České vysoké učení technické v Praze

Fakulta elektrotechnická

Diplomová práce

Detekce tautologie

Bc. Jan Schindler

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, navazující magisterský

Obor: Výpočetní technika

květen 2009

Poděkování

Rád bych na tomto místě poděkovat všem, kteří mi jakýmkoli způsobem pomáhali při tvorbě této diplomové práce. Zvláště pak děkuji svému vedoucímu, Ing. Petru Fišerovi, Ph.D., za odbornou pomoc, podnětné nápady a čas, který mi věnoval při návrhu a implementaci zde popisovaného projektu.

Prohlášení

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

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

V Hradci Králové, dne 15.5. 2009 .....

Abstract

This thesis deals with the design and the implementation of the tautology detection tool, which is based on algorithms from the Espresso, accepts input data in the Espresso PLA format and is integrable to the kernel of the programm BOOM II. Emphasis is placed on the time efficiency of solution found in the diversity of the input logic functions.

Abstrakt

Tato práce se zabývá návrhem a realizací nástroje na detekci tautologie, který je založen na algoritmech z nástroje Espresso, vstupní data přijímá ve formátu Espresso PLA a je integrovatelný to jádra programu BOOM II. Důraz je kladen na časovou efektivitu nalezeného řešení při různorodosti charakteru logických funkcí na vstupu.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Obrovský rozvoj výpočetní techniky a počítačů ve druhé polovině 20. století byl podmíněn nejen rozvojem nejrůznějších výrobních technologií, ale také pokrokem v teoretických disciplínách, které poskytly potřebný aparát pro navrhování, vývoj, ladění a další související činnosti. Jednou z disciplín, které se v tomto směru uplatnily nejvíce, je i matematická logika. V roce 1854 přišel anglický logik George S. Boole (1815-1864) s takovým modelem matematické logiky, ve kterém vystačil jen se třemi základními operátory (and, or a not), a s jejich pomocí dokázal z jednotlivých výroků sestavovat složitější formule stejným způsobem, jakým se v matematice (konkrétně v algebře) sestavují matematické vzorce. Svou logiku pak mohl formálně vybudovat jako algebru, které se dodnes říká Booleova algebra.

Booleova algebra se stala základním teoretickým aparátem pro modelování kombinačních obvodů číslicových počítačů, protože technicky nejsnadněji realizovatelné jsou takové konstrukční prvky, které mají jen dva možné stavy (0, 1). Těm bude odpovídat taková Booleova algebra, která má právě jen tyto dva prvky. Velmi rozsáhlé integrované obvody - VLSI[1] hrají dnes hlavní roli při návrhu komplexních elektronických systémů. K jejich realizaci je zapotřebí aparát, který pomocí efektivních algoritmů dokáže minimalizovat složitost návrhu a umožní tak konečnou implementaci VLSI na čip.

Navrhované systémy lze tedy reprezentovat pomocí logických funkcí Booleovy algebry a tyto funkce pak lze minimalizovat prostřednictvím softwarových nástrojů, které jsou založeny na souborech algoritmů zpracovávajících jednotlivé operace s logickými funkcemi. Jednou ze základním otázek při minimalizaci návrhu je zodpovězení otázky, zda-li je daná funkce zadaná ve formě DNF[2] tautologií[3]. Návrh a realizace takovéhoto nástroje, který bude schopen vyhodnotit, zda-li je logická funkce na vstupu tautologií, je cílem této diplomové práce. Výsledný nástroj by dále měl být začlenitelný do jádra minimalizačního nástroje BOOM II [6]. Hlavní důraz by pak měl být kladen na časovou efektivitu a univerzálnost vůči různorodosti vstupních instancí.

[1] Very Large Scale Integration

[2] Disjunctive normal form - „součty součinů“

[3] Funkce je tautologií právě tehdy, nabývá-li pravdivé hodnoty při jakémkoliv ohodnocení vstupních proměnných.

2 Popis problému, specifikace cíle

Cílem této kapitoly bude vymezení požadavků, které byly stanoveny na navrhovaný systém. Dále bude následovat rešeršní srovnání již existujících implementací a nakonec budou definovány některé základní pojmy, na které se budeme v této diplomové práci odkazovat.

2.1 Vymezení cílů DP a požadavků na navrhovaný systém

Byly stanoveny tyto požadavky na navrhovaný systém:

- Navrhnout a v jazyce C++ implementovat nástroj, který bude schopen vyhodnotit, zda-li je funkce na vstupu zadaná ve formě součtu součinů (DNF[1]) tautologií.
- Důraz by měl být kladen na časovou efektivitu výpočtu a uvažovat by se mělo i s možností kombinace různých algoritmů pro detekci tautologie, za účelem optimalizace rychlosti.
- Nástroj by měl být optimalizován tak, aby efektivně pracoval s různorodými instancemi vstupních logických funkcí. Každou funkci lze charakterizovat mj. pomocí tří základních kriterií (počet vstupních proměnných, počet termů, procentuelní zastoupení DC[2] znaků). Více se touto problematikou, různorodostí instancí, zabývá kapitola 5.
- Program by měl být primárně založen na algoritmech použitých v nástroji na minimalizaci logických funkcí, Espresso [2].
- Výsledný program by měl tvořit zapouzdřený celek, který bude možné začlenit do jádra nástroje na minimalizaci logických funkcí, BOOM II [6]. Jako vstup bude používat struktury vlastní programu BOOM II. Více o těchto datových strukturách se zle dočíst v podkapitole 2.2.2.

Cílem této diplomové práce je tedy vytvořit takto výšše specifikovaný funkční celek a otestovat jej na různých testovacích úlohách. Tyto výsledky pak budeme porovnávat s

[1] Viz podkapitola 2.2.1.

[2] „Don't Care stav“ - na hodnotě tohoto vstupu v daném termu nezáleží.

řešeními již hotových nástrojů na detekci tautologie. Tyto programy jsou blíže popsány v podkapitolách 2.4 - 2.7.

2.2 Definice základních pojmů

Následující podkapitoly si kladou za cíl přiblížit některé základní pojmy, na které se budeme v této diplomové práci odkazovat. Podrobnější definice těchto pojmů jsou pak k dispozici v uvedených zdrojových materiálech.

2.2.1 Logická funkce

Logickou funkci lze popsat následujícím způsobem [4]:

Každé formuli o n logických proměnných odpovídá pravdivostní tabulka. Na tuto tabulku se můžeme dívat jako na zobrazení, které každé n-tici 0 a 1 přiřazuje 0 nebo 1. Řádek pravdivostní tabulky je popsán n-ticí 0 a 1, hodnota je pak pravdivostní hodnota formule pro toto dosazení za logické proměnné. Zobrazení z množiny všech n-tic 0 a 1 do množiny $\{0, 1\}$ se nazývá *Booleova funkce*. Naopak platí, že pro každou *Booleovu funkci* existuje formule, která této funkci odpovídá.

Booleova funkce. *Booleovou funkcí* n proměnných, kde n je přirozené číslo, rozumíme každé zobrazení $f : \{0,1\}^n \to \{0,1\}$ tj. zobrazení, které každé n-tici $(x_1, x_1, ..., x_n)$ nul a jedniček přiřazuje nulu nebo jedničku (označenou $f(x_1, x_1, ..., x_n)$).

Disjunktivní normální tvar (DNF). Literál je logická proměnná nebo negace logické proměnné. Řekneme, že formule je v disjunktivním normálním tvaru, zkráceně v DNF, jestliže je disjunkcí jedné nebo několika formulí, z nichž každá je literálem nebo konjunkcí literálů. Poznamenejme, že literálu nebo konjunkci literálů se také říká minterm. Jestliže každý minterm obsahuje všechny proměnné, říkáme, že se jedná o úplnou DNF.

2.2.2 Formát PLA

Jako zdroj dat pro načítání logických funkcí budou použity soubory typu PLA formátu Espresso, proto se krátce zmíníme o jejich struktuře. Pomocí PLA, lze reprezentovat

logickou funkci ve formátu DNF.

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).

