup a to použití apply operací, jak bude vysvětleno později.

2.4 Shannonova expanze

Výsledek funkce po dosazení konstantní hodnoty k (0 nebo 1) za proměnnou x funkce f se nazývá restrikce nebo někdy bývá označován jako kofaktor a značí se $f\mid_{x\leftarrow k}$. Použitím této restrikce se jakákoli funkce f dá zapsat jako :

$$f = \overline{x}\cdot f\mid_{x\leftarrow 0} + x\cdot f\mid_{x\leftarrow 1}$$

Tento poznatek se označuje jako Shannonova expanze funkce f s ohledem na proměnnou x, ačkoli to bylo objeveno Boolem.

Spojením operace restrikce s dalšími algebraickými operacemi se dají vyjádřit další užitečné operace. Jednou z nich je třeba operace kompozice. Při kompozici funkce g nahrazuje proměnnou x funkce f :

$$f\mid_{x\leftarrow g} = \overline{g}\cdot f\mid_{x\leftarrow 0} + g\cdot f\mid_{x\leftarrow 1}$$

2.5 Konstrukce a manipulace s BDD

2.5.1 Apply funkce

Mnoho operací nad booleovskými funkcemi může být implementováno jako grafové algoritmy aplikované na (R)OBDD. Tyto algoritmy vyhovují jedné důležité vlastnosti a tou je vlastnost uzavřenosti uspořádání proměnných. Pokud před provedením operace

(R)OBDD odpovídá určitému uspořádání, pak provedení této operace na tom nic nezmění.

Apply operace generuje booleovskou funkci aplikováním algebraických operací na jiné funkce. Mějme například funkce *f* a *g* a binární operátor <*op*>, potom apply vrací funkci *f* <*op*> *g*. Funkce apply má pro nás ústřední význam, protože s jejím využitím můžeme spočítat množství jiných funkcí (např f xor 1 je komplement funkce).
Apply algoritmy procházejí graf do hloubky. Abychom zefektivnili výpočet apply operace, používá se hashovací tabulka. Druhá hashovací tabulka se používá k zajištění generování maximálně zredukovaného grafu.
Implementace apply operace se opírá o Shanonovu expanzi:

$$f < op > g = \bar{x} \cdot (f|_{x \leftarrow 0} < op > g|_{x \leftarrow 0}) + x(f|_{x \leftarrow 1} < op > g|_{x \leftarrow 1})$$

Ještě ovšem nevíme, jak získat restrikci daného (R)OBDD s ohledem na proměnnou x. Zcela jednoduše podle následujícího předpisu, kde r_f je kořen OBDD a x<=var(r_f) :

$f|_{x \leftarrow b}$

r_f	x<var(r_f)
low(r_f)	x=var(r_f) a b=0
high(r_f)	x=var(r_f) a b=1

to znamená, že restrikce je reprezentována tím samým grafem nebo jedním z podgrafů reprezentujícím potomka.

3 Vlastní implementace

Tak jako každý komplexnější problém tak i problém, jak správně implementovat nástroj pro práci s BDD, má více cest, kterými se může ubírat.
Musel jsem řešit několik otázek přímo souvisejících s implementací :

- načítat BDD přímo ze souboru PLA bez vnitřní reprezentace?
- nebo raději převést soubor PLA do vnitřní reprezentace?
- jakou datovou strukturou reprezentovat BDD?
- jak z vnitřní reprezentace PLA vytvořit BDD v přijatelném čase?

V následujících odstavcích se pokusím odpovědět.

3.1 Vnitřní reprezentace PLA souboru

Načítání BDD přímo z PLA bez nějakého druhu vnitřní reprezentace jsem celkem záhy zavrhl. Důvodů bylo hned několik :

- přístup k jednotlivým literálům by byl značně komplikovaný
- museli bychom vystačit se sekvenčním přístupem
- všechny činnosti související s PLA by byly z časového hlediska značně neefektivní, protože bychom stále přistupovali k souboru uloženému ve vnější paměti

Jak ale co nejefektivněji reprezentovat PLA soubor v programu? Vyzkoušel jsem více variant z kterých jsem nakonec vybral jednu, kterou popisuje kapitola 3.1.1.

3.1.1 Popis datové struktury PLAtab

Chtěl jsem vytvořit třídu, která by co nejlépe odrážela vlastnosti PLA souboru. Jelikož PLA soubor má formu po řádcích uložených termů, vznikla třída, která do jistá míry připomíná matici. Je ovšem doplněna o množství funkcí, které jsou nutné ke správnému načtení BDD nebo které jsem během psaní programu potřeboval a používal.
Celou třídu popisuje následující deklarace:

```
class PLAtab
  {
    int ** data;
    int    rad;
    int    sl;
   public:
            PLAtab ( int r = 0, int s=0, int load = 0 );
            PLAtab ( const PLAtab & src );
           ~PLAtab ( );

    PLAtab   & operator= ( const PLAtab & src );
    PLAtab     addR      ( );
    PLAtab     delR      ( int radek );
    PLAtab     delS      ( int sloupec );
    PLAtab     expanse1  (Node * parent, BDD * dest_bdd);
    PLAtab     expanse0  (Node * parent, BDD * dest_bdd);
    void       expEnd1   (Node * parent, BDD * dest_bdd);
    void       expEnd0   (Node * parent, BDD * dest_bdd);
    void       bdd_z_termu(BDD * dest_bdd);
    int        operator == ( const PLAtab & x )  const;
    int        operator != ( const PLAtab & x )  const;
    int *      operator[]  ( int ind );
    const int *  operator[]  ( int ind )  const;
    friend ostream & operator<< ( ostream & out, const PLAtab & x );
  };
```

Třídní proměnná ***data*** odpovídá dvojrozměrnému poli celočíselných hodnot, kde jednotlivé řádky pole odpovídají řádkům termů v PLA souboru. Proměnná ***rad*** udává počet řádků matice a je získána z PLA souboru z hodnoty za klíčovým slovem **.p** nebo je spočtena z počtu řádků PLA. A konečně proměnná ***sl*** udává počet sloupců matice a odpovídá počtu proměnných funkce a její hodnota je získána z PLA souboru z hodnoty za klíčovým slovem **.i**, které v souboru nesmí chybět.
Z metod samozřejmě nechybí konstruktor, kopírující konstruktor, destruktor (vrátí dynamicky přidělenou paměť pro ***data***) a operátor pro výstup do proudu.

Navíc jsou implementovány metody, které odlišují třídu PLAtab od obyčejné matice.

PLAtab addR() – slouží k přidání řádku do matice ***data***

PLAtab delR(**int** radek) – slouží ke smazání řádku, jehož číslo je v argumentu *radek*

PLAtab delS(**int** sloupec) – slouží ke smazání sloupce, jehož číslo je v argumentu *sloupec*

PLAtab expanse1 (Node * parent, BDD * dest_bdd)

PLAtab expanse0 (Node * parent, BDD * dest_bdd)

void expEnd1 (Node * parent, BDD * dest_bdd)

void expEnd0 (Node * parent, BDD * dest_bdd)

Všechny tyto metody slouží k načítání BDD za použití Shannonovy expanze. Vstupními argumenty jsou ukazatel na uzel (na kořen BDD) a ukazatel na BDD, které se vytváří. Podrobnější popis fungování těchto metod je uveden v kapitole 3.3.1.1.

Pro snazší manipulaci s PLAtab jsou přetíženy operátory = =,!= a pro indexování pole ***data*** operátor [].

Ovšem nejdůležitější metodou je bezesporu ***bdd_z_termu(BDD *)***, která způsobem popsaným v kapitole 3.3.1.2 vytvoří BDD z jednoho řádku PLA souboru, tedy z jednoho součinového termu.

3.2 Reprezentace BDD

Existuje mnoho způsobů jak implementovat BDD popsané v kapitole 2.2. Každá se principielně liší a jak už to bývá, každá má své výhody a nevýhody. Já jsem šel cestou, na rozdíl od knihovny CUDD, reprezentace BDD jako binárního stromu, kde každý vnitřní uzel má právě dva potomky.

Jak se ukázalo později, není tato volba příliš šťastná. Tento způsob reprezentace je sice nejnázornější a nejpřímočařejší, ale v žádném případě není nejrychlejší ve smyslu vytváření, mazání a modifikace BDD.

