České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Převod víceúrovňové logické sítě na dvouúrovňovou pomocí BDD

Přemysl Rucký

Vedoucí práce: ing. Petr Fišer

Studijní program: Informatika a výpočetní technika

leden 2007

Poděkování

Chtěl bych poděkovat svým rodičům za pevné nervy a podporu při studiu. Dále pak paní Marii Váňové za jazykovou korekturu. V neposlední řadě pak ing. Petru Fišerovi za cenné konzultace, nápady a připomínky vedoucí ke zkvalitnění celé práce.

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 Praze dne 19.1.2007 .....

Abstract

This work deals with a transformation of a multi-level logic network into a two-level logic network, which means the computation of a circuit's truth table. It uses structures called BDD (Binary Decision Diagrams). The usage of BDDs is closely analyzed and evaluated in terms of the variable ordering and the number of generated terms. The method is compared with the logic synthesis system called MVSIS. The work also includes a brand new algorithm for minimization of logic functions that is also closely analyzed and compared with the system for logic minimization called ESPRESSO. The bench format is used as the input format and the PLA format is used as the output format.

Abstrakt

Práce se zabývá převodem víceúrovňové logické sítě na dvouúrovňovou, tedy výpočtem pravdivostní tabulky obvodu. Využívá k tomu struktur BDD (Binary Decision Diagram), jejichž využití je důkladně analyzováno a vyhodnoceno, jak z hlediska pořadí proměnných, tak z hlediska počtu generovaných termů. Metoda je srovnána se systémem pro logickou syntézu MVSIS. Součástí práce je také nový algoritmus pro minimalizaci logických funkcí, který je také analyzován a porovnán se systémem pro minimalizaci logických funkcí ESPRESSO. Jako vstupní formát je použit formát BENCH; výstup probíhá do souboru ve formátu PLA.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

1.1 Motivace

Při řešení různých úloh z oblasti syntézy logických obvodů je často potřeba provádět převod mezi různými reprezentacemi logických obvodů. V práci [11], realizované na katedře K336, byl součástí řešení převod logické sítě z víceúrovňové na dvouúrovňovou a poté naopak. Tato diplomová práce se zabývá prvním případem, tedy převodem víceúrovňové logické sítě na dvouúrovňovou, jinak také *collapsingem*. V důsledku to znamená redukci mnoha logických operací na jednoduché zobrazení ohodnocení vstupních proměnných na hodnotu výstupních proměnných, tzn. výpočet *pravdivostní tabulky*. Protože toto je, zejména u rozsáhlých logických obvodů, velmi obtížné, je třeba vytvořit co nejefektivnější implementaci řešení. K řešení toho problému existuje několik přístupů. Jedním z nich je například metoda průchodu obvodu po hradlech, kdy je obvod procházen od nejnižší (vstupy) nebo nejvyšší (výstupy) úrovně a výstupní proměnné každého hradla jsou postupně nahrazovány jejich vstupními proměnnými. Tímto způsobem je vygenerována logická funkce ve formě součtu součinů a ta je poté minimalizována. V této práci byl ale zvolen odlišný přístup. V dnešní době se totiž ukazuje, že velmi efektivní metodou práce s logickými funkcemi je využití tzv. *binárních rozhodovacích diagramů* (BDD - Binary Decision Diagrams). Jejich využití je mnohé; od testování obvodů, přes rychlé vyhodnocování logických funkcí až třeba po řešení problému splnitelnosti booleovské formule. Proto bylo rozhodnuto, že při řešení tohoto problému budou tyto diagramy využity. Cílem tedy bude analyzovat možnosti využití BDD při řešení tohoto problému. V průběhu práce se objevilo mnoho zajímavých podnětů a problémů, které mimo jiné vyústily ve vytvoření zcela nového algoritmu pro minimalizaci logických funkcí. Ten řeší problém minimalizace pomocí ternárního stromu. Jeho výsledky jsou velmi slibné a samotný algoritmus se může stát základem pro zajímavou budoucí práci v této oblasti. Pro práci s BDD existuje několik různých knihoven, ale po dohodě s vedoucím práce byl zvolen balík CUDD. Jeho vlastnosti budou detailněji popsány v kapitole 5.2.

1.2 Cíl práce

Primárním cílem práce je tedy navrhnout a implementovat metodu převodu víceúrovňové logické sítě na dvouúrovňovou využívající BDD a detailně analyzovat a vyhodnotit její vlastnosti. Druhým cílem práce je vyhodnocení efektivity použití BDD pro řešení tohoto problému, proto bude součástí práce také porovnání tohoto přístupu s jinými přístupy k řešení téhož problému. Budu tedy mou implementaci porovnávat se systémem pro syntézu logických obvodů `mvsis` [3]. Součástí implementace je i nový algoritmus pro minimalizaci logických funkcí, který bude také důkladně analyzován a srovnán se systémem pro minimalizaci logických funkcí `espresso` [4].

1.3 Základní pojmy

V této práci budu pracovat s několika základními pojmy, které je nutno vysvětlit čtenáři.

Logická proměnná - proměnná logické funkce; může nabývat pouze hodnot "0" a "1"
Logická (booleovská) funkce - vztah závislých a nezávislých logických proměnných; popisuje chování obvodu; dále jen *funkce*
Logický prvek - fyzický člen realizující nějakou logickou funkci
Logický obvod - souhrn logických prvků propojených v rámci realizace složitější logické funkce
Kombinační logický obvod - takový logický obvod, ve kterém závisí výstupní hodnoty všech logických prvků pouze na kombinaci jejich vstupních proměnných; v našem případě se z důvodu

jednoduchosti omezíme pouze na takový typ logických obvodů
Logická síť - teoretický model logického obvodu, kde fyzické logické prvky a spoje nahrazují značky a jejich propojení
***K*-úrovňová logická síť** - taková logická síť, ve které je největší počet značek mezi vstupními a výstupními proměnnými roven K
Pravdivostní tabulka - taková reprezentace logické funkce, ve které je definován konkrétní vztah mezi ohodnocením vstupních proměnných a hodnotami výstupních proměnných; příklad pravdivostní tabulky pro funkci $f = x_1 \cdot x_2$ je v tabulce 1.1

Tabulka 1.1: Pravdivostní tabulka pro funkci $f = x_1 \cdot x_2$

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

Term - v některých literaturách se o termech mluví jako o vektorech booleovské funkce; složen z logických proměnných, vázaných spolu nějakým logickým operátorem; podle operátoru se dělí na součinový (např. $x_1 \cdot x_2 \cdot x_3$) a součtový (např. $x_1 + x_2 + x_3$)
Disjunktní normální forma - algebraická forma reprezentace logické funkce skládající se ze součtu součinových termů
Implikant logické funkce - jedná se o výraz ve tvaru součinového termu, pro který platí, že danou funkci implikuje, tzn. jestliže nabývá hodnoty "1", daná funkce nabývá též hodnoty "1"; implikant nazveme přímým implikantem právě tehdy, když po vypuštění libovolného literálu (proměnné) přestává být implikantem. Podstatný implikant je takový implikant, který je součástí každého minimálního řešení dané logické funkce
On-set - množina termů s hodnotou implikující výslednou hodnotu funkce "1"
Off-set - množina termů s hodnotou implikující výslednou hodnotu funkce "0"
Dc-set - don't care set; termy, pro které není výsledná funkce specifikována
Binární rozhodovací diagramy - grafové struktury určené pro efektivní manipulaci s logickými funkcemi, publikovány v práci [2]; dále jen BDD

2 Rozbor problému

V této kapitole popíši problém výpočtu pravdivostní tabulky podrobněji. Jaké problémy skýtá použití BDD, jak efektivně vytvářet BDD ze souborů v zadaném formátu, jak z BDD generovat termy výstupní pravdivostní tabulky a proč je třeba pravdivostní tabulku minimalizovat.

2.1 Binární rozhodovací diagramy

V této kapitole nejdříve popíši BDD, jejich přednosti a nedostatky, co znamenají zkratky OBDD a ROBDD a další podrobnosti týkající se využití BDD při řešení tohoto problému.

2.1.1 Základní struktura BDD

BDD je, jak již bylo zmíněno výše, grafová struktura navržená k rychlému vyhodnocování logických funkcí. Má tvar orientovaného acyklického kořenového grafu. Jednotlivé uzly odpovídají iteracím Shannonova teorému o rozkladu, který říká, že každou logickou funkci lze rozložit podle následujícího vzorce:

$$f(a, \bar{a}, b, \bar{b}, ...) = a \cdot f(1, 0, b, \bar{b}, ...) + \bar{a} \cdot f(0, 1, b, \bar{b}, ...)$$

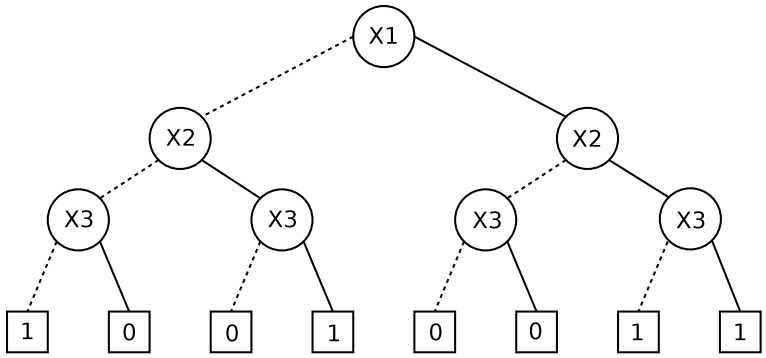
Převedeno do lidské řeči to znamená, že každou funkci můžeme rozdělit na dvě. Před první funkci vytkneme její pozitivní reprezentaci a ve funkci její výskyty nahradíme jedničkou. Negované potom nahradíme nulou. To samé, pouze s negovanou formou, uděláme s druhou funkcí.

Do struktury samotného BDD se to promítne následovně. Každý vnitřní uzel (neterminál) grafu má 2 následníky (tj. rozklad funkce). Jeden odpovídá funkci, kde proměnná, kterou daný uzel reprezentuje, nabývá hodnoty "1" a druhý odpovídá funkci, kde proměnná nabývá hodnoty "0". Pro přehlednost budu označovat vnitřní uzly jmény proměnných, na které se v dané iteraci aplikoval Shannonův teorém. Obrázek 2.1 ukazuje příklad BDD pro funkci $f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_2 + x_2 \cdot x_3$, kde čárkovaná čára značí hodnotu proměnné "0" a plná čára hodnotu proměnné "1". Listy uzlu (terminály) mohou nabývat hodnot "1" nebo "0" a určují hodnotu funkce pro takové ohodnocení proměnných, které vzniklo průchodem grafu od jeho kořene až do listu. Jako příklad je možno uvést první list zleva na obrázku 2.1, který má hodnotu "1". Je vidět, že odpovídá hodnotě funkce pro ohodnocení proměnných $x_1 = x_2 = x_3 = 0$. Z kořene grafu jsme totiž postupovali vždy po hraně označující nulové ohodnocení dané proměnné. Pohledem na zadanou funkci zjistíme, že tento výsledek je správný, protože při nulovém ohodnocení všech tří proměnných bude mít první term logickou hodnotu "1", tudíž je hodnota celé funkce také "1".

2.1.2 Uspořádané BDD - OBDD

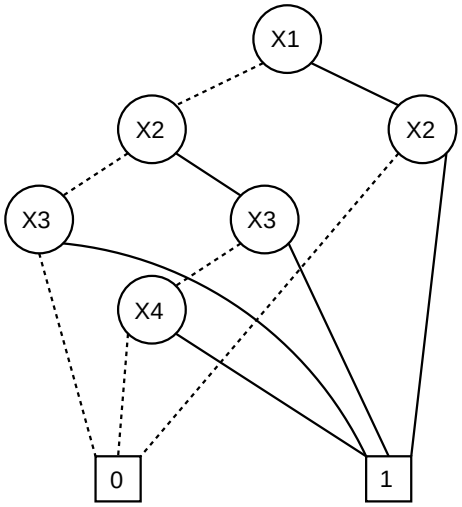
Zkratka OBDD znamená Ordered BDD, tj. uspořádané binární rozhodovací diagramy. O BDD řekneme, že je uspořádaný, resp. že zachovává uspořádání $<$, právě tehdy, když pro každé dvě proměnné, pro které platí uspořádání $x_i < x_j$, narazíme při jakémkoliv průchodu grafem od kořene k listu dříve na proměnnou x_i než na proměnnou x_j. V praxi je tato vlastnost velmi důležitá, protože zásadně ovlivňuje velikost (tj. počet uzlů) výsledného OBDD.

Mějme funkci $f = x_1 \cdot x_2 + \bar{x}_1 \cdot x_3 + x_2 \cdot \bar{x}_3 \cdot x_4$. Na obrázku 2.2 vidíme, jaký OBDD vznikne pro uspořádání proměnných $x_1 < x_2 < x_3 < x_4$. Výsledný OBDD obsahuje celkem 8 uzlů, z toho 6 neterminálů a 2 terminály. Prostou změnou tohoto uspořádání na $x_1 < x_3 < x_4 < x_2$ (obrázek 2.3) se podařilo snížit počet uzlů OBDD na 6, z toho 4 jsou neterminály a 2 terminály. Počet neterminálů se tedy snížil o třetinu. Toto je u rozsáhlých OBDD naprosto zásadní pro efektivní



Obrázek 2.1: Příklad binárního rozhodovacího diagramu

práci s OBDD. Obecně nelze předem říci, jaké uspořádání proměnných povede k minimálnímu počtu uzlů v OBDD. Existují však heuristické algoritmy, které dokáží efektivně určit minimální počet uzlů nebo se mu alespoň přiblížit. Tyto heuristiky budou podrobněji popsány v kapitole 3.5.



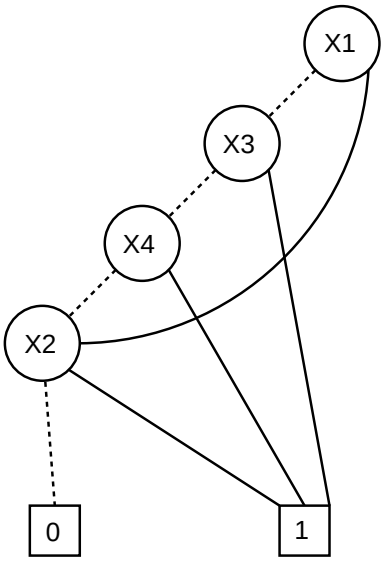
Obrázek 2.2: OBDD zachovávající uspořádání $x_1 < x_2 < x_3 < x_4$

2.1.3 Redukované OBDD - ROBDD

Zkratka ROBDD znamená Reduced OBDD, tzn. redukované uspořádané binární rozhodovací diagramy. Při tvorbě BDD dochází často k redundancím. Ty nám komplikují život hlavně tím, že zvyšují počet uzlů v grafu, což s sebou nese mnoho negativních důsledků. Redundance vyskytující se běžně v BDD lze rozdělit v zásadě na 3 druhy:

- opakování terminálních uzlů
- levý a pravý potomek neterminálního uzlu je ten samý uzel, tzn. daný neterminální uzel je nadbytečný
- výskyt identických podgrafů v BDD

Abychom odstranili tyto redundance musíme podniknout následující kroky:



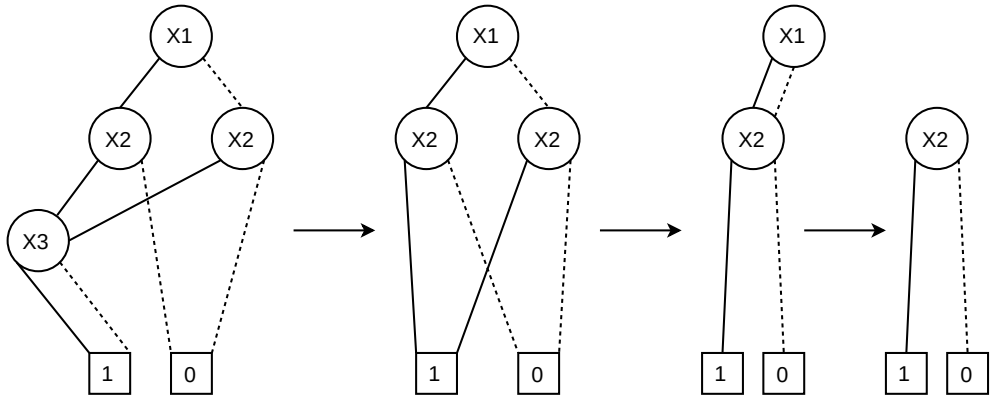
Obrázek 2.3: OBDD zachovávající uspořádání $x_1 < x_3 < x_4 < x_2$

- opakování terminálních uzlů - odstraníme všechny terminály stejné hodnoty, kromě jednoho; do něj potom svedeme všechny hrany vedoucí původně do zrušených terminálů; toto provedeme pro oba druhy terminálů (jedničku a nulu)
- levý a pravý potomek neterminálního uzlu je ten samý uzel, tzn. daný neterminální uzel je nadbytečný - pokud se levý a pravý potomek uzlu u rovnají, přesměrujeme hranu z rodiče uzlu u na libovolného potomka uzlu u a zrušíme uzel u
- uvnitř grafu mohou být některé podgrafy identické - pokud se leví potomci uzlů u a v rovnají a zároveň se rovnají i jejich praví potomci, zrušíme uzel u a všechny hrany vedoucí do uzlu u přesměrujeme do uzlu v

Tato redukční pravidla provádíme do té doby, dokud přinášejí nějaký efekt, tj. snižují počet uzlů v grafu. Na obrázku 5.4 vidíme průběh redukčního algoritmu krok po kroku. První obrázek již reprezentuje graf po aplikování prvního redukčního kroku, odstranění přebytečných terminálů. Na další části je vidět, že uzel označený jako x_3, splňuje podmínku pro vyřazení (obě hrany vedou do stejného potomka). Po vyřazení uzlu x_3 zjistíme, že oba uzly označené x_2 jsou identické, a proto můžeme jeden z nich vyřadit. Tím se dostaneme do situace, kdy uzel x_1 je nadbytečný, oba jeho potomci vedou do stejného uzlu. Vidíme, že výsledný ROBDD je radikálně menší než původní OBDD.

2.1.4 Tvorba ROBDD

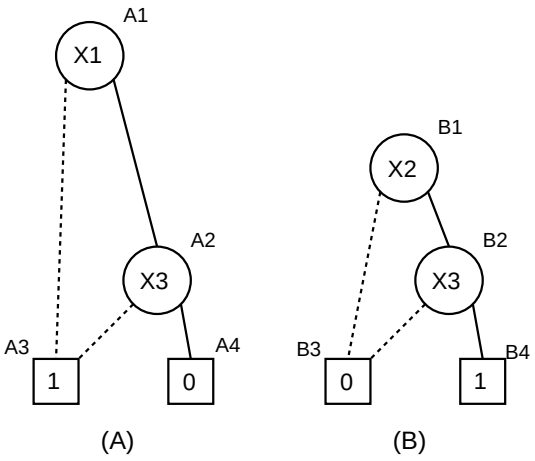
Teď víme, jak z nějakého BDD sestavit ROBDD. Jak ale sestavit ROBDD, když známe pouze předpis logické funkce? Kdybychom postupovali intuitivně podle výše zmíněného Shannonova teorému a postupně rozkládali funkci skládající se z mnoha termů, dostali bychom se do problémů kvůli velikosti vytvářeného ROBDD. Pokud bude funkce zadána v disjunktní normální formě (součet součinů, dále jen DNF), můžeme využít toho, že ROBDD reprezentuje logickou funkci zadanou v DNF a rozdělit tvorbu ROBDD na vytvoření dílčích ROBDD a jejich následné spojování. V praxi se často stává, že jednotlivé termy obsahují jen malou podmnožinu z celkového počtu proměnných logické funkce. Proto budou dílčí ROBDD relativně malé a pokud najdeme efektivní operaci na jejich spojování, velikost výsledného ROBDD



Obrázek 2.4: Příklad redukce OBDD na ROBDD

bude menší. Taková operace samozřejmě existuje a jmenuje se APPLY.

Jak vypadá použití operace APPLY v praxi, si ukážeme na následujícím příkladu. Mějme funkci $h = \overline{x_1 \cdot x_3} + x_2 \cdot x_3$, pro kterou chceme vytvořit odpovídající ROBDD. Vytvoříme tedy 2 ROBDD pro funkce $f = \overline{x_1 \cdot x_3}$ a $g = x_2 \cdot x_3$ a poté je spojíme do jediného výsledného ROBDD operací APPLY. Vytvoříme tedy ROBDD pro funkci $h = f + g$. Důležitou vlastností tohoto algoritmu je, že zachovává uspořádání proměnných, tzn. pro 2 ROBDD s uspořádáním $x_i < x_j$ platí, že výsledný ROBDD bude zachovávat stejné uspořádání. Na obrázku 2.5 jsou původní ROBDD pro funkce f (A) a g (B). Na obrázku 2.6 (A) je vidět OBDD po operaci APPLY, na 2.6 (B) je výsledný ROBDD po redukci.