Ukázkový příklad PLA

Příklad 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
```

Výše uvedené PLA představuje funkci: $y = a\overline{bd} + \overline{b}de + \overline{abd} + bc\overline{d}e$

2.2.3 Asymptotická složitost ve vztahu k detekci tautologie

Asymptotická složitost slouží jako nástroj pro porovnávání efektivity a rychlosti vykonávání jednotlivých algoritmů a definuje několik základních tříd [11] např.: P, NP, $co - NP$.

Budeme-li chtít zařadit problém detekce tautologie, do konkrétní třídy dojdeme k závěru, že spadá mezi úlohy kategorie $co - NP$, která je protějškem třídy NP, a to protože u problémů třídy NP dostáváme při kladné odpovědi na ověření navrhovaného řešení (tzv. *konfigurace*) kladnou odpověď na zadání problému (např. Je daná funkce splnitelná? - stačí při ověřování najít jedno ohodnocení vstupních proměnných-konfiguraci, při kterém je funkce splněna a získáme tím kladnou odpověď na původní otázku - ověříme certifikát). Budeme-li se ptát např. na otázku: Je daná funkce vždy splněna (je tau-

tologií)? - nezískáme kladným ohodnocením konfigurace certifikát. Tento problém je tedy z třídy $co - NP$.

2.2.4 Kofaktor logické funkce

Při analýze a realizaci algoritmu na detekci tautologie budeme často používat pojmu *kofaktor*. Nyní uvedeme jeho formální definici [1].

Nechť f je logická funkce a $x = (x_1, x_2, ..., x_n)$ jsou její proměnné, pak kofaktor f_a z funkce f a jejího literálu $a = x_i$ nebo $a = \overline{x_i}$ je:

$f_{x_i}(x_1, x_2, ..., x_n) = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$,

$f_{\overline{x_i}}(x_1, x_2, ..., x_n) = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$.

Budeme-li kofaktor f_C vytvářet z logické funkce f s ohledem k zadanému termu C budou součástí vzniklého kofaktoru ty termy, které se shodují v hodnotách svých proměnných s daných termem C. Tento postup lze sledovat na následujícím příkladu:

$f = ab\overline{c} + \overline{a}b\overline{d} + abcd$, $C = ab$,

$f_C = f_{ab} = \overline{c} + cd$.

2.3 Rešeršní zpracování existujících implementací

Po vymezení požadavků na navrhovaný systém začalo testování již existujících programů na detekci tautologie, které doporučil vedoucí této práce, pan Fišer. Cílem bylo zjistit, v jakých časech dokáží tyto programy řešit jednotlivé vstupní instance. Toto úvodní testování nebylo nijak rozsáhlé ani systematické, šlo především o získání představy, v jakých řádech se budou pohybovat časy potřebné pro nalezení řešení. Mnohem rozsáhlejší a systematičtější testování se zaměřením na vlivy jednotlivých kritérií charakterizujících danou logickou funkci na vstupu, bude provedeno po naprogramování vlastního nástroje na detekci tautologie, který je předmětem této diplomové práce. Tomuto testování je věnována kapitola 5.

Následující podkapitoly se zabývají základním popisem implementací, se kterými bude porovnáván náš program.

2.4 Výpočet komplementu

Tato implementace nástroje na detekci tautologie vznikla jako semestrální projekt kolegy Radka Chromého [3]. V dokumentaci tohoto projektu, je podrobně popsáno jak celý algoritmus pracuje. Hlavní myšlenka je taková, že dochází k postupnému sekvenčnímu zpracování všech termů na vstupu. Každý term se pak porovnává po znacích se všemi termy v paměti. Mohou nastat dvě situace:

1. term již je v paměti obsažen a je možné jej vypustit
2. term v paměti není celý obsažen, je třeba jej rozdělit na patřičný počet termů, které se pak přidají do paměti.

Na konci výpočtu jsou v paměti termy, které tvoří dohromady komplement vstupní funkce. Je-li komplement prázdny, byla původní funkce tautologie.

Asymptotická složitost tohoto programu je exponencielní, protože algoritmus prochází při každém kroku celou paměť, tzn. N porovnání na M kroků. Paměť se sice dynamicky mění, nicméně v nejhorším případě v každém kroku se může term v paměti násobit nejhůře mohutností množiny a tato mohutnost je menší než $POCET\_PROMENNYCH$. To znamená, že nejhorší časová složitost je pak:

$$M * POCET\_PROMENNYCH * 2^{POCET\_PROMENNYCH-1}.$$

Hodnocení:

Tento algoritmus na detekci tautologie založený na hledání komplementu zadané logické funkce najde řešení velmi rychle u malých instancí zhruba do 20 vstupních proměnných a 500 termů. S rostoucí velikostí instancí se pak čas potřebný na nalezení řešení rapidně zvyšoval. Někdy program skončil neočekávanou výjimkou. S roustoucím procentem DC znaků se rychlost programu zvyšovala.

2.5 Binární rozhodovací diagramy

Tento projekt na detekci tautologie vznikl jako bakalářská práce kolegy Michala Navrkala [9]. V dokumentaci této práce je podrobně popsáno, jak celý program pracuje. V následujícím

odstavci připomeneme pouze základní princip.

2.5.1 Definice BDD

Pod zkratkou BDD se skrývá stromová struktura, která může být velmi vhodná pro detekci tautologie a to v případě, že se nám ji podaří „uspořádat a redukovat“[3]. Poté můžeme v konstantním čase detekovat, zda-li je takto reprezentovaná logická funkce tautologie, protože pokud ano, pak její graf má podobu jedničkového terminálu.

Každá funkce je reprezentovaná jako orientovaný acyklický graf, který tvoří vnitřní uzly (neterminální uzly). Ty pak korespondují vždy s určitou proměnnou funkce, kterou dané BDD[4] 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 může posloužit obrázek 2.1 [12] ilustrující reprezentaci funkce $f(x1,x2,x3)$ definovanou pravdivostní tabulkou. Každý neterminální uzel v je označen proměnnou var(v) a má dvě hrany směřující ke dvěma potomkům :

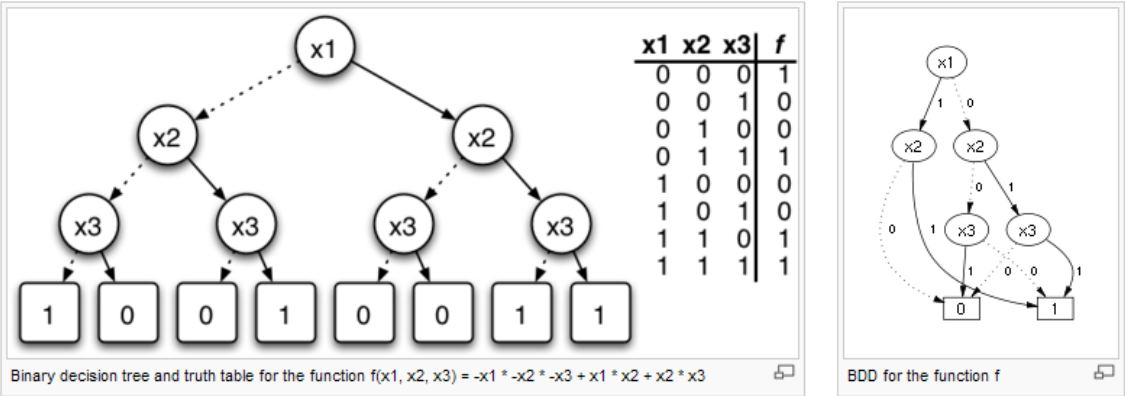
- low(v) označen čárkovanou čarou (případ, kdy proměnná je nulová),
- high(v) označen plnou čarou (případ, kdy proměnná je jedničková).

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.

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.

[3] Viz podkapitola 2.5.2.

[4] Binary Decision Diagram, Binární rozhodovací diagram



Obrázek 2.1: Příklad BDD

2.5.2 Uspořádání a redukce

Pro snazší práci s BDD je třeba provést následující dvě operace: uspořádání a redukci. Po použití redukce získáme OBDD (ordered BDD). Aby se BDD stalo OBDD musíme vyžadovat určité uspořádání proměnných. Pro každý uzel u a jeho potomka v musí platit následující nerovnost:

$$var(u) < var(v).$$

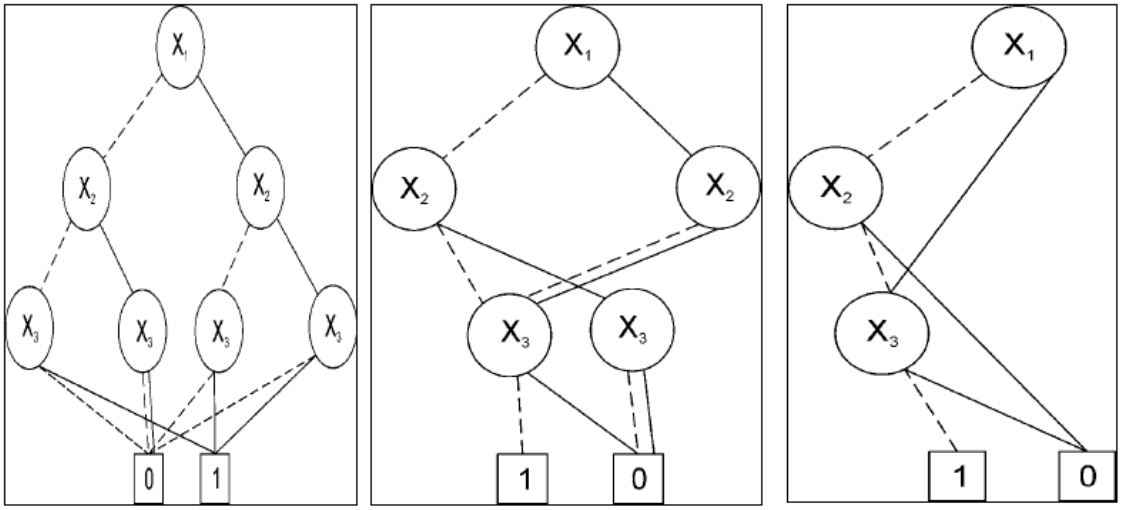
Takto popsaný OBDD má však ještě jednu nevýhodu a tou je možná redundance dat. Abychom ji odstranili, nadefinujeme tři transformační pravidla pro takovýto graf:

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-li redukovaný uspořádaný binární rozhodovací diagram.



Obrázek 2.2: Použití redukčních pravidel na BDD

Jak je vidět z obrázku 2.2 [9] první pravidlo zredukovalo počet terminálních uzlů z 8 na 2. Druhé pravidlo odstranilo dva uzly označené proměnnou $x3$ a nakonec třetí pravidlo odstranilo dva redundantní uzly, jejichž vstupní hrany jsme „přemostili“ do jednoho z potomků.

ROBDD má hned několik výhod oproti BDD, např. můžeme v konstantním čase detekovat, zda-li je takto reprezentovaná logická funkce tautologie, protože pokud ano, pak její graf má podobu jedničkového terminálu.

Hodnocení:

Tento nástroj, rovněž jako algoritmus popisovaný v části 2.4, poskytuje řešení v relativně rozumných časech pro malé instance. Jednu testovanou funkci z asi 30 testovaných však program označil za tautologii, i když jí nebyla. S rostoucím procentem zastoupení DC znaků výkonnost klesá.

2.6 Knihovna s nástrojem Espresso

Cílem bakalářské práce [8] kolegy Martina Miklánka bylo vytvořit knihovnu z nástroje Espresso, který slouží k minimalizaci logických funkcích. Práce, kromě návrhu a implementace rozhraní knihovny, zahrnuje také transformaci původních zdrojových kódů z

jazyka K&R C do jazyka C++. Knihovna je použitelná pod operačními systémy Microsoft Windows a Linux, a to jak ve statické, tak i dynamické verzi.

Využijeme tedy tuto knihovnu, zapouzdřenou do vlastního spouštěcího mechanizmu k detekci tautologie.

Naše vlastní implementace nástroje na detekci tautologie, která je předmětem této DP bude též založena na algoritmech popsaných v nástroji Espresso. Bude tedy velmi důležité, porovnat rychlosti jednotlivých implementací: naše vlastní vs. knihovna z BP [8] pana Miklánka, ve které se uvádí, cituji:

„Rozhodně se nepovedlo odhalit a opravit všechny chyby. Taková úprava by byla časově náročná, vyžádala by si značnou refaktorizaci zdrojových kódů a výsledek by dle mého názoru za to nestál. Osobně bych se přiklonil k vytvoření zcela nového projektu, který by si z původního Espressa odnesl jenom algoritmy a poučení, že by se rozhodně neměla podceňovat kontrola alokování paměti.“

Hodnocení:

Tato implementace dokáže řešit i větší instance než algoritmy popisované v podkapitolách 2.4 a 2.5. Avšak pro velká vstupní PLA např. při 100 vstupních proměnných o 20000 termech již výpočetní čas začíná značně narůstat.

2.7 Manipulace s logickými funkcemi pomocí grafů

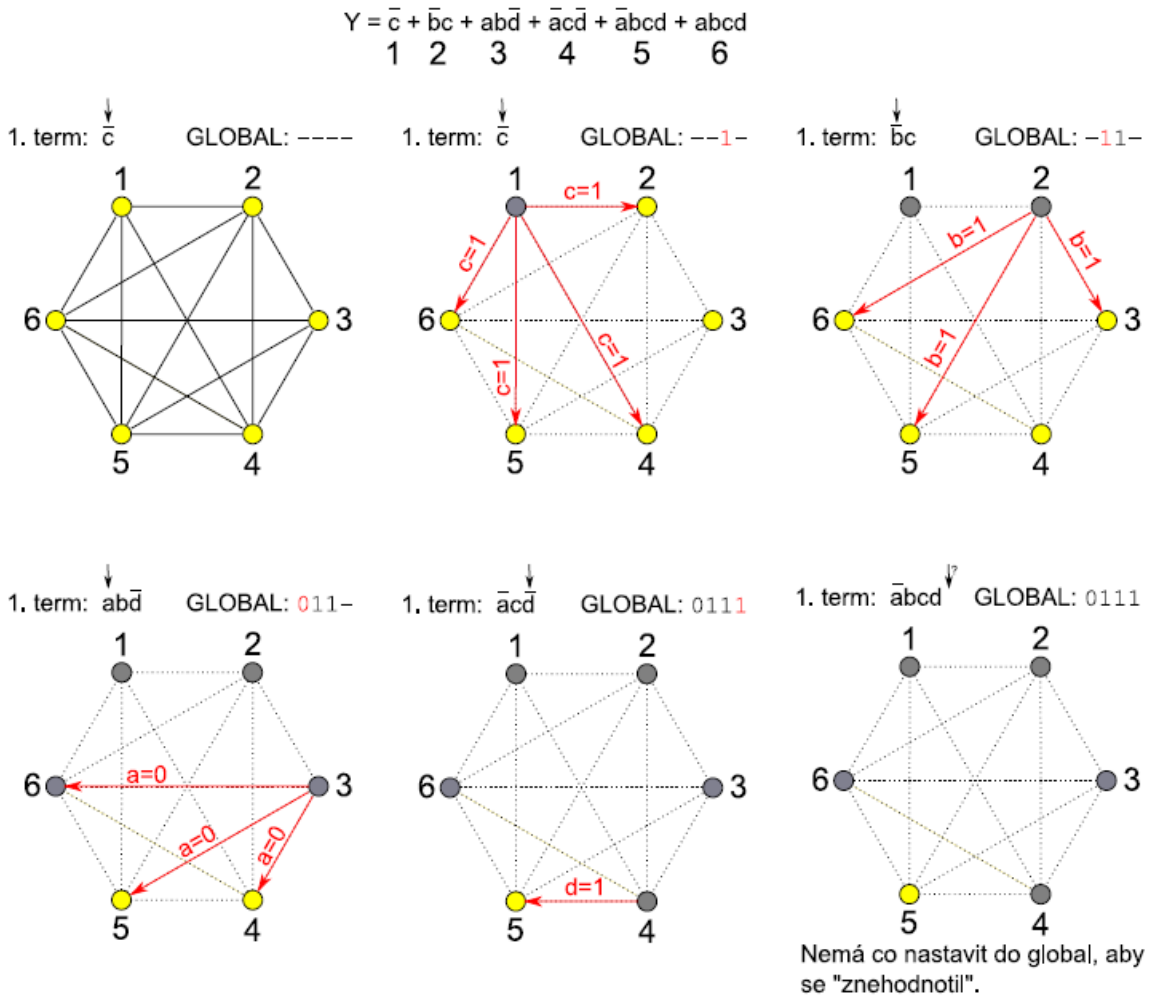
Dalšími nástroji pro detekci tautologie budou dva algoritmy popsané v diplomové práci [10] kolegy Petra Váli, který se zabýval manipulací s logickými funkcemi pomocí grafů. V dokumentaci této práce je podrobně popsáno, jak algoritmy pracují. V následujícím odstavci připomeneme pouze základní princip.

Popis algoritmu

Algoritmus pracuje tak, že se snaží dokázat, že funkce tautologií není. Tedy hledá takové ohodnocení proměnných, které vede k tomu, že výsledek žádného z termů není jedna. Vykonává to tak, že vhodně v iteracích ohodnocuje jednotlivé proměnné tak, aby tímto krokem vždy vyřadil nějaký ze zbývajících termů coby kandidátů na splnění aktuálního ohodnocení. V počátečním stavu jsou kandidáty všechny termy. Algoritmus se vždy

zaměří na některý z nich a nastaví některé z proměnných tohoto termu takovou polaritu, aby výsledkem tohoto termu již nemohla být pravda. Takto „znehodnocuje“ jednotlivé termy až do chvíle, kdy nezbývá žádný platný term. V takovém případě končí, neboť nenalezl ohodnocení, jehož výsledkem není pravda, nebo do chvíle, kdy zbývající kandidáty není schopen vyřadit ze seznamu kandidátů. Pak je tato větev již skutečně tautologická a v takovém případě vrací některá rozhodnutí o ohodnocení zpět (backtracking) a volí jiné způsoby, jakými proměnné ohodnocovat. Jestliže algoritmus prošel všechny větve stavového prostoru, ve kterých se mohlo nacházet řešení a nezdařilo se mu najít ohodnocení, které hledal, prohlásí funkci za tautologii.

Příklad první fáze postupu tohoto algoritmu, lze sledovat na obrázku 2.3 [10], kdy se dostaneme do stavu, kdy už není způsob, jakým by se dalo zabránit splnění termu č. 5.



Obrázek 2.3: Algoritmus řešící tautologii pomocí grafů

Druhým testovaným algoritmem pana Vály pak je algoritmus s předpřípravou PLA, která spočívá ve volání metody *simplification*, která efektivně zjednodušuje PLA na základě zákonu absorbce, absorbce negace a idempotence.

Hodnocení:

Oba algoritmy z diplomové práce pana Vály nalézají řešení relativně rychle pro instance s procentuálním zastoupením DC znaků větším jak 70%. U velkých instancí jejich výkonnost rapidně klesá. Druhý popisovaný algoritmus (s předpřípravou PLA) poskytuje řešení většinou pomaleji.

2.8 Závěr rešeršního zpracování

Po otestování pěti výšše popsaných algoritmů jsem nenalezl žádný, který by byl schopen řešit velké vstupní instance[5] a přitom by byl univerzálně použitelný co do různorodosti procentuálního zastoupení DC symbolů. Našim cílem tedy bude vytvořit nástroj, který bude schopen toto splnit, bude tedy nalézat řešení v přijatelných časech i pro větší zadání, než která byla řešitelná zde srovnávanými algoritmy.

Tabulka 2.1: Srovnání implementací na detekci tautologie

algoritmus	**chybovost**	**velikost instance**	**% DC**
komplement	neošetřené výjimky	pouze pro malá PLA	více %DC : rychlejší
BDD	1 špatný výsledek	pouze pro malá PLA	více %DC : pomalejší
knih. Espresso	bez chyb	i středně velká PLA	pomalé pro DC kolem 60%
graf	bez chyb	i středně velká PLA	vhodné pro DC přes 70%
graf2	bez chyb	i středně velká PLA	vhodné pro DC přes 70%

[5] vstupní PLA větší 5MB (1000 proměnných, 5000 termů)

3 Analýza a návrh řešení

Cílem této kapitoly bude teoretický rozbor problému detekce tautologie a přiblížení základních funkčních principů algoritmů na tuto detekci, které používá minimalizační nástroj Espresso. Nakonec bude představen návrh pro vlastní implementaci jejíž realizace bude popsána v kapitole 4.

3.1 Volba implementačního prostředí

Jedním z požadavků na navrhovaný systém bylo, aby byl implementován v programovacím jazyce C++. Proto se jako nejvíce vyhovující vývojové prostředí jevilo MS Visual Studio 2005, které je pro potřeby výuky volně ke stažení ze serveru [5]. Jako další alternativa pro volbu vývojového prostředí se nabízela freewareová aplikace Dev-C++, která však na první pohled nenabízí tak intuitivní uživatelské rozhraní a *debuggovací*[1] možnosti. Proto bylo pro vývoj zvoleno právě MS Visual Studio. Každé z těchto dvou zmiňovaných vývojových prostředí obsahuje vlastní překladač, bude proto velmi zajímavé vzájemné srovnání těchto různých kompilací stejného projektu. Tomuto porovnání výpočetních časů se budeme věnovat v podkapitole 5.8.

3.2 Problém tautologie

Zodpovězení otázky, zda-li je daná logická funkce tautologií, je jedna ze základních operací využívaná v algoritmech na minimalizaci logických funkcích. Např. nástroj Espresso, který bude představen v následující podkapitole 3.3, využívá metodu na detekci tautologie jako základ mnoha svých algoritmů[2]. Asi největší význam má detekce tautologie v metodě na určování přímých implikantů, kde je třeba ověřit, zda-li daný term je celý obsažen v původní funkci. A toto ověření se používá pro minimalizaci logických funkcí.

[1] Možnosti pro ladění programu.

[2] např. $IRREDUNDANT\_COVER, REDUCE, ESSENTIAL\_PRIMES, LAST\_GASP$

Tautologii, lze definovat jako vlastnost logické funkce. Logická funkce je tautologií, právě tehdy a jen tehdy pokud nabývá pravdivé hodnoty při jakémkoliv ohodnocení vstupních proměnných. Příkladem takovéto funkce může být:

$$Y = a + \overline{a},$$

hodnota Y bude vždy jedna nezávisle na ohodnocení proměnné a z množiny $\{0, 1\}$.

Obecně je detekce tautologie problém třídy $co - NP$ (viz podkapitola 2.2.3). Pro zodpovězení této otázky bychom museli „hrubou silou“ ověřit všech 2^n možných ohodnocení n vstupních proměnných a pro každé z nich říct, zda-li je zadaná funkce při tomto ohodnocení pravdivá či nikoli. Pokud by se nám podařilo najít jedno ohodnocení, při kterém by funkce vracela nepravdivou hodnotu, nemuseli bychom již dále pokračovat a mohli bychom o dané funkci prohlásit, že není tautologií. Máme-li funkci zadanou ve formě DNF (viz podkapitola 2.2.1), a to bude náš případ, neboť vstupní formát PLA (viz podkapitola 2.2.2) obsahuje na každém řádku jeden součinový term, můžeme využít následující vlastnost, že pokud najdeme jeden term, který je pro dané ohodnocení splněný, je splněna celá funkce. Pro detekci tautologie lze využít mnoho dalších pravidel vyplývajících z vlastností DNF a využít je pro zefektivnění práce výpočetního algoritmu. Návrhem takovéhoto postupu se bude zabývat podkapitola 3.4.

3.3 Úvod do Espressa

Tato podkapitola se bude snažit přiblížit funkci a význam nástroje Espresso resp. Espresso-II [2]. Popis vychází z práce kolegy Miklánka [8], který se zabýval vytvořením knihovny z tohoto nástroje.

Program Espresso je heuristický nástroj pro dvouúrovňovou minimalizaci logických funkcí. Jeho autorem je Richard Rudell působící na University of California Berkeley v Kalifornii. Tento nástroj je široce používán - výjimkou není ani Katedra počítačů Fakulty elektrotechnické ČVUT v Praze.

Hlavní přínos Espressa spočívá v použití nových[3] algoritmů, které zrychlují a obecně optimalizují heuristickou minimalizaci. Další výhodou je, že formát vstupních a výstupních

[3] upravovaná verze programu byla z roku 1988-1989

souborů je kompatibilní se standardním formátem pro fyzický popis PLA[4].

Espresso má nicméně i své nevýhody. Mezi největší patří použití příkazové řádky, která je podle současných standardů považována za uživatelsky nepřátelskou a dále chybí možnost použít Espresso jako modul, který by byl využíván nějakým komplexnějším programem. Pro jeho pokrokové metody se Espresso vyplatí modernizovat, rozšířit tak jeho pole působnosti a prodloužit tím dobu používání.

Této úpravě nestojí v cestě ani licence, podle které se Espresso může volně šířit a upravovat pod podmínkou zachování copyrightu[5].

3.4 Rozbor algoritmů z Espressa

V dokumentaci k minimalizačnímu nástroji Espresso-II [2] je popsána kostra algoritmu na detekci tautologie, která bude základem naší vlastní implementace (viz podkapitola 2.1). V následujících odstavcích se budeme tomuto algoritmu věnovat.

3.4.1 Základní rekurzivní schéma

Celá detekce je založena na rekurzivním opakování základního schématu, kde se využivají obecné vlastnosti DNF a logických funkcí (viz podkapitola 2.2.1). Tuto rekurzivní kostru můžeme pro zjednodušení popsat následujícím pseudokódem:

[4]PLA - Programmable logic array

[5]Copyright ©1988, 1989, Regents of the University of California. All rights reserved.