3.2.1 Třída pro reprezentaci uzlu stromu

Binární strom pro reprezentaci BDD se skládá ze dvou druhů uzlů – terminálních a neterminálních. Neterminální vnitřní uzly jsou v mém případě instance třídy Node, která vypadá takto:

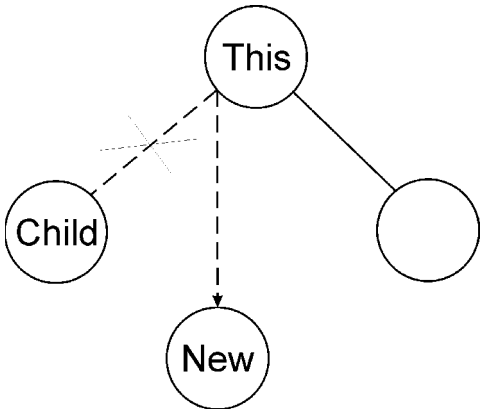
```
class Node {

protected:
  static unsigned int idCount;
  Node * Low;
  Node * High;
  unsigned int varId;
  unsigned int nodeId;
  NodeList  parentList;
public:
      Node();
      Node(int vid, Node * l=NULL, Node * h=NULL);

 void bypass(Node * child, Node * newNode);
 unsigned int getId(){ return nodeId; }
 virtual int getValue(){ return -1; }
 virtual bool isTerminal(){ return false; }
 virtual ostream& printInfo(ostream & out)const;
 friend ostream& operator<<(ostream & out,const Node & n);
};
```

Statická třídní proměnná ***idCount*** má dvě funkce. Jednak udává celkový počet vytvořených uzlů (ne počet uzlů BDD) a jednak řeší potřebu jednoznačného číslování uzlů v rámci BDD. Tato jednoznačná identifikace uzlu je uchována v proměnné ***nodeId***. Proměnné ***Low*** a ***High*** jsou odkazy na low a high potomka (ve smyslu naznačeném v kapitole 2.2) daného uzlu.
Každý uzel musí korespondovat s výhradně jednou proměnnou funkce, kterou BDD reprezentuje. Označení této proměnné je uchováno v třídní proměnné ***varId.***
Protože na uzel může odkazovat více rodičovských uzlů (uzlů které mají za low či high potomka tento uzel), potřebujeme během operací nad BDD znát všechny uzly, která na něj tímto způsobem ukazují. To nám zajišťuje zřetězený seznam ukazatelů na uzel ***parentList***.

Co se metod týče, nechybí konstruktory(bez parametru i s parametry), přetížený operátor pro výstup do proudu, který využívá virtuální metodu ***printInfo( )***, která vypíše všechny potřebné informace do proudu, který dostane jako svůj parametr.
Dalšími virtuálními metodami jsou ***getValue( ),*** která v případě neterminálního uzlu vrací -1, a metoda ***isTerminal( )***, která jak je vidět vrací **false**, protože se nejedná o terminální uzel.
Zajímavou je metoda ***bypass(Node * child, Node * newNode)***, která „ přemostí “ ukazatel na potomka child na newNode. Lépe je to snad vidět na Obrázku 3.
Poslední je metoda ***getId( )***, která jak název napovídá, vrací unikátní označení uzlu.



Obrázek 3 - Ukázka funkce metody bypass třídy Node

Terminální uzly(listy) představují instance třídy Term, která je odvozená od třídy Node. Její deklarace vypadá takto:

```
class Term : public Node{
  int value;

  public :
    Term( int vid, int val, Node * l=NULL, Node * h=NULL) : Node(vid,l,h) {value =
val; }
  virtual int getValue(){ return value; }
  virtual bool isTerminal(){ return true; }
  virtual ostream& printInfo(ostream& out) const;
};
```

Třída dědí vše z třídy Node a má navíc třídní proměnnou ***value***, která odpovídá hodnotě logické konstanty, kterou instance Termu reprezentuje.
Jsou také odpovídajícím způsobem přetíženy virtuální funkce zděděné z Node, abych mohl vhodně využít polymorfismu.

3.2.2 Třída pro reprezentaci BDD

Když už víme, jak vypadají uzly, z kterých se BDD skládá, můžeme si ukázat, jak vypadá třída BDD. Její deklarace:

```
class BDD {
  int varCt;
  Node * root;
  Mapa tab;
  Term * oneT;
  Term * zeroT;
  NodeList * arrayLevel;
void   copy_bdd(BDD & src_BDD);
void   del_nodes();

public:
      BDD(int var = 0);
      BDD(int *tt,int n);
      BDD(BDD & src);
     ~BDD();

BDD    operator=(BDD  &src);
  void   reduce();
  bool   isTautology();
  void   vypisLevely(ostream& out = std::cout) const;
  string nacti(char * dst );
  void   applyOR(BDD & sec, BDD & vysl);
  void   applyAND(BDD & sec,BDD & vysl);
  void   applyXOR(BDD & sec,BDD & vysl);
  void   print_to_file(char * dst);
```

```
  float  count_nodes();
 friend ostream& operator<<(ostream & out,const BDD & dia);
};
```

Každé BDD musí vědět, kolik proměnných má funkce, kterou reprezentuje. Tato hodnota je uchována v proměnné ***varCt.*** V proměnné ***root*** je ukazatel na kořen binárního stromu reprezentujícího vlastní logickou funkci. V souladu s redukčním pravidlem o duplicitních terminálech, obsahuje každé BDD pouze jeden terminální uzel od každé logické hodnoty – tedy ***oneT***(log 1) a ***zeroT***(log 0).

Pro snadnější manipulaci s BDD existuje třídní proměnná ***arrayLevel***, což je pole zřetězených seznamů ukazatelů na uzly. Každá položka pole(arrayLevel[0] až arrayLevel[varCt-1]) odpovídá jedné proměnné, obsahuje tedy uzly z jedné úrovně binárního stromu. ***Tab*** je hashovací tabulka, která se využívá při redukci BDD k zajištění odstranění redundantních uzlů.

Privátní funkce ***copy_bdd(*** BDD & ***)*** vytvoří mělkou kopii (nevytvoří fyzicky nové BDD, jen překopíruje ukazatele) BDD zadaného jako vstupní parametr. Je privátní, protože není vhodné, aby k ní uživatel měl přístup. Stejně tak ***del_nodes( )***, která smaže všechny uzly stromu.

Nechybí implicitní konstruktor, který pouze vytvoří terminální uzly a nastaví počet proměnných logické funkce. Zajímavější je konstruktor ***BDD(int*** *tt,***int** n***)***, který umí načíst BDD z pole reprezentující pravdivostní tabulku, kde *n* je počet proměnných logické funkce.

Je naimplementována metoda na minimalizaci BDD ***reduce( )***, metoda pro zjištění, zda je BDD tautologie ***isTautology( )***, metoda ***vypisLevely(*** ostream& ***)***, kterou využívá přetížený operátor pro výstup do proudu.

Metoda ***nacti(char*** *** dst)*** načte z PLA souboru *dst* BDD s využitím algoritmu popsaného v kapitole 3.3.1

Funkce ***applyOR(***BDD &, BDD &***), applyAND(***BDD &,BDD &***)*** a ***applyXOR(***BDD &, BDD &***)*** provádějí logické funkce odpovídající jejich názvům mezi dvěmi BDD.

Metoda ***print_to_file(char*** *** dst)*** slouží k výpisu BDD do souboru daného jako parametr ***dst.*** Metoda ***count_nodes( )*** spočítá vnitřní uzly BDD, její složitost není konstantní, ale je lineární v závislosti na počtu uzlů.

3.3 Popis nejdůležitějších algoritmů

V tomto odstavci bych rád popsal algoritmy, které jsem použil při načítání, redukci a implementaci logických operací nad BDD.

3.3.1 Algoritmus načítání BDD z PLA souboru

Jak načíst BDD ze zadaného PLA byl asi nejsložitější úkol, který mně při psaní této práce potkal. Přesněji řečeno problém nebyl BDD načíst, ale načíst ho v relativně krátkém čase.