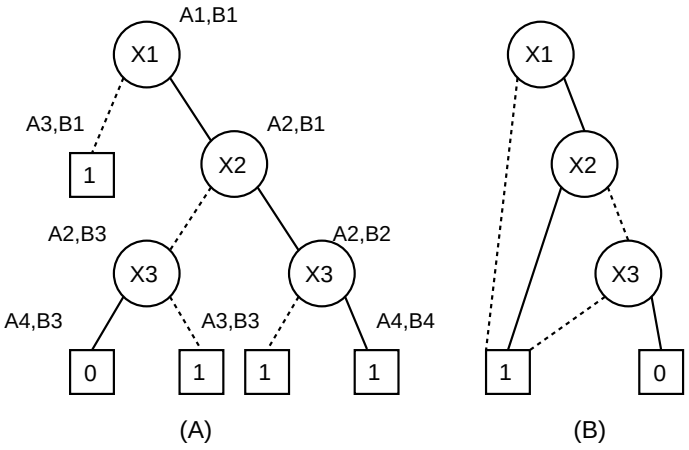


Obrázek 2.5: ROBDD pro funkce $f = \overline{x_1 \cdot x_3}$ a $g = x_2 \cdot x_3$ před operací APPLY

Základní myšlenka algoritmu spočívá v následující rovnosti:

$$f_1 < op > f_2 = x_i \cdot (f_1 | x_i = 0 < op > f_2 | x_i = 0) + \bar{x}_i \cdot (f_1 | x_i = 1 < op > f_2 | x_i = 1)$$

Ta nám říká, že Shannonův teorém o rozkladu lze rekurzivně použít i na binární operace s BDD. Algoritmus začne svůj průběh v kořenech obou ROBDD, kde uzel v prvním grafu označíme v_1 a uzel z druhého grafu označíme v_2 a postupuje dolů k listům. Mezitím vytváří uzly ve výsledném grafu vždy, když narazí na následující kombinace uzlů v_1 a v_2:



Obrázek 2.6: Výsledek operace APPLY

- v_1 a v_2 jsou oba terminální uzly - výsledný graf se skládá z uzlu, jehož hodnota je $hodnota(v_1) < op > hodnota(v_2)$
- v_1 a v_2 jsou neterminály a mají stejný index (reprezentují stejnou proměnnou) - ve výsledném grafu vytvoříme uzel u a rekurzivně spustíme algoritmus na $low(v_1)$ (potomek uzlu v_1 přes hranu určující hodnotu proměnné "0"; opakem je $high(v_1)$) a $low(v_2)$; tímto vznikne podgraf, jehož kořenem bude uzel $low(u)$; to samé opakujeme pro $high(v_1)$ a $high(v_2)$
- $index(v_2) > index(v_1)$ nebo je v_2 terminál - toto znamená, že funkce reprezentovaná grafem s kořenem v uzlu v_2 nezávisí na proměnné s indexem $index(v_1)$; v tomto případě vytvoříme ve výsledném grafu uzel u s indexem $index(v_1)$, ale rekurzivně spustíme algoritmus na uzlech $low(v_1)$ a v_2, abychom vygenerovali podgraf, jehož kořenem se stane $low(u)$ a poté na $high(v_1)$ a v_2, abychom vygenerovali podgraf, jehož kořenem se stane $high(u)$; to samé se děje, pokud je vztah mezi uzly obrácený.

Obecně lze říct, a je to vidět i na obrázku 2.6, že BDD vytvořený tímto postupem nebude redukovaný. Je třeba ho tedy před vrácením nového BDD redukovat. Kdybychom aplikovali tento postup přímo, dostali bychom se na algoritmus s exponenciální složitostí, protože každé zavolání algoritmu pro uzly, z nichž alespoň jeden je neterminál generuje další dvě rekurzivní volání. Abychom se tomu vyhnuli, uděláme dvě vylepšení.

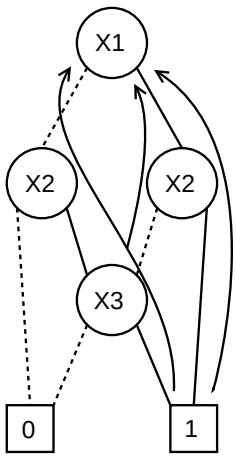
Za prvé, algoritmus nemusí vyhodnocovat stejnou dvojici uzlů vícekrát. Namísto toho si můžeme vytvořit tabulku uzlů skládající se z prvků ve tvaru (v_1, v_2, u), která nám říká, že výsledný graf pro uzly v_1 a v_2 je graf s kořenem v u. Poté, před zavoláním algoritmu na danou dvojici uzlů, zkontrolujeme, zda už se tato dvojice nevyskytuje v tabulce. Pokud ano, máme vyhráno a můžeme ihned vrátit výsledek. Pokud ne, algoritmus výsledek spočítá a uloží jej do tabulky. Tato úprava redukuje složitost algoritmu na $O(|G_1| \cdot |G_2|)$, kde $|G_1|$ je počet uzlů prvního ROBDD a $|G_2|$ je počet uzlů druhého ROBDD. V praxi se experimentálně dokázalo, že poměr úspěšných pokusů při hledání výsledku v tabulce se pohybuje mezi 40-50%.

Druhé vylepšení vychází z dominance některých logických hodnot v určitých případech. Když víme, že jedna z hodnot logického součinu je "0", už nás nemusí zajímat, jaká je hodnota druhého operandu; výsledek bude totiž vždy "0". Když tuto myšlenku přeneseme do našeho algoritmu, dojdeme k následujícímu výsledku. Pokud je v_1 neterminál a uzel v_2 terminál s

takto dominující hodnotou, můžeme vytvořit ve výsledném grafu uzel s hodnotou, kterou tento terminál implikuje, a vyhodnocování podgrafu s kořenem v uzlu v_1 můžeme s klidným svědomím zastavit. V praxi se ukázalo, že poměr takových případů je zhruba 10%.

2.2 Generování pravdivostní tabulky

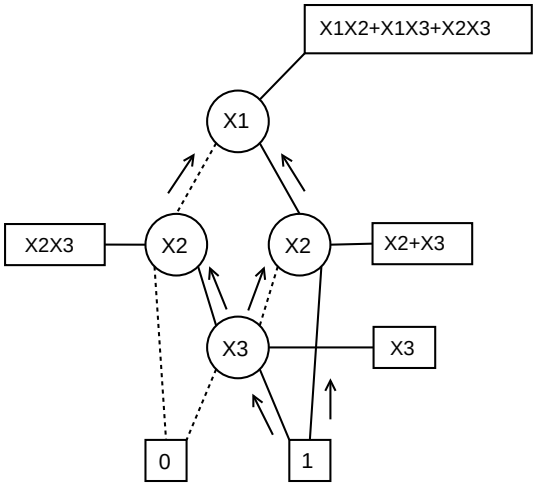
Máme tedy vytvořený ROBDD pro nějakou funkci. Jak ale vygenerovat její pravdivostní tabulku? V zásadě existují 2 způsoby, jak toho dosáhnout. První spočívá v průchodu grafem shora od kořene, kdy postupně, na základě logických hodnot v Shannonově rozkladu, generujeme term. Na počátku inicializujeme všechny hodnoty proměnných v termu na *don't care*. Poté začneme s průchodem ROBDD; procházíme graf do hloubky a při každém rekurzivním sestupu aktualizujeme term. Když sestupujeme po hraně označující rozklad pro hodnotu "0" dané proměnné, vložíme do termu na místo odpovídající aktuálně procházené proměnné "0" a pokračujeme dále. Analogicky toto provádíme s hodnotou "1". Když narazíme na terminál, zkontrolujeme, jestli je jeho hodnota "1" a pokud ano, úspěšně jsme vygenerovali jeden term, tzn. máme hotový jeden řádek pravdivostní tabulky. Pokud je hodnota terminálu "0", vracíme se o krok zpět, protože generujeme funkci zadanou jako on-set, tedy pouze prostřednictvím termů implikujících hodnotu výstupní funkce "1". Pokaždé když se rekurzivně vracíme zpět, aktualizujeme odpovídající místo v termu opět na *don't care*. Když takto projdeme celý graf, máme vygenerovány všechny on-set termy, které se v daném ROBDD vyskytují. Tento postup je velmi jednoduchý a intuitivní. Bohužel má také své nevýhody. Abychom totiž vygenerovali všechny termy, jejichž funkční hodnota je "1", musíme projít úplně celý graf, tudíž je tato metoda časově poměrně náročná. Příklad ROBDD a jemu odpovídající pravdivostní tabulky ja na obrázku 2.9 resp. v tabulce 2.1.



Obrázek 2.7: Generování termů průchodem grafu od listu ke kořeni

Druhou metodou je generování termů průchodem grafu od listů ke kořeni. Vybereme list s hodnotou "1" (generujeme on-set zadání tabulky) a postupujeme směrem ke kořeni. V každém uzlu aktualizujeme generovaný term a pokračujeme dál do rodičovských uzlů. Tento přístup je výhodnější v tom, že uvažuje pouze termy, vedoucí na hodnotu logické funkce "1". To znamená značnou časovou úsporu. Jak vypadá průchod grafem tímto způsobem, ilustruje obrázek 2.7. Vidíme, že tato metoda průchodu vygeneruje pouze 3 termy (11-, 101, 011), což jsou přesně ty termy, které implikují na výstupu hodnotu "1". Metoda se dá ještě vylepšit tím, že nebudeme graf procházet do hloubky a generovat termy po jednom, ale budeme jej procházet do šířky

a v každém uzlu vyhodnotíme celou logickou funkci, kterou tento uzel reprezentuje. Přitom je ovšem potřeba v každém uzlu provést minimalizaci. Tento přístup ukazuje obrázek 2.8. Výhodou je, že pokud najdeme efektivní způsob minimalizace, můžeme se v kořeni dostat až k minimálnímu počtu termů. Problém je ona zmíněná minimalizace, jejíž složitost může být značně vysoká. Tento způsob generování termů není ještě dostatečně prozkoumán. Další problém vězí v orientaci grafu, která je opačná, tedy od kořene k listům. Abychom mohli využít výhody, kterou poskytuje tento přístup, museli bychom si do uzlů ROBDD ukládat nejen informaci o potomcích, ale i o předchůdcích. To je velmi paměťově náročné, protože předchůdců může být velmi mnoho. V použitém balíku CUDD pro tento přístup podpora není, proto se budeme muset spolehnout na první způsob průchodu od kořene k listům.



Obrázek 2.8: Generování termů průchodem grafu od listu ke kořeni s aplikováním minimalizace

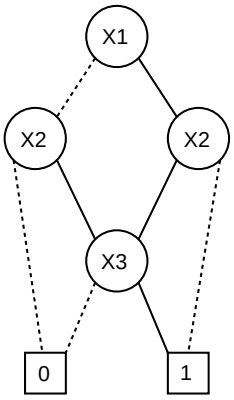
2.3 Minimalizace pravdivostní tabulky

Jak již bylo řečeno, výstupem programu bude pravdivostní tabulka popisující chování obvodu. Tato tabulka bude navíc, jak již bylo řečeno, zadána on-set, tedy budou v ní zobrazeny pouze termy, pro které je hodnota logické funkce "1". Termy, pro které je hodnota logické funkce "0", získáme jako komplement on-set zadané funkce. Opakem tohoto přístupu je off-set, kdy je funkce zadána pomocí nulových termů, tj. termů, pro které je hodnota logické funkce "0". Další možností je funkce zadaná jak on-set, tak off-set.

Tabulka 2.1: Pravdivostní tabulka generovaná ROBDD na obrázku 2.9

x_1	x_2	x_3	f
0	0	-	0
1	1	0	0
0	1	0	0
1	0	-	1
0	1	1	1
1	1	1	1

Tabulku získáme průchodem ROBDD a postupným generováním termů, tak jak jsme popsali



Obrázek 2.9: Příklad ROBDD generující neminimalizovanou pravdivostní tabulku

v minulé kapitole. Na tabulku můžeme nahlížet jako na jinou reprezentaci logické funkce, kde řádky s jedničkovým ohodnocením odpovídají jednotlivým termům tvořící logickou funkci zadanou v DNF. Pokud je u dané proměnné hodnota "1", vyskytuje se v daném termu v přímé formě, pokud "0", vyskytuje se v negované formě a pokud je hodnota proměnné "-", daná proměnná se v termu nevyskytuje. Symbol "-" značí tzv. *don't care*, což je symbol reprezentující neurčený stav proměnné. Vlastnosti *don't care* se dají popsat také tak, že na hodnotě proměnné označené jako *don't care* nezáleží. Ať danou proměnnou ohodnotíme jakkoli, hodnota výstupní logické funkce se nezmění. Naším cílem je, aby výsledný počet termů byl co nejnižší možný a jelikož obecně nelze zajistit, aby při průchodu ROBDD vznikala přímo minimalizovaná pravdivostní tabulka, tedy tabulka s minimálním počtem termů, je potřeba výstup ještě minimalizovat. Lépe je to vidět na příkladu; na obrázku 2.9 je jednoduchý příklad ROBDD, který může při řešení konkrétního obvodu vzniknout. Je vidět, že pravdivostní tabulka generovaná tímto ROBDD vypadá tak, jak ji zachycuje tabulka 2.1. Nás zajímají pouze řádky, ve kterých je hodnota logické funkce "1", z nichž poslední 2 řádky jsou viditelně slučitelné. Liší se totiž pouze v hodnotě první proměnné, takže termy s ohodnocením 011 a 111 lze sloučit do jediného termu s ohodnocením -11.

Na tomto jednoduchém případě je tedy vidět, že v praxi je třeba zabývat se problémem minimalizace pravdivostní tabulky. Tím vyvstává další dílčí problém na cestě k řešení, tj. navrhnout efektivní algoritmus pro minimalizaci pravdivostní tabulky získané průchodem námi vytvořeného ROBDD. Jelikož je tento problém známý a existuje několik nástrojů, které ho různými způsoby řeší, můžeme následně porovnat tyto různé přístupy s naším.

3 Návrh řešení

3.1 Volba programovacího jazyka a struktura řešení

Vzhledem k použití balíku CUDD, který je napsán v jazyku C s možností využít objektové rozhraní jazyka C++, jsem měl na výběr z jazyků C a C++. Z důvodu bezpečnosti a také díky standardní knihovně STL dostupné pro jazyk C++, byl pro implementaci zvolen jazyk C++. Vzhledem k možné budoucí integraci implementace do výukového systému EDA vyvíjeného na katedře počítačů, bylo třeba dekomponovat problém na několik modulů. Proto byl problém rozdělen na následující podproblémy:

- načtení obvodu ze vstupního souboru
- tvorba potřebných ROBDD
- minimalizace ROBDD
- generování pravdivostní tabulky
- minimalizace počtu řádků v pravdivostní tabulce
- výpis pravdivostní tabulky do výstupního souboru

První modul se bude starat o načtení obvodu ze souboru, druhý o tvorbu ROBDD, jejich minimalizaci a generování pravdivostní tabulky, třetí o minimalizaci pravdivostní tabulky a čtvrtý o export výsledné minimalizované pravdivostní tabulky do souboru. V následujících podkapitolách budou popsány základní principy řešení jednotlivých modulů. Výjimku bude tvořit pouze algoritmus pro minimalizaci pravdivostní tabulky, který bude, díky své rozsáhlosti a významu vysvětlen v samostatné kapitole. Minimalizační algoritmus neměl být původně součástí řešení, nicméně při snaze snížit počet termů generovaných z vytvořených ROBDD se mi podařilo přijít s novou myšlenkou na řešení tohoto problému. Výhodou algoritmu je i to, že není závislý na ostatních částech programu, je proto možné jej použít jako samostatný produkt.

3.2 Použité formáty

Před začátkem popisu návrhu jednotlivých modulů je třeba ještě popsat vstupní a výstupní formáty, které budou použity. Jako vstupní formát bude použit formát bench a jako výstupní formát bude použit formát PLA Espresso.

3.2.1 Formát BENCH

V souboru ve formátu bench se mohou vyskytnout následující konstrukce:

- # komentář - znak # uvozuje komentář; končí znakem konec řádky
- INPUT(x_1) - definuje vstupní proměnnou jménem x_1
- OUTPUT(y_1) - definuje výstupní proměnnou jménem y_1
- y_1 = NOT(x_1) - definuje vztah mezi proměnnými (hradlo v logickém obvodu); obecně lze mít ve vztahu více vstupních proměnných s více výstupními proměnnými; v našem případě budeme pracovat pouze s obvody, ve kterých je v tomto vztahu vždy pouze jedna výstupní proměnná

Jak je vidět, formát bench je velice intuitivní a jednoduchý. V podstatě kopíruje intuitivní chápání logického obvodu, jakožto souhrnu logických prvků; zde logické prvky realizují jednotlivé vztahy. Množina vztahů, se kterými budeme pracovat, je následující:

- AND - logický součin; může mít více vstupních proměnných
- NAND - negovaný logický součin; může mít více vstupních proměnných
- OR - logický součet; může mít více vstupních proměnných
- NOR - negovaný logický součet; může mít více vstupních proměnných
- XOR - nonekvivalence; jen dvě vstupní proměnné
- XNOR - ekvivalence; jen dvě vstupní proměnné
- BUF - identita; jen jedna vstupní proměnná
- NOT - negace; jen jedna vstupní proměnná

Nyní si ukážeme jednoduchý příklad. Jedná se o obvod c17.bench ze standardní sady testovacích obvodů [5].

```
# c17 iscas example (to test conversion program only)
# -----------------------------------------------
#
#
# total number of lines in the netlist ..............  17
# simplistically reduced equivalent fault set size = 22
# lines from primary input gates .......  5
# lines from primary output gates .......  2
# lines from interior gate outputs ......  4
# lines from ** 3 ** fanout stems ...  6
#
INPUT(G1gat)
INPUT(G2gat)
INPUT(G3gat)
INPUT(G6gat)
INPUT(G7gat)
OUTPUT(G22gat)
OUTPUT(G23gat)

G10gat = nand(G1gat, G3gat)
G11gat = nand(G3gat, G6gat)
G16gat = nand(G2gat, G11gat)
G19gat = nand(G11gat, G7gat)
G22gat = nand(G10gat, G16gat)
G23gat = nand(G16gat, G19gat)
```

Na tomto příkladu vidíme popis jednoduchého obvodu s pěti vstupy a dvěmi výstupy. Je také vidět, že je zde použito jen dvouvstupové hradlo NAND. Identifikátory proměnných mohou obsahovat nejen alfanumerické znaky, ale i další znaky jako například ".". Jelikož není formát bench nijak standardizován, je na nás a naší implementaci, co všechno dokážeme korektně načíst. S tím souvisí jedna velmi důležitá vlastnost, která poměrně výrazně ovlivnila

implemenetaci. U tohoto formátu totiž není definováno, zda může být nějaká proměnná použita dříve než je "definována", tzn. dříve než je definována jako vstupní proměnná celého obvodu nebo jako výstupní proměnná nějakého hradla. Tento zápis je naprosto korektní:

```
INPUT(X1)
INPUT(X2)
OUTPUT(Y)

Y = nand(X3, X4)
X3 = nand(X1, X2)
X4 = xor(X1, X2)
```

Vidíme, že proměnné X3 a X4 byly použity jako vstupní proměnné pro výpočet výstupní proměnné Y dříve, než byl definován jejich vztah s vstupními proměnnými. Je proto zřejmé, že bude nutno vytvořit nějakou datovou strukturu pro načtení souboru, než začneme s tvořením ROBDD. Jinak bychom se dostali do problémů hned při prvním výskytu "nedefinované" proměnné. Jak se tento problém podařilo vyřešit bude popsáno v kapitole 3.3.

3.2.2 Formát PLA

Na začátku souboru se vyskytují klíčová slova určující strukturu a vlastnosti logické funkce, kterou daný soubor reprezentuje. Zde je seznam nejdůležitějších z nich.

- *.model jméno* - definuje jméno popisovaného obvodu
- *.i počet* - číslo *počet* označuje počet vstupních proměnných
- *.o počet* - číslo *počet* označuje počet výstupních proměnných
- *.ilb* $i_1, i_2, i_3, ..., i_n$ - označuje seznam jmen vstupních proměnných
- *.ob* $o_1, o_2, o_3, ..., o_n$ - označuje seznam jmen výstupních proměnných
- *.type typ* - určuje typ zadání logické funkce; nabývá hodnot *f, r, fd, fr, dr* nebo *fdr*
- *.p počet* - číslo *počet* označuje celkový počet termů
- *.end* - označuje konec souboru

Po definici těchto základních vlastností popisované funkce následují již samotné termy v součinovém tvaru. První část řádku je popsána vektorem ohodnocení jednotlivých proměnných v termu, kde proměnné mohou nabývat hodnot "0","1" a "-" (*don't care*). Ohodnocení "1" znamená, že se daná proměnná v daném termu vyskytuje v přímé formě, ohodnocení "0" znamená, že se vyskytuje v negované formě a ohodnocení "-" znamená, že se daná proměnná v daném termu nevyskytuje. Na druhé části řádku, po oddělení oddělovačem, následuje vektor ohodnocení výstupních funkcí. Jelikož generujeme funkce zadané on-set, význam ohodnocení výstupních funkcí ve formátu PLA je následující. Ohodnocení "1" značí hodnotu logické funkce "1" a ohodnocení "0" znamená, že daná výstupní proměnná nemá pro danou výstupní funkci význam.

Jak takový soubor ve formátu PLA vypadá, si ukážeme na jednoduchém příkladu.

```
.model example
.i 7
```

```
.o 2
.ilb f b c d a h g
.ob f0 f1
.p 9
.type fd
-1--1-- 10
1-11--- 10
-001--- 10
01---1- 10
-0--0-- 01
1---0-- 01
0-----0 01
01--1-- 01
10-0--- 01
.e
```

Tento soubor tedy popisuje obvod jménem example se sedmi vstupními a dvěmi výstupními proměnnými. Celkový počet termů je devět a typ zadání funkce je fd. U typu se ještě na chvíli zastavím. Jak již bylo výše popsáno, parametr *.type* může nabývat hodnot *f* , *r*, *fd*, *fr*, *dr* nebo *fdr*. Nejpoužívanější hodnoty jsou následující:

- fd - základní typ; je nastaven, pokud není určeno jinak; určuje takové zadání funkce, kde termy s hodnotou výstupní proměnné "1" patří do on-set dané funkce, termy s hodnotou výstupní proměnné "0" nemají pro výstupní funkci význam a termy s hodnotou "-" patří do dc-set dané funkce
- f - určuje takové zadání funkce, kde termy s hodnotou výstupní proměnné "1" patří do on-set dané funkce, termy s hodnotami "0" nebo "-" nemají pro danou výstupní funkci význam
- fr - určuje takové zadání funkce, kde termy s hodnotou výstupní proměnné "1" patří do on-set dané funkce, termy s hodnotou výstupí proměnné "0" patří do off-set dané funkce a termy s hodnotou výstupní proměnné "-" nemají pro danou funkci význam
- fdr - určuje takové zadání funkce, kde termy s hodnotou výstupní funkce "1" patří do on-set dané funkce, termy s hodnotou výstupní proměnné "0" patří do off-set dané funkce a termy s hodnotou výstupní proměnné "-" patří do dc-set

3.3 Načtení vstupního souboru

V kapitole 3.2.1 jsem popsal formát bench, který bude použit jako vstupní formát. V této kapitole popíši jak tento formát načíst do mnou navržené vnitřní struktury.
Na načtení vstupního souboru ve formátu bench jsem použil lexikální analyzátor z diplomové práce Vlastimila Kozáka [10]. Z důvodu zmíněného v kapitole 3.2.1 bylo nutné před vlastní tvorbou ROBDD uložit strukturu obvodu do nějaké dočasné datové struktury. Rozeberu nejdříve požadavky, které jsou na tuto strukturu kladeny. Musí umět reprezentovat tyto tři druhy vstupních prvků:

- vstupní proměnná
- výstupní proměnná

- logický prvek (hradlo) - tento prvek se navíc dělí na typy AND, NAND, OR, NOR, XOR, XNOR, BUF a NOT

Je zřejmé, že základní vlastností každého prvku, ať už vstupní proměnné, výstupní proměnné nebo hradla, je nějaké pojmenování, podle kterého později při tvorbě vlastních ROBDD poznáme, o který prvek jde. Pro vstupní a výstupní proměnnou je to, spolu s typem prvku, jediné, co nás zajímá. Pro hradlo ovšem potřebujeme znát ještě jeho vstupní proměnné. Proto potřebujeme ještě seznam, do kterého uložíme jména všech vstupních proměnných daného hradla.