```
boolean TAUTOLOGY(cover F) {
  T := SPECIAL_CASES()                 // testování speciálních případů

  if(T == -1) UNATE_REDUCTION ()       // volání unátní redukce
  if(T == -1) COMPONENT_REDUCTION ()   // detekce komponent

  if(T != -1) return T

  j := BINATE_SELECT ()                // výběr nejvíce binátní proměnné
  if(0 == TAUTOLOGY(cofactor(j)) {     // musíme hledat dále v rekurzi
    return 0                           // není-li splněn, nejde o tautologii
  }
  if(0 == TAUTOLOGY(neg_cofactor(j)){ // totéž pro neg. kofaktor
    return 0
  }
  return 1                             // pokrytí této úrovně je tautologie
}
```

```
  //         SPECIAL_CASES               |   VÝSLEDEK   |  NÁVRAT
  // ---------------------------------------------------------------
  // 1) řádek samých DC                 |  tautologie  |    1
  // 2) sloupec samých 0 nebo 1         |  není taut.  |    0
  // 3) nedostatečný počet uzlů         |  není taut.  |    0
  // 4) méně jak 7 vstupů -> hrubá síla |     ---      |    ?
  // 5) nic z předchozích               |   rekurze    |   -1
```

Vstupní data přicházejí ve formě PLA[6] struktury. Pro jednoduchost si lze představit, že algoritmus pracuje s dvourozměrným polem, kde řádky představují jednotlivé termy a toto pole má tolik sloupců, kolik je vstupních proměnných. Používáme tři symboly: ”0”, ”1” a ”2”, kde ”1” značí danou proměnnou v daném termu, ”0” její negaci a ”2” zas-

[6]Viz kapitola 2.2.2.

tupuje DC. Příklad reprezentace zadané logické funkce a jejich kofaktorů[7] je symbolicky znázorněn na obrázku 3.1.

Z výše uvedeného pseudokódu, který popisuje základní rekurzivní schéma našeho algoritmu je patrné, že nejprve dochází k testování daného pokrytí[8] na sérii speciálních případů viz výše uvedený pseudokód rekurzivního schématu. Rozlišujeme čtyři základní případy:

1. **řádek samých DC:** obsahuje-li dané pokrytí řádek, kde se vyskytují pouze DC symboly resp. "2" viz obr. 3.1. Je toto pokrytí tautologické, neboť jak vyplývá z definice DNF[9] stačí najít jeden term (tento řádek), který bude splněn vždy (a to tento term bude, jsou zde pouze DC), aby byla logická funkce tautologií,

2. **sloupec samých "0" nebo "1":** obsahuje-li dané pokrytí sloupec, ve kterém se vyskytují pouze symboly "1" resp. "0", pak tato funkce není tautologií, neboť se v ní daná proměnná (příslušna tomuto sloupci) vyskytuje buď v přímé resp. negované formě. Abychom našli ohodnocení, pro které funkce vrátí nepravdivou hodnotu, stačí tuto proměnnou ohodnotit jak „nepravdu“ resp. „pravdu“,

3. **nedostatečný počet uzlů:** jako spodní hranici pro minimální počet mintermů[10] můžeme stanovit hodnotu, kterou získáme jako součet, do kterého každý term přispěje 2^k, kde k je rovno počtu DC v dané řádce tohoto termu. Je-li tato suma menší než 2^n, kde n představuje počet proměnných pak daná funkce nemůže být tautologií,

4. **méně jak 7 vstupů:** má-li testované pokrytí méně jak 7 vstupů, použije se na detekci tautologie tzv. „hrubá síla“, která představuje rekurzivní funkci, která projde celou pravdivostní tabulku a zodpoví otázku, zda-li je testovaná funkce tautologie. Volání hrubé síly, je zde přípustné, neboť pravdivostní tabulka má pro 6 vstupů $2^6 = 64$ řádků, které se dají otestovat ve velmi krátkém čase. Hranici "7" lze chápat jako parametr a v kapitole 5.6, kde se budeme zabývat testováním, zkusíme měnit tuto hodnotu a sledovat vliv na výsledný čas potřebný k nalezení řešení,

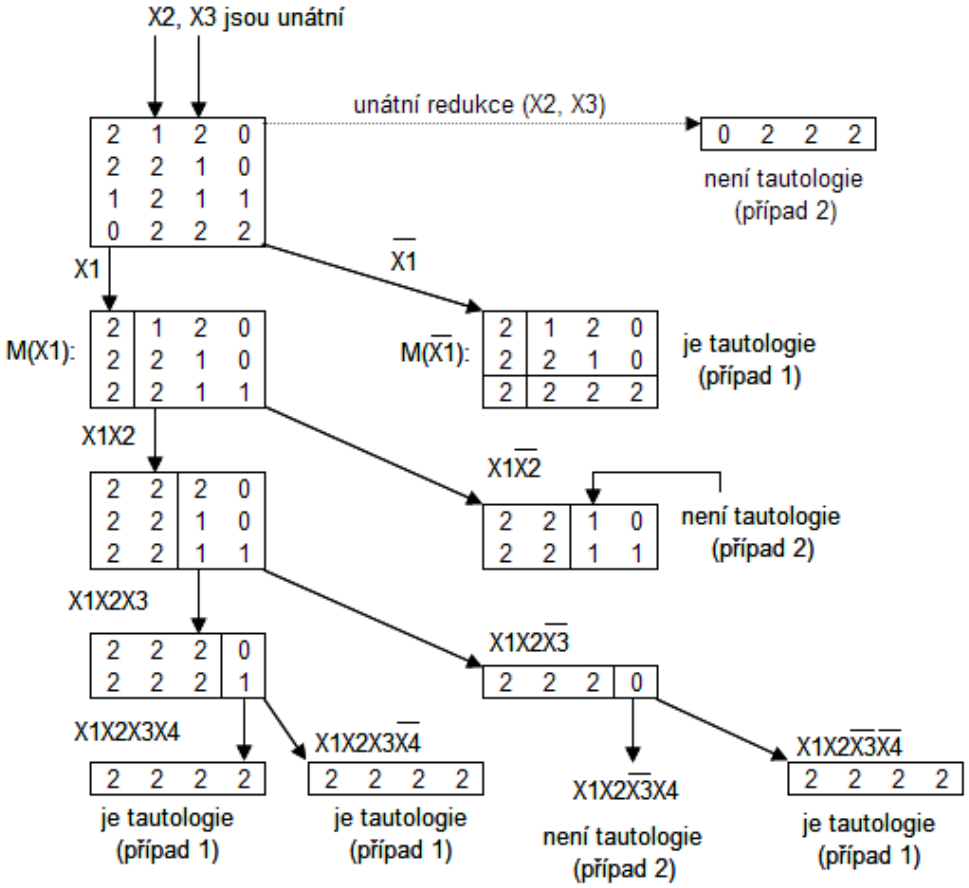
[7] Viz kapitola 3.4.2.

[8] Z angl. slova *cover*, viz reprezentace logické funkce na obr. 3.1.

[9] Viz podkapitola 2.2.1

[10] Viz podkapitola 2.2.1.

5. **nic z předešlých případů:** v daném pokrytí se nevyskytuje žádný ze speciálních případů, je třeba pokračovat dále podle schématu.



Obrázek 3.1: Ukázka rekurzivní detekce tautologie

Dostali jsme se nyní do stavu, kdy jsme otestovali naši funkci na sérii speciálních případů. Dále předpokládejme, že prozatím nebudeme uvažovat funkce na zefektivnění činnosti algoritmu nazvané $UNATE\_REDUCTION$ a $COMPONENT\_REDUCTION$, kterým se budeme věnovat v podkapitole 3.4.3 resp. 3.4.4. Jako další krok následuje výběr tzv. nejvíce binátní proměnné a jejímu výběru se věnuje podkapitola 3.4.2 a obrázek 3.2. Heuristický výběr nejvíce binátní proměnné je velmi důležitý pro konstrukci kofaktoru[11] z dané funkce. Pro efektivní práci algoritmu je vhodné, abychom dokázali detekovat sloupce unátní[12], proto je výhodné se „zbavovat“ sloupců binátních a v našem případě

[11] Viz podkapitola 3.4.2.

[12] Viz podkapitola 3.4.3.

toho nejvíce binátního.

Z vybrané nejvíce binátní proměnné se vytvoří dva kofaktory: přímý a negovaný a nad každým z těchto kofaktorů se zavolá opět ta samá rekurzivní detekce tautologie, popsaná výše ovedeným pseudokódem. Původní funkci pak můžeme prohlásit za tautologii právě tehdy a jen tehdy, pokud je otázka tautologie zodpovězena kladně pro obě větve vzniklých kofaktorů.

Tento rekurzivní postup lze názorně sledovat na obrázku 3.1, kde je nejprve za nejvíce binátní proměnnou zvoleno x_1. Negovaný kofaktor vpravo můžeme ihned prohlásit za tautologii, neboť obsahuje řádek samých DC (případ 1). Levý kofaktor se dále rozdělí na dva podle proměnné x_2 atd. Na obrázku 3.1 je pro každý kofaktor vyznačeno, který speciální případ u něho nastal. Je patrné, že původní funkce není tautologie, protože tautologií nebyly všechny z ní rekurzivně vzniklé kofaktory. Stejnou odpověď jsme mohli získat, ale v mnohem kratším čase, rozpoznáním toho, že proměnné x_2 a x_3 jsou unátní. Tato skutečnost je na obrázku 3.1 znázorněna přerušovanou čarou v horní části. Tomuto zefektivnění se bude věnovat podkapitola 3.4.3.

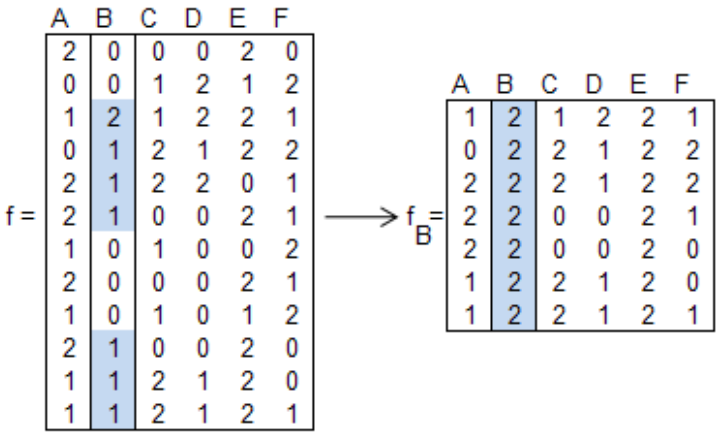
3.4.2 Kofaktor, nejvíce binátní proměnná

Řekli jsme, že náš nástroj na detekci tautologie obdrží na vstupu logickou funkci popsanou pomocí PLA[13] struktury. Tuto logickou funkci lze pak reprezentovat dvourozměrným polem, kde řádky představují jednotlivé termy a toto pole má tolik sloupců kolik je vstupních proměnných. Používáme pak tři symboly: "0", "1" a "2", kde "1" značí danou proměnnou v daném termu, "0" její negaci a "2" zastupuje DC. Kofaktor dané funkce, který je formalně definován v podkapitole 2.2.4, pak hledáme vzhledem k zvolené proměnné a to buď k její přímé nebo negované formě. Abychom získali kofaktor původní funkce z přímé formy její jedné proměnné, přiřadíme do kofaktoru ty řádky, které obsahují v sloupci příslušejícímu dané proměnné "1" nebo "2" a jako hodnotu v kopírovaném řádku příslušející sloupci dané proměnné dosadíme "2". Pro negovaný kofaktor pak analogicky ty řádky, kde se nacházejí v daném sloupci "0" nebo "2". Vše lze ilustrovat obrázkem 3.2, kde je znázorněna konstrukce přímého kofaktoru proměnné "B".

[13] Viz kapitola 2.2.2

Heuristická otázka volby proměnné, ze které budeme vytvářet kofaktor, je vyřešena pomocí detekce tzv. nejvíce binátní proměnné. Tato detekce bude velmi často využívána, viz podkapitola 3.4.1. Nejvíce binátní proměnnou lze definovat jako tu, která má největší počet zpárovatelných dvojic "0" a "1" v daném sloupci ze všech. Tuto detekci lze popsat následujícím pseudokódem:

```
begin
  maxDvojic01 = 0;
  nejviceBinatniPromenna = 0;
  for ( i = 0; i < pocetPromennych; i++){
    if(min(pocet0veSloupci[i], pocet1veSloupci[i]) > maxDvojic01){
      maxDvojic01 = min(pocet0veSloupci[i], pocet1veSloupci[i]);
      nejviceBinatniPromenna = i;
    }
  }
  return nejviceBinatniPromenna;
end
```



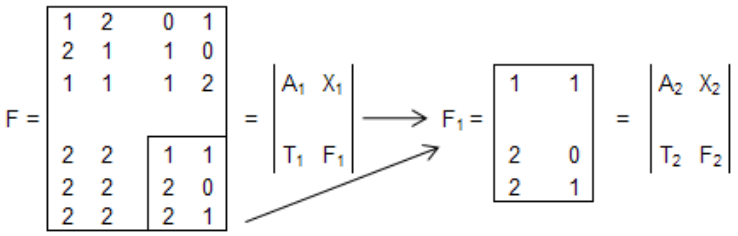
Obrázek 3.2: Ukázka vzniku kofaktoru z nejvíce binátní proměnné

3.4.3 Unátní redukce

Unátní redukce je jedním ze způsobů, jak zvýšit efektivitu algoritmu na detekci tautologie. Na obrázku 3.1 jsme si ukázali, že rozpoznání unátních sloupců, může vést k velmi rychlému nalezení výsledku. Nyní ukážeme, v čem unátní redukce spočívá.

Základem unátní redukce je „odhalení“ unatních sloupců. Sloupec je unátní pokud odsahuje pouze symboly ”0” a ”2” nebo ”1” a ”2” (viz obrázek 3.1) nebo první dva sloupce na obrázku 3.3. Pokud zadaná logická funkce neobsahuje žádný unátní sloupec, žádná akce se v unátní redukci nevykonává. V případě, že jsou detekovány unátní sloupce vychází se z teorému, který lze popsat pomocí obrázku 3.3.

Logickou funkci F lze rozdělit do čtyř částí $\{A_1, X_1, T_1, F_1\}$ přičemž blok A_1 není tautologie a blok T_1 obsahuje pouze samé symboly ”2”. Lze tvrdit, že funkce F je tautologie právě tehdy a jen tehdy, pokud blok F_1 je tautologie. Tento postup je základem unátní redukce, která pak probíhá následujícím způsobem. Z unátních sloupců překopírujeme ty termy, které na pozicích unátních sloupců obsahují pouze symboly ”2”. Neexistuje-li, žádný takovýto řádek, daná funkce není tautologie.



Obrázek 3.3: Schéma unátní redukce

3.4.4 Redukce komponent

Redukce komponent je další způsob jak zvýšit efektivitu algoritmu na detekci tautologie a je založen na obdobném principu jako unátní redukce popisovaná v předchozí podkapitole 3.4.3. Nyní ukážeme v čem redukce komponent spočívá.

Základem redukce komponent, je detekce bloků v pokrytí zadané funkce, které budeme nazývat komponentami. Předpokládejme, že se nám podaří najít řádkovou a sloupcovou permutaci v původní matici tak, že se nám tuto matici podaří upravit na tvar, který vystihuje obrázek 3.4. Danou funkci pak můžeme rozložit na bloky - komponenty $F^1 ... F^k$, zbytek matice je vyplněn pouze symboly "2". Poté testujeme na tautologii pouze jednotlivé komponenty, protože původní logická funkce je tautologií, pokud alespoň jedna z komponent je tautologie. Tato skutečnost může mít za následek radikální změnšení prostoru, který musíme procházet při detekci tautologie.

$$F' = \begin{bmatrix} F^1 & 2 & . & . & 2 \\ 2 & F^2 & . & . & . \\ . & . & . & . & . \\ . & . & . & . & 2 \\ 2 & 2 & . & 2 & F^k \end{bmatrix}$$

Obrázek 3.4: Schéma redukce komponent

3.5 Návrh implementace

Shrneme-li nyní požadavky[14] na navrhovaný nástroj na detekci tautologie, závěr z rešeršního zpracování[15] a analýzu algoritmů z Espressa[16], dostaneme následující návrh implementace, jejíž realizací se bude zabývat kapitola 4.

Základem celého nástroje na detekci tautologie bude jedna třída, která bude nést informace o datech reprezentujících logickou funkci, bude definovat metody potřebné pro detekci tautologie a bude uchovávat pomocné datové struktury potřebné pro zefektivnění

[14] Viz podkapitola 2.1.
[15] Viz podkapitola 2.8.
[16] Viz podkapitola 3.4.

práce algoritmu. Pro komunikaci s jádrem programu BOOM II se bude využívat interface, definující dvě metody:

- bool isPlaTautology(PLA* pla);
- bool isSoPlaTautology(SoPLA* pla);

První z nich se bude používat pro práci s PLA strukturou, která je reprezentována vnitřní třídou nástroje BOOM II. Druhá metoda je určena pro práci se strukturou SoPLA, která je speciálním případem jedno-výstupového[17]. Schéma těchto tříd je znázorněno na zjednodušeném UML diagramu popisujícím třídy jádra programu BOOM II na obrázku 4.3.

Úkolem realizace bude:

- najít nejvýhodnější způsob pro vnitřní reprezentaci dané logické funkce s ohledem na paměťovou náročnost a s možností časově efektivního náhodného přístupu k datům na základě adresace konkrétní proměnné v konkrétním termu,
- časově efektivní detekce speciálních případů,
- vytváření kofaktorů z nejvíce binátní proměnné,
- časově efektivní detekce unátních sloupců a následná tvorba unátního derivátu podle schématu na obrázku 3.3,
- vytvoření zcela vlastního efektivního algoritmu na detekci komponent a jejich následná tvorba podle schématu na obrázku 3.4.

[17]SoPLA = Single Output PLA

4 Realizace

Cílem této kapitoly bude popis průběhu implementace nástroje na detekci tautologie, který je cílem této diplomové práce. Realizace je založena na návrhu implementace vycházejícího z analýzy uvedené v předchozí kapitole 3.

4.1 Průběh implementace

Vývoj celého projektu probíhal ve třech základních fázích. V první z nich došlo k vytvoření základní funkční kostry algoritmu, který pracoval podle schématu vyjádřeného pseudokódem v podkapitole 3.4.1. Tento nástroj byl schopen detekovat tautologii ze zadaných PLA struktur na vstupu, avšak výsledný čas detekce nebyl optimální. Ve většině případů byly časy potřebné pro nalezení řešení horší v porovnání s knihovnou z nástroje Espresso, která je popsaná v podkapitole 2.6, a která je hlavním srovnávacím prvkem pro efektivitu našeho projektu.

Jako další krok vývoje následovala druhá fáze, která si kladla za cíl zvýšit výkonnost algoritmu, tak aby řešení bylo nalezeno v co nejkratším čase. Za tímto účelem byly do hlavní třídy přidány pomocné dynamické datové struktury, umožňujících efektivnější indexování prvků a uchování informací o jednotlivých termech a proměnných. Při vzniku odvozených pokrytí (kofaktor, unátní derivát, komponenta) - viz obrázek 3.1 se inicializují informace o jednotlivých řádcích a sloupcích takto vzniklých derivátů a tyto informace se pak využívají např. pro vyhodnocování speciálních případů - viz podkapitola 3.4.1. Dále došlo k optimalizaci cyklických procházení datových polí především pomocí přidání ukončovacích podmínek. Tyto úpravy vedly k požadovanému zvýšení výkonu a náš nástroj byl již téměř ve všech případech rychlejší, než knihovna z nástroje na minimalizaci logických funkcí, Espressa, což byl náš původní záměr. Následné testování však ukázalo, že existuje možnost, jak náš projekt zrychlit mnohem více a to zhruba pětinásobně. Tento efekt by měl být umožněn přechodem na výhodnější vnitřní reprezentaci zadané logické funkce a to díky rychlejšímu náhodnému přístupu ke zpracovávaným datům.

O začátku vývoje se pro vnitřní reprezentaci dat používala třída PLA z jádra programu BOOM II. Při testech na časovou optimálnost náhodného přístupu k datům,

se ukázalo, že mnohem výhodnějším způsobem, než je zde používaný STL[1] systém generických kolekcí, je přímý přístup k paměti. Poslední, třetí, fáze vývoje projektu tedy měla za cíl implementovat nový systém vnitřní reprezentace dat pomocí dynamických dvourozměrných polí. Výsledným efektem pak bylo, že nová verze nástroje na detekci tautologie přinesla několikanásobné zrychlení, které je kvantifikováno v podkapitole 5.5.

4.2 Popis realizace se zaměřením na nestandardní části řešení

Následující podkapitoly se budou zabývat konkrétním způsobem realizace těch částí nástroje na detekci tautologie, které jsme vymezili v části 3.5.

4.2.1 Vnitřní reprezentace logické funkce

Na samotném začátku implementace tohoto projektu se jako nejvýhodnější jevilo použít pro vnitřní reprezentaci dat představujících logickou funkci právě objektovou strukturu definovanou v jádru programu BOOM II - třídu PLA. Tato struktura využívá třídy představující jednotivé termy (*class Term*) a v třídě PLA jsou pak data uložena jako typované kolekce jednotlivých termů. Pro efektivní přístup k jednotlivým řádkům (termům) je zde zavedena i podpora jejich indexování a lze i adresovat jednotlivé proměnné pro daný term. Tato struktura tedy umožňuje sekvenční procházení pokrytí dané funkce, což je důležitým předpokladem pro samotnou realizaci jednotlivých částí[2] algoritmu na detekci tautologie.

Použitím vnitřní třídy PLA programu BOOM II se podařilo implementovat nástroj na detekci tautologie, který poskytoval řešení ve velmi krátkých časem ve srovnání s jinými implementacemi[3]. Dokonce se dařilo, řešit i relativně velké instance vstupních logických funkcí, které nebyly ostatními nástroji detekovatelné (hledání řešení trvalo např. více než 200s). Během testování se však zjistilo, že by mohla existovat cesta k dalšímu zrychlení chodu algoritmu a to pomocí přechodu na alternativní reprezentaci pokrytí dané logické funkce. Toto zjištění bylo umožněno na základě srovnání časů sekvenčního průchodu

[1] Standard Template Library - standardní knihovna šablon jazyka C++

[2] Viz podkapitola 3.4.1.

[3] Viz podkapitola 2.3.

dvou stejně velkých dat logických funkcí. V prvním případě se jednalo o již použitou třídu PLA (nabízí se dvě možnosti průchodu termů - pomocí iterátoru, nebo pomocí indexu), v druhém případě pak šlo o dynamické dvourozměrné pole. Tyto průchody a adresace jednotlivých prvků pro tyto tři případy jsou vyjádřeny následujícími kódy:

```
// průchod PLA pomocí iterátoru
  PLA* pla;
  list<Term>::iterator ti;
  TermList *data = pla->GetData();
  for ( ti = data->begin(); ti != data->end(); ti++ ){
    for(int j=0; j< numberOfVariables; j++){
      (*ti).GetIM(j);
    }
  }

// průchod PLA pomocí indexů
  for(int i=0; i< numberOfTerms; i++){
    for(int j=0; j< numberOfVariables; j++){
      pla->GetTerm(i)->GetIM(j);
    }
  }

// průchod dynamického 2D pole
  char **cover;
  for(int i=0; i < numberOfTerms; i++){
    for(int j=0; j < numberOfVariables; j++){
      cover[i][j];
    }
  }
```

Měření bylo prováděno na procesoru AMD Athlon XP-M 2800+ 2,12GHz, 512MB RAM s operačním systémem Windows XP SP3. Výsledkem tohoto testu bylo, že poslední způsob

trval u reprezentace logické funkce o velikosti 5MB (např. funkce o 1000 proměnných a 5000 termech) asi 0,01s, při procházení termů pomocí iterátoru asi 0,09s a při přístupu k termům pomocí indexů asi 0,26s. Použít průchod termy pomocí iterátoru je možné pouze v případě, že musíme projít všchny termy a tato situace nenastává vždy. I tak je zřejmé, že použití dvourozměrného dynamického pole pro reprezentaci dané logické funkce je téměř o jeden řád rychlejší než průchod pomocí iterátoru a zhruba 30x rychlejší než přistupování k termům pomocí indexů.

Na základě tohoto zjištění se vytvořila nová verze nástroje na detekci tautologie využívající pro vnitřní reprezentaci logické funkce dynamické pole. Zrychlení, které tato změna způsobila bylo natolik významné, že se rozhodlo o přepracování jádra programu BOOM II, kde se místo nyní používaných generických kolekcí z knihovny STL začne využívat také přímý přístup do paměti. Zrychlení nástroje pomocí této úpravy je kvantifikováno v podkapitole 5.5.

Po aplikaci těchto změn do jádra programu BOOM II bude opět moci být využívána původní verze našeho nástroje na detekci tautologie, protože struktura PLA, se kterou pracuje, již bude využívat onu efektivnější reprezentaci dat.

4.2.2 Detekce speciálních případů

V první funkční verzi našeho nástroje na detetci tautologie se pro detekci speciálních případů[4] používalo sekvenční procházení pokrytí analyzované funkce. I když tyto detekční cykly obsahovaly podmínky na předčasné ukončování při nalezení nebo vyvrácení daného kritéria, byl tento způsob detekce speciálních případů velmi neefektivní. Došlo tedy k zavedení dynamických struktur, které nesou informace a jednotlivých řádcích a sloupcích testovaného pokrytí. Pro vyhodnocení všech speciálních případů je zapotřebí informace o počtu "0" a "1" v každém sloupci a počtu "2" v každém řádku. K rozhodnutí, zda-li nastává některý speciální případ, pak stačí v nejhorším případě jeden průchod těmito daty a ne již průchod celým dvourozměrným polem.

Při detekci třetího speciálního případu dochází k zajímavému problému: dosáhne-li počet proměnných více jak 3000, tak hodnota 2^{3000}, kterou je potřeba porovnávat se součtem

[4] Viz podkapitola 3.4.1.

2^k, kde k je počet DC v každém termu, začne jít mimo rozsah i toho největšího datového typu C++ (*long double* - rozsah do 10^{999}). Pro zachování univerzálnosti našeho nástroje, bylo nutné vymyslet způsob jak porovnávat takto velká čísla a tím bylo použití bitového pole o délce počtu proměnných, kde každý bit přísluší mocnice čísla 2 dané pořadím tohoto bitu. Sčítání přes všechny řádky hodnot 2^k, kde k je počet DC v každém termu, poté probíhá tak, že se program pokusí uložit hodnotu "1" na pozici k, pokud je obsazená, vynuluje ji a propaguje přenos do řádu $k+1$ atd. dokud nenalezne bit z hodnotou "0". Na konci stačí porovnat maximální dosažený řád v tomto poli s hodnotou počtu proměnných.

Pro vyhodnocení čtvrtého případu, kdy nastává situace, že dané pokrytí má již méně jak sedm proměnných, bylo třeba implementovat algoritmus na detekci tautologie tzv. „hrubou silou". Tato realizace je založena na rekurzivním algoritmu generujícím postupně všechny možné konfigurace ohodnocení jednotlivých proměnných a jejich následným testováním na výslednou hodnotu dané funkce. Rekurze se zastaví pokud nalezne ohodnocení, pro které ja daná funkce *nepravda*, nenajde-li takové, označí pokrytí za tautologii.

4.2.3 Tvorba kofaktorů z nejvíce binátní proměnné

Pro vytvoření kofaktoru je nezbytné určit proměnnou podle, které budeme její dva kofaktory (v přímé a negované formě) vytvářet. Proč je výhodné, aby touto proměnnou byla právě ta, kterou nazýváme „nejvíce binátní" popisuje podkapitola 3.4.1. A detekcí nejvíce binátní proměnné a tvorbou kofaktoru se zabývá podkapitola 3.4.2.

Jak je vidět z pseudokódu na detekci nejvíce binátní proměnné v podkapitole 3.4.2, využívají se zde také stejné struktury popisující dané sloupce jako v případě detekce speciálních případů. Díky těmto informacím o počtu "0" a "1" v jednotlivých sloupcích je určení nejvíce binátní proměnné možné jedním průchodem pole o délce rovné počtu proměnných. Bez této pomocné struktury bychom musel projít celým pokrytím.

Máme-li nyní určenou nejvíce binátní proměnnou, je třeba vytvořit dva kofaktory - viz schéma na obrázku 3.2. Nejprve je třeba určit nové rozměry pokrytí představující vzniklý kofaktor. To provedeme spočítáním termů, které budou obsaženy v novém kofaktoru a uložením indexů těchto řádků. Počet sloupců (počet proměnných) poté bude vždy o jedna měnší, nemá cenu již dále uvažovat s původně nejvíce binátním sloupcem, který by měl

v kofaktoru podobu samých "2".

Po určení rozměrů pokrytí kofaktoru alokujeme potřebnou pamět a inicializujeme pomocné dynamické datové struktury podle rozměrů daného kofaktoru. Poté naplníme kofaktor daty, viz obrázek 3.2, za pomoci uložených indexů kopírovaných termů. V tomto případě nelze použít rychlejší přístup k termům pomocí iterátoru[5], protože neprocházíme postupně všechny termy a jsme tedy odkázáni pouze na přístup k termům pomocí indexů. Při tomto plnění dat kofaktoru současně detekujeme hodnoty jednotlivých symbolů a aktualizujeme informace v pomocných strukturách popisujích jednotlivé řádky a sloupce.

Nad takto vzniklými dvěma kofaktory se poté rekurzivně spustí stejná analýza jako na původní pokrytí, ze kterého tyto kofaktory vznikly - viz pseudokód základního schématu v podkapitole 3.4.1. Heuristickou otázkou by nyní mohlo být: nad kterým kofaktorem (větví rekurze), spustit detekci dříve? Nabízí se řešení vyplývající z třetího specialního případu, který může prohlásit dané pokrytí za netautologické pokud obsahuje nedostatečný počet DC symbolů. Proto budeme volat nejprve vyhodnocení kofaktoru s nižším počtem DC, abychom urychlili případnou detekci netautologií, protože v případě, že se o tautologii jedná, musí se projít obě rekurzivní větve.

4.2.4 Realizace unátní redukce

Pro efektivní využití přínosu, který nám nabízí unátní redukce, jejíž princip je popsán v podkapitole 3.4.3, je třeba v co možno nejkratším čase rozpoznat přítomnost unátních sloupců. Při této detekci se opět uplatní struktura nesoucí informace o jednotlivých sloupcích. Unátní sloupce rozpoznáme jedním průchodem tímto polem o délce rovné počtu proměnných, protože unátní sloupec je takový, který neobsahuje žádné symboly "0" resp. "1" a přitom obsahuje alespoň jednu hodnotu "1" resp. "0".

V případě, že pokrytí neobsahuje žádné unátní sloupce, žádná další akce se v tomto bodě nekoná a toto zjištění nás nestálo příliš mnoho času. V druhém případě, když jsme detekovali unátní sloupce, vede toto zjištění k velmi významnému zmenšení prostoru, který musíme dále procházet a testovat na tautologii. Z původního pokrytí vytvoříme unátní derivát podle schématu na obrázku 3.3. Nejprve zjistíme rozměry nového pokrytí,

[5] Viz podkapitola 4.2.1.

které vznikne po unátní redukci a přitom si ukládáme indexy těch řádků (obsahujících pouze "2" na pozicích unátních proměnných) a sloupců (unátní sloupce), které se stanou součástí unátního derivátu.

Po určení rozměru nového pokrytí a vymezení rozsahu dat, který se budou dále používat, alokujeme potřebné datové struktury a naplníme je daty obdobně jako při vzniku kofaktoru - viz podkapitola 4.2.3.

Nad takto vzniklým unátním derivátem se poté rekurzivně spustí stejná analýza jako na původní pokrytí, ze kterého vznikl - viz pseudokód základního schématu v podkapitole 3.4.1.

4.2.5 Detekce a tvorba komponent

Princip a využití detekce komponent je popsán v podkapitole 3.4.4. Naším úkolem bude nyní vymyslet zcela vlastní algoritmus, který bude schopen tyto komponenty detekovat a vytvářet.

Detekci komponent založíme na faktu, že každý řádek může příslušet maximálně jedné komponentě. Vyjdeme-li na počátku od libovolného řádku a budeme-li prohledávat celou oblast pokrytí a detekovat rozsah komponenty příslušející tomuto řádku dostaneme na závěr tohoto postupu dvě možná řešení. Buď bude nalezená komponenta shodná s původním pokrytím a v tomto případě již nelze toto pokrytí dělit na menší komponenty a žádné urychlení běhu detekce tautologie nás nečeká a nebo v druhém případě nalezneme komponentu vymezenou mřížkou danou protnutím se konkrétních řádků a sloupců. V tomto druhém případě jsme z původního pokrytí vybrali některé řádky, které náleží menší komponentě, než je původní pokrytí a tento postup detekce komponent můžeme zopakovat, když začneme hledat od libovolného dalšího řádku, který ještě nebyl použit pro žádnou z komponent. Tento postup opakujeme do té doby, než všechny řádky z původního pokrytí budou příslušet nalezeným komponentám. Je tedy jedno, od kterého řádku jsme na počátku začali deketci odvíjet.

Která data však spadají do jedné komponenty? Je třeba od počátečního řádku procházet všechny sloupce, které v tomto řádku neobsahují symbol "2" a v těchto sloupcích je třeba uložit na zásobník unikátní indexy těch řádků, které obsahují symboly "0" nebo

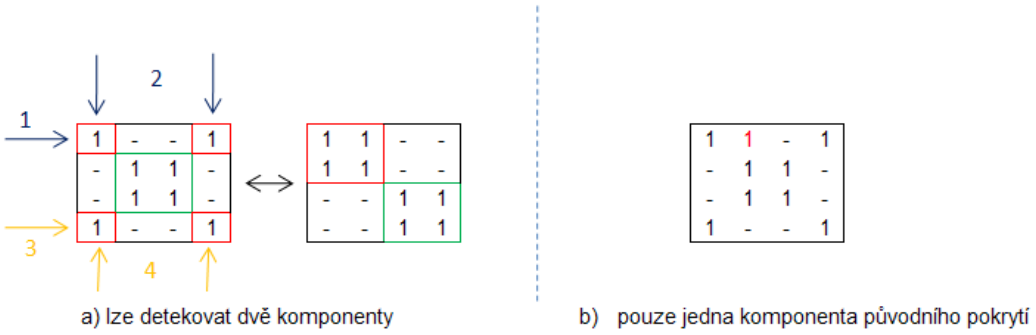
”1” tzv. incidují v ortogonálních směrech s výchozím řádkem. Ukládáme indexy takto „proscanovaných“ sloupců. Poté tento postup opakujeme pro další řádek podle indexu z vrcholu zásobníku a „scanujeme“ ty sloupce, které dosud nebyly kontrolovány.

Tento postup lze sledovat na obrázku 4.1, kde v prvním případě máme dva ekvivalentní způsoby zápisu jednoho pokrytí logické funkce. Tyto dvě matice lze mezi sebou převést pomocí záměny příslušných řádků a sloupců, což jsme zde udělali pouze pro názornost, aby v druhém případě byly na první pohled vidět dvě komponenty, které lze získat z původního pokrytí. Pro funkci algoritmu na detekci komponent nemá zaměňování řádků a sloupců žádný vliv. V první fázi vyznačené na obrázku 4.1 modrou číslicí jedna dojde k určení startovacího řádku, ve kterém budeme „scanovat“ všechny sloupce, které neobsahují v tomto řádku DC symbol (znázorněn pomocí ”-”). Při procházení těchto sloupců (na obr. fáze 2) přidáváme na zásobník unikátní indexy těch řádků, které v tomto procházeném sloupci neobsahují DC symboly (obsahují ”0” nebo ”1”) a zároveň si uložíme informaci, které sloupce už jsme takto „proscanovali“. Ve fázi 3 opakujeme postup pro řádek z vrcholu zásobníku a procházíme dosud „neproscanované“ sloupce. Postup opakujeme do té doby, než je zásobník prázdný. V tuto chvíli jsme našli oblast komponenty obsahující startovací řádek a tuto komponentu máme vymezenou rozsahem indexů řádků a sloupců, které jí příslušejí. Obsahuje-li takto vymezená komponenta všechny řádky z původního pokrytí (případ *b* na obrázku 4.1 - symbol 1 způsobil propojení obou komponent z případu *a* do jedné), nelze toto pokrytí rozdělit na menší komponenty. V opačném případě, když detekovaná komponenta obsahuje pouze některé řádky, opakujeme postup na detekci jedné komponenty pro startovací řádek, kterým muže být kterýkoliv z řádků původního pokrytí, který doposavaď nebyl zařazen do žádné jiné komponenty a to do té doby, než všechny řádky původní matice jsou přiřazeny konkrétním komponentám.

Po úspěšném detekování dvou a více komponent je nutné vytvořit jejich objektové reprezentace v programu pro další zpracování algoritmem na detekci tautologie. Alokujeme potřebné datové struktury a naplníme je daty obdobně jako při vzniku kofaktoru - viz podkapitola 4.2.3. Účel a přínos detekce komponent je popsán v podkapitole 3.4.4 a vyplývá z něho, že podaří-li se nám detekovat dvě a více komponent, tak původní pokrytí je tautologie právě tehdy a jen tehdy, pokud alespoň jedna z komponent je tautologie. Metoda na detekci komponent:

```
vector<Tautology*>  Tautology::findComponents();
```

vrací po úspěšné dektekci dvou a více komponent jejich seznam v podobě STL vektoru nesoucí ukazatele na jednotlivé komponenty - instance třídy *Tautology*, která reprezentuje dané pokrytí logické funkce a obsahuje metody a pomocné datové struktury pro detekci tautologie viz následující podkapitola 4.3. Máme-li nyní k dospozici seznam komponent, stačí je postupně otestovat na tautologii stejným způsobem jako původní logickou funkci. V případě, že kterákoliv z komponent je tautologie, bylo tautologií i pokrytí, ze kterého tyto komponenty vznikly.



Obrázek 4.1: Detekce komponent

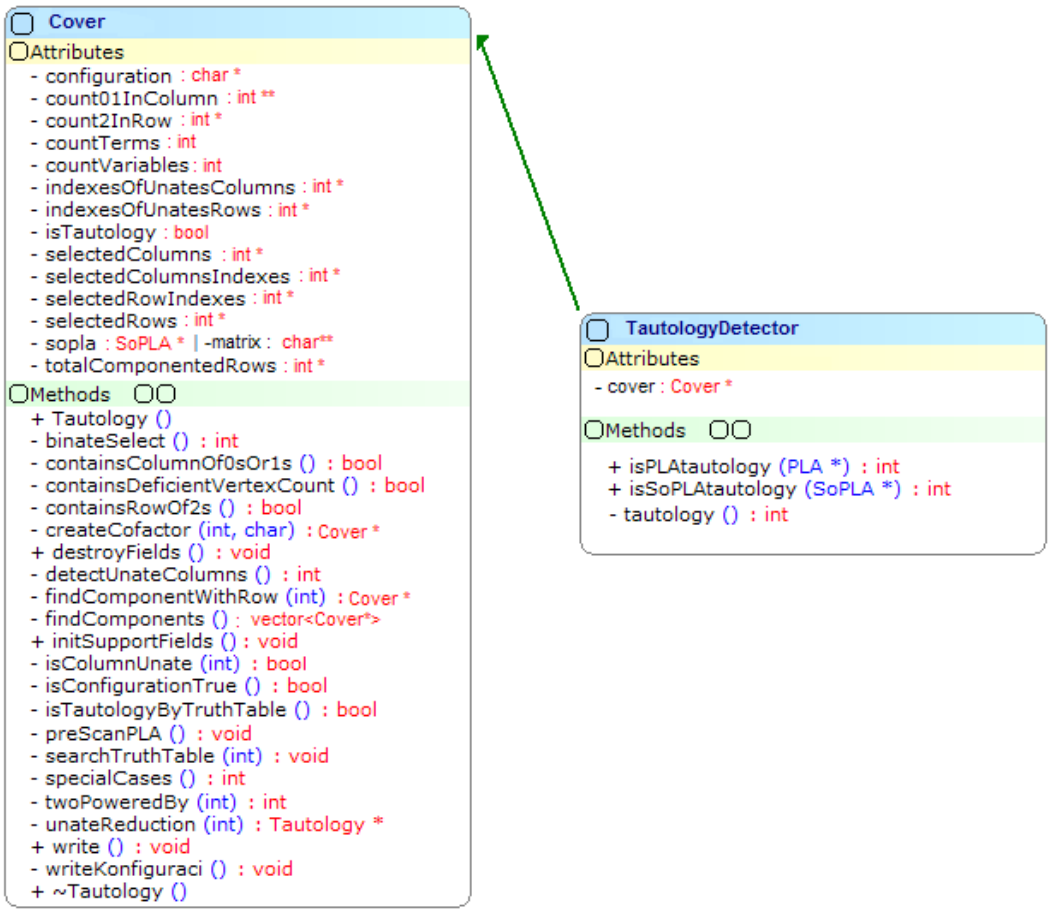
Následující kód ukazuje výše popisovaný hlavní cyklus detekující rozsah jedné komponenty:

```
vector<int> stack;
stack.push_back(rowIndex);
while(!stack.empty()){
  int exploringRow = stack.back();
  stack.pop_back();
  selectedRows[exploringRow] = 1;
  totalComponentedRows[exploringRow] = 1;
  selectedRowIndexes[componentRows] = exploringRow;
  componentRows++;
  for (int j = 0; j < countVariables; j++){
    if(sopla->GetTerm(exploringRow)->GetIM(j) != '-'
       && selectedColumns[j] == -1){
      selectedColumns[j] = 1;
      selectedColumnsIndexes[componentColumns] = j;
      componentColumns++;
      for (int i = 0; i < countTerms; i++){
        if(sopla->GetTerm(i)->GetIM(j) != '-' && selectedRows[i] == -1){
          stack.push_back(i);
          selectedRows[i] = 1;
        }
      }
    }
  }
}
```

Nyní, když víme, jak komponenty detekovat a vytvářet, zůstává otázkou: u jakých funkcí se vyplatí tuto deketci volat, protože samotná detekce, je časově náročná, v nejhorším případě dosahuje složitosti obdobné jako sekvenční průchod celým pokrytím funkce. Komponenty lze detekovat pouze v relativně „řídkých“ maticích. Nastavením ideální meze, od kdy se vyplatí tuto detekci volat, se bude věnovat podkapitola 5.7.

4.3 Popis datových struktur

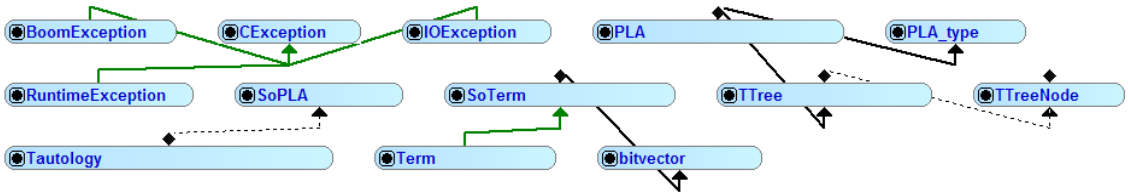
V podkapitole 3.5 zabývající se návrhem implementace jsme vymezili zde popisovaný nástroj na detekci tautologie jako jeden zapouzdřený celek. Ten bude rozdělen do dvou tříd: první třída *Cover* nese informace o datech reprezentujících logickou funkci, definuje metody potřebné pro detekci tautologie a uchovává pomocné datové struktury potřebné pro zefektivnění práce algoritmu. Druhá třída *TautologyDetecor* pak definuje metody pro komunikaci s okolním prostředím a realizuje detekci tautologie pomocí rekurzivního volání metody *tautology*, viz podkapitola 3.4.1. Tato třída je rovněž integrovatelná do jádra programu BOOM II a pro komunikaci s ním bude využívat interface definující dvě metody pro práci se strukturami PLA a $SoPLA$.



Obrázek 4.2: Diagram tříd TautologyDetection a Cover

Schéma tříd *Cover* a *TautologyDetector* je znázorněno pomocí class diagramu na obrázku 4.2. Rozdělení programu na tyto dvě třídy bylo nezbytné kvůli potřebě dealokování paměti

po vyhodnocení daneho pokrytí při rekurzivním běhu programu. Hlavní třídy jádra programu BOOM II, které jsou potřebné pro detekci tautologie a základní vztahy mezi jsou pak vyjádřeny pomocí obrázku 4.3.



Obrázek 4.3: Třídy jádra programu BOOM II

4.4 Přenositelnost na jiné platformy

Cílem tohoto projektu bylo vytvořit nástroj na detekci tautologie, který bude inegrovatelný do jádra programu na minimalizaci logických funkcí, BOOM II, jehož zdrojové kódy nejsou vázány na konkrétní operační systém. Náš nástroj na detekci tautologie by měl tedy být rovněž platformově nezávislý a to se podařilo splnit. Jelikož byl projekt vyvíjen pod operačním systémem Windows, byl z testovacích důvodů pro účely měření doby vykonávání výpočtu použit hlavičkový soubor *windows.h*. Při skutečném nasazení našeho nástroje pak již tento „#include“ nebude potřeba a projekt bude možné kompilovat nezávisle na druhu platformy (ověřeno pro Linux). Samotná funkce měřící čas, který výpočetní proces strávil na CPU[6] je použita z webových stránek předmětu $X36PAA$ [13].

4.5 Odhad asymptotické složitosti

V podkapitole 2.2.3 jsme zařadili problém detekce tautologie do třídy $co - NP$. V nejhorším možném případě nelze obecně tautologii detekovat v polynomiálním čase a jinak to tedy nebude ani s použitím našeho nástroje na detekci tautologie, který pracuje na rekurzivním principu podle schématu definovaném v podkapitole 3.4.1. Budeme se tedy nyní ptát na asymptotickou složitost našeho algoritmu, tedy na počet kroků, které algoritmus bude muset vykonat, aby nalezl řešení instance problému o velikosti N v tom

[6] Central Processing Unit

nejméně příznivém případě, kdy nedochází k detekci speciálních případů. V rekurzivním běhu algoritmu dochází v každém cyklu k rozdělení daného pokrytí na dvě další, přičemž počet proměnných klesne o jedna (nejvíce binátní proměnná, podle které vzniknout dva kofaktory). Hloubka rekurzivního stromu je tedy N a každý uzel se nám rozděluje na dvě větve. Výsledná asymptotická složitost pak vychází 2^N, tedy stejná jakou má algoritmus deteketující tautologii pomocí „hrubé síly“.

Efektivnost našeho algoritmu je však založena na detekci speciálních případů, unátní redukci a detekci komponent, které jsou popsány v podkapitole 4.2. Díky těmto opatřením dosahujeme mnohem vyšší výkonnosti, která je však závislá na charakteru konkrétního zadání. Výkonnost lze demostrovat v následující kapitole 5, která se zabývá testováním.

Budeme-li se dále ptát na paměťovou složitost zjistíme, že v nejhorším možném případě bude program potřebovat udržet v paměti tolik reprezentací kofaktorů, jakou má maximální hloubka rekurzivního stromu a tou je N, kde N je počet proměnných resp. $N-7$ hranice, od kdy se volá hrubá síla. Dále budeme uvažovat, že počet proměnných a termů se pro vzniklý kofaktor sníží o jedna. Pak paměťovou složitost můžeme popsat následujícím vztahem:

$$O(N^2 * T).$$

5 Testování

Cílem této kapitoly bude srovnání našeho nástroje na detekci tautologie s ostatními implementacemi vyjmenovanými v podkapitolách 2.4 - 2.7. Testování je rozděleno do několika kategorií, ve kterých budeme sledovat vliv konkrétních parametrů na čas potřebný k nalezení řešení u jednotlivých programů.

V grafech budeme na časové ose ve většině případů používat logaritmické měřítko a to kvůli řádovým rozdílům ve výkonnostech jednotlivých programů. U naměřených výsledcích budeme uvádět časy v sekundách. Hodnota „chyb. výsl.“ znamená, že program označil řešení za tautologii, i když jí nebyla. Hodnota „$> x$“ znamená, že program nenalezl řešení ani po xs. Hodnota „chyba“ znamená neočekávanou výjimku, která nastala při běhu programu. Měření bylo prováděno na procesoru AMD Athlon XP-M 2800+ 2,12GHz, 512MB RAM s operačním systémem Windows XP SP3. Pro případy, kdy jednotlivé nástroje nalézali řešení v různých časech pro konkrétní vygenerované instance, které se nelišily pouze svými parametry, jsou pak úváděné časy průměrem z takto získaných hodnot. V sloupci nadepsaném symbolem T je uvedeno, zda-li byly instance z dané testované skupiny tautologiemi. Testované programy jsou vyjmenovány v následující tabulce 5.1.

Tabulka 5.1: Přehled testovaných programů

Název programu	**Popis**
Kompl.	metoda výpočtu komplementu, kap. 2.4
BDD	detekce tautologie pomocí BDD, kap. 2.5
Graf	manipulace s funkcemi pomocí grafů, kap. 2.7
Graf II	totéž co předchozí + předpříprava PLA, kap. 2.7
Knih.-Espresso	knihovna z nástroje Espresso, kap. 2.6
Tautology	náše vlastní implementace nástroje na detekci tautologie

5.1 Generátor instancí logických funkcí

Před samotným testováním bylo třeba získat instance zadání logických funkcí ve formátu Espresso PLA. Pro tyto účely byl použit generátor PLA, který vznikl jako semestrální práce [3] kolegy Radka Chromého. Tento generátor umožňuje vytváření PLA na základě nastavení tří parametrů, kterými jsou: počet proměnných, počet termů a procentuální zastoupení DC symbolů. Testované instance jsou pak součástí CD přílohy této diplomové práce a jsou zařazeny podle jednotlivých kategorií.

5.2 Závislost doby výpočtu na počtu proměnných

Jedním z hlavních parametrů, kterým se vyznačuje logická funkce je počet proměnných, které se v ní vyskytují. Podkapitola 4.5 uvádí, že obecně asymptotická složitost detekce tautologie roste exponencielně s počtem proměnných logické funkce, neboť pravdivostní tabulka, kterou musíme v nejhorším možném případě celou projít má 2^N řádků, kde N představuje počet proměnných. Bude tedy velmi zajímavé sledovat, jaký vliv bude mít tento parametr na výpočetní časy testovaných programů.

Nejprve provedeme testování pro relativně malé instance (do 50 proměnných), aby je bylo možné řešit všemi implementacemi. Větší instance (do 1000 proměnných) jsou pak již řešitelné pouze některými algoritmy.

Pro malé instance byly zvoleny pevné hodnoty počtu termů - 500 a procentuální zastoupení DC symbolů na 30%. Jediným pohyblivým parametrem je tedy počet proměnných. V tabulce 5.2 jsou časy potřebné k nalezení řešení u šesti testovaných programů.

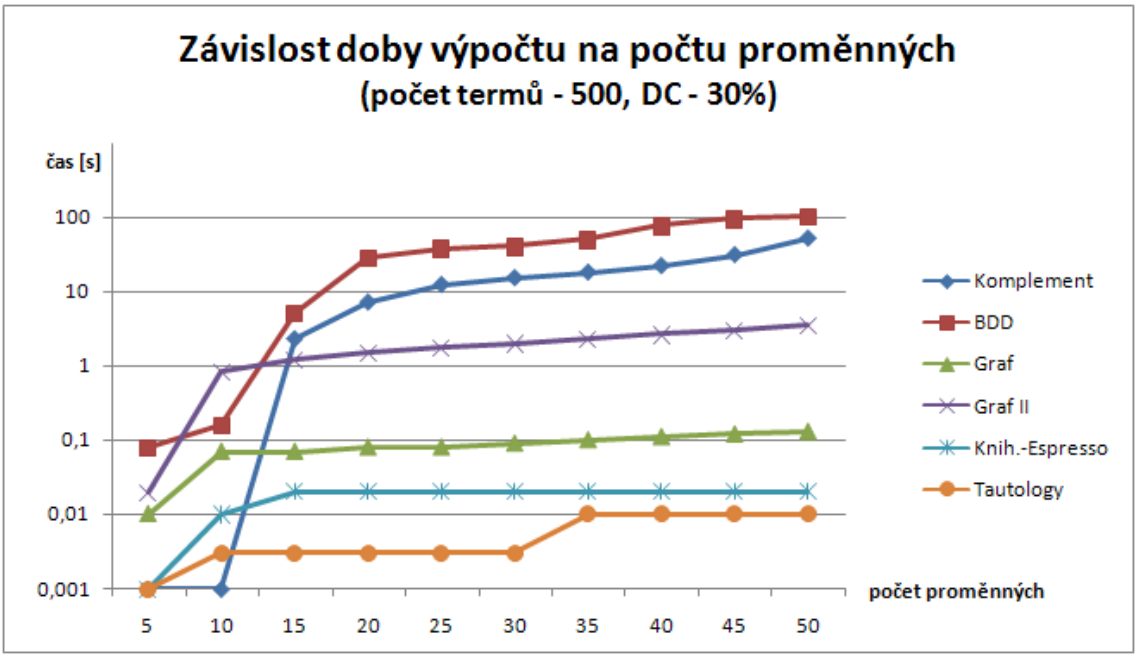
V grafu na obrázku 5.1 je znázorněna závislost doby výpočtu na počtu proměnných.

Pro větší instance byly zvoleny pevné hodnoty počtu termů - 1000 a procentuální zastoupení DC symbolů na 50%. Jediným pohyblivým parametrem je tedy opět počet proměnných. V tabulce 5.3 jsou časy potřebné k nalezení řešení u šesti testovaných programů.

V grafu na obrázku 5.2 je znázorněna závislost doby výpočtu na počtu proměnných pro relativně větší instance.

Tabulka 5.2: Závislost doby výpočtu na počtu proměnných

var.	T	Kompl.	BDD	Graf	Graf II	Knih.-Espresso	Tautology
5	1	chyba	0,08	0,01	0,02	0,001	0,001
10	0	chyba	0,16	0,07	0,83	0,01	0,003
15	0	2,32	5,08	0,07	1,24	0,02	0,003
20	0	7,23	28,58	0,08	1,52	0,02	0,003
25	0	12,45	38,05	0,08	1,78	0,02	0,003
30	0	15,34	40,28	0,09	2,01	0,02	0,003
35	0	18,23	50,41	0,10	2,35	0,02	0,01
40	0	22,32	77,23	0,11	2,67	0,02	0,01
45	0	31,34	94,57	0,12	3,01	0,02	0,01
50	0	52,12	102,39	0,13	3,52	0,02	0,02

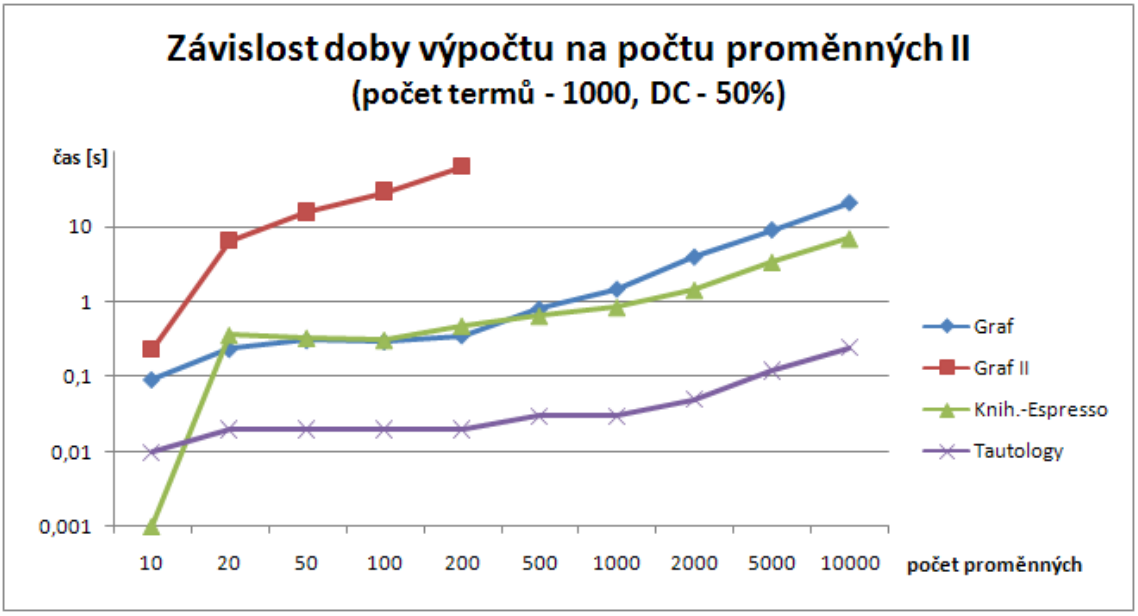


Obrázek 5.1: Závislost doby výpočtu na počtu proměnných

Malé instance byly měřitelné všemi algoritmy, výpočetní čas však výrazně narůstal s rostoucím počtem proměnných u metody *komplementu* a *BDD*. Program pracující s *grafy* poskytoval velmi dobré výsledky, varianta *graf II* byla o jeden řád horší. Nejrychlejší byl

Tabulka 5.3: Závislost doby výpočtu na počtu proměnných II

var.	T	Kompl.	BDD	Graf	Graf II	Knih.-Espresso	Tautology
10	1	chyba	0,25	0,09	0,23	0,001	0,01
20	0	chyba	chyb. výsl.	0,23	6,55	0,36	0,02
50	0	> 200	> 200	0,30	15,81	0,33	0,02
100	0	> 200	> 200	0,29	29,63	0,31	0,02
200	0	> 200	> 200	0,35	64,02	0,48	0,02
500	0	> 200	> 200	0,79	> 200	0,65	0,03
1000	0	> 200	> 200	1,46	> 200	0,85	0,03
2000	0	> 200	> 200	3,94	> 200	1,45	0,05
5000	0	> 200	> 200	9,07	> 200	3,43	0,12
10000	0	> 200	> 200	20,91	> 200	7,03	0,24



Obrázek 5.2: Závislost doby výpočtu na počtu proměnných II

náš nástroj spolu s knihovnou Espresso, tyto dva programy dosahovaly zhruba stejných a konstantních časů pro tento rozsah proměnných.

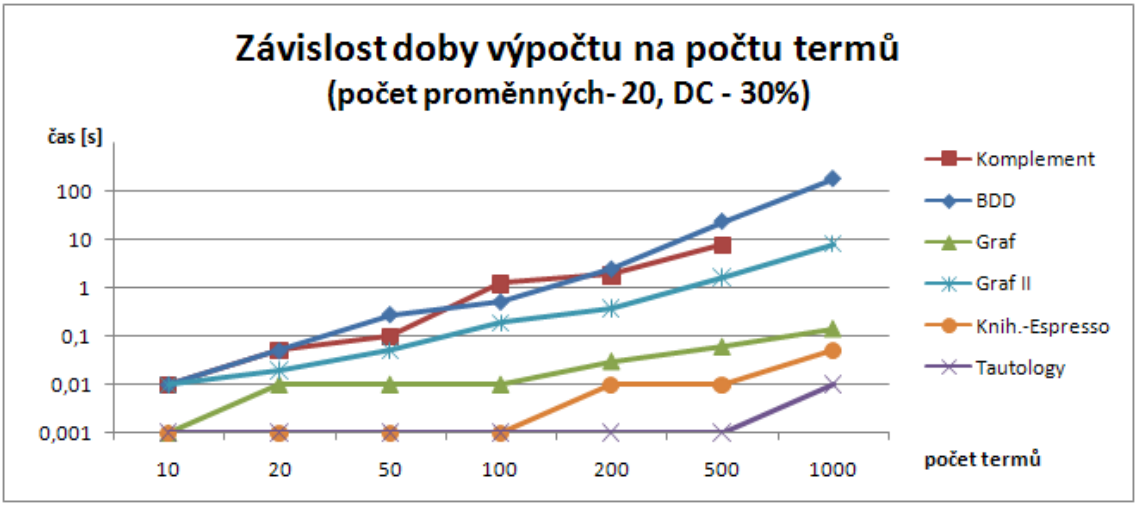
Relativně větší instance pak byly řešitelné jen třemi programy a to: naším nástrojem, knihovnou Espresso a nástrojem pracujícím pomocí grafů. Nejrychlejší byl náš nástroj,

zhruba 3x pomalejší pak byla knihovna Espresso a zhruba 2-3x pomalejší než knihovna Espresso byl nástroj pracujícím pomocí grafů.

5.3 Závislost na počtu termů

Tabulka 5.4: Závislost doby výpočtu na počtu termů

termů	T	Kompl.	BDD	Graf	Graf II	Knih.-Espresso	Tautology
10	0	0,05	0,01	0,001	0,01	0,01	0,001
20	0	0,10	0,05	0,01	0,02	0,001	0,001
50	0	0,02	0,28	0,001	0,05	0,001	0,001
100	0	1,2	0,51	0,01	0,19	0,001	0,001
200	0	1,8	2,44	0,03	0,37	0,01	0,001
500	0	7,5	23,61	0,06	1,61	0,01	0,001
1000	0	chyba	179,61	0,14	7,84	0,5	0,01



Obrázek 5.3: Závislost doby výpočtu na počtu termů

Dalším z parametrů, kterým lze charakterizovat danou logickou funkci, je počet jejich součinových termů, nebo-li počet řádků PLA. Víme, že čím více těchto řádků bude, tím vyšší je šance, že daná funkce bude tautologie a tím vyšší by měl být i výpočetní čas na