3.3.1.1 Načítání s využitím Shannonovy expanse

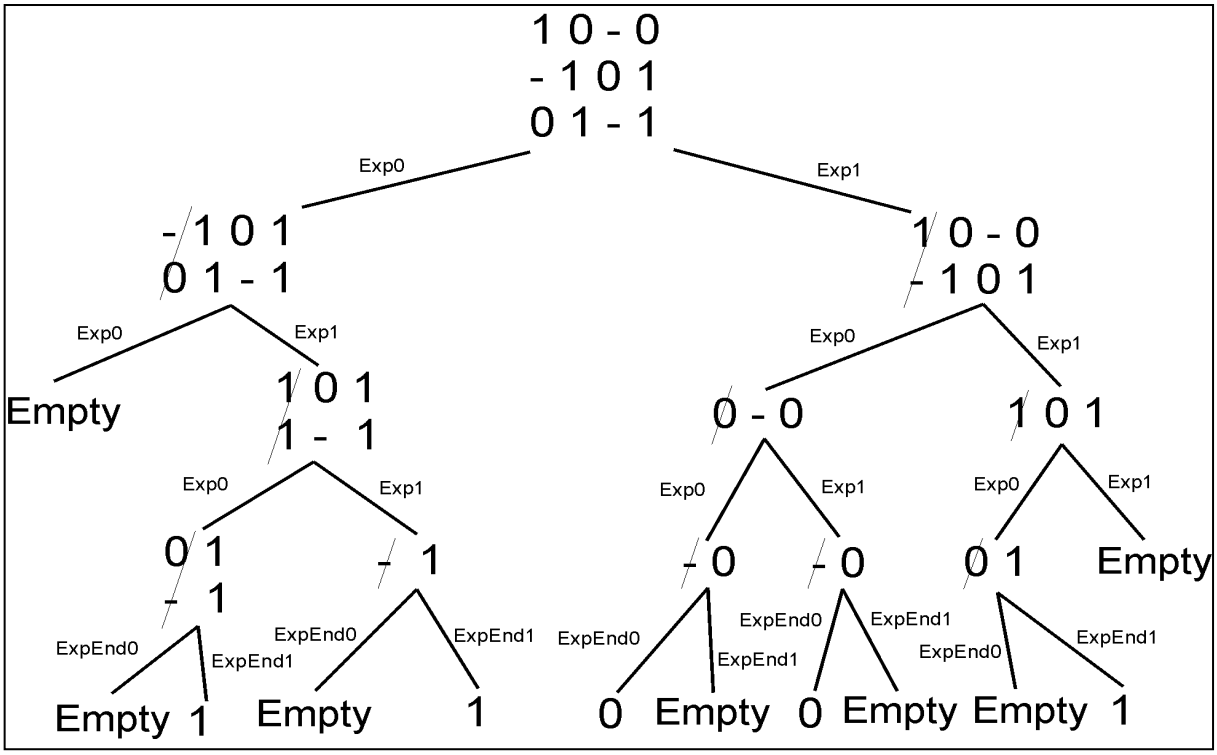
Jako první pokus jsem zkusil přesnou implementaci Shannonovy expanze logické funkce. Celý soubor se načte do vnitřní formy – do PLAtab. Poté stačí zavolat nad touto strukturou rekurzivní metody *expanse1(root)* a *expanse0(root)*, které vytvoří pravý a levý podstrom uzlu, který byl dodán jako parametr.
V každém kroku volání expansní funkce *expanse1( )* mně zajímají pouze termy(řádky), které mají v prvním sloupci(v proměnné podle které expanduji) hodnotu 1 nebo `-`. Řádky začínající nulou mohu s klidným svědomím vypustit. Jak jsem řekl jedná se o rekurzivní metody, takže se aplikují do té doby, dokud má tabulka nějaké řádky a sloupce. Před zavoláním expanzních funkcí na takto upravenou tabulku, odstraním první sloupec tabulky a posunu se tím na další proměnnou. Pokud zbývá už jediná proměnná, zavolají se funkce *expEnd1( )* a *expEnd0( )*, které celou expansi ukončí. Stejně funguje *expanse0( )*, s tím rozdílem, že ji zajímají řádky začínající 0 nebo `-`.

Nejlépe bude asi vše vidět na příkladě:
Mějme PLA tabulku (tab1.pla) :

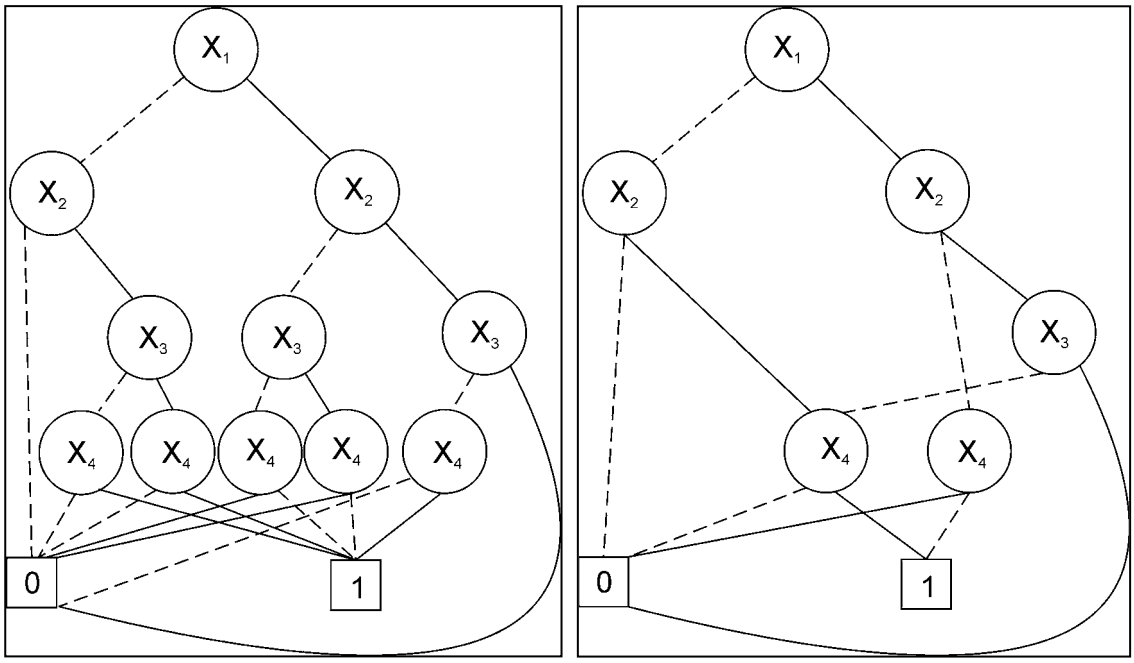
10-0 1
-101 1
01-1 1

Potom hierarchie volání metod *expanse1( )* a *expanse0( )* je vidět na Obrázku 4. Na Obrázku 5 je vidět binární strom, který tyto metody vytvoří.
Vytváření BDD takovýmto způsobem je značně neefektivní. Nejenom, že vznikne BDD, které rozhodně není minimalizované (jak je vidět na rozdílu mezi Obrázkem 5a a Obrázkem 5b), ale celý postup vytváření stromu z PLAtab má značnou výpočetní a

paměťovou náročnost. Každá metoda expanse musí pracovat se svojí kopií aktuální PLAtab (jak je vidět z Obrázku 4) a to v případě tabulek velkých rozměrů zdržuje. Proto jsem od této metody načítání upustil, nicméně i tento způsob je implementován a je plně funkční.



Obrázek 4 – Strom rekurzivního volání metod expanse



a) bez aplikace redukce b) po aplikaci redukce

Obrázek 5 – Strom vzniklý aplikací Shannonovy expanse na tab1.pla

3.3.1.2 Načítání s využitím apply funkce

Tento způsob je o poznání rychlejší. Spočívá v tom, že načítám postupně BDD ne z celé PLA tabulky, ale jen z jednoho řádku (tedy z jednoho termu) a na jednotlivé řádky aplikuji logický součet pomocí apply funkce OR (v duchu principu popsaném v kapitole 2.5.1) .

Načtení BDD z jednoho součinového termu je značně jednodušší než aplikace Shannonovy expanse na součet součinových termů. U každé proměnné provádíme test právě jednou. Nemůže se tedy stát, že by vznikl nějaký redundantní uzel. Pokud je proměnná v logické 1, směruji low potomka do nulového terminálu. Pokud je v logické 0, high potomek musí být nulový terminál a pokud je proměnná v hodnotě `-` (don't care), jednoduše se v BDD nevyskytne a přejdu na test další proměnné. Tento postup je implementován v metodě ***bdd_z_termu( )*** třídy PLAtab.

U apply funkce se s úspěchem dají využít vyhledávací nebo hashovaní tabulky. Jedna se využívá k tomu, abychom produkovali již redukované BDD a nemuseli explicitně po načtení BDD minimalizovat. Stačí si v tabulce uchovávat uzly, které již strom obsahuje a předtím než vytvoříme nový uzel, zkontrolujeme tabulku, zda takový uzel již neexistuje. Pokud ano, nevytváříme nový uzel, ale odkaz, který měl vézt do nového

uzlu, přesměrujeme do uzlu z tabulky. Pokud v tabulce ještě není, vytvoříme ho a přidáme do tabulky. Takto máme zajištěnou minimalizaci výsledného BDD, protože BDD načtené z jednoho řádku je redukované a apply funkce tuto redukci zachovává.

Druhá tabulka se využívá pro dynamické programování, abychom zefektivnili výpočet. Apply funkce je funkce rekurzivní a proto se může stát, že ji budeme volat s naprosto totožnými parametry vícekrát. To je samozřejmě časově neefektivní a lze tomu snadno zabránit. Do této druhé tabulky si ukládáme dvojice uzlů, pro kterou byla funkce apply počítána, spolu s výsledkem, který funkce vrátila. Poté stačí na začátku každé iterace funkce zkontrolovat tabulku, zda jsme pro dané argumenty již funkci nepočítali. Využitím této tabulky se dostáváme se složitostí operace apply na $O(|u_1| * |u_2|)$, kde |u1| je počet uzlů grafu u1 a |u2| počet uzlů grafu u2. Tímto je také daná maximální velikost výsledného grafu, potažmo velikost hashovaní tabulky, protože každé volání funkce přispěje maximálně jedním uzlem do výsledného grafu.