Jak tedy probíhá vlastní načítání vstupního souboru? Pomocí lexikálního analyzátoru čteme soubor a překládáme ho na lexikální elementy. Podle nich poznáme, jaký typ vstupního prvku byl načten. Pokud byla načtena vstupní proměnná obvodu, uložíme ji do seznamu vstupních prvků (v případě vstupní proměnné nastavíme pouze typ a jméno). Pokud byla načtena výstupní proměnná obvodu, uděláme totéž, pouze nastavíme typ prvku v seznamu na typ odpovídající výstupní proměnné. Pokud bylo načteno hradlo, uložíme ji taktéž do seznamu vstupních prvků, nicméně k jménu a typu musíme ještě přidat seznam vstupních proměnných daného hradla. Pokud tedy bude v souboru např. konstrukce `Y = nand(X3, X4)`, vytvoříme prvek v seznamu vstupních prvků, kterému nastavíme jméno na `Y`, typ na `T_NAND` a do seznamu vstupních proměnných uložíme jména `X3` a `X4`. Po dokončení průchodu celého souboru budeme mít vytvořený seznam s přesným popisem všech prvků. Poté můžeme přistoupit k vlastní tvorbě ROBDD.

3.4 Tvorba ROBDD

Nyní je třeba popsat, jak ze seznamu vstupních prvků vytvořit vlastní ROBDD. Základní postup je takový, že budeme procházet seznam vstupních prvků od začátku do konce a podle toho, na jaký prvek narazíme, rozhodneme o další činnosti. Máme následující tři možnosti:

- narazili jsme na prvek typu vstupní proměnná (`T_INPUT`) - jelikož je zřejmé, že žádný ROBDD pro tento prvek nemůže ještě existovat, vytvoříme elementární ROBDD pro jednu proměnnou a uložíme ho do seznamu, do kterého budeme ukládat všechny vytvořené ROBDD; tento ROBDD budeme dále používat pokaždé, když se daná proměnná bude vyskytovat na vstupu nějakého hradla
- narazili jsme na prvek typu výstupní proměnná - přeskočíme ho a neděláme nic; zatím pro nás nemá žádný význam
- narazili jsme na prvek typu hradlo - tato možnost je nejzajímavější, protože v sobě skýtá mnoho záludností; proto bude popsána podrobněji v samostané podkapitole

3.4.1 Rekurzivní tvorba ROBDD

Nyní jsme v situaci, kdy jsme při průchodu seznamem vstupních prvků narazili na prvek typu hradlo. Abychom věděli, kterou z dostupných funkcí balíku CUDD implementující funkci APPLY pro danou operaci máme použít, musíme určit typ hradla. Ten však máme v prvku uložen, takže můžeme bez problémů pokračovat. Poté projdeme seznam vstupních proměnných hradla a zkontrolujeme, zda všechny prvky v tomto poli mají ve spojovém seznamu, kam ukládáme vytvořené ROBDD, svůj odpovídající prvek. Pokud ano, pomocí implementace operace APPLY vytvoříme nový ROBDD a vložíme jej do seznamu. Předtím musíme ještě vyřešit problém s počtem vstupních proměnných hradla. Pokud chceme vytvořit např. nový ROBDD reprezentující třívstupové hradlo typu NAND, můžeme udělat následující jednoduchou úpravu:

$$f = \overline{x_1 \cdot x_2 \cdot x_3} = \bar{x}_1 + \bar{x}_2 + \bar{x}_3$$

Problém je v tom, že operace APPLY má vždy pouze dva operandy, takže v případě více než dvouvstupových hradel s negovaným výstupem si musíme vybrat ze dvou možností tvorby ROBDD. Buď budeme postupovat tak, že sekvencí operací APPLY daného typu, avšak bez negace (pokud se jedná o hradlo NAND, použijeme APPLY s operací AND) vytvoříme nový ROBDD a ten poté znegujeme nebo využijeme rovnosti uvedené výše (De-Morganův zákon) a vytvoříme ROBDD za její pomoci. Z experimentů vyplynulo, že i když jsou obě metody srovnatelné, metoda využívající De-Morganových zákonů je mírně úspornější ve smyslu počtu uzlů ve výsledném ROBDD. Mírně výhodnější je tedy použití De-Morganových zákonů.

Takto tedy postupujeme, pokud mají všechny vstupní proměnné daného hradla vytvořen již svůj ROBDD. Pokud ale některá z proměnných toto nesplňuje, musíme pro ni ROBDD nejdříve vytvořit. To uděláme tak, že najdeme výskyt dané proměnné v seznamu vstupních prvků a pokusíme se pro ni vytvořit ROBDD (budeme hledat její výskyt na levé straně přiřazovacího výrazu; např. hledáme výskyt proměnné x_5, proto budeme hledat výraz např. v tomto tvaru: $x_5 = \mathrm{NAND}(x_1,\ x_2)$). Píši, že se pokusíme, protože může nastat stejná situace, která nás dovedla sem, tj. opět nebudou vytvořeny ROBDD pro všechny vstupní proměnné daného hradla. Pokud toto nastane, provedeme rekurzivní sestup pro proměnnou bez odpovídajícího ROBDD. Je zřejmé, že toto se může opakovat nejvýše do té doby, než narazíme na hradlo, které má jako vstupní proměnné vstupní proměnné celého obvodu. Elementární ROBDD pro tyto vstupní proměnné jsou totiž bezpochyby vytvořeny. Poté se vracíme rekurzivně zpět a po cestě tvoříme všechny potřebné dílčí ROBDD, až se dostaneme k výrazu, z kterého jsme vyšli. Jelikož již máme všechny ROBDD, které jsou potřebné, vytvořeny, můžeme pokračovat dále a vytvořit nový ROBDD odpovídající dané proměnné. Takto postupujeme do té doby, dokud neprojdeme celý seznam vstupních prvků.

Poslední velice důležitá část tohoto algoritmu je fakt, že na začátku každé iterace průchodu seznamu vstupních prvků zkontrolujeme, zda již není v seznamu dílčích ROBDD vytvořen ROBDD pro proměnnou aktuální v dané iteraci (proměnnou, pro kterou chceme vytvořit ROBDD). Pokud nenastane výše popsaná situace s nutností rekurzivního sestupu, není to třeba, protože ROBDD se budou tvořit v "přirozeném" pořadí, tedy vždy když budeme chtít vytvořit nový ROBDD, budeme mít k dispozici dílčí ROBDD odpovídající vstupním proměnným daného hradla. Jak jsem popsal dříve, toto závisí pouze na struktuře vstupního souboru. Obecně ale nelze říci, zda se bude v daném souboru tato struktura vyskytovat, a proto je nutné tyto případy ošetřit.

Na konci této části algoritmu, jsme tedy v situaci, kdy máme vytvořen seznam všech doposud vytvořených ROBDD, tedy i těch dočasných, které nereprezentují výstupní proměnné. Ty ale pro generování pravdivostní tabulky nepotřebujeme (potřebovali jsme je jen pro rychlejší tvrobu ROBDD), můžeme se jich proto zbavit. Projdeme tedy celý seznam a všechny prvky, které nereprezentují výstupní proměnnou obvodu, smažeme. Takto tedy dostaneme seznam všech ROBDD pro výstupní funkce. Z těch potom generujeme výstupní pravdivostní tabulku.

3.5 Snížení velikosti ROBDD pomocí změny pořadí proměnných

Po dokončení úkonů popsaných v minulé kapitole, vytvoření všech ROBDD pro výstupní funkce, můžeme přistoupit k dalšímu kroku na cestě k řešení. Zkusíme co nejvíce snížit počet uzlů ve všech ROBDD. Balík CUDD poskytuje několik algoritmů pro minimalizaci počtu uzlů v ROBDD. Stručně popíši jejich principy.

3.5.1 Náhodný algoritmus

Algoritmus využívající náhodného výběru se v balíku CUDD vyskutuje ve dvou variantách: s tzv. pivotem a bez pivota. Varianta bez pivota vybere náhodně dvě proměnné a prohodí jejich pořadí. Prohození pořadí vybraných proměnných probíhá prostřednictvím prohazování sousedních proměnných. Po každé iteraci vyhodnotí algoritmus aktuální velikost grafu a pořadí s nejnižším počtem uzlů ponechá jako vstup pro další iteraci. Počet dvojic, které se takto vyberou, je roven počtu promněnných.
Varianta s pivotem dělá to samé jako varianta bez pivota s tím rozdílem, že je vybrán tzv. pivot, což je proměnná s největším počtem uzlů. Při výběru dvojice proměnných k prohození se vybere vždy jedna proměnná s pozicí v grafu nad pivotem a druhá pod pivotem.

3.5.2 Sifting algoritmus

Algoritmus navržený R. Rudellem v práci [15]. Jeho myšlenka spočívá ve spočítání nejvhodnější pozice v grafu pro jednu konkrétní proměnnou za předpokladu, že ostatní proměnné zůstanou na svých místech. Daná proměnná se nejdříve postupným prohazováním se sousedními proměnnými přesouvá grafem směrem dolů k listům. Během tohoto procesu si poznamenáváme aktuální počet uzlů s touto proměnnou a nejlepší prozatímní výsledek si pamatujeme. Když dojdeme na předposlední pozici (na poslední jsou terminální uzly), postupujeme stejným způsobem zpět nahoru. Když dorazíme do kořene, známe nejvhodnější pozici pro tuto konkrétní proměnnou, a proto ji opět postupným přehazováním se sousedními proměnnými přesuneme na tuto pozici s nejnižším počtem uzlů. Toto provedeme pro všechny proměnné. Algoritmus je v balíku CUDD implementován v základní i konvergentní verzi. Konvergentní verze spouští základní algoritmus do té doby, dokud přináší nějaké zlepšení.

3.5.3 Symmetric sifting algoritmus

Algoritmus publikovaný v práci [14], jehož autory jsou S. Panda, F. Somenzi a B.F. Plessier. Je založen na siftingu, pouze s jedním rozdílem. Při přehazováni sousedních proměnných testuje tyto proměnné na symetrii. Funkce je symetrická v proměnných x_i a x_j tehdy, když se po jejich prohození na funkci nic nezmění. Tato Shannonova definice vede k tvrzení, že symetrie platí i pro BDD; tedy když přehodíme pozici dvou symetrických proměnných, velikost BDD se nezmění. Algoritmus toto testuje a pokud narazí na dvojici proměnných, které splňují podmínku symetrie, seskupí je do skupiny a dále s nimi zachází jako s jednou. Jinak algoritmus pracuje stejně jako standradní sifting. Také tento algoritmus je v balíku CUDD implementován v základní i konvergentní verzi.

3.5.4 Group sifting algoritmus

Algoritmus publikovaný v práci [13], jehož autory jsou S. Panda a F. Somenzi. Jde o rozšíření metody symmetric sifting o další kritéria seskupování proměnných do skupin. Jde o metody rozšířené symetrie a druhé diference. První z nich, rozšířená symetrie, spočívá v rozšíření kritérii pro detekci symetrie. Existují dva druhy symetrie: pozitivní a negativní. Řekneme, že funkce g je pozitivně symetrická v proměnných x_i a x_j, právě tehdy, když platí rovnost $g_{x_i x'_j} = g_{x'_i x_j}$. Druhý případ, negativní symetrie, nastává tehdy, když pro funkci g platí rovnost $g_{x_i x_j} = g_{x'_i x'_j}$. Rozšíření symetrie spočívá za prvé v tom, že dovolíme kombinace těchto dvou druhů symetrií a za druhé v tom, že povolíme určité procento porušení těchto pravidel. Dovolíme např. 10% uzlů, aby nesplnilo ani jedno kritérium symetrie.

Pro metodu druhé diference si nejprve zadefinujeme následující rovnost: $S(i) = \frac{N(i+1)}{N(i)} - \frac{N(i)}{N(i-1)}$, kde $N(i)$ označuje počet uzlů s indexem i. Pokud je tento výraz záporný, proměnné x_i a x_j

budou seskupeny.

3.5.5 Window permutation algoritmus

Algoritmus založený na permutaci oken, publikovaný v pracech [8] a [9].

3.5.6 Algoritmus využívající simulované ochlazování

Algoritmus velmi volně inspirovaný algoritmem publikovaným v práci [1]. Jako všechny implementace tohoto typu iterativních heuristik, je i tato potenciálně velmi pomalá.

3.5.7 Algoritmus využívající genetické algoritmy

Algoritmus inspirovaný prací [6]. Stejně jako v případě simulovaného ochlazování, je i tato metoda potenciálně velmi pomalá.

Úkolem je otestovat všechny popsané metody a rozhodnout, které z nich mají nejlepší vlastnosti, a budou proto dále používány. Všechny metody budou otestovány a zhodnoceny v kapitole 6.1.

3.6 Generování pravdivostní tabulky

Princip generování pravdivostní tabulky byl popsán v kapitole 2.2. Pro připomenutí uvedu, že jsou v zásadě dvě možnosti, jak tabulku generovat: průchodem ROBDD od kořene k listům nebo průchodem od listu ke kořeni. Jelikož v balíku CUDD není podpora pro rychlý průchod směrem od listu ke kořeni, používáme generování průchodem od kořene k listům. Procházíme tedy ROBDD shora od kořene do hloubky a udržujeme si řetězec, reprezentující generovaný term, do kterého ukládáme aktuální ohodnocení proměnných. Při průchodu každým uzlem si tedy načteme index proměnné, kterou uzel reprezentuje, na toto místo do řetězce vložíme "0" a rekurzivně sestoupíme do potomka pro hodnotu "0" (*low*). To samé uděláme pro hodnotu "1" (*high*). Když je celý podgraf vyhodnocen, vrátíme hodnotu proměnné v řetězci zpět na výchozí hodnotu "-". Pokud se nám podaří dojít až do listu s hodnotou "1", úspěšně jsme vygenerovali jeden term. Generované termy ukládáme do struktury pro minimalizaci. Jak tato struktura vypadá a jak pracuje vlastní algoritmus minimalizace pravdivostní tabulky bude popsáno v samostatné kapitole 4.

3.7 Výpis do výstupního souboru

Modul exportu do výstupního formátu již jen převezme strukturu od algoritmu pro minimalizaci pravdivostní tabulky a stejným způsobem jako byl popsán v kapitole 3.6 vygeneruje a vypíše termy do výstupního souboru. Zároveň k vlastním termům připíše statistické údaje o daném souboru (počet vstupním proměnných, počet výstupních proměnných a počet termů).

4 Minimalizace pravdivostní tabulky

V kapitole 2.3 jsem popsal problém minimalizace pravdivostní tabulky a proč má cenu snažit se o vytvoření efektivních algoritmů pro jeho řešení. Při řešení problému se mi podařilo navrhnout poměrně kvalitní a rychlý algoritmus pro minimalizaci logických funkcí. V této kapitole jej podrobněji popíši.

Jako vstup pro algoritmus máme neminimalizovanou pravdivostní tabulku. Příklad takové tabulky vidíme v tabulce 4.1. Základní myšlenka algoritmu je následující. Pokud tabulku seřadíme při nějakém zadefinovém uspořádání (např. "0" < "1" < "-") dostaneme tabulku, ve které jsou všechny termy, které se liší pouze v ohodnocení poslední proměnné pod sebou. Na příkladu tabulky 4.1 vidíme, že je již seřazená, takže vidíme celkem 4 dvojice kandidátů na sloučení. První podmínka ale je, že oba termy musí implikovat stejnou hodnotu logické funkce, což nesplňuje první a druhý term, kteří se sice liší pouze v ohodnocení poslední proměnné, ale implikují jinou hodnotu výstupní logické funkce. Tím nám zbývají 3 dvojice vhodné na sloučení. Všimněme si, že dvojic na sloučení je v tabulce více, nicméně ohodnocení proměnné, ve kterém se liší, se vyskytují jinde než na posledním místě. Teď můžeme přistoupit k samotnému slučování termů. Jelikož máme tabulku seřazenou, nemusíme porovnávat termy "každý s každým", ale můžeme je srovnávat jen se svými sousedy. Takže pokud zjistíme, že se dva sousední termy liší jen v ohodnocení poslední proměnné, sloučíme je do jednoho, ve kterém odpovídající ohodnocení proměnné nastavíme na "-".

Tabulka 4.1: Příklad neminimalizované pravdivostní tabulky pro funkci $f = x_1 + x_2 + x_3$

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabulku po první iteraci algoritmu zachycuje tabulka 4.2.

Tabulka 4.2: Pravdivostní tabulka po první iteraci minimalizačního algoritmu

x_1	x_2	x_3	f
0	0	1	1
0	1	-	1
1	0	-	1
1	1	-	1

Term, který implikoval hodnotu výstupní logické funkce "0" jsme vyškrtli, protože již není pro algoritmus důležitý. Navíc, jak již bylo řečeno, výstupem z převodu víceúrovňové logické sítě na dvouúrovňovou bude pravdivostní tabulka zadána on-set, tedy budou v ní obsaženy pouze termy implikující hodnotu výstupní logické funkce "1". V tabulce po první iteraci vidíme, že se nám sloučily 3 řádky, čímž se počet řádků v tabulce snížil na polovinu. Nyní musíme provést

rotaci celé tabulky, tzn. přeskupit tabulku tak, že celý první sloupec vložíme na poslední místo. Výsledek vidíme na tabulce 4.3.

Tabulka 4.3: Pravdivostní tabulka po rotaci

x_2	x_3	x_1	f
0	1	0	1
1	-	0	1
0	-	1	1
1	-	1	1

Nyní musíme tabulku znovu seřadit, nicméně jelikož je již částečně seřazená, není již seřazení tak časově náročné a je možné ho zvládnout v lineárním čase. Tabulku po opětovném seřazení zachycuje tabulka 4.4.

Tabulka 4.4: Pravdivostní tabulka po opětovném seřazení

x_2	x_3	x_1	f
0	1	0	1
0	-	1	1
1	-	0	1
1	-	1	1

Nyní jsme opět v situaci, kdy můžeme sloučit termy, lišící se pouze v ohodnocení posledních proměnných. Poté opět provedeme rotaci, seřazení a opět sloučení. Takto postupujeme minimálně N-krát, kde N značí počet proměnných. Maximální smysluplný počet iterací je N^2, což je počet, který je potřeba, abychom porovnali termy systémem každý s každým. V praxi se ukázalo, že hodnota počtu iterací, která je vhodným kompromisem mezi rychlostí a kvalitou řešení je rovna N. Zdůvodnění tohoto tvrzení si uvedeme v kapitole 6.2.

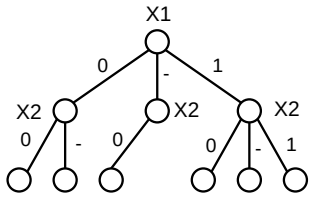
Výsledná tabulka po doběhu algoritmu je zobrazena v tabulce 4.5.

Tabulka 4.5: Výsledná pravdivostní tabulka

x_1	x_2	x_3	f
0	0	1	1
1	0	-	1
-	1	-	1

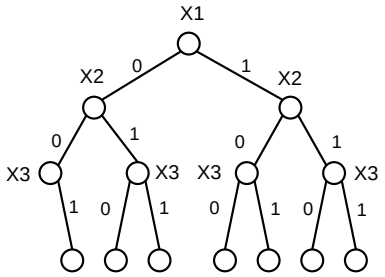
Jak vidíme, idea algoritmu je poměrně jednoduchá a výsledky stojí za další zkoumání. Předně by bylo třeba vyřešit problém řazení, který je zejména na počátku poměrně problematický, protože jeho složitost bude minimálně $O(N \cdot P \cdot logN)$, kde N je počet termů a P je počet vstupních proměnných (platí pro první seřazení tabulky, následné řazení už částečně seřazených termů bude mít lineární časovou složitost), navíc operace porovnání bude ještě záviset na počtu proměnných v termu. Proto by bylo vhodné navrhnout nějakou jinou strukturu, která by zmíněné problémy řešila. Řešením se ukázal být tzv. ternární strom (publikovaný v práci P. Fišera a J. Hlavičky [7]), jehož příklad vidíte na obrázku 4.1. Strom na obrázku obsahuje termy

00, 0-, -0, 10, 1- a 11. Tato struktura problémy s řazením řeší. Jaký je průběh algoritmu při využití této struktury si ukážeme na stejném příkladu jako pro verzi s dvourozměrným polem (obrázek ternárního stromu odpovídajícímu stejné funkci je na obrázku 4.2).



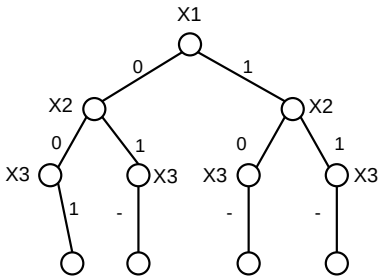
Obrázek 4.1: Příklad ternárního stromu

Jak generujeme termy z ROBDD, stavíme postupně ternární strom. Jak probíhá vkládání termu do stromu je zřejmé; procházíme term po proměnných, zároveň procházíme strom a pokud je třeba (pro dané ohodnocení proměnné ještě ve stromu neexistuje uzel), vytváříme nové uzly. První výhoda je hned vidět, mnoho uzlů bude společných pro několik termů. Tím snížíme paměťovou náročnost algoritmu. Druhá výhoda je také zřejmá, prohledáváním stromu do hloubky při definovaném pořadí uzlů generujeme seřazenou tabulku termů, takže odpadá problém řazení.