ověření takovéto funkce na tautologii. Následující test byl opět rozdělen pro malé a větší instance.

Pro menší instance byly zvoleny pevné hodnoty počtu proměnných - 20 a procentuální zastoupení DC symbolů na 30%. Jediným pohyblivým parametrem je tedy počet termů. V tabulce 5.4 jsou časy potřebné k nalezení řešení u šesti testovaných programů.

V grafu na obrázku 5.3 je znázorněna závislost doby výpočtu na počtu termů.

Pro větší instance byly zvoleny pevné hodnoty počtu proměnných - 100 a procentuální zastoupení DC symbolů na 50%. Jediným pohyblivým parametrem je tedy opět počet termů. V tabulce 5.5 jsou časy potřebné k nalezení řešení u šesti testovaných programů.

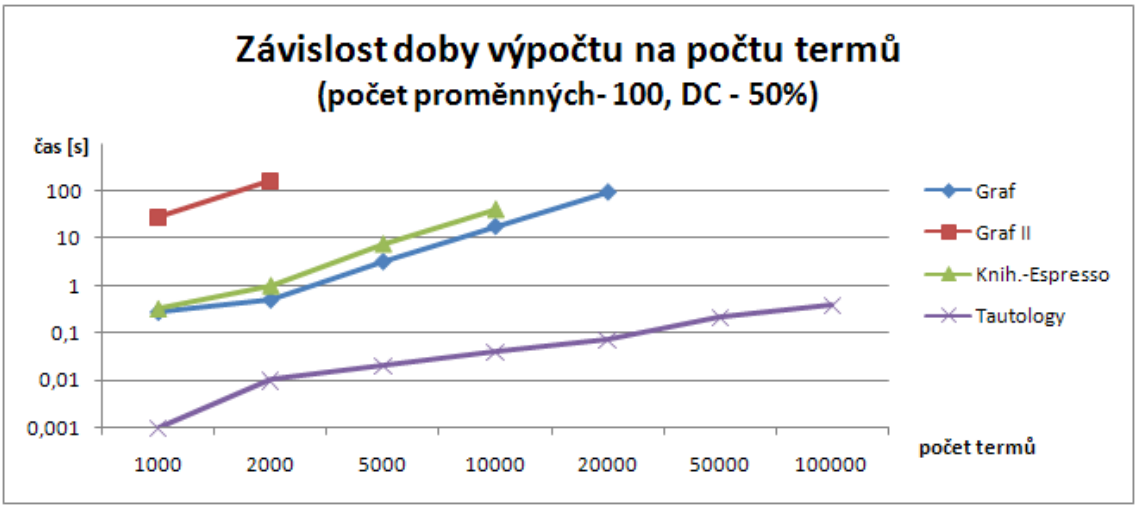
Tabulka 5.5: Závislost doby výpočtu na počtu termů II

termů	**T**	**Kompl.**	**BDD**	**Graf**	**Graf II**	**Knih.-Espresso**	**Tautology**
1000	0	> 200	> 200	0,28	29,12	0,32	0,001
2000	0	> 200	> 200	0,51	167,17	1,001	0,01
5000	0	> 200	> 200	3,28	> 200	7,651	0,02
10000	0	> 200	> 200	18,07	> 200	41,88	0,04
20000	0	> 200	> 200	96,04	> 200	> 200	0,07
50000	0	> 200	> 200	> 200	> 200	> 200	0,21
100000	0	> 200	> 200	> 200	> 200	> 200	0,38

V grafu na obrázku 5.4 je znázorněna závislost doby výpočtu na počtu termů pro větší instance.

Malé instance byly měřitelné všemi algoritmy, výpočetní čas však výrazně narůstal s rostoucím počtem termů u metody *komplementu* a *BDD*. Program pracující s *grafy* poskytoval velmi dobré výsledky, varianta *graf II* o jeden řád horší. Nejrychlejší byl náš nástroj spolu s knihovnou Espresso, tyto dva programy dosahovaly zhruba stejných a konstantních časů pro tento rozsah proměnných.

Zajímavé výsledky jsme pak obdrželi při testování větších instancí, kde jako nejrychlejší byl ve všech případech náš nástroj na detekci tautologie. U ostatních programů začal čas s rostoucím počtem termů velmi rapidně narůstat a to dokonce i u knihovny Espresso.



Obrázek 5.4: Závislost doby výpočtu na počtu termů II

Velmi zajímavé bude testování v následující podkapitole 5.4, která se zabývá testováním doby výpočtu v závislosti na procentuální zastoupení DC symbolů a to z toho důvodu, abychom podtvrdili nebo vyvrátili zde vzniklou domněnku, že knihovna Espresso řeší obtížně instance zadání s procentuálním zastoupením DC symbolů kolem 50%.

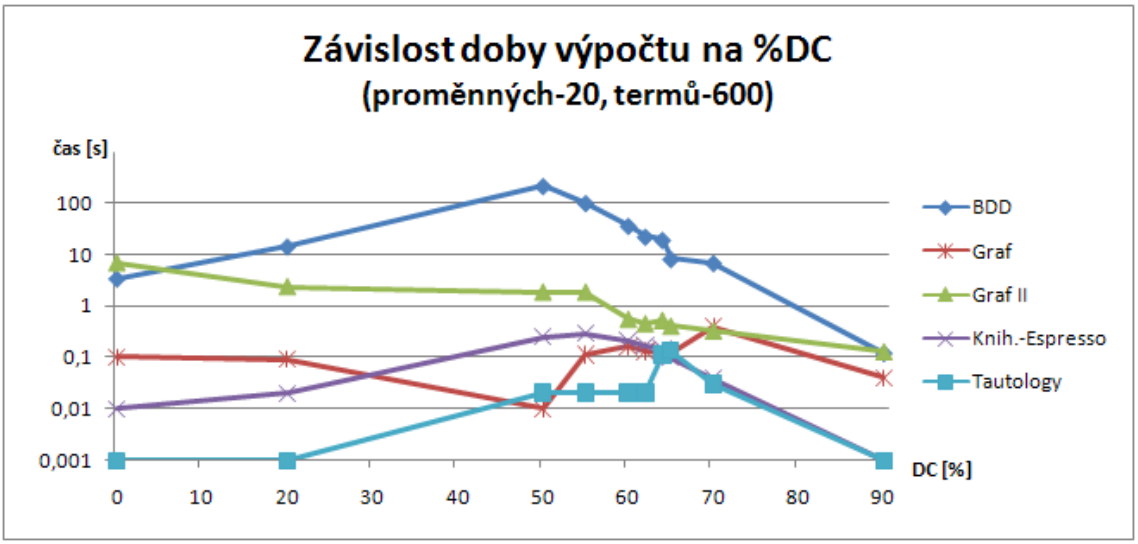
5.4 Závislost doby výpočtu na procentuálním zastoupení DC symbolů

Třetím z parametrů, kterým lze charakterizovat instanci zadání logické funkce, je procentuální zastoupení DC symbolů. V předchozí podkapitole 5.3 jsme z naměřených hodnot vyvodili předpoklad, že knihovna Espresso řeší obtížně instance zadání, kde zastoupení DC symbolů dosahuje kolem 50%. Zaměříme se tedy podrobnění na testování v tomto rozsahu hodnot. Následující test byl opět rozdělen do tří skupin podle velikosti instancí a do z důvodu možnosti sledování velmi významné závislosti mezi procentuálním zastoupením DC a dobou výpočtu.

Pro první skupinu instancí byly zvoleny pevné hodnoty počtu proměnných - 20 a počtu termů - 600. Jediným pohyblivým parametrem je tedy procentuální zastoupení DC symbolů. V tabulce 5.6 jsou časy potřebné k nalezení řešení u šesti testovaných programů. V grafu na obrázku 5.5 jsou pak tyto výsledky znázorněny graficky.

Tabulka 5.6: Závislost doby výpočtu na procentuálním zastoupení DC

DC [%]	T	Kompl.	BDD	Graf	Graf II	Knih.-Espresso	Tautology
0	0	2,07	3,41	0,10	7,01	0,01	0,001
20	0	5,03	14,37	0,09	2,41	0,02	0,001
50	0	chyba	217,22	0,01	1,89	0,25	0,02
55	0	chyba	101,03	0,11	1,88	0,29	0,02
60	0	chyba	36,29	0,16	0,56	0,21	0,02
62	0	chyba	22,32	0,13	0,46	0,17	0,02
64	0	chyba	19,32	0,12	0,53	0,15	0,11
65	1	chyba	8,21	0,12	0,41	0,10	0,13
70	1	chyba	6,85	0,39	0,33	0,04	0,03
90	1	0,01	0,12	0,04	0,13	0,001	0,001



Obrázek 5.5: Závislost doby výpočtu na procentuálním zastoupení DC

Pro druhou skupinu instancí byly zvoleny pevné hodnoty počtu proměnných - 30 a počtu termů - 4096. Jediným pohyblivým parametrem je tedy opět procentuální zastoupení DC symbolů. V tabulce 5.7 jsou časy potřebné k nalezení řešení u šesti testovaných programů.

V grafu na obrázku 5.6 je znázorněna závislost doby výpočtu na procentuálním zastoupení DC symbolů pro druhou skupinu instancí.

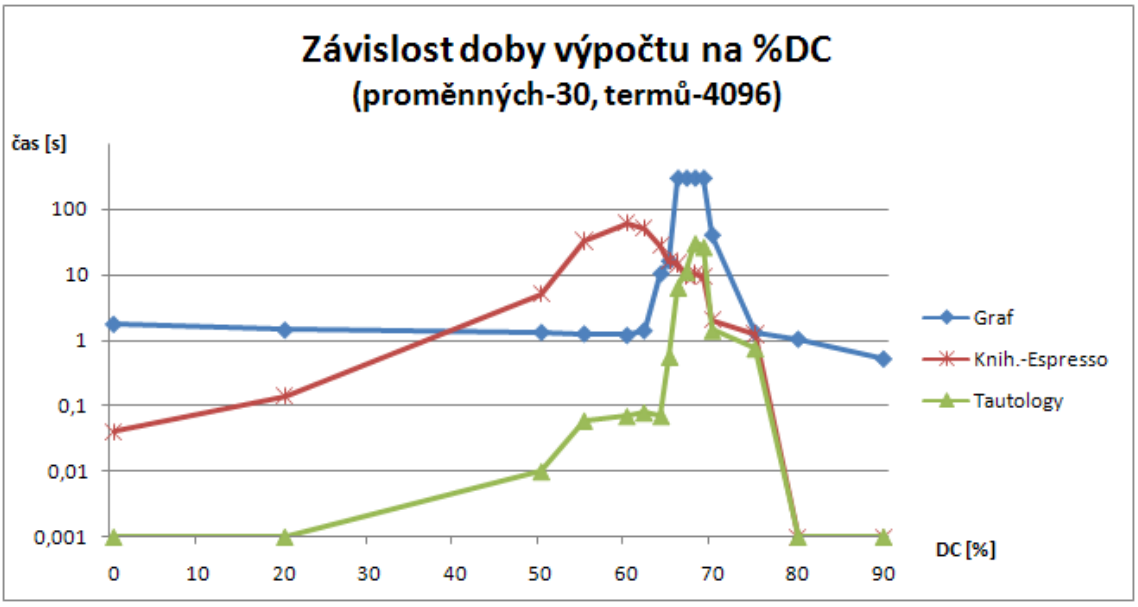
Tabulka 5.7: Závislost doby výpočtu na procentuálním zastoupení DC II

DC [%]	**T**	**Kompl.**	**BDD**	**Graf**	**Graf II**	**Knih.-Espresso**	**Tautology**
0	0	> 300	> 300	1,79	> 300	0,04	0,001
20	0	> 300	> 300	1,46	> 300	0,14	0,001
50	0	> 300	> 300	1,35	> 300	5,05	0,01
55	0	> 300	> 300	1,28	> 300	32,35	0,06
60	0	> 300	> 300	1,21	> 300	60,27	0,07
62	0	> 300	> 300	1,43	> 300	50,21	0,08
64	0	> 300	> 300	10,61	> 300	27,28	0,07
65	0	> 300	> 300	16,48	> 300	16,12	0,57
66	0	> 300	> 300	> 300	> 300	14,52	6,52
67	0	> 300	> 300	> 300	> 300	9,29	11,34
68	1	> 300	> 300	> 300	> 300	10,21	30,76
69	1	> 300	> 300	> 300	> 300	9,37	27,11
70	1	> 300	> 300	40,82	> 300	2,03	1,43
75	1	> 300	> 300	1,34	> 300	1,22	0,76
75	1	> 300	> 300	1,34	> 300	1,22	0,76
80	1	> 300	> 300	1,06	> 300	0,001	0,001
90	1	> 300	> 300	0,53	> 300	0,001	0,001

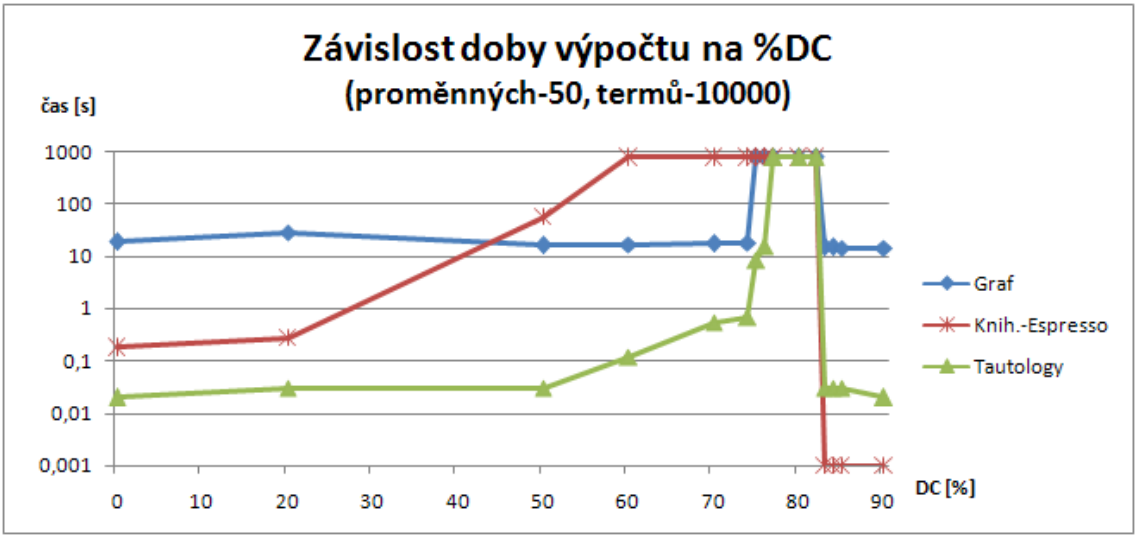
Pro třetí skupinu instancí byly zvoleny pevné hodnoty počtu proměnných - 50 a počtu termů - 10000. Jediným pohyblivým parametrem je tedy opět procentuální zastoupení DC symbolů. V tabulce 5.8 jsou časy potřebné k nalezení řešení u šesti testovaných programů.

V grafu na obrázku 5.7 je znázorněna závislost doby výpočtu na procentuálním zastoupení DC symbolů pro větší instance.

Z výsledků tohoto testování vyplývají především dvě významné skutečnosti, které jsou nejlépe vidititelné v grafu na obrázku 5.6. Pro malá procentuální zastoupení DC symbolů nejsou testované instance tautologiemi a programy nemají problémy s nalezením řešení. Začne-li však výskyt DC dosahovat kolem 65% dojde k efektu, že generované instance



Obrázek 5.6: Závislost doby výpočtu na procentuálním zastoupení DC II



Obrázek 5.7: Závislost doby výpočtu na procentuálním zastoupení DC III

začnou být tautologiemi. S roustoucím procentem DC jsou pak již všechny generované instance tautologiemi. Pro kontrolu byl pro generování instancí použit nástroj, který vznikl jako diplomová práce [7] kolegy Tomáše Měchury.

Tento princip lze zobecnit pro každou instanci tak, že pokud neobsahuje dostatečný počet DC symbolů, nemůže být tautologií. Pokud se však pohybujeme na rozmezí, jsou vygenerované instance tzv. „nejhůře detekovatelné“, jejich analýza vyžaduje procházení

Tabulka 5.8: Závislost doby výpočtu na procentuálním zastoupení DC III

DC [%]	T	Kompl.	BDD	Graf	Graf II	Knih.-Espresso	Tautology
0	0	> 780	> 780	18,53	> 780	0,18	0,02
20	0	> 780	> 780	27,99	> 780	0,27	0,03
50	0	> 780	> 780	16,13	> 780	57,22	0,03
60	0	> 780	> 780	16,64	> 780	> 780	0,12
70	0	> 780	> 780	17,46	> 780	> 780	0,54
74	0	> 780	> 780	17,65	> 780	> 780	0,69
75	0	> 780	> 780	> 780	> 780	> 780	8,5
76	0	> 780	> 780	> 780	> 780	> 780	15,42
77	0?1	> 780	> 780	> 780	> 780	> 780	> 780
80	0?1	> 780	> 780	> 780	> 780	> 780	> 780
82	0?1	> 780	> 780	> 780	> 780	> 780	> 780
83	1	> 780	> 780	14,71	> 780	0,001	0,03
84	1	> 780	> 780	15,01	> 780	0,001	0,03
85	0	> 780	> 780	14,57	> 780	0,001	0,03
90	0	chyba	> 780	14,41	> 780	0,001	0,02

rekurzivního stromu s nejvyšší hloubkou. Dokonce je možné pro dané rozměry instance (počet proměnných a počet termů) předpovědět, u jaké hranice nastane tento přechod. Vyjdeme přitom z předpokladů třetího speciálního případu[1], který definuje spodní mez pro počet DC symbolů. Nebudeme-li uvažovat překryv termů lze tuto hranici spočítat ze vztahu:

$$DC = \frac{pocetPromennych - log_2(pocetTermu)}{pocetPromennych}$$

Např. u testování středních instancí vychází tato hranice jako 60%. V grafu na obrázku 5.6 pak můžeme sledovat jaký má tato skutečnost vliv na výpočetní časy testovaných programů. Každá implementace je citlivá na tuto hranici v jiné míře. Knihovna Esppresso

[1]Viz podkapitola 4.2.2.

má problémy v rozmezí 55-70% procent, program detekující tautologii pomocí grafů v rozmezí 64-70% a náš nástroj v intervalu 66-70%, což významně souvisí s teoreticky vypočtenou hranicí, připočteme-li vliv překryvu termů. Při testování vliv překryvu posouval teoretickou hranici zhruba o 5-6%.

5.5 Srovnání přístupů k vnitřním reprezentacím

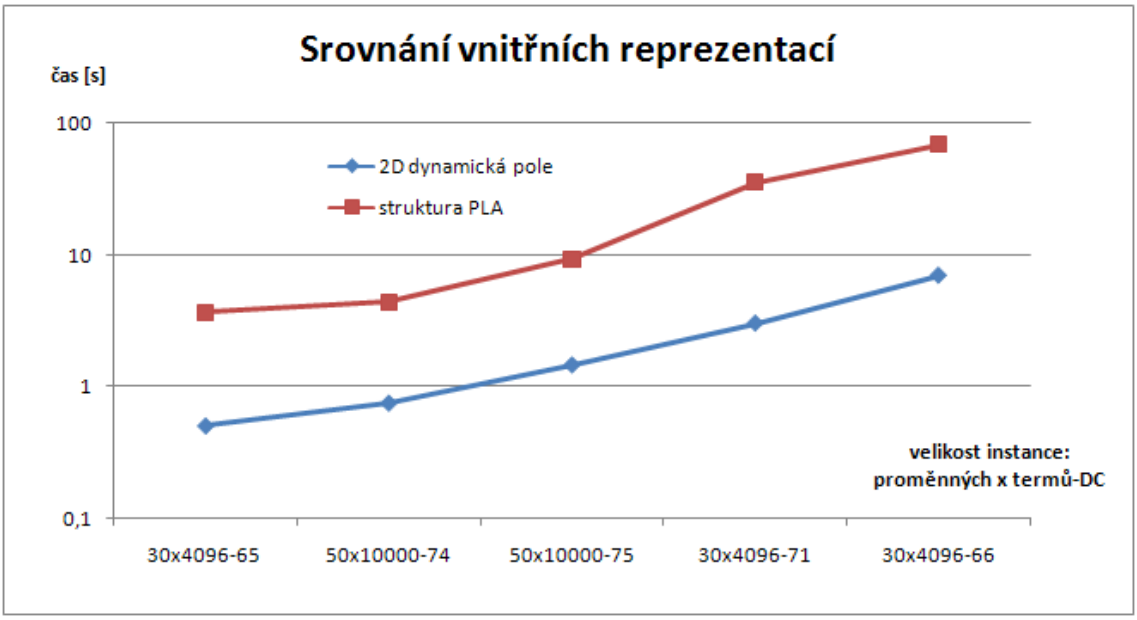
V podkapitole 4.2.1 jsme uvedli, že v průběhu vývoje tohoto nástroje na detekci tautologie vznikly dvě verze, z nichž každá používá jinou reprezentaci pro uchování pokrytí zpracovávané logické funkce. Nyní otestujeme výkonnosti obou variant.

V tabulce 5.9 jsou uvedeny časy potřebné pro vyhodnocení řešení dvou verzí nástroje na detekci tautologie. První používá k vnitřní reprezentaci logické funkce strukturu PLA z jádra programu BOOM II, druhá pak dynamická 2D pole. Graficky je pak toto srovnání vyjádřeno na obrázku 5.8.

Tabulka 5.9: Srovnání vnitřních reprezentací

Velikost instance proměnných x termů - DC	**T**	**struktura PLA**	**2D dynamické pole**
30x4096-65	0	3,72	0,51
50x10000-74	0	4,46	0,76
50x10000-75	0	9,36	1,78
30x4096-71	1	35,76	3,07
30x4096-66	0	68,91	7,07

Z naměřených hodnot vyplývá, než přechod na novou reprezentaci pomocí dvourozměrných dynamických přinesl očekávané zrychlení našeho nástroje na detekci tautologie. Toto zrychlení je natolik významné, že se rozhodlo o obdobné změně vnitřní reprezentace v jádru programu BOOM II, kde se bude pro uchovávání dat logické funkce ve třídě PLA rovněž používat tato efektivnější reprezentace.



Obrázek 5.8: Srovnání vnitřních reprezentací

5.6 Mez pro použití hrubé síly

V podkapitole 3.4.1, která se zabývá detekcí speciálních případů, jsme uvedli, že pokud bude testované pokrytí obsahovat sedm a méně proměnných použije se pro jeho vyhodnocení tzv. hrubé síly. Tuto hranici 7 bude nyní chápat jako parametr a ověříme, jak se budou měnit výpočetní časy našeho programu u několika zkušebních instancí. Výsledky jsou shrnuty v tabulce 5.10.

Tabulka 5.10: Mez pro použití hrubé síly

Velikost instance proměnných x termů - DC	**T**	**5**	**7**	**9**	**11**
30x4096-65	0	0,56	0,51	0,54	0,82
50x10000-74	0	0,68	0,67	0,67	0,67
50x10000-75	0	1,49	1,48	1,48	1,89
30x4096-71	1	3,09	3,07	3,07	3,11
30x4096-66	0	7,36	7,07	7,21	11,85

Tento test potvrdil, že jako nejvýhodnější hranice, od kdy lze použít k vyhodnocení

daného pokrytí hrubou sílu, je počet sedmi proměnných. Jiné hranice nepřinášely ve většině případů o mnoho horší výsledky, avšak byla-li by tato mez vyšší jak deset, vedlo by to k zhoršení efektivnosti našeho programu, viz např. poslední řádek v tabulce 5.10 pro mezní hodnotu 11.

5.7 Přínos unátní redukce a redukce komponent

V této podkapitole se zaměříme na unátní redukci a redukci komponent a budeme zjišťovat, jaký mají tyto funkce vliv na efektivnost našeho algoritmu. Test provedeme na čtyřech vygenerovaných instancích s parametry 50 proměnných, 10000 termů a 76% DC. Analyzovat je budeme naším nástrojem a to ve čtyřech variantách: s unátní redukcí i redukcí komponent (U+K), pouze s unátní redukcí (U-), pouze s redukcí komponent (-K) a nakonec ani s jednou touto funkcí (–). Výsledky tohoto testu jsou v tabulce 5.11. Dále jsou v této tabulce uvedeny i počty komponent, unátních sloupců a rekurzivních volání.

Tabulka 5.11: Vliv unátní redukce a redukce komponent na výkonnost

instance	**T**	**U+K**	**U-**	**-K**	**–**	**# un. sloupců**	**# komp.**	**# rekurzí**
I	0	3,54	2,70	3,53	2,63	915	24	41932