V pseudo kódu vypadá apply funkce nějak takto:

```
apply(u1, u2, operatorOR)
{
// vysledky – tabulka s již počítanými dvojicemi uzlů a s výsledkem jaký funkce vrátila
// tabUzlu – tabulka s již existujícími uzly stromu

if(vysledky.isIn(u1,u2)) return vysledky.get(u1,u2);
//pokud už jsem pro u1 a u2 apply funkci počítal, vrátím hodnotu z tabulky výsledků

if(u1.isTerminal() and u2.isTerminal()) return operatorOR(u1,u2);
//pokud jsou oba uzly terminály, tak jednoduše spočítám návratovou hodnotu

else if(var(u1) = var(u2))
{
        TmpA = apply(u1->Low,u2->Low,operatorOR); // levý podstrom
        TmpB = apply(u1->High,u2->High,operatorOR); // pravý podstrom
        if(TmpA == TmpB) return TmpA; //uplatnění 3.redukčního pravidla z kap.2.3.2

        //podívej se do tabulky uzlů, zda neexistuje uzel, který by měl low-potomka
```

```
          // TmpA a high-potomka TmpB
          if(tabUzlu.isIn(TmpA,TmpB))
           {
            return tabUzlu.get(TmpA,TmpB);
           }
        else
          {
            u = new Node(TmpA,TmpB);
            tabUzlu.insert(TmpA,TmpB,u)
          }

}

else if(var(u1) < var(u2))
{
        TmpA = apply(u1->Low,u2,operatorOR); // levý podstrom
        TmpB = apply(u1->High,u2,operatorOR); // pravý podstrom
        if(TmpA == TmpB) return TmpA; // uplatnění 3. redukčního pravidla z kap.2.3.2

          // podívej se do tabulky uzlů, zda neexistuje uzel, který by měl low potomka
          // TmpA a high potomka TmpB
          if(tabUzlu.isIn(TmpA,TmpB))
           {
             return tabUzlu.get(TmpA,TmpB);
           }
           else
             {
              u = new Node(TmpA,TmpB);
              tabUzlu.insert(TmpA,TmpB,u)
             }

else if(var(u1) > var(u2))
{
         TmpA = apply(u1,u2->Low,operatorOR); // levý podstrom
```

```
        TmpB = apply(u1,u2->High,operatorOR); // pravý podstrom
        if(TmpA == TmpB) return TmpA; //uplatnění 3. redukčního pravidla z kap.2.3.2

        //podívej se do tabulky uzlů, zda neexistuje uzel, který by měl low potomka
        // TmpA a high potomka TmpB
        if(tabUzlu.isIn(TmpA,TmpB))  return tabUzlu.get(TmpA,TmpB);
         else
            {
             u = new Node(TmpA,TmpB);
             tabUzlu.insert(TmpA,TmpB,u)
            }
}
}
```

3.3.2 Algoritmus pro redukci(minimalizaci) BDD

Pro účely načítání BDD Shannonovou expansí jsem implementoval algoritmus, který ze zadaného OBDD udělá ROBDD. Jedná se o přímou implementaci redukčních pravidel z kapitoly 2.3.2.

Jde o metodu implementovanou nad strukturou BDD. Prochází všechny uzly stromu, začíná od nejnižší úrovně v poli ***arrayLevel***, a hledá redundance.

Pro každý uzel určuje, zda daný uzel nemá v tabulce ***tab*** svůj ekvivalent. Pokud ano, potom všechny vstupní hrany (využití seznamu ***parentList***) přesměruje do tohoto ekvivalentu, který získá z tabulky, a kontrolovaný uzel smaže. Po projití všech uzlů stromu máme jistotu, že dané OBDD je už ROBDD!

Metoda popsaná pro přehlednost v částečném pseudokódu:

```
void BDD::reduce(){
tab.clear();
for(int i = početProměnných-1; i>0; i--)
  {
   pro všechny uzly u z arrayLevel[i] do
     {
       // 2. redukční pravidlo
```

```
        if(u->Low == u->High)
          {
           //iteruj přes všechny rodičovské uzly
           pro všechny uzly p z u->parentList do
             {
              p->bypass(u,u->Low);
             }
           delete u;
           continue;
          }
         // 3. redukční pravidlo
          if(tab.isEkvivalentIn(u)
             {
               //iteruj přes všechny rodičovské uzly
                pro všechny uzly p z u->parentList do
                 {
                  p->bypass(u,tab.getEkvivalent(u));
                 }
                delete u;
             }
           else //ekvivalent není v tabulce
             {
              tab.insert(u);
             }
        }
   }
// aby při tautologii zůstal opravdu jen terminál 1
  if (root->Low == root->High)
    {
      delete root;
      root = root->Low;
    }
}
```

3.3.3 Algoritmy pro logické operace mezi dvěmi BDD

Implementoval jsem funkce pro logický AND, OR a XOR. Všechny funkce využívají funkci apply, jak byla popsána v kapitole 3.3.1.2. Liší se jen operátorem, s kterým je apply funkce volána.

Proto pokud by bylo potřeba implementovat jinou operaci než tyto tři, stačí připsat příslušnou funkci a jsme hotovi.

4 Porovnání s jinými implementacemi

V této kapitole se pokusím porovnat moji implementaci s jinými dostupnými. Po dohodě s ing. Fišerem jsme vybrali knihovnu CUDD, která taktéž využívá BDD a potom metodu kolegy Chromého, která postupuje odlišným způsobem.

4.1 Knihovna CUDD

Knihovna CUDD je počin Fabia Somenziho z University of Colorado. Jde o nástroj pro manipulaci zejména s BDD, ale umí pracovat i s algebraickými rozhodovacími diagramy (ADD) a zero-suppressed diagramy (ZDD).

Tato knihovna je značně rozsáhlá a propracovaná. Pokud bych měl porovnat způsob implementace s mojí, nacházím velké rozdíly. Acyklický graf je zde reprezentován tabulkou, ne binárním stromem jako u mne. Jednotlivým uzlům grafu odpovídají řádky v této tabulce. Veškeré operace nad BDD jsou řešeny operacemi nad touto tabulkou a nutno říci, že tento způsob je mnohem rychlejší oproti mé implementaci.
Výrazným rozdílem je i použití komplementárních hran. Zatímco já používám pouze jeden druh hran, CUDD používá kromě normálních i negované hrany.
Načítání souborů PLA není implementováno, nicméně v rámci bakalářské práce Ondřeje Kološe[9] byla tato funkce doplněna a mohu proto v rámci praktických měření obě metody porovnat. Jak se ukázalo, je balík CUDD výpočetně velice rychlý a kompaktní.

4.2 Test tautologie výpočtem komplementu logické funkce

Tato implementace zjišťování tautologie vznikla jako semestrální projekt kolegy Radka Chromého[8].

Celý algoritmus funguje takto:
1. přečte se term ze souboru
2. podíváme se do paměti a zpracujeme ho podle pravidel
3. pokud je ještě něco v souboru, skočí na 1.
4. uloží výsledky do souboru a ukončí se.

Program postupně ořezává přečtené termy od termů uložených v paměti.
Na počátku je v paměti term obsahující samé "-", tj. term který je tautologie.

Nejdříve si zavedeme několik pojmů a identifikatorů:

Zaveďme nad množinou symbolů "-,0,1" operace "<,==,!=,NEG" definované takto:

1.) '0' != '1';

2.) '0' < '-'; '1' < '-';

3.) '0' == '0'; '1' == '1'; '2' == '2';

4.) NEG(0) = 1; NEG(1) = 0;

Nechť ***i*** je index booleovské proměnné v rozsahu <1..PROM_COUNT>, indexujme je v hranatých závorkách [i], kde PROM_COUNT je počet proměnných,
N je počet nově vzniklých termů
A je seznam indexů, pro které platí podmínka v pravidle 3., indexovatelný indexem j
j je "index" pro který platí podmínka v pravidle 3.
h[i] je hodnota i-té proměnné.
t_cten je přečtený term ze vstupního souboru
t_mem je aktuální term v paměti z množiny všech termů v paměti T_MEM
t_new(t_kopie) je nově vzniklý term, mající stejné ohodnocení h[i] jako term t_kopie, pro všechny i
t_mem.h[i] je hodnota i-té proměnné termu v paměti

Při přečtení termu se v paměti term zpracuje podle následujících pravidel:

1.) iff t_mem.h[i] == t_cten.h[i] nebo t_mem.h[i] < t_cten.h[i] pro všechny i v daném rozsahu, pak je čtený člen v paměti a z paměti se vypustí

2.) iff t_mem.h[i] != t_cten.h[i] pro nějaké i, pak se nic nemění(term nepokrývá žádnou proměnou v paměti)

3.) iff t_cten.h[i] < t_mem.h[i] pro nějaké i a současně neplatí ani 1. ani 2., pak dochází k rozštěpení termu:

Term se rozštěpí na N termů takto:

t_new_1(t_mem).h[A[j]] = NEG(t_cten.h[A[j]]) pro i pro které platí podmínka 3.

t_new_j(t_mem).h[A[j]] = NEG(t_new1.h[A[j]])

t_new_j(t_mem).h[A[j+1]] = NEG(t_cten.h[A[j+1]]) ... pro všechny 0 < j <= N

Tyto termy se uloží do paměti.