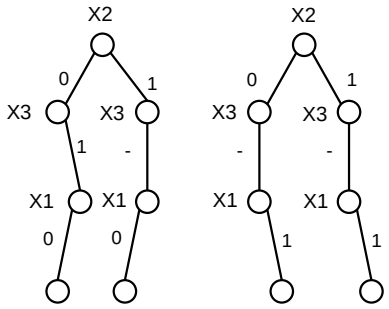
Obrázek 4.2: Ternární strom pro pravdivostní tabulku 4.1

Nyní k samotnému průběhu algoritmu. Máme tedy postavený ternární strom a chceme provést sloučení termů lišících se pouze na ohodnocení poslední proměnné. Je zřejmé, že takové termy mají ve stromu listy, které mají společného předchůdce. Zjistíme tedy, které uzly na poslední úrovni před listy mají více než jednoho potomka a provedeme sloučení, tzn. potomek z těchto uzlů bude jen jeden a bude ukazovat na "-". Po aplikaci tohoto pravidla, bude náš strom vypadat takto (obr. 4.3):



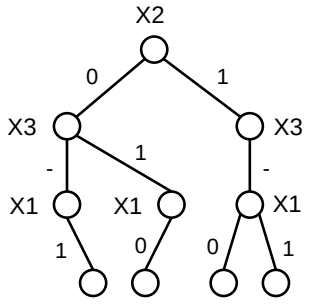
Obrázek 4.3: Ternární strom po sloučení listů se stejným předchůdcem

Teď musíme provést "rotaci prvního sloupce", jak jsem tento krok nazval při popisu postupu s dvourozměrným polem, což v případě stromu znamená "utrhnout" kořen od zbytku stromu a odpovídající hodnoty uložit na místa nových listů. Tímto krokem nám vzniknou maximálně 3 stromy (vznikne tolik stromů, kolik měl kořenový uzel potomků). Poté musíme vytvořit nové potomky všech listů ve všech stromech a to tak, že do stromu, jehož kořen byl potomkem původního "odtrženého" kořene přes hranu označující ohodnocení proměnné "0", vložíme nový uzel na místo "0". Analogicky toto provedeme pro ostatní stromy. Situaci po aplikování těchto dvou operací zachycuje obrázek 4.4.



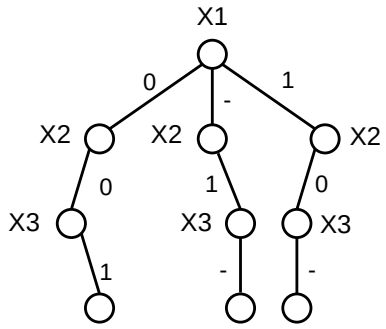
Obrázek 4.4: Ternární strom po odtržení kořene a vložení nových listů

Nyní máme před sebou 2 stromy a abychom mohli pokračovat dále, musíme tyto stromy opět sloučit do jednoho. Postup je jednoduchý. Nejprve si zvolíme jeden ze stromů jako cílový, tedy strom, do kterého budeme slučovat ostatní. Poté procházíme paralelně oba stromy a každý nalezený uzel, který se v cílovém stromu nevyskytuje, vložíme do cílového stromu. Tímto nám vznikne opět jeden strom, v našem příkladu vypadající tak, jak ho zachycuje obrázek 4.5.



Obrázek 4.5: Ternární strom po spojení obou stromů

Toto je konec jedné iterace algoritmu. Nyní budeme opakovat znovu kroky: sloučení listů, odtržení kořene, vložení nových listů a sloučení stromů. Po N opakováních se opět dostaneme do situace, kdy máme v kořeni stromu stejnou proměnnou jako na počátku. Výsledný strom zachycuje obrázek 4.6. Vidíme, že jsme dospěli ke stejnému výsledku, jako algoritmus využívající dvourozměrné pole. Podrobnou analýzu tohoto algoritmu uvedeme v kapitole 6.2.



Obrázek 4.6: Výsledný ternární strom po N iteracích

4.1 Složitost algoritmu

Nyní je již princip algoritmu znám, můžeme tedy přistoupit k určení jeho teoretické asymptotické časové složitosti. Algoritmus provádí nejprve vložení termů do stromu a poté P-krát, kde P je počet vstupních proměnných dané logické funkce, operace sloučení listů, rozdělení stromů, vložení nových listů a sloučení stromů. Operace vkládání termů do stromu trvá $N \cdot P$, kde N je počet termů. Operace sloučení listů, vložení nových listů a sloučení stromů všechny trvají v nejhorším případě opět $N \cdot P$. Rozdělení stromů má konstantní složitost. Výsledná složitost tedy vypadá následovně:

$$T(N, P) = N \cdot P + P \cdot (N \cdot P + N \cdot P + N \cdot P) = O(N \cdot P^2)$$

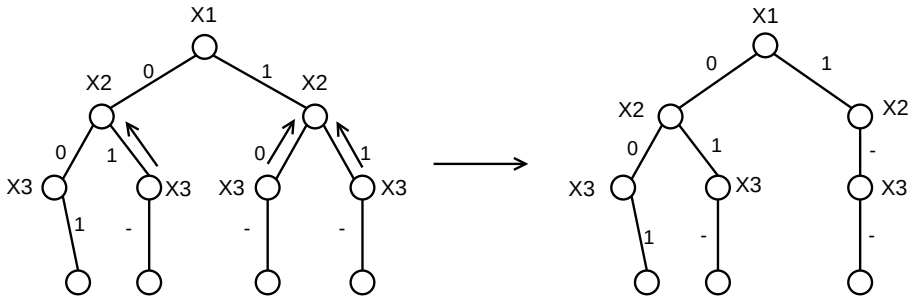
4.2 Možná vylepšení

4.2.1 Nalezení podmnožin - absorpce

Algoritmus se dá pomocí dvou vylepšení ještě zkvalitnit. První vylepšení spočívá v následující úvaze: máme např. dva termy o třech proměnných; ohodnocení proměnných v prvním termu je 010, v druhém termu 0–. Je vidět, že předvedeným algoritmem tyto dva termy nesloučíme. Liší se totiž ve více než jednom ohodnocení proměnné. Správně by se ale tyto termy sloučit měly, protože první term je podmnožinou druhého. Když expandujeme term 0– do všech termů, které reprezentuje, dostaneme termy 000, 001, 010 a 011. Vidíme, že term 010 je v termu 0– také obsažen, takže jej můžeme z tabulky vymazat. Abychom tyto případy zachytili, budeme si při běhu algoritmu uchovávat seznam kandidátů na nadmnožiny, tedy termů s největším počtem *don't care*. Počet takových termů můžeme různě měnit, čímž se bude měnit rychlost výpočtu a kvalita řešení. Testování tohoto vylepšení bude součástí kapitoly 6.2. Na závěr algoritmu projdeme ještě jednou celý strom a vymažeme z něj všechny termy, které jsou podmnožinou některého z termů v seznamu termů s největším počtem *don't care*.

4.2.2 Propagace sloučení listů

Druhé vylepšení je o něco komplikovanější a je spíše námětem na budoucí práci, jelikož není implementováno. První vylepšení bylo zaměřeno na zkvalitnění vypočteného řešení za cenu co nejmenšího zpomalení algoritmu, druhé je zaměřeno pouze na zrychlení algoritmu. Základní myšlenka je tato: při každém sloučení listů zkusíme propagovat tuto informaci z rodičovského uzlu sloučených listů o jednu úroveň výše. Pokud ji uzel, kterému tuto informaci posíláme, obdrží od všech svým potomků, sloučíme celý jeho podstrom. Postup ilustruje obrázek 4.7.



Obrázek 4.7: Propagace sloučení listů do rodičovských uzlů

Levý obrázek zachycuje strom po prvním sloučení listů z výše popsaného průběhu algoritmu. Na levém obrázku vidíme šipkami naznačenou propagaci sloučení listů do rodičovkých uzlů. Vidíme, že v levém podstromu uzlu X1 nebyly sloučeny oba podstromy uzlu X3, a proto je nemůžeme sloučit. Naopak v pravém podstromu uzlu X1 bylo provedeno sloučení obou podstromů uzlů X3, informace o tomto přišla do uzlu X2, a proto můžeme celý podstrom uzlu X2 sloučit.

5 Implementace

V této kapitole budou popsány detaily implementace, zejména použité datové struktury a návrh tříd pro implementaci jednotlivých modulů tak, jak jsem je popsal v kapitole 3. Nejprve ale popíši datové struktury pro uložení samotných ROBDD použité v balíku CUDD a ukáži několik druhů přístupu k tvorbě programů při jeho použití.

5.1 Datové struktury pro ROBDD

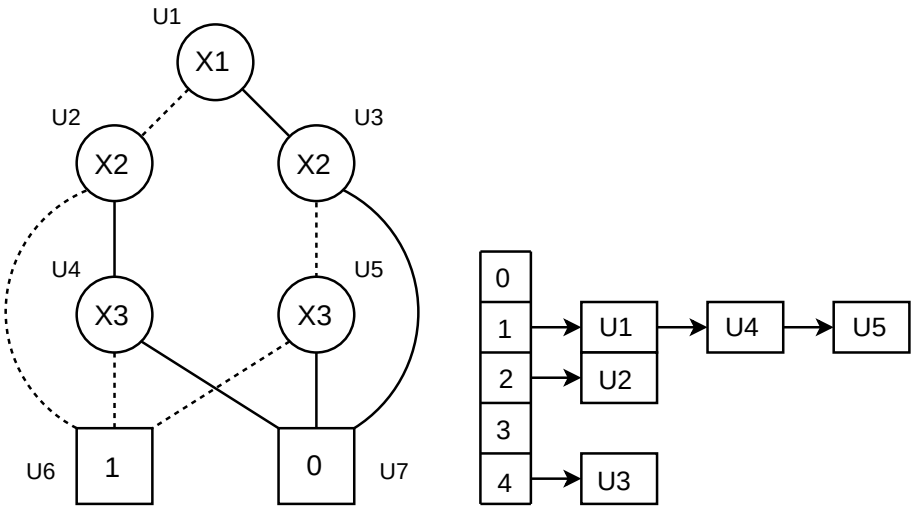
Existuje více různě efektivních datových struktur navržených pro práci s (RO)BDD. Vzhledem k tomu, že pro řešení tohoto problému byl použit balík CUDD (podrobněji o něm v kapitole 5.2), popíši zde pouze struktury v něm použité.

Základní strukturou je struktura *DdNode* - uzel. Obsahuje položky `index`, `počet referencí`, `hodnota` a ukazatele na potomky. Položka `index` označuje neměnné jméno uzlu, které určuje, jakou proměnnou daný uzel reprezentuje. Položka `počet referencí` slouží pro určení, zda je daný uzel potřebný, nebo je možné jej vymazat; pokud její hodnota klesne na 0, je možné uzel vymazat (o to se v balíku CUDD stará integrovaný garbage collector). Položka `hodnota` slouží k určení hodnoty terminálních uzlů, tzn. mají ji nastaveny pouze terminální uzly.

Z těchto uzlů se skládá tzv. *unique table*, což je speciální hash-tabulka určená pro uchovávání ROBDD. Důležité je, že uchovává ROBDD, tedy redukované OBDD. Jak je z názvu *unique table* vidět, je to struktura, navržená tak, aby uzly v ní uložené byly jedinečné - unikátní. V důsledku to znamená, že v tabulce nejsou žádné dva uzly označené stejným indexem se stejnými potomky.

Operuje se zde se třemi hodnotami: index uzlu, index následníka pro nulu (*low*) a index následníka pro jedničku (*high*). Indexy znamenají, jak píši výše, v podstatě jména proměnných. ROBDD se tvoří odspodu, jinak bychom při umisťování uzlu do tabulky neznali indexy potomků. Pokud chceme vytvořit nový uzel, nejdříve se podíváme do tabulky, zda už se v ní nevyskytuje stejný uzel (uzel ze stejnými všemi třemi indexy) a pokud ano, nový uzel se nevytvoří a použije se stávající. Pokud uzel neexistuje, vložíme ho do tabulky nějakou hashovací funkcí. Hashovací funkci můžeme definovat různými způsoby, např. pokud reprezentujeme *unique* tabulku jako pole o velikosti n, můžeme definovat funkci jako $f(x_i, x_j, x_k) = (i + j + k)\%n$, kde index i značí index vkládaného uzlu, index j značí index uzlu *low* a index k značí index uzlu *high*. V případě, že na stejném místě se již nějaký uzel vyskytuje, použije se zřetězený seznam.

Příklad ROBDD a jemu odpovídající *unique* tabulky je na obrázku 5.1. Je důležité uvědomit si rozdíl mezi označením proměnných (Xi) a uzlů (Ui). Každý uzel v *unique* tabulce je unikátní (nejsou žádné 2 stejné uzly se stejným indexem a stejnými potomky), ale více uzlů může reprezentovat stejnou proměnnou. Do *unique* tabulky ukládáme uzly, takže vždy spočítáme na které místo v tabulce daný uzel patří (např. uzel U1 reprezentuje proměnnou X1, index je tedy 1 (počítáme z indexu proměnné) a oba potomci uzlu U1 mají indexy 2; proto je místo, kam uzel uložíme místo v poli s indexem 1 - $index\_v\_poli = (1 + 2 + 2)\%5 = 1$. Podíváme se na tento index v poli a zjistíme, jestli se na něm již nevyskytuje nějaký jiný uzel. Pokud ano, projdeme zřetězený seznam, který začíná na vypočítaném místě v poli a zřetězíme nový uzel za poslední uložený uzel.



Obrázek 5.1: Příklad ROBDD a jemu odpovídající *unique* tabulky

5.2 Balík CUDD

V této kapitole bude popsán balík CUDD (Colorado University Decision Diagram), který patří k nejrozšířenejším balíkům používaným k práci s BDD. Pracuje i s ADD a ZDD, což jsou jiné druhy rozhodovacích diagramů, které ale nejsou pro řešení našeho problému potřebné, proto je zde nebudeme popisovat. Tento balík vytvořil Fabio Somenzi z univerzity v Coloradu. Samotný balík je ke stažení ve formě zdrojových souborů na adrese http://vlsi.colorado.edu/~fabio/CUDD/, momentálně ve verzi 2.4.1. Obsahuje mnoho funkcí pro tvorbu ROBDD, jejich spojování, optimalizaci velikosti a další užitečné funkce, celkem přes 400. Používat se dá třemi způsoby:

- jako černá skříňka - základní způsob, kdy programátor používá pouze externí funkce balíku a nezajímá ho, co se děje uvnitř; vzhledem k tomu, že množství funkcí je více než dostatečné, je tento postup plně postačující
- jako průhledná skříňka - při psaní sofistikovaných aplikací založených na rozhodovacích diagramech je občas potřeba přímý přístup k vnitřním funkcím (funkcím, které jsou volány externími funkcemi) pro operace s diagramy, namísto využívání externích funkcí; pokud je pro naši aplikaci výhodné použít i vnitřní funkce balíku, můžeme použít tento postup
- přes objektové rozhraní - objektově orientované jazyky, jako C++ nebo Perl5 mohou použít objektové rozhraní balíku CUDD, které umožňuje programátorovi hodit za hlavu starosti o správu paměti, referencování uzlů a podobné nízkoúrovňové nepříjemnosti a využívat pestré palety přetížených operátorů

5.2.1 Základní manipulace s ROBDD

Nyní ukáži jak v CUDD vytvořit ROBDD pro jednoduchou funkci $f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4$.

5.2.1.1 Rozhraní jazyka C

Pokud budeme s CUDD zacházet jako s černou skříňkou, bude vypadat úsek programu vytvářející ROBDD výše zmíněné funkce následovně:

```
DdManager *manager;
DdNode *f, *var, *tmp;
int i;

f = Cudd_ReadOne(manager);
Cudd_Ref(f);
for (i = 3; i >= 0; i--) {
    var = Cudd_bddIthVar(manager,i);
    tmp = Cudd_bddAnd(manager,Cudd_Not(var),f);
    Cudd_Ref(tmp);
    Cudd_RecursiveDeref(manager,f);
    f = tmp;
}
```

Proměnná `manager` reprezentuje strukturu, která je souhrnem *unique* tabulky a dalších, pomocných, dat. Je srdcem celého balíku. Jelikož není, kromě několika statistických dat, v CUDD žádná globální proměnná, je možné používat několik instancí manageru zároveň. V našem případě to ale není potřeba. Pro tvorbu ROBDD potřebujeme tři proměnné typu ukazatel na `DdNode`, což, jak je z názvu zřejmé, je struktura popisující uzel v ROBDD.

Funkce `Cudd_ReadOne()` vrátí konstantu "1" uloženou v *unique* tabulce. Uložíme si tedy tuto konstantu do proměnné `f` a funkcí `Cudd_Ref()` zvýšíme počet referencí. BDD se staví odspodu, proto je cyklus od 3 do 0 a ne obráceně. V každé iteraci si nejprve do proměnné `var` uložíme funkcí `Cudd_bddIthVar()` proměnnou s indexem `i`. Pokud taková proměnná neexistuje, bude vytvořena. Následně funkcí `Cudd_bddAnd()` spojíme (operace APPLY) negovanou proměnnou `var`, reprezentující kořen elementárního ROBDD pro jednu proměnnou, s proměnnou `f`. Ta v první iteraci reprezentuje konstantu "1" a v každé následující výsledek spojení `var` a `f` z předchozí iterace. Dostaneme výsledek, který uložíme do proměnné `tmp`. Jelikož jsme právě vytvořili nový uzel, musíme zvýšit počet referencí na něj. Následně provedeme dereferenci proměnné `f` funkcí `Cudd_RecursiveDeref()`, což způsobí snížení počtu referencí na uzel, který proměnná `f` reprezentuje. Pokud snížením počtu referencí klesne tento počet na nulu, jsou dereferencovany i potomci. Tato funkce se používá pro ROBDD, které už nejsou potřeba. Tímto se nám tedy uvolnila proměnná `f`. Výraz `f = tmp`, znamená v podstatě to samé jako:

```
f = tmp;
Cudd_Ref(f);
Cudd_RecursiveDeref(manager,tmp);
```

pouze je efektivnější. Reference na `tmp` je přesunuta na proměnnou `f` a `tmp` je znovu připravena k použití.
Tímto postupem tedy budeme na konci mít v proměnné `f` uložen ukazatel na kořen výsledného ROBDD pro funkci $f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4$.

5.2.1.2 Rozhraní jazyka C++

Pokud budeme chtít použít objektové rozhraní pro tvorbu ROBDD, bude úsek programu vytvářející ROBDD pro stejnou funkci jako v minulém případě vypadat takto:

```
Cudd mgr(0,0);
BDD x = mgr.bddVar();
BDD y = mgr.bddVar();
BDD z = mgr.bddVar();
```

```
    BDD f = !x * !y * !z;
```

Proměnná `mgr` reprezentuje manager z minulého příkladu. Pomocí manageru `mgr` vytvoříme tři proměnné, ze kterých potom přetíženým operátorem `*` spojíme do výsledné funkce. Je zřejmé, že tento způsob je mnohem přehlednější. Navíc se nemusíme se starat o referencování a dereferencování, o to vše se stará CUDD sám.

Mohlo by se zdát, že rozhraní jazyka C++ je zcela jasně lepší způsob jak přistupovat k řešení problému za pomoci knihovny CUDD, nicméně z důvodu nedostatečné dokumentace objektového rozhraní jsem přesto zvolil první způsob. Celý manuál je psán právě pro tento způsob programování, takže pokud bych chtěl řešit nějaké problémy při použití objektového rozhraní, musel bych hledat nápovědu ve zdrojových kódech CUDDu a to by znamenalo značnou ztrátu času. Proto jsem se nakonec dal cestou dobře popsaného, byť ne tak elegantního a bezpečného přístupu k programování.

5.3 Načtení vstupního souboru

Postup načtení vstupního souboru byl popsán v kapitole 3.3, zde budou popsány navržené datové struktury. Potřebujeme do struktury uložit jméno načítaného prvku, jeho typ a v případě, že se jedná o hradlo, i seznam jeho vstupních proměnných. Výsledná struktura se jmenuje `Input_elem` a její deklarace vypadá následovně:

```
struct Input_elem {
    MStr name;
    int type;
    list<MStr> in_vars;
};
```

Proměnná `type` reprezentuje typ prvku; může nabývat hodnot `T_INPUT, T_OUTPUT, T_AND, T_NAND, T_OR, T_NOR, T_XOR, T_XNOR, T_BUF, T_NOT`. Proměnná `name` reprezentuje jméno prvku. Seznam řetězců `in_vars` reprezentuje seznam vstupních proměnných prvku typu hradlo. Jak je ze struktury vidět, používáme prvky standardní knihovny STL (Standrad Template Libray), která obsahuje řadu šablon tříd a algoritmů pro práci s nejčastěji používanými strukturami, jako je např. spojový seznam, pole, fronta nebo zásobník. Práce s touto knihovnou je navíc velice efektivní, a to nejen díky své jednoduchosti, ale i díky dobré efektivitě implementovaných algoritmů. Proto byla tato knihovna při implementaci této práce velmi často využívána.

Realizaci načtení vstupního souboru do popsané struktury má na starosti třída `Import`. Její deklarace vypadá následovně:

```
class Import {
    Lexan *l;
    vector<Input_elem> var_list;
    int token;
    int input_num;
    int output_num;

    void inputs();
    void outputs();
    void assigns();
};
```

Pro jednoduchost zde neuvádím všechny metody, jen ty které provádějí vlastní import. Nejdůležitějším prvkem třídy `Import` je dynamické pole prvků `Input_elem` jménem `var_list`. Do této struktury se při průchodu souborem ukládají nalezené prvky. Ostatní prvky struktury slouží k lexikální analýze (`l, token`) a k uložení pomocných dat.

Metody `inputs()`, `outputs()` a `assigns()` provádějí čtení souboru, tvorbu vstupních prvků a jejich ukládání do seznamu `var_list`.

5.4 Tvorba ROBDD

Nyní ukáži struktury, použité pro tvorbu a manipulaci s ROBDD. Nejprve ukáži jednoduchou strukturu pro uložení vytvořených ROBDD.

```
struct Bdd_elem {
    MStr name;
    DdNode * node;
    int type;
};
```

Tato struktura má jen tři prvky. Jméno proměnné `name`, odkaz na kořenový uzel v ROBDD `node` a typ proměnné `type`. Typ musíme znát proto, abychom odlišili vnitřní proměnné obvodu (výstupy hradel uvnitř obvodu) od výstupních proměnných celého obvodu. To se nám bude hodit v dalších fázích.

O samotnou tvorbu ROBDD se stará třída `Function` a její struktura vypadá následovně:

```
class Function {
    DdManager *manager;
    list<DdNode*> temp;
    list<Bdd_elem> bdd_list;
    list<Bdd_elem> output_list;
    Pupik *pupik;
};
```

Uvádím zde pro jednoduchost jen nejdůležitější prvky. První proměnná `manager` reprezentuje základní strukturu balíku CUDD (*unique* tabulka + pomocná data). Spojový seznam uzlů ROBDD jménem `temp` slouží k dočasnému uložení seznamu kořenových uzlů ROBDD potřebných k operaci APPLY, tzn. do tohoto seznamu se ukládají ROBDD pro vstupní proměnné hradla, pro které chceme vytvořit odpovídající nový ROBDD. Spojový seznam `bdd_list` obsahuje všechny vytvořené ROBDD, resp. odkazy na jejich kořenový uzel. Proměnná `pupik` je ukazatel na třídu `Pupik`, která implementuje minimalizační algoritmus, popsaný v kapitole 4. Implementace tohoto algoritmu bude popsána v kapitole 5.6.