II	0	1,68	1,31	1,67	1,32	494	16	19410
III	0	1,21	0,89	1,23	0,87	234	4	7110
IV	0	3,55	2,71	3,58	2,74	763	26	41474

Z výsledků totoho testu vyplývá, že redukce komponent u těchto instancí výkonnost mírně zhoršuje. To je způsobeno zřejmě tím, že dochází pouze k detekci nejvíce dvou komponent z nichž jedna je téměř stejně velká jako celé původní pokrytí, což nepřinese velké zrychlení v důsledku očekávaného snížení počtu rekurzivních volání. Navíc je detekování takovéto komponenty časově náročnější než zrychlení, které nám detekce přinese. Detekci komponent je proto výhodné provádět pouze u některých instancí (s vysokým procentem DC).

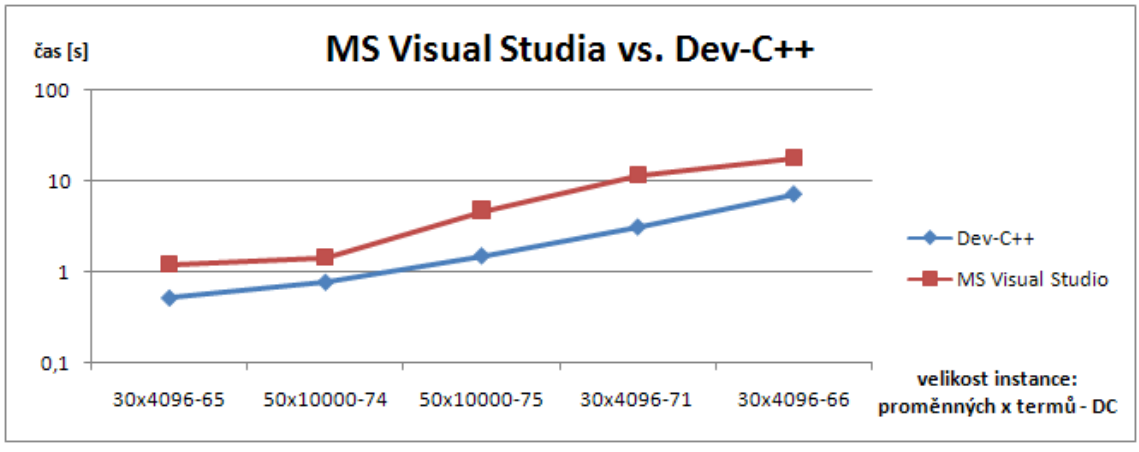
Unátní redukce přinesla jen velmi malý nárůst výkonnosti a to zřejmě kvůli tomu, že počet detekovaných unátních sloupců je zhruba 2000 menší než počet rekurzivních volání, proto touto redukcí ušetříme u testovaných instancí jen pouze několik málo procent z celkového počtu rekurzivních volání.

5.8 Srovnání kompilací MS Visual Studia a Dev-C++

V podkapitole 3.1 jsme uvedli, že podrobíme testu náš nástroj, bude-li zkompilován pomocí vývojového prostředí Dev-C++ a nebo MS Visual Studia. V tabulce 5.12 jsou pak časy potřebné pro nalezení řešení pro obě varianty našeho programu. Tyto hodnoty jsou pak vyneseny do grafu na obrázku 5.9

Tabulka 5.12: Srovnání kompilací MS Visual Studia a Dev-C++

Velikost instance proměnných x termů - DC	**T**	**MS Visual Studio**	**Dev-C++**
30x4096-65	0	1,21	0,51
50x10000-74	0	1,45	0,76
50x10000-75	0	4,73	1,78
30x4096-71	1	11,59	3,07
30x4096-66	0	17,82	7,07



Obrázek 5.9: Srovnání kompilací MS Visual Studia a Dev-C++

Ukázalo se, že kompilace, kterou jsme vytvořily pomocí MS Visual Studia 2005 je zhruba 3x pomalejší než program přeložený pomocí Dev-C++.

5.9 Zhodnocení naměřených výsledků

Shrneme-li výsledky, které jsme obdrželi v předcházejících kapitolách zabývajících se testováním programů uvedených v tabulce 5.1, můžeme tvrdit, že jako nejuniverzálnější nástroj na detekci tautologie je náš program, který dosahoval nejlepších výsledků u drtivé většiny případů. Velmi dobrých výsledků dosahovala i knihovna z nástroje Espresso, která však byla více citlivá na procentuální zastoupení DC, počet termů i počet proměnných. Ostatní programy dosahovaly výrazně horší výkonnosti, výjimečně poskytoval rychlá řešení program detekující tautologii pomocí grafů a to někdy i rychleji než knihovna z nástroje Espresso - viz např. graf 5.6.

Velmi významným poznatkem z testování byl objev hranice procentuálního zastoupení DC symbolů u níž dochází při generování instancí k přechodu mezi tautologiemi a netautologiemi. V podkapitole 5.4 jsme ukázali způsob, jak tuto hranici vypočítat. Budeme-li generovat instance s procentuálním zastoupením v oblasti blízké ($\pm$ jednotky procent) tomuto přechodu, budeme získávat logické funkce, které budou obtížněji vyhodnotitelné než ty se stejným počtem proměnných a termů. Každý program byl jinak citlivý na tuto hranici, viz graf 5.6. Nejuzší interval, ve kterém algoritmus potřeboval výrazně vyšší čas, měl náš nástroj na detekci tautologie.

Dalším významným přínosem testování bylo potvrzení efektivnosti nového způsobu vnitřní reprezentace analyzované logické funkce. Toto zrychlení je natolik významné, že se rozhodlo o obdobné změně vnitřní reprezentace v jádru programu BOOM II.

Zjistilo se také, že kompilace našeho projektu pomocí vývojového prostředí Dev-C++ je zhruba asi 3x rychlejší než pomocí MS Visual Studia 2005.

5.10 Možnosti kombinace různých algoritmů

Podle zadání této diplomové práce jsme měli uvažovat také možnost kombinace různých algoritmů pro detekci tautologie, za účelem optimalizace rychlosti. Jediným případem, kdy náš program nenalézal řešení nejrychleji, byl interval zhruba 5% v okolí přechodu mezi tautologiemi a ne-tautologiemi, viz podkapitola 5.4. V těchto případech byla rychlejší knihovna z nástroje Espresso. Zde se nabízejí dva možné postupy, buď se pokusit analyzovat proč je právě knihovna Espresso v této oblasti výkonnější (a to i případným prostudováním zdrojových kódů Esppressa) a doplnit náš nástroj o tyto poznatky a nebo můžeme náš nástroj upravit tak, aby při rozpoznání, že pracuje s instancí, která spadá do tohoto intervalu, použil k její analýze funkci z nástroje Espresso. Identifikace takovýchto logických funkcí by pak probíhala pomocí vztahu, který jsme uvedli v podkapitole 5.4.

6 Závěr

Cílem této diplomové práce bylo navrhnout a v jazyce C++ implementovat nástroj, který bude schopen vyhodnotit, zda-li je zadaná logická funkce tautologií. Důraz při této realizaci byl kladen na časovou efektivitu výpočtu a uvažovalo se i s možností kombinace různých algoritmů pro detekci tautologie za účelem optimalizace rychlosti. Výsledný nástroj je optimalizován tak, aby efektivně pracoval s různorodými vstupními instancemi logických funkcí co do počtu proměnných a termů či procentuálního zastoupení DC symbolů. Program je primárně založen na algoritmech použitých v nástroji Espresso a tvoří zapouzdřený celek, který je možné začlenit do jádra nástroje na minimalizaci logických funkcí, BOOM II. Lze tedy říci, že touto realizací nástroje na detekci tautologie byly splněny všechny cíle zadání.

V průběhu realizace byl použit odlišný způsob pro vnitřní reprezentaci analyzovaných logických funkcí, pomocí dvourozměrných dynamických polí, což přineslo natolik významné zrychlení oproti použití generických STL kolekcí ve třídách programu BOOM II, že se rozhodlo o obdobné změně vnitřní reprezentace logických funkcí ve třídách jádra programu BOOM II. Dále byl navržen a implementován vlastní způsob na detekci a tvorbu komponent v maticovém pokrytí analyzované logické funkce.

Ze závěrečného testování vyplývá, že jako nejuniverzálnější nástroj na detekci tautologie je náš program, který dosahoval nejlepších výsledků u drtivé většiny testovaných instancí. Velmi dobrých výsledků dosahovala i knihovna z nástroje Espresso, která však byla více citlivá na procentuální zastoupení DC symbolů, počet termů i počet proměnných.

Dalším významným poznatkem z testování byl objev hranice procentuálního zastoupení DC symbolů, u níž dochází při generování instancí k přechodu mezi tautologiemi a ne-tautologiemi. Odvodili jsme i vztah, jak tuto hranici vypočítat. Výsledky testování dokládají, že tato hranice má zásadní vliv na výpočetní časy jednotlivých algoritmů. Testování také přineslo poznatek, že kompilace programu pod vývojovým prostředím Dev-C++ je významně rychlejší než překlad pomocí MS Visual Studia 2005.

Nástroj nabízí možnosti na vylepšení především v oblastech unátní redukce a detekce komponent, eventuelně by se daly implementovat obdobné prvky zvyšující časovou efektivitu.

7 Literatura

[1] R. K. Brayton. Logic synthesis, 2000.
www.eecs.berkeley.edu/~brayton/courses/219b/ppslides/.

[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelii. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[3] R. Chromý. *Detekce tautologie, výpočet komplementu booleovské funkce*. Semestrální projekt, ČVUT v Praze, FEL, 2004.

[4] M. Demlová. Přednášky z předmětu X01DML, 2009.
http://math.feld.cvut.cz/demlova/teaching/dml/pred02.pdf.

[5] E-academy. Student's downloads, 2009.
http://www.e-academy.com/.

[6] P. Fišer and H. Kubátová. Flexible Two-Level Boolean Minimizer BOOM-II and its applications. *In Proceedings of 9th Euromicro Conference on Digital System Design. Los Alamitos: IEEE Computer Society,ISBN 0-7695-2609-8*, 2006.

[7] T. Měchura. *Parametrizovaný generátor náhodných logických obvodů*. Diplomová práce, ČVUT v Praze, FEL, 2008.

[8] M. Miklánek. *Úprava minimalizačního nástroje ESPRESSO*. Bakalářská práce, ČVUT v Praze, FEL, 2008.

[9] M. Navrkal. *Detekce tautologie pomocí BDD*. Bakalářská práce, ČVUT v Praze, FEL, 2006.

[10] P. Vála. *Manipulace s logickými funkcemi pomocí grafů*. Diplomová práce, ČVUT v Praze, FEL, 2009.

[11] Wikipedia. Asymptotická složitost, 2009.
http://cs.wikipedia.org/wiki/Asymptotick%C3%A1_slo%C5%BEitost.

[12] Wikipedia. Binary decision diagram, 2009.
http://en.wikipedia.org/wiki/Binary_decision_diagram.

[13] X36PAA. Měření CPU času ve windows v C++, 2006.
http://service.felk.cvut.cz/courses/X36PAA/CPU_time_C++.html.

A Obsah přiloženého CD

Na CD, které je přiloženo k této diplomové práci, jsou uložena tato data:

- Zdrojový kód programu na detekci tautologie (dvě verze).
- Spustitelná verze programu (dvě varianty).
- Spustitelné verze všech porovnávaných programů.
- Testované instance logických funkcí ve formátu PLA.
- Text diplomové práce ve formátu pdf.
- Zdrojový kód textu diplomové práce ve formátu LaTeX.
- Obrázky použité v textu DP.

B Malý slovník použitých pojmů a zkratek

- ”0” - daná proměnná v přímém tvaru (např. a)
- ”1” - daná proměnná v negovaném tvaru (např. $\overline{a}$)
- ”2” - na hodnotě dané proměnné v daném termu nezáleží (totéž co DC)
- BDD - binární rozhodovací diagram, viz podkapitola 2.5.
- binátní - lze použít ve spojení pro sloupec nebo proměnnou, říkáme, že binátní je takový sloupec (proměnná), který obsahuje alespoň jeden symbol ”0” a současně i alespoň jeden symbol ”1”
- DC - na hodnotě dané proměnné v daném termu nezáleží, z angl. „Don’t care“ (totéž co ”2”)
- kofaktor - podmnožina z původní logické funkce, ze které je odvozen na základě daného termu, viz podkapitola 2.2.4.
- komponenta - souvislý blok pokrytí logické funkce, kde se souvislost chápe na základě ortogonální incidence symbolů ”0” nebo ”1”, viz podkapitola 4.2.5.
- PLA - Programmable logic array, nebo vnitřní třída programu BOOM II reprezentující logickou funkci, zadanou ve formátu Espresso PLA
- pokrytí - dvourozměrné pole, kde sloupce tvoří proměnné dané logické funkce a řádky její termy
- unátní - lze použít ve spojení pro sloupec nebo proměnnou, říkáme, že unátní je takový sloupec (proměnná), který obsahuje buď pouze symboly ”0” a ”2” a nebo pouze ”1” a ”2”.

C Formát Espresso PLA

Jako zdroj dat pro načítání logických funkcí jsou použity soubory typu PLA formátu Espresso, zde se krátce zmíníme o jejich struktuře. Pomocí PLA, lze reprezentovat logickou funkci ve formátu DNF.

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 [d1 ... dn].

.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 $[s_0$ až $[s_1$ jsou považovány za jednu vícehodnotovou proměnnou.

.symbolic-output $[s_0][s_1]...[s_n]$, $[t_0][t_1]...[t_m]$ specifikuje, že výstupní funkce [s0] až [sn] 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 hodnota funkce (pro naše účely bude funkce zadána onsetem, tudíž výstup bude vždy 1).

D Integrace do jádra BOOM II

Náš nástroj na detekci tautologie tvoří zapouzdřenou třídu, které je integrovatelná do jádra nástroje na minimalizaci logických funkcí, BOOM II. Vzájemná komunikace probíhá pomocí *interface*, který definuje dvě metody:

- bool isPlaTautology(PLA* pla);
- bool isSoPlaTautology(SoPLA* pla);

První z nich se používá pro příjem vstupních dat ve formě PLA struktury, která je vnitřní třídou nástroje BOOM II. Druhá metoda je určena pro práci se speciálním případem jedno-vstupového PLA tzv. SoPLA[1].

Příklad integrace a inicializace detekce tautologie je demonstrována následující úkázkou kódu:

```
PLA* pla = new PLA();
pla->LoadPLA("zadani.pla");
TautologyDetector *tauDetector = new TautologyDetector();
bool isTautology;
isTautology = tauDetector->isPLAtautology(pla);
```

[1]SoPLA = Single Output PLA