Jak se to dělá v příkladu :

nechť toto je obsah vstupního souboru:

00--

-10-

1111

-01-

Inicializace:

paměť: ----

krok1:

přečte: 00--

porovnání: 00-- ... ---- pravidlo 3. rozdělí na 2: množina A = {1,2}

rozdělení: 1--- + 01--

paměť: 1--- + 01--

krok2:

přečte: -10-

porovnání: 1--- ... -10- pravidlo 3. rozdělení na 2: množina A = {2,3}

rozdělení: 10-- 111-

porovnání: 01-- ... -10- pravidlo 3. rozdělení na 1: množina A = {1}

rozdělení: 011-

paměť: 10-- + 111- + 011-

krok3:

přečte: 1111

porovnání: 10-- ... 1111 pravidlo 2. nic neprovádí

porovnání: 111- ... 1111 pravidlo 3. rozdělení na 1: množina A = {4}

rozdělení: 1110

porovnání: 011- ... 1111 pravidlo 2. nic neprovádí

paměť: 10-- + 1110 + 011-

krok4:

přečte: -01-

porovnání: 10-- ... -01- pravidlo 3. rozdělení na 1: množina A = {1}

rozdělení: 100-

porovnání: 1110 ... -01- pravidlo 2. nic neprovádí

porovnání: 011- ... -01- pravidlo 2. nic neprovádí

paměť: 100- + 1110 + 011-

krok5:

nic se nečte

uložení do souboru:

"funkce neni tautologie"

100-

1110

011-

Složitost algoritmu:

Tento algoritmus prochází při každém kroku celou paměť, tzn. N porovnání na M kroků. Paměť se ale dynamicky mění a je těžké odhadnout složitost. Nicméně v každém kroku se může term v paměti násobit nejhůře mohutností množiny A (3. pravidlo) a tato mohutnost je menší než PROM_COUNT.

Maximální velikost potřebné paměti na uchování termu je v nejhorším případě $2^{(PROM\_COUNT\ -1)}$.

To znamená, že nejhorší časová složitost je pak $M * PROM\_COUNT * 2^{(PROM\_COUNT\ -1)}$ operací porovnání.

4.3 Naměřené hodnoty

Pro vytváření testovacích PLA souborů jsem využil generátor, jež vznikl jako bakalářská práce na naší fakultě a jehož autorem je Tomáš Měchura[7].

K testování byl použit počítač AMD Athlon XP 2200+ s 512 MB RAM.

Metoda výpočtu komplementu funkce podává velice dobré výsledky i pro rozsáhlé funkce, u kterých ovšem nepřesáhne don't care vstupních hodnot 50%. Pokud se tak stane, nedočkáme se v reálném čase výsledku i pro funkce s poměrně malým počtem vstupních proměnných (řádově už pro 50 vstupních proměnných).

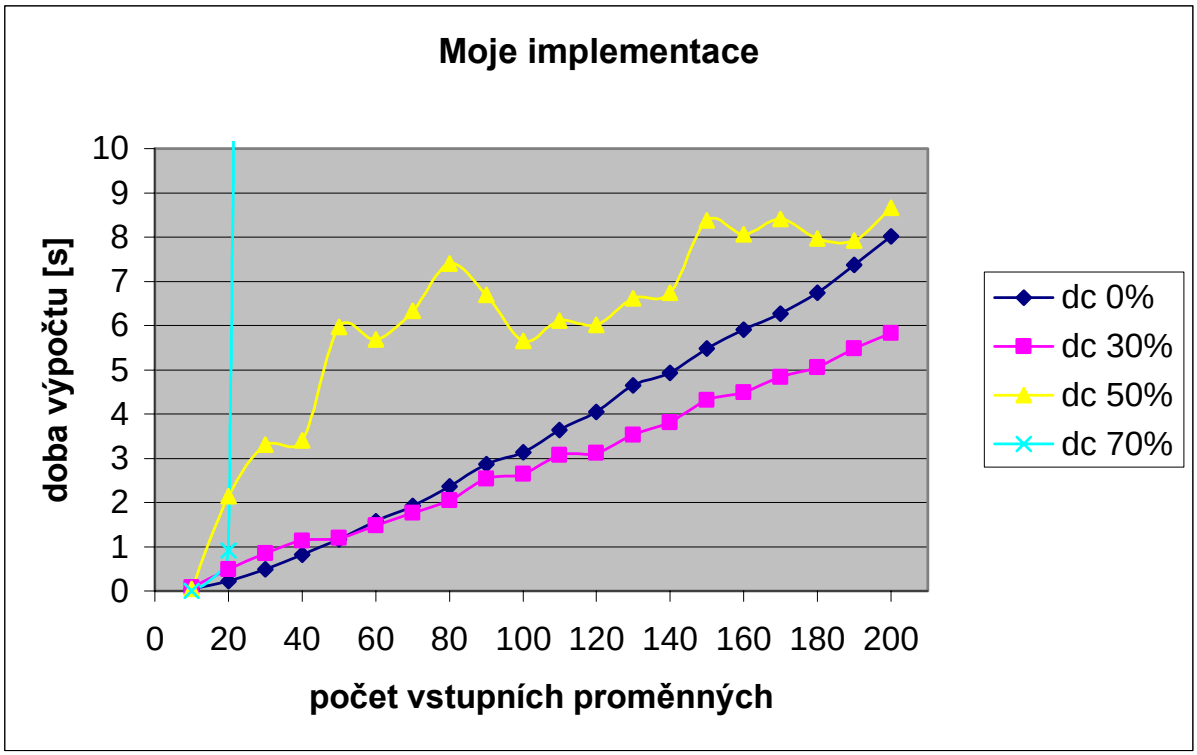
Empiricky získané výsledky poměrně jasně ukazují, že knihovna CUDD je velice dobře zpracovaná. Její problémy začínají, až když procento zastoupení don't care vstupů přesáhne 70%. Poté vzniká již BDD s takovým množstvím uzlů, že je není možno v přijatelném čase zpracovat. Ale pro funkce, které toto kritérium splňují, dává funkce výsledky v téměř konstantním čase i pro funkce s řádově stovkami vstupních proměnných.

Moje implementace podává solidní výsledky i pro velké funkce, pokud ovšem nemají na vstupu více jak 50% don't care hodnot. Při překročení této hranice musíme počítat s delším výpočetním časem. Nicméně jak je vidět z tabulek naměřených hodnot, výsledek na rozdíl od metody počítání komplementu funkce dostaneme.
Složitost mé implementace je dána několika dílčími složitostmi. Načtení BDD z jednoho termu má složitost lineární s počtem proměnných O(i), protože každá proměnná se testuje pouze jednou. Toto načítání se aplikuje p-krát, kde p je počet termů v PLA souboru. Po každém kroku se aplikuje apply funkce se složitostí O(|u1| * |u2|), kde u1 je počet uzlů jednoho BDD a u2 počet uzlů druhého. Dostáváme tak výpočetní složitost O(p * (i + |u1| |u2|)). Maximální počet uzlů u redukovaného BDD je dáno vztahem :

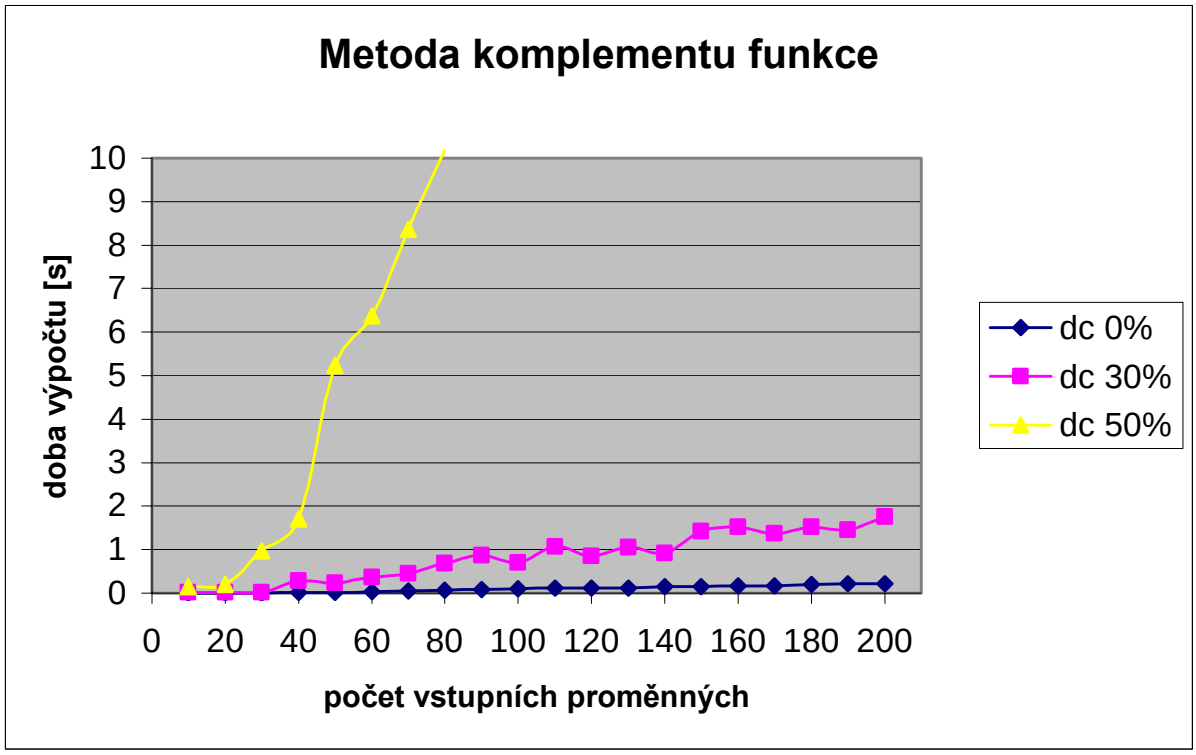
$$\Phi_{\max} = \sum_{K=0}^{i-1} \min(2^K, 2^{2^{i-K}} - 2^{2^{i-K-1}})$$