Ještě je třeba zastavit se u pojmu *sekvence operací APPLY*. Takhle vypadá fragment kódu provádějící sekvenci operací APPLY pro vícevstupové hradlo NAND:

```
    DdNode *tmp, *f, *g;

    i = bdd_list.begin();

    tmp = Cudd_Not(*i);
```

```
Cudd_Ref(tmp);
i++;
if (i != _bdd.end())
    g = Cudd_Not(*i);

while(i != _bdd.end()){
    f = Cudd_bddOr(manager, tmp, g);
    i++;
    g = Cudd_Not(*i);
    Cudd_Ref(f);
    Cudd_IterDerefBdd(manager, tmp);
    tmp = f;
}
```

Tento kód projde celý seznam `_bdd`, který obsahuje odkazy na kořenové uzly ROBDD reprezentující vstupní proměnné hradla, pro které právě vytváříme nový ROBDD a sekvencí operací APPLY (funkce `Cudd_bddOr()`) postupně tvoří nové ROBDD. Tato část je důležitá zejména proto, že se v ní skrývá menší záludnost. Pokud bychom totiž zapomněli dereferencovat uzel `tmp`, dostali bychom se do situace, kdy vytváříme dílčí ROBDD, ale poté, co už je nepotřebujeme, je nemažeme z paměti (resp. nedáme snížením počtu referencí znamení garbage collectoru, že uzel může smazat) a jelikož na ně ztratíme ukazatel, už se k nim nikdy nedostaneme. To by nebyl až takový problém, stejně je přeci nepotřebujeme. Problém je ovšem v tom, že stále zabírají místo v unique tabulce, což kromě zbytečně vysoké pamětové náročnosti znamená také neúnosnou časovou náročnost algoritmů pro uspořádání proměnných. Ty totiž musí pracovat s unique tabulkou s třeba i o řád vyšším počtem uzlů než by mohly, kdybychom nepotřebné ROBDD ihned mazali. Proto je třeba si na toto dávat velký pozor.

Na konci této části algoritmu, jsme tedy v situaci, kdy máme uloženy v seznamu `bdd_list` všechny doposud vytvořené ROBDD, tedy i ty dočasné, které nereprezentují výstupní proměnné. Ty ale pro generování pravdivostní tabulky nepotřebujeme, můžeme se jich proto zbavit. Projdeme tedy seznam `bdd_list` a všechny prvky, které nemají nastaven atribut `type` na hodnotu `T_OUTPUT`, smažeme. Smazání probíhá ve dvou fázích. Nejprve zavoláme na kořen aktuálně rušeného ROBDD funkci `Cudd_IterDerefBdd()`, která sníží počet referencí na daný uzel a pokud tento počet klesne na nulu, rekurzivně dereferencuje i jeho potomky. Její primární použití je uvolnění nepotřebných ROBDD z paměti. Druhá fáze spočívá ve smazání celého prvku ze seznamu `bdd_list`. To zajistí metoda `erase()` ze standardní knihovny STL, kterou obsahuje šablona `list`. Ta provede vymazání prvku, uvedeného jako parametr a vrátí ukazatel na další prvek v seznamu. Takto tedy dostaneme seznam všech ROBDD pro výstupní funkce. Z těch potom generujeme výstupní pravdivostní tabulku.

5.5 Snížení velikosti ROBDD pomocí změny pořadí proměnných

Algoritmy pro uspořádání proměnných jsem stručně popsal v kapitole 3.5, takže v této kapitole se můžu věnovat tomu, jak správně využít funkce balíku CUDD pro tuto oblast. Balík CUDD pro toto nabízí dvě možnosti realizace. První je realizován zavoláním funkce `Cudd_AutodynEnable()`, která zapne automatickou změnu pořadí proměnných a vždy po dosažení nějakého prahu v počtu uzlů spustí algoritmus pro změnu pořadí proměnných. Práh si na počátku CUDD inicializuje a po každém spuštění algoritmu pro změnu pořadí proměnných jej aktualizuje. Typ algoritmu je funkci předán jako parametr. Druhou možností je nasazení změny pořadí proměnných po až po vytvoření všech ROBDD, k čemuž slouží funkce `Cudd_ReduceHeap()`. Ta po zavolání spustí algoritmus, zadaný jako parametr a jednorázově

se pokusí o snížení počtu uzlů ve všech ROBDD, které jsou aktuálně uloženy v unique tabulce. Mým úkolem je zhodnotit tyto dva přístupy a rozhodnout, který z nich je výhodnější. Je zřejmé, že každý má své plusy a mínusy. První přístup dělí uspořádání proměnných rozsáhlých ROBDD na menší části, díky čemuž je možno ušetřit výpočetní čas. Na druhou stranu, tento algoritmus počítá s tím, že uspořádání proměnných probíhá již během vlastní tvorby ROBDD, tzn. v situaci, kdy máme uloženy v unique tabulce ještě všechny ROBDD, tedy i dočasné ROBDD potřebné pouze pro vytváření ROBDD pro výstupní proměnné. Tyto ROBDD budou ale do změny pořadí proměnných zahrnuty také, což zmíněnou výhodu obrací v nevýhodu. Pokud totiž chceme postupovat druhým způsobem, můžeme funkci `Cudd_ReduceHeap()` zavolat až poté, co provedeme smazání nepotřebných ROBDD z unique tabulky, následkem čehož bude běh algoritmů pro změnu pořadí proměnných probíhat rychleji, protože počet ROBDD bude nižší. Jako nejvýhodnější se ukázalo použití funkce `Cudd_AutodynEnable()` pro ROBDD, jejichž počet uzlů je extrémně vysoký a je tudíž nutnost snižovat jejich velikost již během tvorby. Pro ostatní je výhodnější použít funkci `Cudd_ReduceHeap()`.

Z implementačního hlediska se tyto metody liší jen jednou věcí, a to pozicí v kódu, kde se funkce `Cudd_AutodynEnable()` a `Cudd_ReduceHeap()` volají. První zmíněnou funkci musíme zavolat ještě předtím než začneme stavět vlastní ROBDD a druhou až poté, co máme všechny potřebné ROBDD postavené, resp. můžeme ji zavolat kdykoliv během tvorby ROBDD, ale tím bychom přišli o výhodu jejího použití, protože až do posledního prvku v poli `var_list` nevíme, které dílčí ROBDD budeme ještě potřebovat, a proto je nemůžeme smazat.

5.6 Minimalizace pravdivostní tabulky

O minimalizaci výstupní pravdivostní tabulky se stará třída `Pupik`. Její deklarace vypadá následovně:

```
class Pupik {
    I_buffer *min_tree;
    long count;
    int inputs_num;
    int outputs_num;
    int same_term;
    vector<Top_term> top_terms;
};
```

Proměnné `count, inputs_num, outputs_num` a `same_term` jsou podpůrné proměnné pro potřeby minimalizačního algoritmu. Dynamické pole `top_terms` slouží k uložení potenciálně vhodných termů pro minimalizaci podmnožinami (absorpce). Ukládají se do něj termy, které mají největší šanci k absorbování ostatních termů. Hlavním prvkem třídy `Pupik` je proměnná `min_tree`, která reprezentuje ukazatel na kořen minimalizačního ternárního stromu. Ternární strom, zde pojmenován jako I-buffer, implementuje třída `I_buffer`.

Nejprve ukáži deklaraci třídy `Top_term`:

```
class Top_term {
    char *term;
    int dc_count;

    int operator<(const Top_term&) const;
};
```

Tato třída popisuje jeden prvek v poli termů s největším počtem ohodnocení "-". Řetězec `term` obsahuje vlastní term a proměnná `dc_count` obsahuje počet ohodnocení "-" v termu. Přetížený operátor $<$ slouží k vyhledávání termu s nejnižším počtem "-". Vlastní vkládání termů do pole `top_terms` probíhá až po dokončení hlavního minimalizačního algoritmu. Projdeme celý strom a vygenerujeme K nejnadějnějších termů, kde s K je parametr algoritmu, kterým můžeme měnit kvalitu řešení a časovou náročnost. Čím větší bude K, tím větší šanci na odhalení potenciálních podmnožin budeme mít, ovšem za cenu větší časové náročnosti. Strom totiž budeme muset projít K-krát.

Nyní ukáži deklaraci třídy `I_buffer` realizující jeden uzel v ternárním minimalizačním stromu:

```
class I_buffer {
    I_buffer *one;
    I_buffer *zero;
    I_buffer *dc;

    bool is_leaf(void);
};
```

Třída obsahuje 3 ukazatele, které slouží k propojení uzlů stromu mezi sebou. Ukazatel `one` určuje potomka aktuálního uzlu pro ohodnocení proměnné "1", ukazatel `zero` určuje potomka pro ohodnocení "0" a ukazatel `dc` určuje potomka pro ohodnocení "-".

Hlavními metodami, které realizují nejdůležitější kroky algoritmu, jsou `mt_merge_leaves()` a `mt_divide_and_merge()`. Z názvů je patrná jejich základní činnost, první z nich projde strom a sloučí všechny listy, které lze sloučit. Druhá metoda provede další kroky algoritmu: odtržení kořene, vložení nových listů do stromů a sloučení všech stromů do jednoho. Tyto kroky se provádějí v cyklu N-krát, kde N je počet proměnných.

5.7 Výpis do výstupního souboru

O výpis do výstupního souboru ve formátu PLA se stará třída Export. Její deklarace vypadá následovně:

```
class Export {
    ofstream PLA_file;
public:
    void PLA_print_header(list<Bdd_elem>&, int, int);
    void PLA_print_footer();
    bool export_to_pla(char *, char *, int, int);
};
```

Proměnná `PLA_file` typu `ofstream` (třída pro výstupní stream) je použita pro samotný výpis do výstupního souboru. Metoda `PLA_print_header()` zapíše do výstupního streamu hlavičku ve formátu PLA, tzn. informaci o počtu vstupních a výstupních proměnných obvodu a jejich jména. Metoda `PLA_print_footer()` zapíše do výstupního souboru patičku, tzn. ukončí PLA soubor značkou `.e`. Metoda `export_to_pla()` vypíše do výstupního streamu jeden řádek ve formátu PLA, tzn. vstupní term a jím implikovaná hodnota výstupu oddělená mezerou.

6 Testování

V této kapitole otestuji vlastnosti použitých algoritmů a posoudím jejich kvality. Po vyhodnocení provedu srovnání s jinými přístupy, které budou reprezentovat zmíněné systémy MVSIS a ESPRESSO.

6.1 Algoritmy pro uspořádání proměnných

Jak již bylo řečeno, balík CUDD obsahuje několik algoritmů pro uspořádání proměnných. Princip těchto algoritmů jsem popsal v kapitole 3.5, nyní je můžu důkladně otestovat. Metody jsem porovnával na vybraném souboru jedenácti testovacích souborů ze standardní sady [5]. V tabulce 6.1 jsou zobrazeny parametry vybraných obvodů.

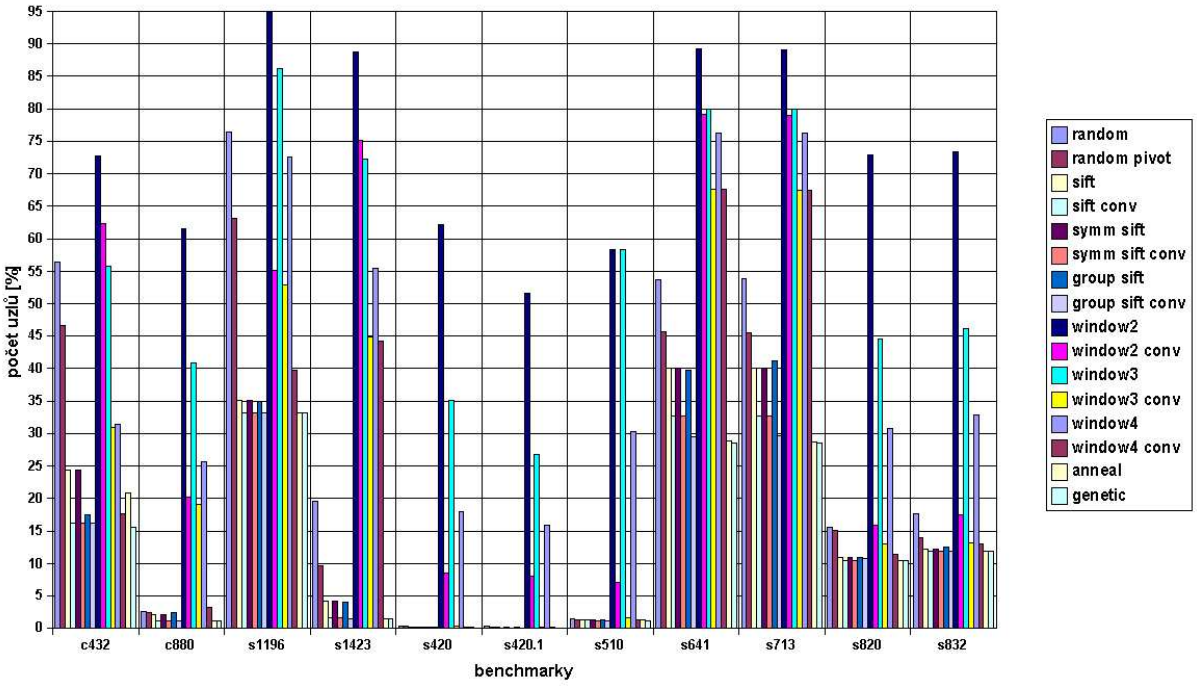
Tabulka 6.1: Parametry vybraných testovacích souborů

benchmark	**počet vstupů**	**počet výstupů**	**počet hradel**
c432	36	7	159
c880	60	26	357
s420	35	18	196
s420.1	34	17	218
s510	25	13	211
s641	54	42	379
s713	54	42	393
s820	23	24	289
s832	23	24	287
s1196	32	32	529
s1423	91	79	657

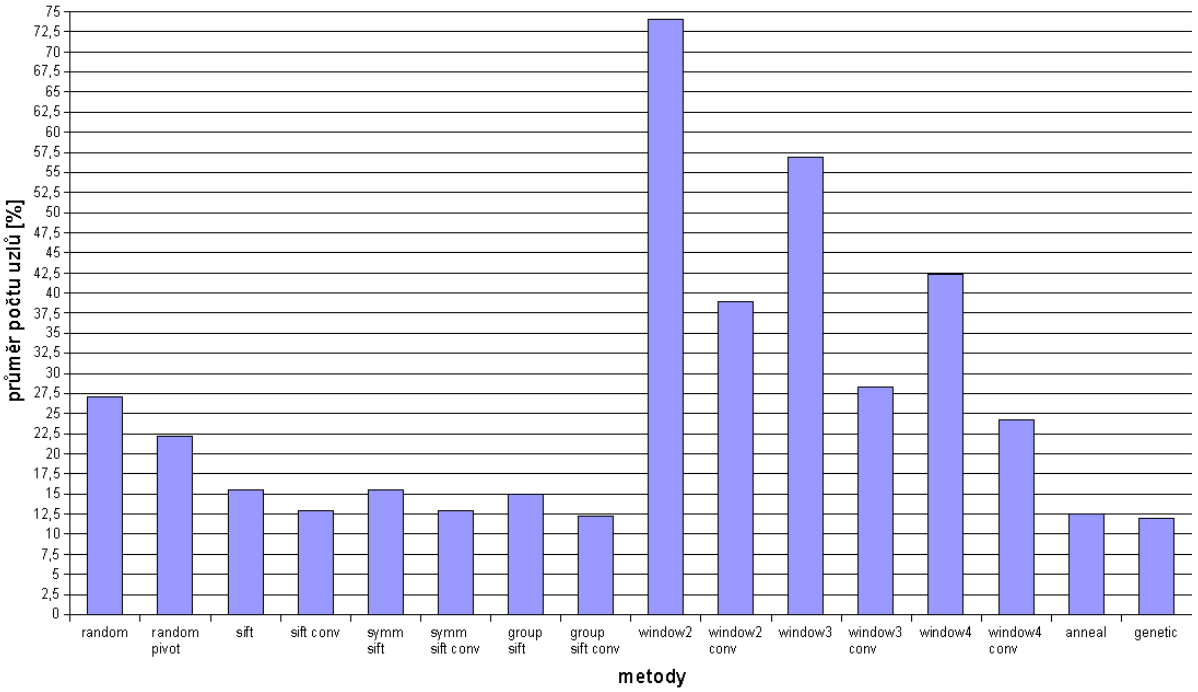
Nejprve se zaměřím na otestování výsledného počtu uzlů v ROBDD po změně pořadí proměnných. Tedy testování schopnosti jednotlivých metod snížit velikost samotného ROBDD. Kvůli prevenci ovlivnění měřeného výsledku budu provádět měření bez použití minimalizace pravdivostní tabulky; to bude důležité zejména v druhé části, kde budu měřit výsledný počet termů, které generují ROBDD po snížení počtu uzlů vlivem změny pořadí proměnných.

Na grafu 6.1 je vidět, že i jednoduché metody postavené na náhodné změně pořadí (`random` a `random pivot`) poskytují nezanedbatelnou úsporu v počtu uzlů výsledného ROBDD. Jejich výhodou je jednoduchost, nevýhodou je prvek náhodnosti, který způsobí nedeterministické chování algoritmu. Naproti tomu algoritmy založené na pracech [8] a [9] (v grafu to jsou metody `window2`, `window3` a `window4`) včetně jejich konvergentních variant nepřinášejí potřebnou efektivitu.

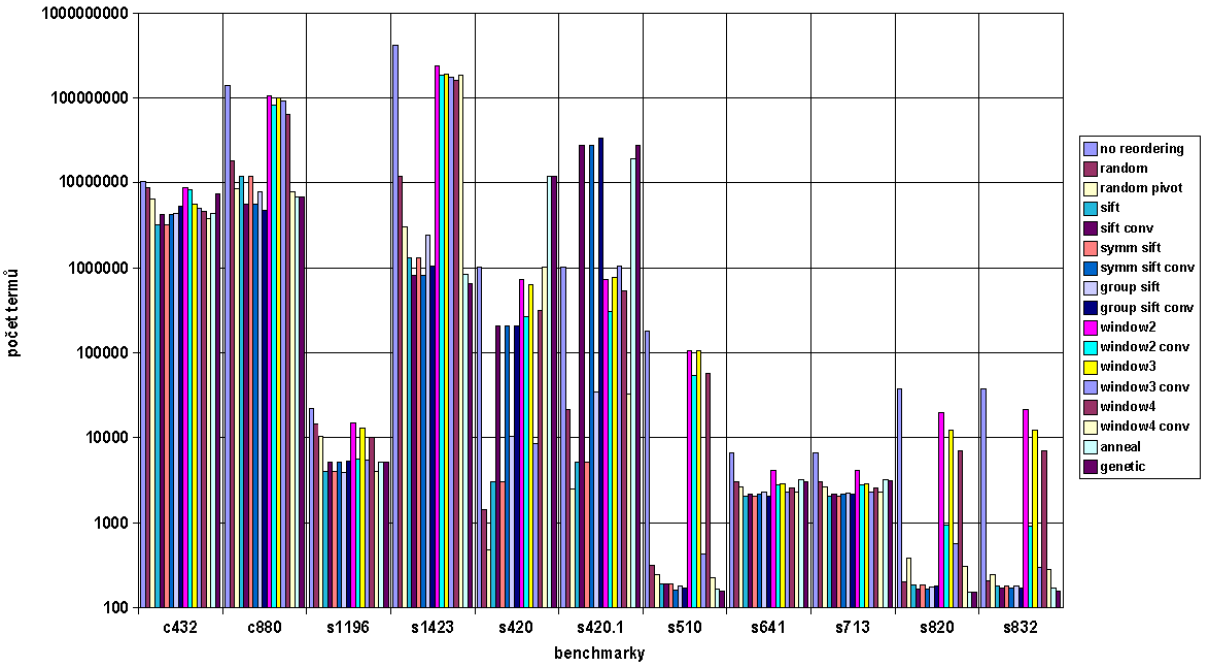
Soubor algoritmů založených na `sifting` algoritmu (`sifting`, `symmetric sifting` a `group sifting`) zároveň s jejich konvergentními variantami a metody založené na simulovaném ochlazování (`anneal`) resp. genetických algoritmech (`genetic`) poskytují velmi vyrovnanou kvalitu řešení. Abych mohl rozhodnout, který z nich je v tomto kritériu nejúspěšnější, vypočtu pro každý algoritmus průměrný počet uzlů po minimalizaci (v procentech) a pro každý algoritmus tak dostanu jedno číslo reprezentující průměrnou kvalitu řešení z hlediska počtu uzlů. Tuto skutečnost zachycuje graf 6.2.



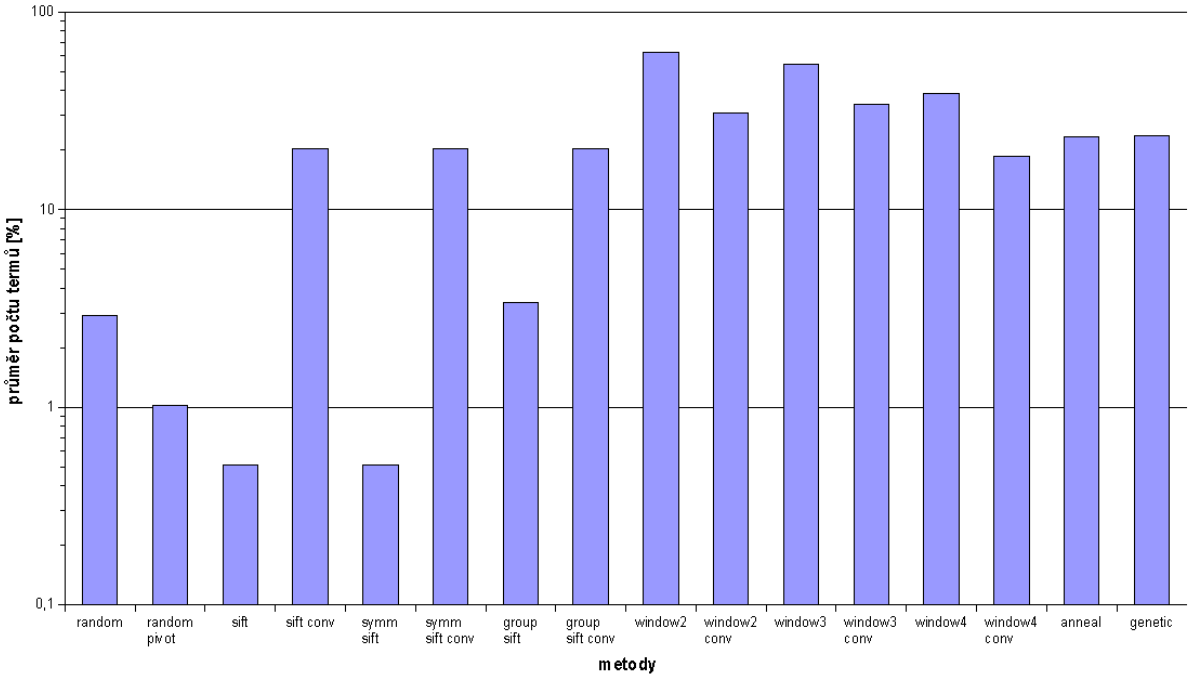
Obrázek 6.1: Porovnání algoritmů pro uspořádání proměnných podle výsledného počtu uzlů



Obrázek 6.2: Porovnání algoritmů pro uspořádání proměnných podle průměrného počtu uzlů



Obrázek 6.3: Porovnání algoritmů pro uspořádání proměnných podle výsledného počtu termů



Obrázek 6.4: Porovnání algoritmů pro uspořádání proměnných podle průměrného počtu termů

Na grafu 6.2 je jasně vidět propad algoritmů založených na permutacích oken. Dále je vidět, že konvergentní metody založené na metodě `sifting` společně s algoritmy založenými na simulovaném ochlazování a genetických algoritmech podávají nejlepší výsledky. Rozdíly mezi nimi jsou zanedbatelné. V této části testování tedy dopadly nejlépe.

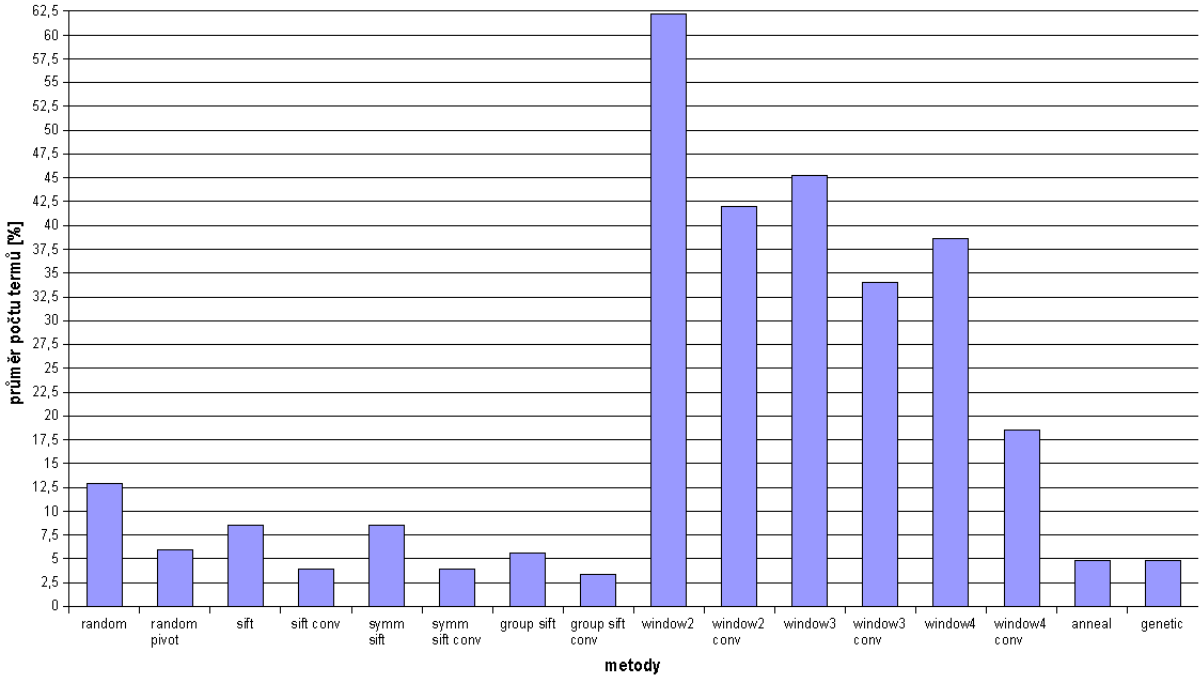
Další testování, které provedu, je ještě mnohem důležitější než pohled čistě na snížení počtu uzlů v ROBDD. Výstupem algoritmu je totiž pravdivostní tabulka a počet řádků (termů) v ní je jedním z nejdůležitěších kritérií (spolu s časem výpočtu), podle kterých posuzujeme kvalitu algoritmu. Porovnám tedy metody pro řazení z hlediska vlivu na výsledný počet termů. Počty termů pro jednotlivé metody a testovací soubory ukazuje graf 6.3.

Pro srovnání jsem jako první metodu vložil do grafu "metodu" `no_reordering`, tedy počet termů bez aplikování kteréhokoli algoritmu pro změnu pořadí proměnných. Velmi dobře tedy vynikne jedna záludnost tohoto problému. Pouhé snížení počtu uzlů v ROBDD nezaručuje, že se sníží i počet termů jím generovaných. Dobře je to vidět na příkladu s420.1, kde počet termů generovaných z ROBDD po změně pořadí proměnných metodami `sift_conv`, `symm_sift_conv`, `group_sift_conv`, `anneal` a `genetic` je dokonce řádově vyšší než před ní! Všechny metody minimalizace ROBDD totiž mají za cíl minimalizovat primárně počet uzlů v grafu, nikoli počet cest z kořene do listu (tedy počet termů). Ve většině případů se podaří obojí; snížením počtu uzlů se sníží i počet cest. Ve výjimečných případech se ovšem i při razantním snížení počtu uzlů může počet cest v grafu naopak zvýšit. Tato skutečnost klade do budoucna požadavek na vytvoření takových algoritmů, které se zaměří spíše než na minimalizování počtu uzlů, právě na minimalizaci počtu cest.

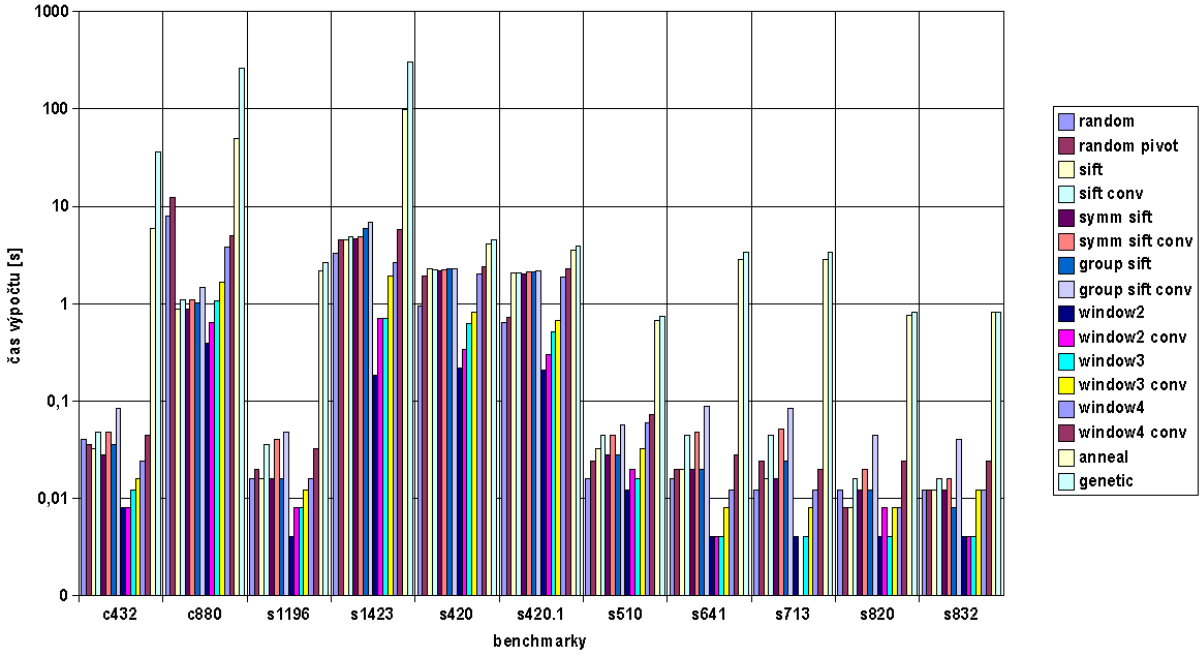
Pro přehlednost uvádím znovu graf průměrého počtu generovaných termů po změně pořadí proměnných jednotlivými metodami. Je vidět, že i přes výše zmíněnou anomálii, je počet termů po změně pořadí proměnných dramaticky nižší (průměr opět vynášíme v procentech; průměrnému počtu termů bez aplikace uspořádání proměnných tedy odpovídá hodnota 100%). Bohužel několik metod, zejména kovergentní varianty metod založených na siftingu a metody simulovaného ochlazování a genetických algoritmů, bylo drasticky ovlivněno benchmarkem s420.1, který se svým chováním naprosto vymyká ostatním. Pro srovnání uvedeme i graf 6.5, zobrazující průměrný počet termů pro jednotlivé metody bez tohoto benchmarku. Při porovnání grafů 6.5 a 6.4 vidíme, jak velký vliv měl benchmark s420.1 na výsledek konvergentních variant algoritmů `sift`, `symm_sift` a `group_sift` spolu s algoritmy simulovaného ochlazování a genetických algoritmů. Závěr, který je z tohoto zjištění možno učinit zní: Metody poskytující nejnižší výsledný počet uzlů v ROBDD, poskytují ve většině případů také nejnižší počet generovaných termů; ve výjimečných případech mohou ovšem vypočíst výsledek, který bude ještě horší než výsledek vypočtený bez jakékoli změny pořadí proměnných. Díky této záludnosti, je tedy jistější v praxi používat spíše metody nekonvergentní, které sice neposkytují tak nízký počet uzlů po změně pořadí proměnných, zato ovšem netrpí výše zmíněnou vlastností. Dále je také vidět velmi špatné chování algoritmů založených na permutaci oken, které se opět jeví jako nejhorší. V tomto kritériu podaly kvalitní výsledek také jednoduché náhodné algoritmy, což staví zmíněné `window` algoritmy do ještě horšího světla. Nejlepšího výsledku v této kategorii testů dosáhly metody `sift` a `symm_sift`, jejichž výsledky jsou velmi vyrovnané.

Poslední oblastí, ve které budu algoritmy testovat, je čas výpočtu nového pořadí proměnných. Graf pro jednotlivé metody a testovací soubory je zobrazen na obrázku 6.6.

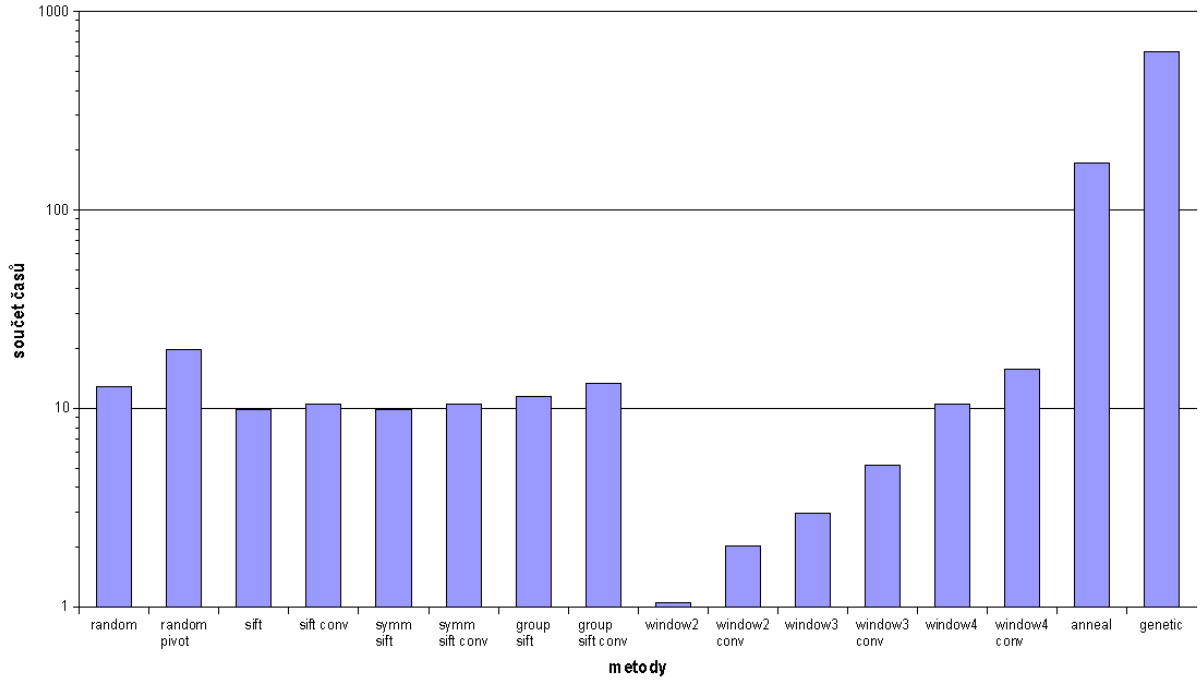
Časy se lišily o mnoho řádů, proto je na ose y použito logaritmické měřítko. Problém způsobují, dle očekávání, hlavně algoritmy využívající principů simulovaného ochlazování a genetických al-



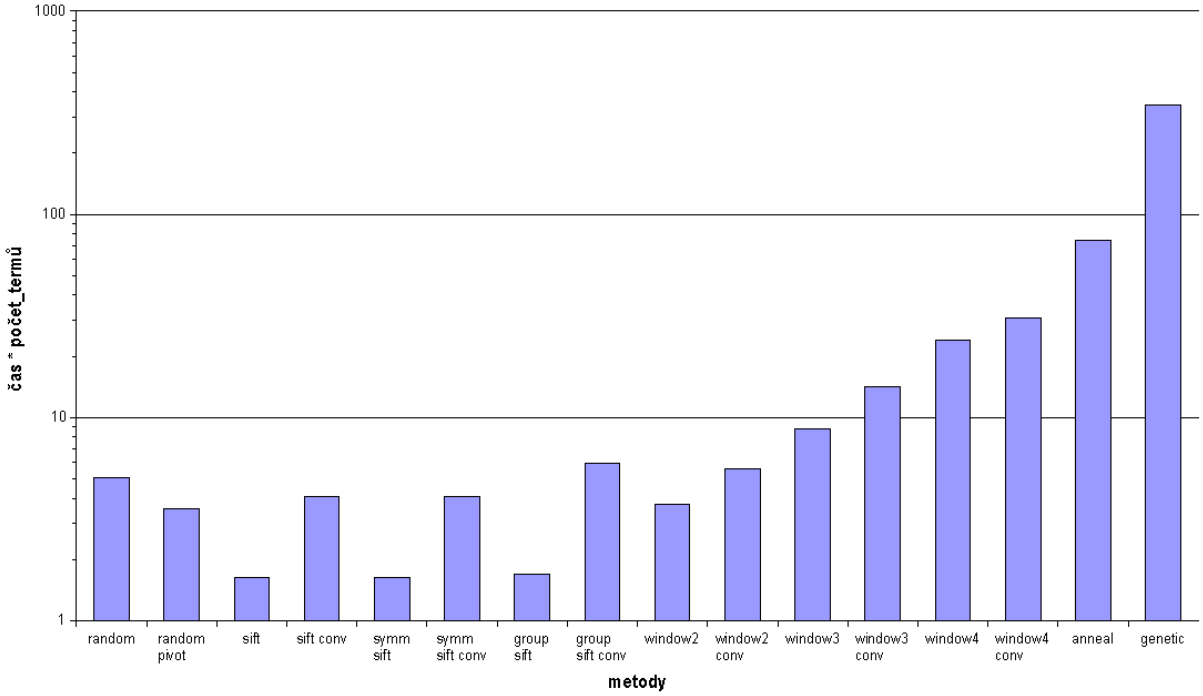
Obrázek 6.5: Porovnání algoritmů pro uspořádání proměnných podle průměrného počtu termů bez benchmarku s420.1



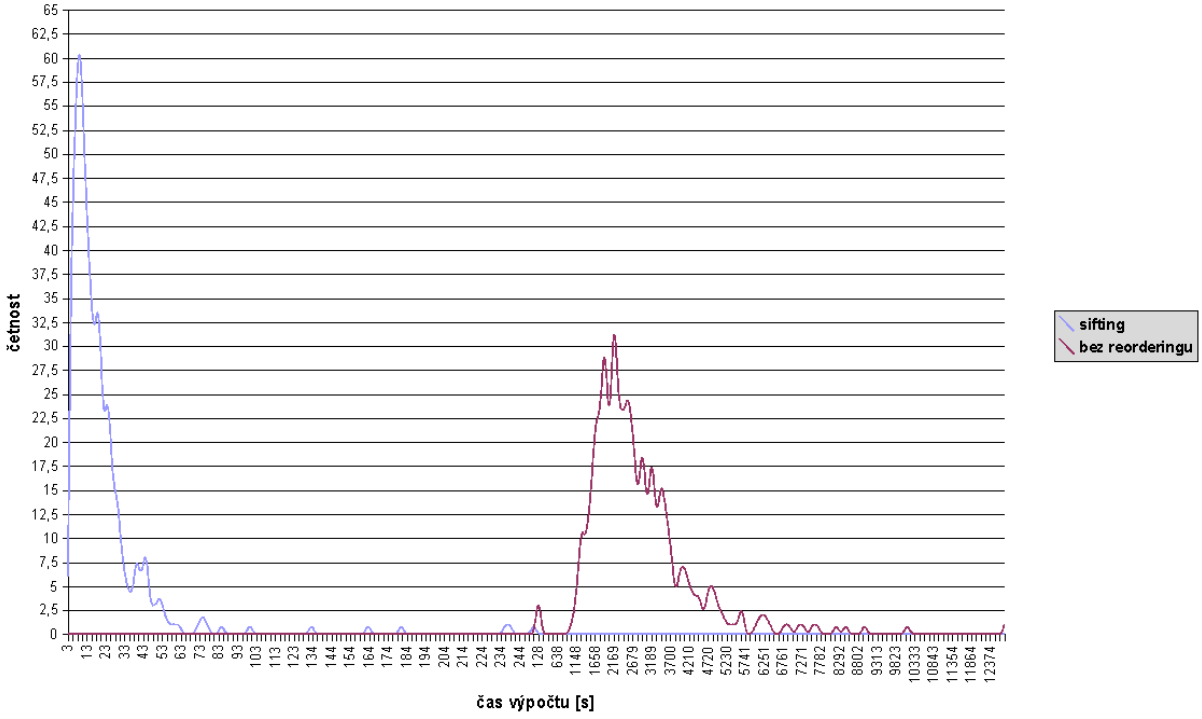
Obrázek 6.6: Porovnání algoritmů pro uspořádání proměnných podle času výpočtu



Obrázek 6.7: Porovnání algoritmů pro uspořádání proměnných podle času výpočtu



Obrázek 6.8: Porovnání algoritmů pro uspořádání proměnných podle poměru času ke kvalitě řešení



Obrázek 6.9: Histogram času výpočtu pro collapsing bez použití změny pořadí proměnných a s použitím algoritmu sifting

goritmů. Tyto metody jsou pro rozsáhlé příklady prakticky nepoužitelné, protože jejich časová náročnost je místy i o dva řády vyšší než časová náročnost ostatních algoritmů při produkci srovnatelně kvalitního řešení. Dále je vidět, že algoritmy `window`, které nepodávaly v minulých měřeních srovnatelně kvalitní výsledky jako ostatní metody, vítězí alespoň v tomto kritériu. Nicméně ostatní algoritmy, zejména ty založené na `siftingu`, poskytují i řádově nižší počet termů, a proto ani nejvyšší rychlost není důvodem k použití této skupiny algoritmů. Abychom rozhodli, který algoritmus bude v tomto kritériu nejlepší, provedeme sumaci naměřených časů pro jednotlivé metody a tento součet vyneseme do grafu. Získáme tak lepší přehled o tom, který z algoritmů vypočetl výsledky nejrychleji. Tuto skutečnost zachycuje graf 6.7.

Na grafu je ještě lépe vidět jednak obrovský časový propad metod iterativních heuristik `anneal` a `genetic` a také vysoká rychlost `window` algoritmů. Výsledky `sifting` algoritmů jsou poměrně vyrovnané. Nejlepších výsledků dosahují algoritmy `sift` a `symm_sift`.

Abych dokázal rozhodnout, který algoritmus je pro náše potřeby nejvhodnější, je nutné předchozí testy zkombinovat, abych dosáhl srovnání v nějakém jednoznačném kvalitativním kritériu - efektivitě. Proto vynesu do grafu poměr `čas / kvalita_řešení`, což značí v podstatě součin času, vypočteného v minulém případě a počtu termů z předminulého případu. Čím nižší výsledek tohoto součinu, tím lepší algoritmus je. Srovnání ukazuje graf 6.8.