kde i je počet proměnných funkce.

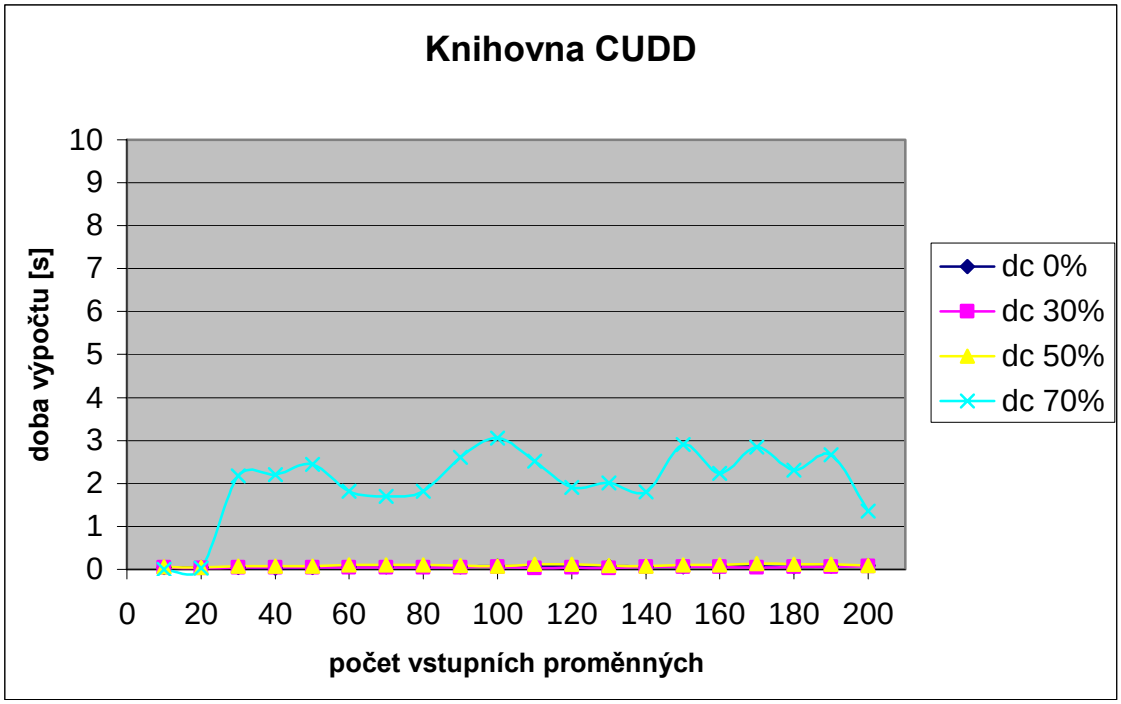
Jak reagují jednotlivé implementace na změnu počtu proměnných a don't care vstupů ukazují následující grafy:



Obrázek 6 – Závislost doby výpočtu na počtu vstupních proměnných u mé implementace



Obrázek 7 – Závislost doby výpočtu na počtu vstupních proměnných u implementace Radka Chromého



Obrázek 8 – Závislost doby výpočtu na počtu vstupních proměnných u knihovny CUDD

5 Závěr

V rámci této bakalářské práce vznikl nástroj, který je schopen o funkci zadané v notaci PLA Espresso rozhodnout, zda jde o tautologii, či nikoliv. Umí i něco navíc – aplikovat logické funkce (or,xor,and) mezi dvěmi BDD. Nevznikl na funkce tak bohatý balík, jakým je např. CUDD. Ani výkonnost není taková, jakou bych si představoval, když jsem přebíral zadání. Ale určitě je na čem stavět a co vylepšovat.

Nabízí se hned několik možných cest :

- Rozšířit pole působnosti i na vícevýstupové logické funkce
- Doplnit nové funkce pro práci s BDD
- Implementovat algoritmy pro změnu uspořádání proměnných (variable reordering)
- Změnit způsob reprezentace grafu z binárního stromu na hashovaní tabulku

Tento program je psaný v čistě objektovém duchu, který dovoluje jazyk C++. Proto by neměl být vážnější problém se s balíkem rychle seznámit a případně doprogramovat potřebné funkce.

6 Seznam literatury

[1] Randal E. Bryant: Symbolic Boolean manipulation with ordered binary decision diagrams, ACM Computing Surveys, 24(3):293-318, September 1992

[2] Henrik Reif Andersen: An Introduction to Binary Decision Diagrams, Lecture notes for 49285 Advanced Algorithms E97, October 1997.
Department of Information Technology, Technical University of Denmark.

[3] Randal E. Bryant: Graph based algorithms for Boolean function manipulation
IEEE Transactions on Computers, 8(C-35):667-691, 1986

[4] S.B. Avers: Binary decision diagrams.
IEEE Transactions on Computers C-27, 6 (Aug.), pp. 509–516. 1978

[5] G. Kolotov, I. Levin, V. Ostrovsky: Techniques for formal transformations of binary decision diagrams
School of Engineering, Tel-Aviv University

[6] M. Virius: Programování v C++, skriptum ČVUT Praha, 1999

[7] Tomáš Měchura: Parametrizovaný generátor náhodných booleovských funkcí, Bakalářská práce, ČVUT v Praze, FEL, 2006

[8] Radek Chromý: Detekce tautologie, výpočet komplementu booleovské funkce, Semestrální projekt, ČVUT v Praze, FEL, 2004

[9] Ondřej Kološ: Port balíku CUDD pod Windows,
Bakalářská práce, ČVUT v Praze, FEL, 2006

[10] http://vlsi.colorado.edu/~fabio/CUDD/

[11] http://www.engineering.uiowa.edu/~switchin/espresso.5.html

A. Formát souboru notace PLA Espresso

Jako zdroj dat pro načítání logických funkcí byly použity soubory typu PLA formátu Espresso, proto se krátce zmíním o jejich struktuře.
Na začátku souboru se nachází klíčová slova, která popisují strukturu a vlastnosti souboru. Po nich jsou vypsány řádky se součinovými termy. Celou funkci potom dostáváme jako logický součet všech těchto řádků (termů).

Klíčová slova

V následujících definicích [d] znamená celé desítkové číslo a [s] znamená řetězec. Každý soubor jich může obsahovat různý počet, ovšem dvě jsou vyžadovány vždy. Jsou to :

.i [d] – udává počet vstupních proměnných
.o [d] - udává počet výstupních proměnných

dále může obsahovat :

.mv [num_var] [num_binary_var] [d_1] . . . [d_n]
specifikuje počet proměnných, počet binárních proměnných a velikost každé vícehodnotové proměnné [d_1 až d_n].

.ilb [s_1] [s_2] . . . [s_n]
pojmenovává binární proměnné. Musí následovat po **.i** a **.o**(nebo po **.mv**). Názvů musí být stejný počet jako vstupních proměnných.

.ob [s_1] [s_2] . . . [s_n]
pojmenovává výstupní funkce. Musí následovat po **.i** a **.o**(nebo po **.mv**). Názvů musí být stejný počet jako výstupních proměnných.

.label var=[d] [s_1] [s_2] ...
u vícehodnotové logiky specifikuje názvy proměnné.

.type [s]

nastavuje logickou interpretaci charakteristické matice. Nabývá hodnot f, r, fd, fr, dr, nebo fdr.

.phase [s]

[s] je string tolika 0 nebo 1, kolik je výstupních funkcí.

Specifikuje polaritu pro každou výstupní funkci, která se má použít pro minimalizaci(1 specifikuje ON-set a 0 specifikuje OFF-set výstupní funkce).