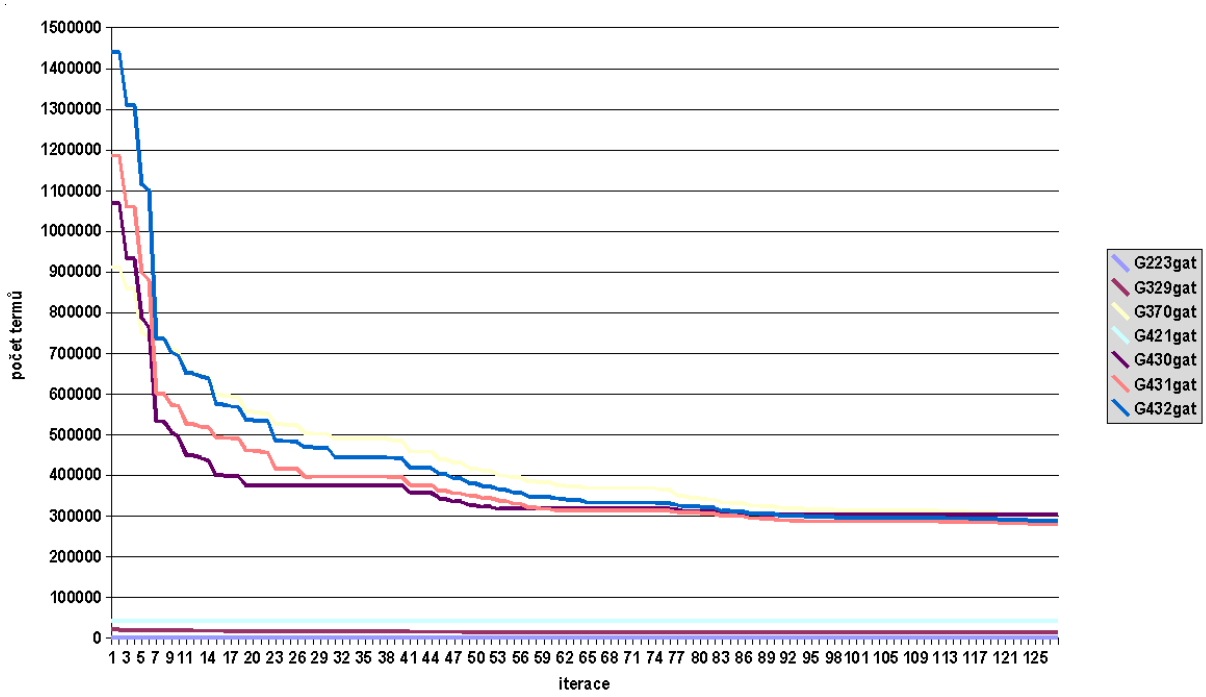
Kvůli řádovým rozdílům ve výsledcích, je nutno opět použít logaritmické měřítko. Vidíme, že metody `anneal` a `genetic` jsou opravdu nevhodné k použití, jelikož jejich efektivita je řádově horší než u ostatních metod. U `window` metod lze pozorovat postupné zhoršování efektivity při zvětšování okna. Čas potřebný k výpočtu totiž roste rychleji než klesá počet termů generovaných minimalizovaným ROBDD. Nejlepší volbou tedy budou metody založené na siftingu

a jak je vidět, je relativně lhostejné, kterou z nich použijeme; všechny tři mají takřka stejnou efektivitu. Dlužno dodat, že efektivita je počítána z hodnot času a počtu termů i s benchmarkem s420.1. Je v ní tedy započtena i ona záludnost ovlivňující zejména konvergentní varianty `sifting` algoritmů.

Na závěr ještě uvedu graf ukazující histogram času výpočtu při převodu víceúrovňové logické sítě na dvouúrovňovou bez změny pořadí proměnných a poté s použitím `sifting` algoritmu, který byl vyhodnocen jako vhodný pro praktické použití. Vygeneroval jsme přes 400 náhodných počátečních pořadí proměnných a zkusil spustit vlastní převod. Nejprve jsem vygeneroval pravdivostní tabulku bez jakékoli změny pořadí proměnných, poté jsem funkcí `Cudd_ReduceHeap()` spustil `sifting` algoritmus a opět vygeneroval pravdivostní tabulku. Na grafu 6.9 je vidět, jak se při náhodném generování pořadí proměnných u benchmarku c432 vyvíjel čas výpočtu, tedy jak často se čas výpočtu vešel do daného časového intervalu. Je jasně vidět, že při použití sifting algoritmu je čas výpočtu radikálně nižší než bez jeho použití. Výhoda využití algoritmů pro změnu pořadí proměnných je tedy jasně patrná.

6.2 Minimalizace pravdivostní tabulky

Nyní otestuji navržený algoritmus pro minimalizaci pravdivostní tabulky. Nejprve se pokusím naleznout optimální nastavení vnitřních parametrů algoritmu, tzn. optimální počet iterací a vliv vylepšení využívající podmnožin (jde v podstatě o zákon absorpce), které bylo uvedeno v kapitole 4.2.1. Pokusím se experimentálně ověřit tvrzení z kapitoly 5.6, že optimální počet iterací je roven počtu proměnných. Poté analyzuji chování algoritmu v závislosti na nastavení parametrů, tj. závislost na počtu proměnných minimalizované funkce, na počtu termů a na poměru zastoupení *don't care* ve vstupních termech. Nakonec porovnám vlastnosti algoritmu se systémem pro minimalizaci logických funkcí `espresso`. Pro generování testovacích souborů se zadanými parametry použiji PLA Generator z bakalářské práce Tomáše Měchury [12]. Tento generátor umožňuje nastavovat všechny potřebné parametry vstupních logických funkcí.



Obrázek 6.10: Závislost počtu termů na počtu iterací

Na grafu 6.10 vidíme závislost počtu termů v minimalizačním stromu na počtu iterací. Testování probíhalo na benchmarku `c432` ze standardní sady testovacích souborů [5]. Tento obvod obsahuje 36 vstupních proměnných a 7 výstupních proměnných. Algoritmus minimalizuje vždy každou výstupní proměnnou zvlášť. Každá křivka v grafu reprezentuje vývoj počtu termů ve stromě odpovídající konkrétní výstupní proměnné. Vidíme, že největší minimalizační efekt se dostavuje hned zpočátku a poté až do počtu iterací odpovídající počtu proměnných zpomaluje. Na počátku další "superiterace" (násobek počtu proměnných) se opět dostaví minimalizační efekt, nicméně již mnohem nižší. Po několika "superiteracích" je již efekt minimalizace zcela zanedbatelný. Díky větší přehlednosti, zde není graf zobrazující závislost počtu termů na iteracích až do počtu iterací rovnému N^2, kde N je počet proměnných. Vidíme, že o počtu iterací vyšším než $2N$ již nemá cenu uvažovat. Vzhledem k efektivitě provedené minimalizace je vhodnou volbou pro počet iterací N.

Na grafu 6.11 vidíme vliv minimalizace přes podmnožiny na čas výpočtu. Na ose x jsou vy-

neseny vybrané výstupní proměnné jednotlivých obvodů opět ze sady [5]. Minimalizace přes podmnožiny probíhá vždy až po dokončení všech iterací minimalizačního algoritmu. Celkový čas je proto vždy vyšší; otázka je, zda její výsledky vyváží tento výpočetní čas navíc. Na grafu je vidět, že vliv na čas výpočtu je ve většině případů vcelku malý, zhruba do deseti procent. Nicméně při pohledu na graf 6.12 je zřejmé, že ani tak malé dopady na celkový čas výpočtu neospravedlňují nulový efekt na výsledný počet termů. Ukázalo se tedy, že na praktických příkladech reálných obvodů není efektivní toto vylepšení používat, a proto bude v základním nastavení vypnuto. Dlužno dodat, že toto zjištění není proti předpokladům, jelikož v reálných obvodech se případy vhodné pro tento typ minimalizace příliš nevyskytují. Uplatnění toto vylepšení najde při minimalizaci náhodně generovaných logických funkcí.

Nyní již jsou nastaveny parametry algoritmu a můžu přistoupit k testování jeho vlastností. Nejprve otestuji chování algoritmu, když budu měnit poměr *don't care* ve vstupních termech. Počáteční nastavení ostatních parametrů je následující: počet vstupních proměnných nastavím na 30 a počet termů na 10000. Poměr *don't care* budu měnit od nuly po 90%.

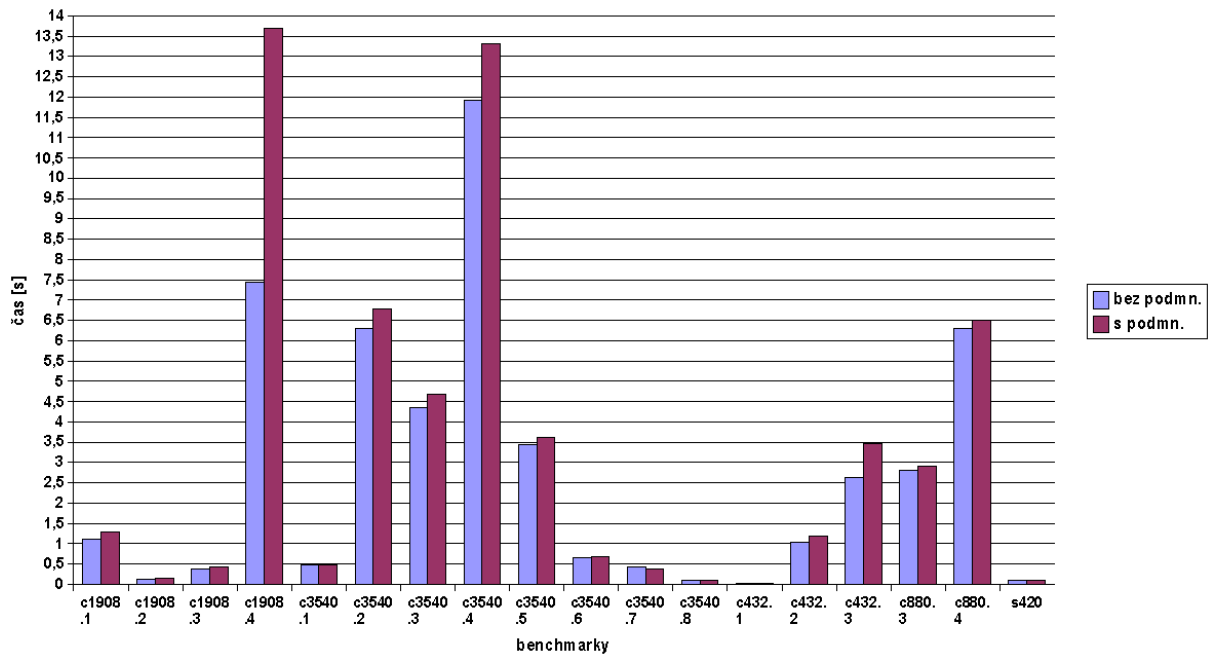
Graf ukazující závislost minimalizovného počtu termů na poměru *don't care* ve vstupních termech ukazuje graf 6.13. Vidíme, že až do 50% *don't care* ve vstupních termech algoritmus vůbec nedokáže minimalizovat počet termů. Počet termů je totiž příliš nízký na to, aby se objevila nějaká dvojice lišící se pouze v jedné proměnné. Čím více se ovšem v termech budou vyskytovat *don't care*, tím větší bude šance, že takové dvojice vzniknou. Počet jedniček a nul v termech totiž klesá. Na grafu je to jasně vidět. U 60% se již vyskytuje první zaznamenáníhodná minimalizace a dále se počet termů po minimalizaci ještě razantněji snižuje. Pokud se bude poměr *don't care* dále zvyšovat, dříve nebo později dojde k situaci, že generátor vygeneruje takovou množinu termů, které tvoří tautologii. Taková situace nastala při 75%. Druhý graf 6.14 ukazuje závislost času výpočtu na poměru *don't care*. Je vidět, že čas výpočtu je vcelku stabilní, razatněji se začne snižovat až při větším poměru *don't care*; počet termů v ternárním stromě totiž razantně klesá, proto klesá i počet operací, které musí algoritmus provést.

Druhým předmětem testování bude parametr počet vstupních proměnných. Otestuji jaká je závislost času výpočtu na počtu vstupních proměnných. Pevné nastavení ostatních parametrů zvolím následovně: poměr *don't care* ve vstupních termech nastavím na 70%, jelikož v minulém testování se ukázalo, že to je hraniční hodnota, kdy algoritmus začíná počet termů skutečně minimalizovat; počet termů nastavíme na 5000.

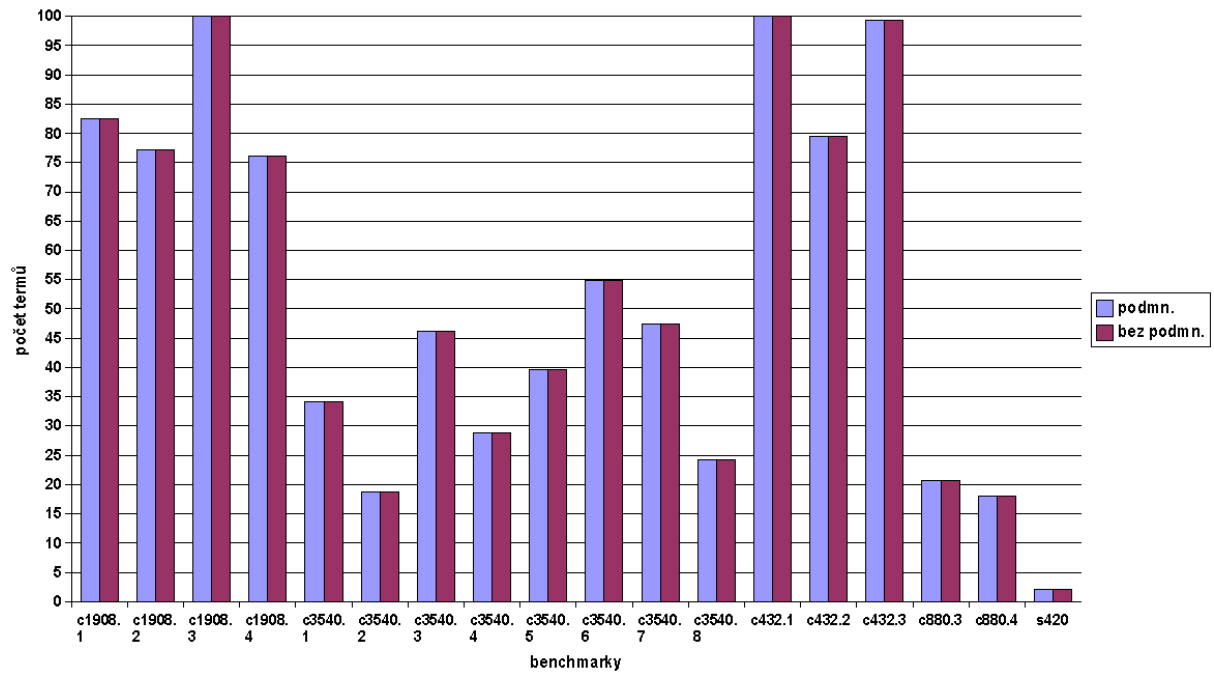
Graf 6.15 ukazuje závislost času výpočtu na počtu vstupních proměnných. Jak jsem popsal v kapitole 4.1, tato závislost je kvadratická, což se v grafu jasně potvrzuje.

Posledním parametrem algoritmu je počet minimalizovaných termů. Budu měnit počet termů na vstupu algoritmu při zachování počtu vstupních proměnných a poměru *don't care* ve vstupních termech. Tyto parametry nastavím následovně: počet vstupních proměnných bude 30 a poměr *don't care* ve vstupních termech bude 70%. Využiji přitom poznatky nasbírané při předchozích testech, kde se tyto hodnoty ukazovaly jako vhodný kompromis pro testování.

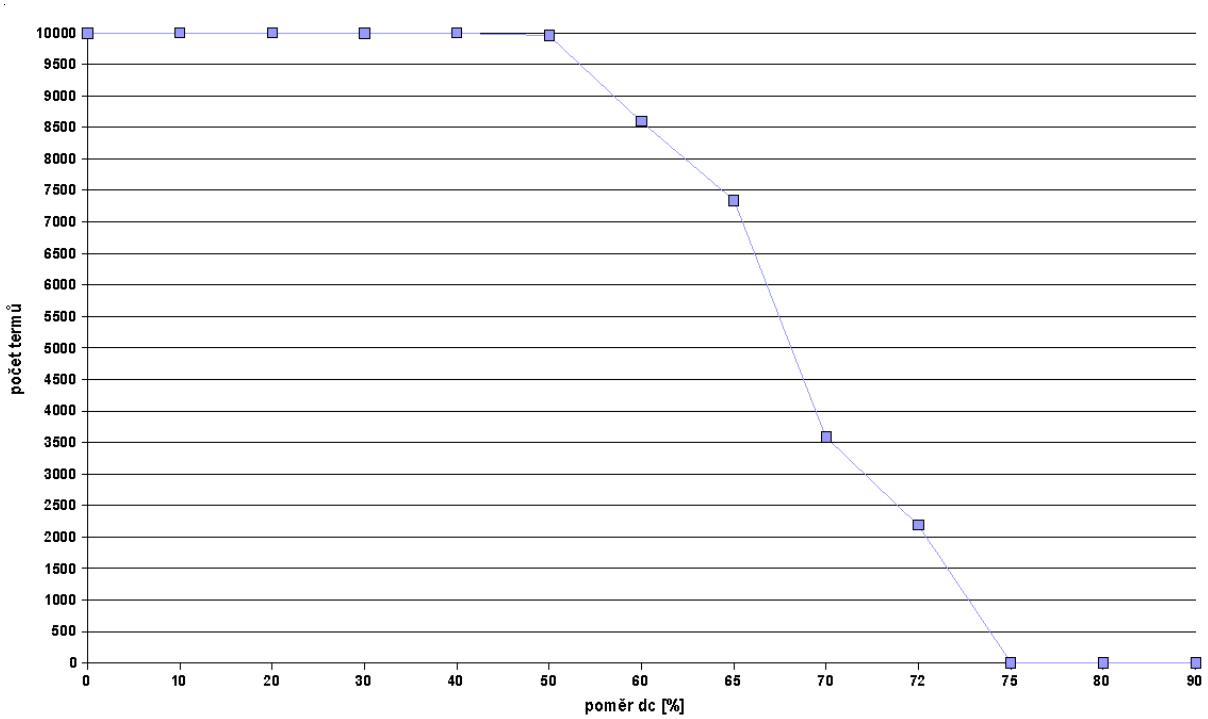
Graf 6.16 ukazuje závislost času výpočtu na měnícím se počtu termů na vstupu. Tato závislost je, jak jsem ukázal v kapitole 4.1, lineární a graf toto potvrzuje.



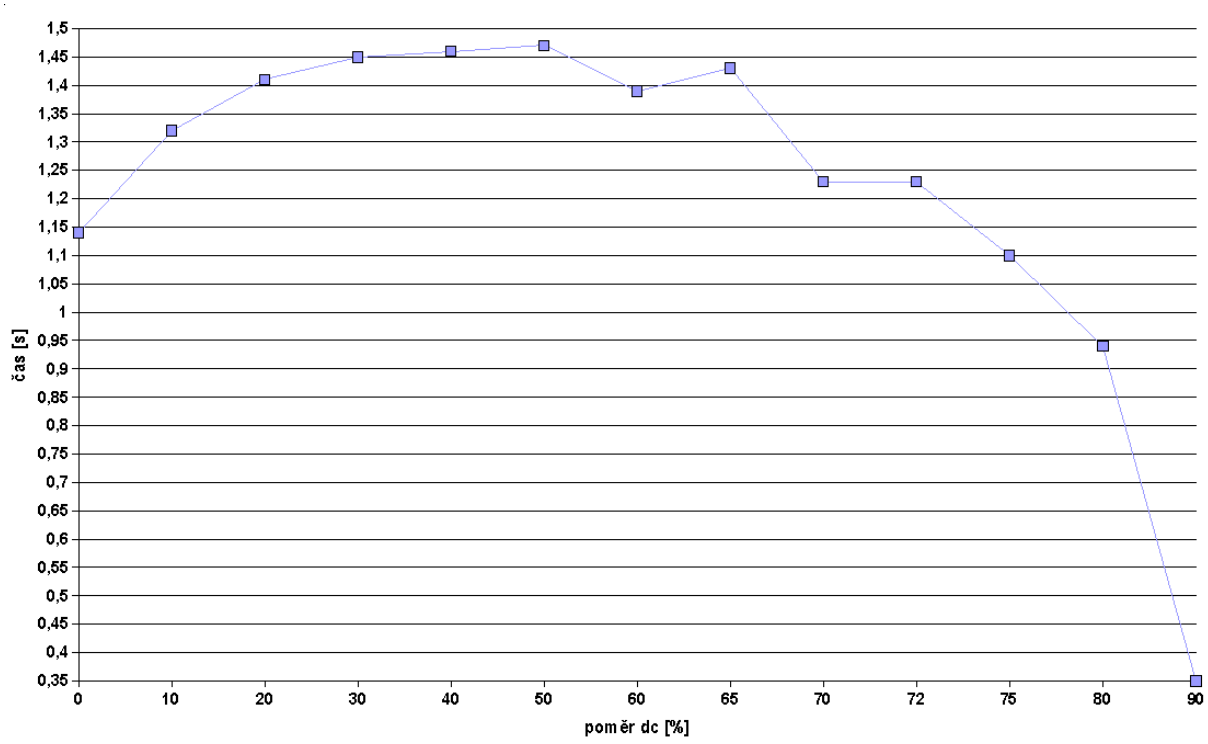
Obrázek 6.11: Vliv minimalizace podmnožinami na čas výpočtu



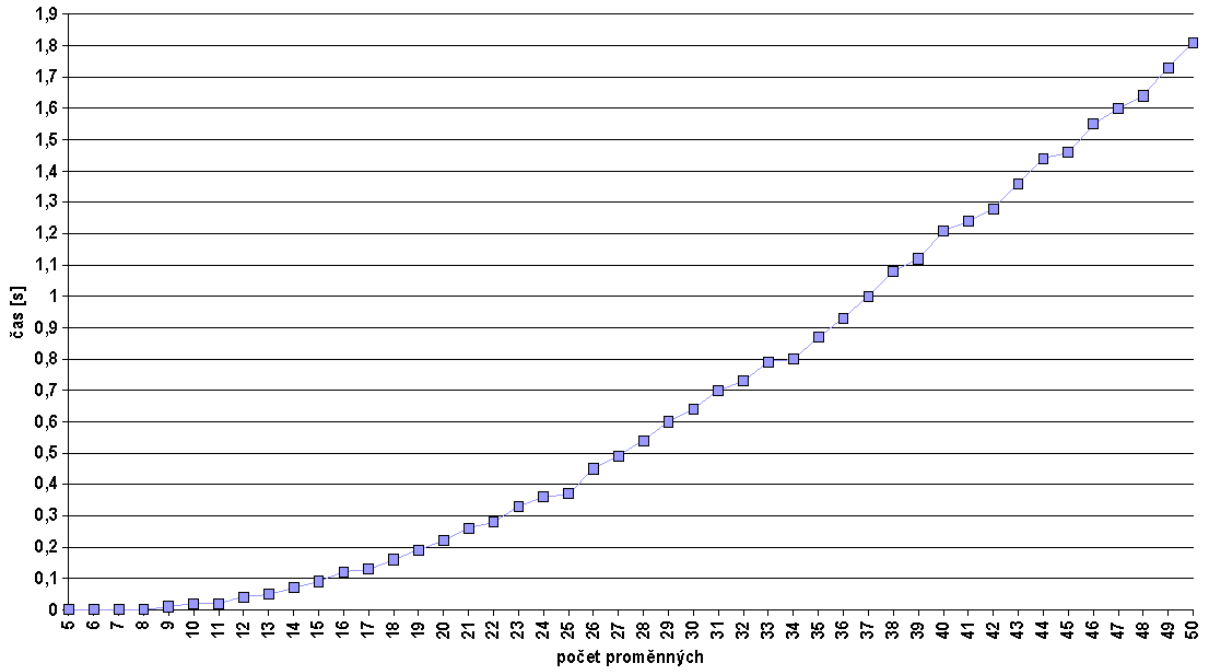
Obrázek 6.12: Vliv minimalizace podmnožinami na počet termů