.pair [d]

specifikuje počet párů proměnných, které spolu budou párovány použitím dvoubitových dekodérů.

.symbolic [s_0] [s_1] . . . [s_n] ; [t_0] [t_1] . . . [t_m] ;

specifikuje, že binární proměnné [s0] až [sn] jsou považovány za jednu vícehodnotovou proměnnou.

.symbolic-output [s_0] [s_1] . . . [s_n] ; [t_0] [t_1] . . . [t_m] ;

specifikuje, že výstupní funkce [s_0] až [s_n] jsou považovány za jednu symbolickou výstupní funkci.

.kiss

nastavuje se pro kiss minimalizaci

.p [d]

specifikuje počet součinových termů. Tyto termy následují ihned po tomto klíčovém slovu.

.e (.end)

značí konec PLA popisu

Pro naše účely vystačíme s **.i**, **.o**(vždy bude 1), může následovat **.ilb** a **.ob**, počet termů může být explicitně vyjádřen klíčovým slovem **.p** nebo se zjistí z počtu řádků termů. Konec PLA popisu také není povinný, potom jeho funkci přebírá konec souboru. Každý řádek (term) obsahuje vektor hodnot vstupních proměnných, které nabývají hodnot 1, 0, a `-`. Jednička znamená, že daný literál se v termu vyskytuje v nenegované podobě, 0 v negované podobě a konečně `-` daný literál se v termu nevyskytuje. Následuje mezera a výstupní hodnotu funkce(pro naše účely bude funkce zadána onsetem – tudíž výstup bude vždy 1).

Ukázkové příklady PLA

Přiklad 1 – Funkce o 5 proměnných, jsou zadány jejich jména, zadáno jméno výstupní proměnné, explicitně vyjádřen počet termů a nechybí ukončení PLA.

```
.i 5
.o 1
.ilb a b c d e
.ob y
.p 4
10-0- 1
-0-11 1
00-0- 1
-1101 1
.e
```

Představuje funkci : $y = a\bar{b}\bar{d} + \bar{b}de + \bar{a}\bar{b}\bar{d} + bc\bar{d}e$

Příklad 2 – funkce o 3 proměnných, minimální forma zadání

```
.i 3
.o 1
10- 1
001 1
101 1
1-0 1
```

Počet termů je získán z počtu řádků.

B. Tabulky naměřených hodnot

File	.i	.o	.p	% dc vstupů	Čas	tautologie
Dc0File10x100.pla	10	1	100	0	0,047	ne
Dc0File20x100.pla	20	1	100	0	0,218	ne
Dc0File30x100.pla	30	1	100	0	0,484	ne
Dc0File40x100.pla	40	1	100	0	0,812	ne
Dc0File50x100.pla	50	1	100	0	1,172	ne
Dc0File60x100.pla	60	1	100	0	1,578	ne
Dc0File70x100.pla	70	1	100	0	1,922	ne
Dc0File80100.pla	80	1	100	0	2,360	ne
Dc0File90x100.pla	90	1	100	0	2,859	ne
Dc0File100x100.pla	100	1	100	0	3,141	ne
Dc0File110x100.pla	110	1	100	0	3,640	ne
Dc0File120x100.pla	120	1	100	0	4,047	ne
Dc0File130x100.pla	130	1	100	0	4,641	ne
Dc0File140x100.pla	140	1	100	0	4,922	ne
Dc0File150x100.pla	150	1	100	0	5,485	ne
Dc0File160x100.pla	160	1	100	0	5,907	ne
Dc0File170x100.pla	170	1	100	0	6,265	ne
Dc0File180x100.pla	180	1	100	0	6,734	ne
Dc0File190x100.pla	190	1	100	0	7,375	ne
Dc0File200x100.pla	200	1	100	0	8,016	ne
Dc30file10x100.pla	10	1	100	30	0,078	ne
Dc30file20x100.pla	20	1	100	30	0,485	ne
Dc30file30x100.pla	30	1	100	30	0,843	ne
Dc30file40x100.pla	40	1	100	30	1,141	ne
Dc30file50x100.pla	50	1	100	30	1,203	ne
Dc30file60x100.pla	60	1	100	30	1,484	ne
Dc30file70x100.pla	70	1	100	30	1,766	ne
Dc30file80100.pla	80	1	100	30	2,047	ne
Dc30file90x100.pla	90	1	100	30	2,531	ne
Dc30file100x100.pla	100	1	100	30	2,641	ne
Dc30file110x100.pla	110	1	100	30	3,078	ne
Dc30file120x100.pla	120	1	100	30	3,125	ne
Dc30file130x100.pla	130	1	100	30	3,531	ne
Dc30file140x100.pla	140	1	100	30	3,813	ne
Dc30file150x100.pla	150	1	100	30	4,313	ne
Dc30file160x100.pla	160	1	100	30	4,484	ne
Dc30file170x100.pla	170	1	100	30	4,828	ne
Dc30file180x100.pla	180	1	100	30	5,053	ne
Dc30file190x100.pla	190	1	100	30	5,484	ne
Dc30file200x100.pla	200	1	100	30	5,828	ne

File	.i	.o	.p	% dc vstupů	Čas	tautologie
Dc50file10x100.pla	10	1	100	50	0,047	ne
Dc50file20x100.pla	20	1	100	50	2,141	ne
Dc50file30x100.pla	30	1	100	50	3,313	ne
Dc50file40x100.pla	40	1	100	50	3,406	ne
Dc50file50x100.pla	50	1	100	50	5,969	ne
Dc50file60x100.pla	60	1	100	50	5,687	ne
Dc50file70x100.pla	70	1	100	50	6,328	ne
Dc50file80100.pla	80	1	100	50	7,407	ne
Dc50file90x100.pla	90	1	100	50	6,687	ne
Dc50file100x100.pla	100	1	100	50	5,656	ne
Dc50file110x100.pla	110	1	100	50	6,109	ne
Dc50file120x100.pla	120	1	100	50	6,015	ne
Dc50file130x100.pla	130	1	100	50	6,609	ne
Dc50file140x100.pla	140	1	100	50	6,735	ne
Dc50file150x100.pla	150	1	100	50	8,375	ne
Dc50file160x100.pla	160	1	100	50	8,062	ne
Dc50file170x100.pla	170	1	100	50	8,407	ne
Dc50file180x100.pla	180	1	100	50	7,969	ne
Dc50file190x100.pla	190	1	100	50	7,922	ne
Dc50file200x100.pla	200	1	100	50	8,657	ne
Dc70file10x100.pla	10	1	100	70	0	ano
Dc70file20x100.pla	20	1	100	70	0,906	ne
Dc70file30x100.pla	30	1	100	70	106,781	ne
Dc70file40x100.pla	40	1	100	70	191,687	ne
Dc70file50x100.pla	50	1	100	70	280,922	ne
Dc70file60x100.pla	60	1	100	70	195,609	ne
Dc70file70x100.pla	70	1	100	70	163,687	ne
Dc70file80100.pla	80	1	100	70	128,125	ne
Dc70file90x100.pla	90	1	100	70	207,125	ne
Dc70file100x100.pla	100	1	100	70	245,219	ne
Dc70file110x100.pla	110	1	100	70	130,625	ne
Dc70file120x100.pla	120	1	100	70	172,422	ne
Dc70file130x100.pla	130	1	100	70	162,688	ne
Dc70file140x100.pla	140	1	100	70	146,015	ne
Dc70file150x100.pla	150	1	100	70	277,625	ne
Dc70file160x100.pla	160	1	100	70	217,547	ne
Dc70file170x100.pla	170	1	100	70	236,188	ne
Dc70file180x100.pla	180	1	100	70	197,359	ne
Dc70file190x100.pla	190	1	100	70	232,094	ne
Dc70file200x100.pla	200	1	100	70	144,203	ne

Tabulka 1 – Naměřené hodnoty pro moji implementaci

File	.i	.o	.p	% dc vstupů	Čas	tautologie
Dc0File10x100.pla	10	1	100	0	0	ne
Dc0File20x100.pla	20	1	100	0	0	ne
Dc0File30x100.pla	30	1	100	0	0	ne
Dc0File40x100.pla	40	1	100	0	0,016	ne
Dc0File50x100.pla	50	1	100	0	0,016	ne
Dc0File60x100.pla	60	1	100	0	0,031	ne
Dc0File70x100.pla	70	1	100	0	0,046	ne
Dc0File80100.pla	80	1	100	0	0,062	ne
Dc0File90x100.pla	90	1	100	0	0,078	ne
Dc0File100x100.pla	100	1	100	0	0,094	ne
Dc0File110x100.pla	110	1	100	0	0,110	ne
Dc0File120x100.pla	120	1	100	0	0,125	ne
Dc0File130x100.pla	130	1	100	0	0,125	ne
Dc0File140x100.pla	140	1	100	0	0,156	ne
Dc0File150x100.pla	150	1	100	0	0,157	ne
Dc0File160x100.pla	160	1	100	0	0,172	ne
Dc0File170x100.pla	170	1	100	0	0,172	ne
Dc0File180x100.pla	180	1	100	0	0,203	ne
Dc0File190x100.pla	190	1	100	0	0,219	ne
Dc0File200x100.pla	200	1	100	0	0,219	ne
Dc30file10x100.pla	10	1	100	30	0,016	ne
Dc30file20x100.pla	20	1	100	30	0,016	ne
Dc30file30x100.pla	30	1	100	30	0,014	ne
Dc30file40x100.pla	40	1	100	30	0,281	ne
Dc30file50x100.pla	50	1	100	30	0,235	ne
Dc30file60x100.pla	60	1	100	30	0,360	ne
Dc30file70x100.pla	70	1	100	30	0,453	ne
Dc30file80100.pla	80	1	100	30	0,688	ne
Dc30file90x100.pla	90	1	100	30	0,875	ne
Dc30file100x100.pla	100	1	100	30	0,703	ne
Dc30file110x100.pla	110	1	100	30	1,078	ne
Dc30file120x100.pla	120	1	100	30	0,859	ne
Dc30file130x100.pla	130	1	100	30	1,047	ne
Dc30file140x100.pla	140	1	100	30	0,922	ne
Dc30file150x100.pla	150	1	100	30	1,422	ne
Dc30file160x100.pla	160	1	100	30	1,515	ne
Dc30file170x100.pla	170	1	100	30	1,375	ne
Dc30file180x100.pla	180	1	100	30	1,516	ne
Dc30file190x100.pla	190	1	100	30	1,453	ne
Dc30file200x100.pla	200	1	100	30	1,750	ne

File	.i	.o	.p	% dc vstupů	Čas	tautologie
Dc50file10x100.pla	10	1	100	50	0,152	ne
Dc50file20x100.pla	20	1	100	50	0,203	ne
Dc50file30x100.pla	30	1	100	50	0,969	ne
Dc50file40x100.pla	40	1	100	50	1,703	ne
Dc50file50x100.pla	50	1	100	50	5,235	ne
Dc50file60x100.pla	60	1	100	50	6,375	ne
Dc50file70x100.pla	70	1	100	50	8,359	ne
Dc50file80100.pla	80	1	100	50	10,187	ne
Dc50file90x100.pla	90	1	100	50	12,859	ne
Dc50file100x100.pla	100	1	100	50	12,265	ne
Dc50file110x100.pla	110	1	100	50	17,047	ne
Dc50file120x100.pla	120	1	100	50	13,938	ne
Dc50file130x100.pla	130	1	100	50	21,469	ne
Dc50file140x100.pla	140	1	100	50	20,766	ne
Dc50file150x100.pla	150	1	100	50	1911,67	ne
Dc50file160x100.pla	160	1	100	50	22,469	ne
Dc50file170x100.pla	170	1	100	50	149,672	ne
Dc50file180x100.pla	180	1	100	50	404,204	ne
Dc50file190x100.pla	190	1	100	50	263,953	ne
Dc50file200x100.pla	200	1	100	50	337,672	ne

Tabulka 2 – Naměřené hodnoty pro implementaci Radka Chromého

File	.i	.o	.p	% dc vstupů	Čas	tautologie
Dc0File10x100.pla	10	1	100	0	0,047	ne
Dc0File20x100.pla	20	1	100	0	0,047	ne
Dc0File30x100.pla	30	1	100	0	0,047	ne
Dc0File40x100.pla	40	1	100	0	0,031	ne
Dc0File50x100.pla	50	1	100	0	0,047	ne
Dc0File60x100.pla	60	1	100	0	0,063	ne
Dc0File70x100.pla	70	1	100	0	0,063	ne
Dc0File80100.pla	80	1	100	0	0,062	ne
Dc0File90x100.pla	90	1	100	0	0,063	ne
Dc0File100x100.pla	100	1	100	0	0,079	ne
Dc0File110x100.pla	110	1	100	0	0,078	ne
Dc0File120x100.pla	120	1	100	0	0,078	ne
Dc0File130x100.pla	130	1	100	0	0,078	ne
Dc0File140x100.pla	140	1	100	0	0,063	ne
Dc0File150x100.pla	150	1	100	0	0,063	ne
Dc0File160x100.pla	160	1	100	0	0,078	ne
Dc0File170x100.pla	170	1	100	0	0,079	ne
Dc0File180x100.pla	180	1	100	0	0,063	ne
Dc0File190x100.pla	190	1	100	0	0,078	ne
Dc0File200x100.pla	200	1	100	0	0,094	ne

File	.i	.o	.p	% dc vstupů	Čas	tautologie
Dc30file10x100.pla	10	1	100	30	0,047	ne
Dc30file20x100.pla	20	1	100	30	0,032	ne
Dc30file30x100.pla	30	1	100	30	0,047	ne
Dc30file40x100.pla	40	1	100	30	0,047	ne
Dc30file50x100.pla	50	1	100	30	0,047	ne
Dc30file60x100.pla	60	1	100	30	0,047	ne
Dc30file70x100.pla	70	1	100	30	0,047	ne
Dc30file80100.pla	80	1	100	30	0,047	ne
Dc30file90x100.pla	90	1	100	30	0,047	ne
Dc30file100x100.pla	100	1	100	30	0,063	ne
Dc30file110x100.pla	110	1	100	30	0,032	ne
Dc30file120x100.pla	120	1	100	30	0,047	ne
Dc30file130x100.pla	130	1	100	30	0,031	ne
Dc30file140x100.pla	140	1	100	30	0,062	ne
Dc30file150x100.pla	150	1	100	30	0,062	ne
Dc30file160x100.pla	160	1	100	30	0,062	ne
Dc30file170x100.pla	170	1	100	30	0,046	ne
Dc30file180x100.pla	180	1	100	30	0,062	ne
Dc30file190x100.pla	190	1	100	30	0,062	ne
Dc30file200x100.pla	200	1	100	30	0,078	ne
Dc50file10x100.pla	10	1	100	50	0,063	ne
Dc50file20x100.pla	20	1	100	50	0,047	ne
Dc50file30x100.pla	30	1	100	50	0,078	ne
Dc50file40x100.pla	40	1	100	50	0,078	ne
Dc50file50x100.pla	50	1	100	50	0,079	ne
Dc50file60x100.pla	60	1	100	50	0,109	ne
Dc50file70x100.pla	70	1	100	50	0,110	ne
Dc50file80100.pla	80	1	100	50	0,109	ne
Dc50file90x100.pla	90	1	100	50	0,094	ne
Dc50file100x100.pla	100	1	100	50	0,078	ne
Dc50file110x100.pla	110	1	100	50	0,125	ne
Dc50file120x100.pla	120	1	100	50	0,125	ne
Dc50file130x100.pla	130	1	100	50	0,094	ne
Dc50file140x100.pla	140	1	100	50	0,078	ne
Dc50file150x100.pla	150	1	100	50	0,109	ne
Dc50file160x100.pla	160	1	100	50	0,110	ne
Dc50file170x100.pla	170	1	100	50	0,141	ne
Dc50file180x100.pla	180	1	100	50	0,125	ne
Dc50file190x100.pla	190	1	100	50	0,125	ne
Dc50file200x100.pla	200	1	100	50	0,094	ne

File	.i	.o	.p	% dc vstupů	Čas	tautologie
zfile10x100.pla	10	1	100	70	0,000	ano
zfile20x100.pla	20	1	100	70	0,031	ne
zfile30x100.pla	30	1	100	70	2,172	ne
zfile40x100.pla	40	1	100	70	2,203	ne
zfile50x100.pla	50	1	100	70	2,437	ne
zfile60x100.pla	60	1	100	70	1,812	ne
zfile70x100.pla	70	1	100	70	1,704	ne
zfile80100.pla	80	1	100	70	1,812	ne
zfile90x100.pla	90	1	100	70	2,609	ne
zfile100x100.pla	100	1	100	70	3,062	ne
zfile110x100.pla	110	1	100	70	2,516	ne
zfile120x100.pla	120	1	100	70	1,906	ne
zfile130x100.pla	130	1	100	70	2,016	ne
zfile140x100.pla	140	1	100	70	1,797	ne
zfile150x100.pla	150	1	100	70	2,907	ne
zfile160x100.pla	160	1	100	70	2,234	ne
zfile170x100.pla	170	1	100	70	2,844	ne
zfile180x100.pla	180	1	100	70	2,312	ne
zfile190x100.pla	190	1	100	70	2,672	ne
zfile200x100.pla	200	1	100	70	1,359	ne

Tabulka 3 – Naměřené hodnoty pro knihovnu CUDD

C. Obsah CD

V kořenovém adresáři přiloženého cd se nachází binární spustitelný soubor tautology.exe a podadresáře:

src - zde jsou uloženy zdrojové soubory
texty - složka obsahuje tuto práci v elektronické podobě
testy - obsahuje testovací soubory a naměřená data