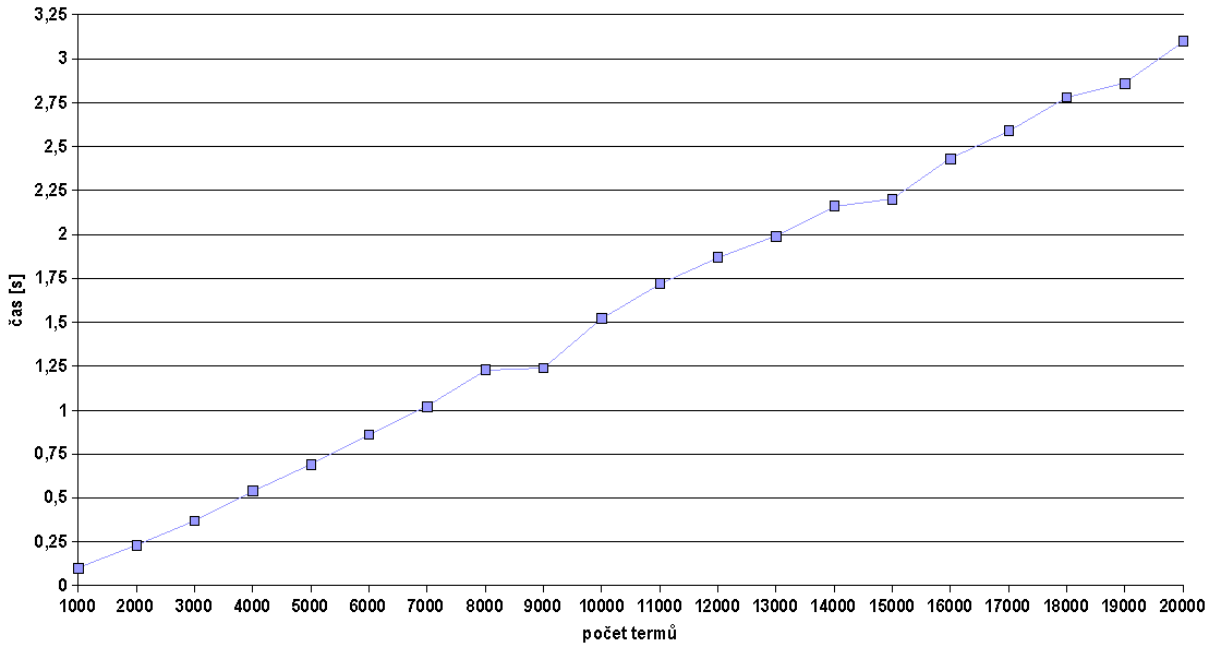
Obrázek 6.13: Závislost počtu termů po minimalizaci na poměru *don't care* ve vstupních termech



Obrázek 6.14: Závislost času výpočtu na poměru *don't care* ve vstupních termech



Obrázek 6.15: Závislost času výpočtu na počtu vstupních proměnných



Obrázek 6.16: Závislost času výpočtu na počtu vstupních termů

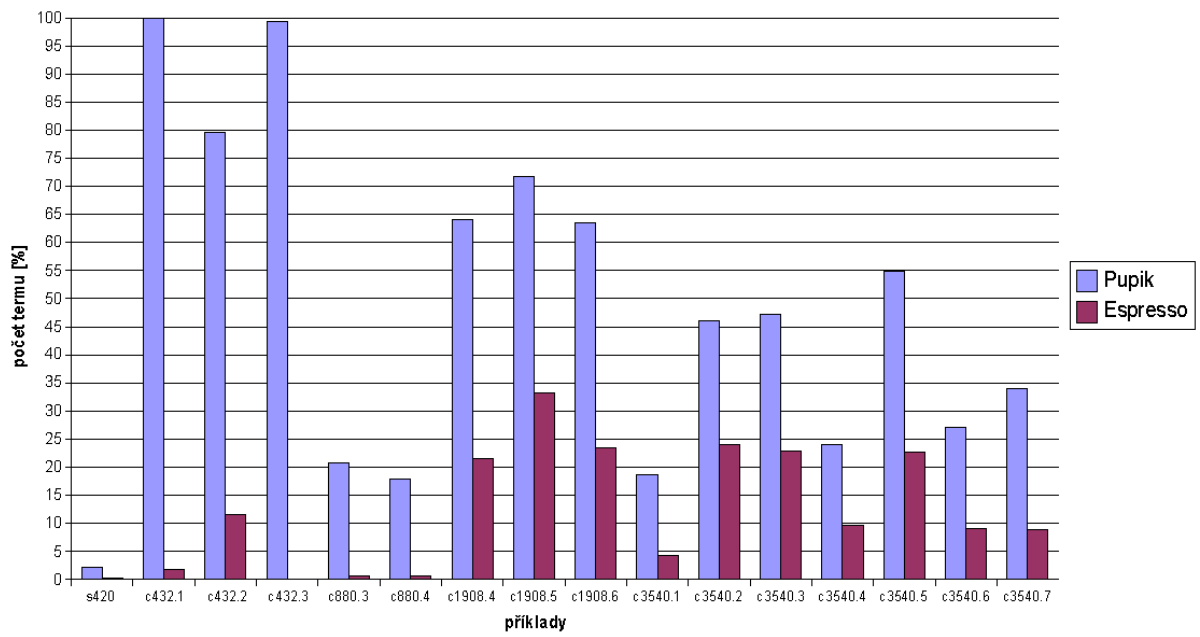
6.2.1 Pupik vs. Espresso

Na závěr ukáži srovnání mého minimalizačního algoritmu (pojmenujeme ho `pupik`) se systémem pro minimalizaci logických funkcí `espresso`. Opět vyberu několik příkladů ze sady [5] a porovnám výpočetní čas, počet termů po minimalizaci a efektivitu obou algoritmů. Aby mělo srovnání smysl, extrahuji z benchmarků několik vybraných výstupních funkcí a minimalizaci provedu odděleně pro každou z nich. Jelikož `espresso` obsahuje algoritmy pro minimalizaci vícevýstupových funkcí, což `pupik` nepodporuje (algoritmus minimalizuje jednotlivé funkce odděleně, nijak je nekombinuje), zkreslil by se tím výsledek srovnání. Abych toho dosáhl, provedu převod na dvouúrovňovou reprezentaci a uložím ji do souboru ve formátu PLA. Ten potom testovací program načte a předá jej algoritmu `pupik`, který provede vlastní minimalizaci. V tabulce 6.2 vidíme detailní přehled o naměřených hodnotách, tj. počet termů před a po minimalizaci a časy výpočtu. Časy jsou měřeny v sekundách. Pro větší přehlednost si naměřené hodnoty vyneseme do několika grafů, abychom mohli lépe nahlédnout na kvalitu testovaných algoritmů.

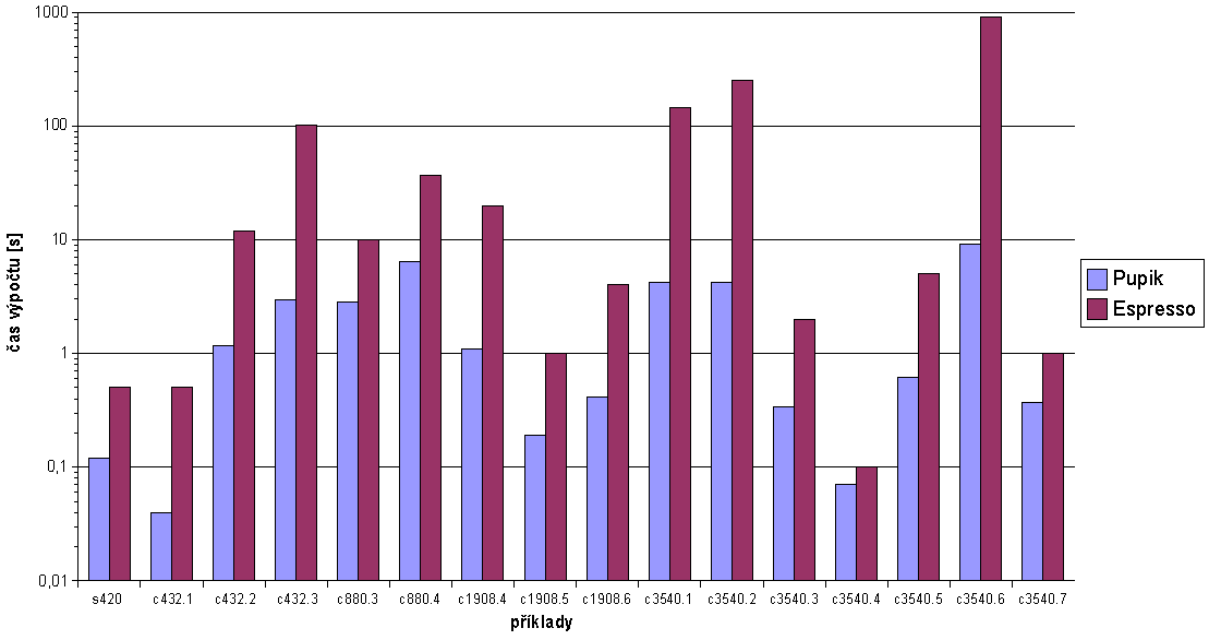
Tabulka 6.2: Naměřené hodnoty při srovnání minimalizačních algoritmů

benchmark	**termy**	**termy pupik**	**termy espresso**	**čas pupik**	**čas espresso**
s420	7146	160	17	0,12	0,5
c432.1	510	510	9	0,04	0,5
c432.2	20126	16008	2313	1,16	12
c432.3	42091	41814	64	2,97	102
c880.3	29577	6129	198	2,86	10
c880.4	67136	12046	440	6,44	37
c1908.4	32768	20990	7040	1,1	20
c1908.5	6464	4640	2144	0,19	1
c1908.6	13700	8704	3200	0,41	4
c3540.1	117160	21883	4961	4,2	146
c3540.2	42568	19620	10202	4,2	254
c3540.3	4464	2112	1022	0,34	2
c3540.4	1912	459	184	0,07	0,1
c3540.5	6657	3654	1516	0,62	5
c3540.6	159920	43442	14397	9,09	900
c3540.7	6933	2360	611	0,37	1

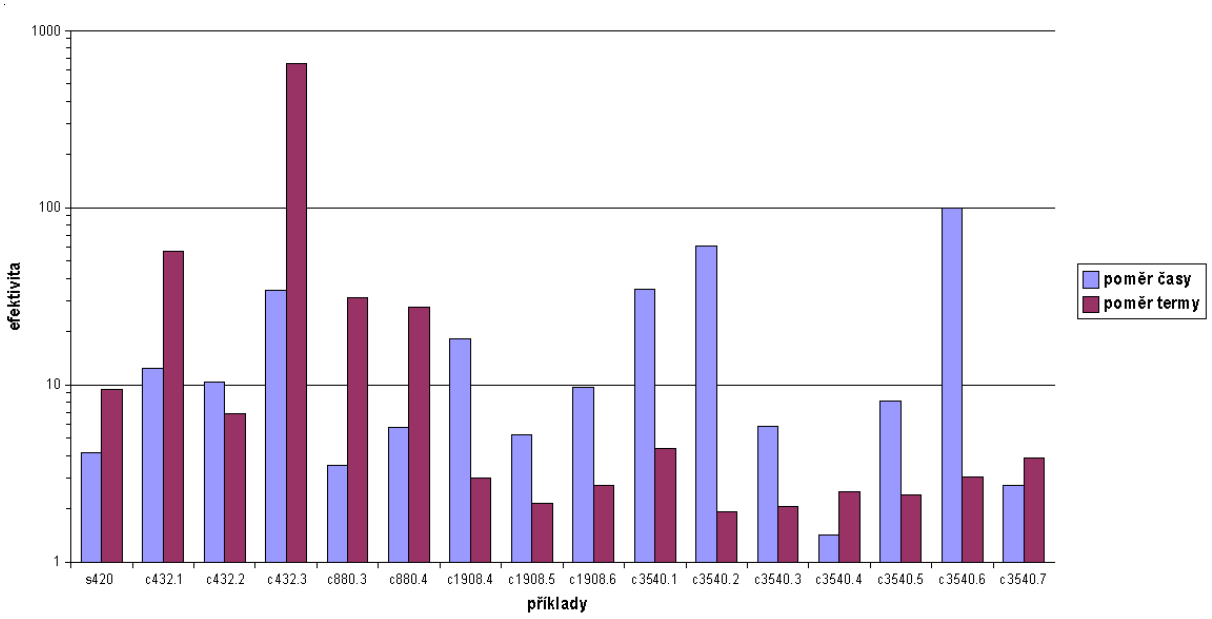
Na prvním grafu 6.17 je zobrazen počet termů po minimalizaci, vypočtený oběma algoritmy. Pro přehlednost je počet vyjádřen v procentech původního počtu termů před minimalizací. Vidíme, že `espresso` poskytuje větší kvalitu minimalizace ve všech případech. V příkladech c432 a c880, které minimalizačnímu algoritmu `pupik` nesedí, je rozdíl skutečně drtivý. V ostatních případech (c1908 a c3540) jsou ovšem výsledky vcelku vyrovnané. Pro vyhodnocení jak kvalitní tento výsledek ve skutečnosti je, ovšem potřebujeme vědět čas výpočtu, za který oba algoritmy došly k výsledkům. To vidíme na grafu 6.18. Kvůli velkým rozdílům mezi oběma algoritmy, bylo nutné nastavit na osu y logaritmické měřítko. Vidíme, že situace je oproti minulému grafu zcela opačná, `espresso` je vždy pomalejší, v některých případech i stonásobně. Je tedy potřeba zjistit, jaká je efektivita obou algoritmů. Pro vyhodnocení tohoto kritéria slouží graf 6.19. To, co tento graf zobrazuje, by se dalo slovně popsat takto: "**kolikrát** pomalejší bylo `espresso`, aby dosáhlo **tolikrát** lepšího výsledku". Jinými slovy, modrý sloupec zobrazuje čas výpočtu pro `espresso` v násobcích času výpočtu algoritmu `pupik` a fi-



Obrázek 6.17: Porovnání minimalizačních algoritmů podle počtů termů



Obrázek 6.18: Porovnání minimalizačních algoritmů podle času výpočtu



Obrázek 6.19: Porovnání minimalizačních algoritmů podle efektivity

alový sloupec zobrazuje počet termů po minimalizaci vypočtený algoritmem `pupik` v násobcích počtu termů vypočtených `espressem`. Jasně se zde ukazuje, že v příkladech, vhodných pro minimalizaci `pupikem` (c1908 a c3540) je efektivita algoritmu `pupik` místy i řádově vyšší! Naopak v příkladech nevhodných pro minimalizaci `pupikem` je efektivita espressa také místy řádově vyšší.

Velice zajímavé jsou tabulky 6.3 a 6.4. V tabulce 6.3 jsou zaneseny výsledky měření kvality minimalizace a času výpočtu na několika vybraných obtížných benchmarcích, se kterými mělo `espresso` již značné problémy. Vybrané benchmarky se skládají opět ze standardních víceúrovňových benchmarků, které byly převedeny na dvouúrovňové. Soubory s názvy ex17 a ex25 jsou logické funkce vygenerované PLA generátorem [12]. Jejich parametry byly nastaveny následovně: pro ex17 byl počet proměnných roven 15, poměr *don't care* byl 10% a počet termů byl nastaven na 50000; pro ex25 byl počet proměných roven 17, poměr *don't care* byl 15% a počet termů byl nastaven na 60000. Výsledky minimalizace těchto dvou funkcí `espressem` nejsou k dispozici, protože na nich `espresso` vůbec nebylo schopno k výsledku dojít. Běh programu byl ukončen operačním systémem, kvůli vyčerpání všech systémových prostředků. Ze zvědavosti jsem provedl ještě jeden experiment a jeho výsledky jsou vidět v tabulce 6.4. Zkusil jsem spustit minimalizaci `espressem` na PLA soubory, vypočtené algoritmem `pupik` z benchmarků z tabulky 6.3. Následně jsem sečetl časy výpočtu pro obě metody a spolu s počtem termů po minimalizaci je zanesl do tabulky. Výsledky jsou opravdu zajímavé. Je vidět, že kvalita minimalizace, tedy výsledný počet termů, je v podstatě stejná jako při běhu samotného `espressa`, ale čas výpočtu se výrazně snížil. V některých případech (c3540.1 a c3540.2) dokonce na třetinu! Minimalizace algoritmem `pupik` tedy dokázala `espressu` velmi výrazně pomoci k urychlení výpočtu, při zachování stejné kvality minimalizace. Toto je rozhodně zajímavé zjištění a ukazuje jednu z možných cest, kam by se mohl minimalizační algoritmus `pupik` ubírat. Kvalitou minimalizace sice pokročilým metodám zatím konkurovat nemůže, ale vzhledem ke své rychlosti jim může velice pomoci něčím, co by se dalo nazvat *preprocessing*.

Shrnu tedy nasbírané poznatky o navrhnutém minimalizačním algoritmu do několika závěrečných vět. Algoritmus se ukázal jako rychlý a poměrně slušně škálovatelný minimalizační nástroj.

Tabulka 6.3: Naměřené hodnoty při minimalizaci obtížných benchmarků

benchmark	**termy původně**	**termy pupik**	**termy espresso**	**čas [s] pupik**	**čas [s] espresso**
c880.1	212290	57853	15673	23,11	470
c3540.1	403298	101512	32455	32,9	4740
c3540.2	159920	43442	14397	9,09	900
ex17	50000	30688	N/A	1,5	N/A
ex25	60000	50439	N/A	20,54	N/A

Tabulka 6.4: Naměřené hodnoty při minimalizaci algoritmem pupik a následně espressem

benchmark	**termy původně**	**termy pupik+espresso**	**čas [s] pupik+espresso**
c880.1	212290	15673	305
c3540.1	403298	32396	1569
c3540.2	159920	14397	365
ex17	50000	N/A	N/A
ex25	60000	N/A	N/A

Závislost na počtu termů je lineární a závislost na počtu vstupních proměnných kvadratická. To se v testech potvrdilo. Kvalita výsledků je sice vždy horší než kvalita výsledků vypočtená `espreseem`, nicméně tato nevýhoda je vyvážena rychlostí, která se nejvíce projeví na velmi rozsáhlých funkcích s mnoha termy. Tam je již `espresso` velmi pomalé, kdežto `pupik` stále poskytuje velmi dobrý čas výpočtu. Navíc se ukázalo, že svoji rychlostí může pokročilým metodám minimalizace, které reprezentuje například `espresso`, výrazně pomoci ke snížení výsledného času výpočtu. Jedná se tedy o dobrý základ pro další vylepšování směrem k vyšší kvalitě výsledků, při zachování vysoké rychlosti.

6.3 Celkový test převodu víceúrovňové sítě na dvouúrovňovou

V této kapitole ukáži výsledky testování celého programu; porovnáme rychlost a kvalitu řešení s minimalizací pravdivostní tabulky i bez ní a připojím i srovnání se systémem pro syntézu logických obvodů `mvsis`. Měření bude probíhat na vybrané skupině standardních testovacích obvodů s následujícími parametry:

Tabulka 6.5: Parametry vybraných testovacích souborů

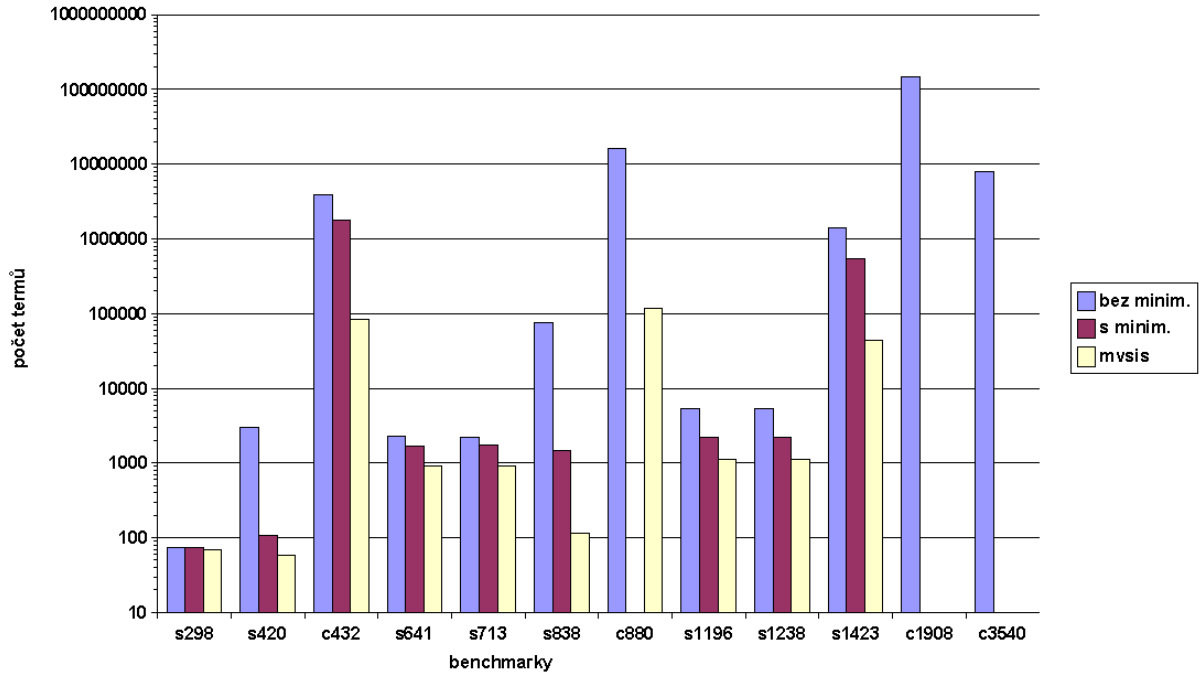
benchmark	počet vstupů	počet výstupů	počet hradel
s298	17	20	75
c432	36	7	159
c880	60	26	357
s1196	32	32	529
s1238	32	32	508
s1423	91	79	657
s420	35	18	196
s641	54	42	379
s713	54	42	393
s838	67	34	390
c1908	33	25	855
c3540	50	22	1647

V tabulce 6.6 vidíme výsledky měření.

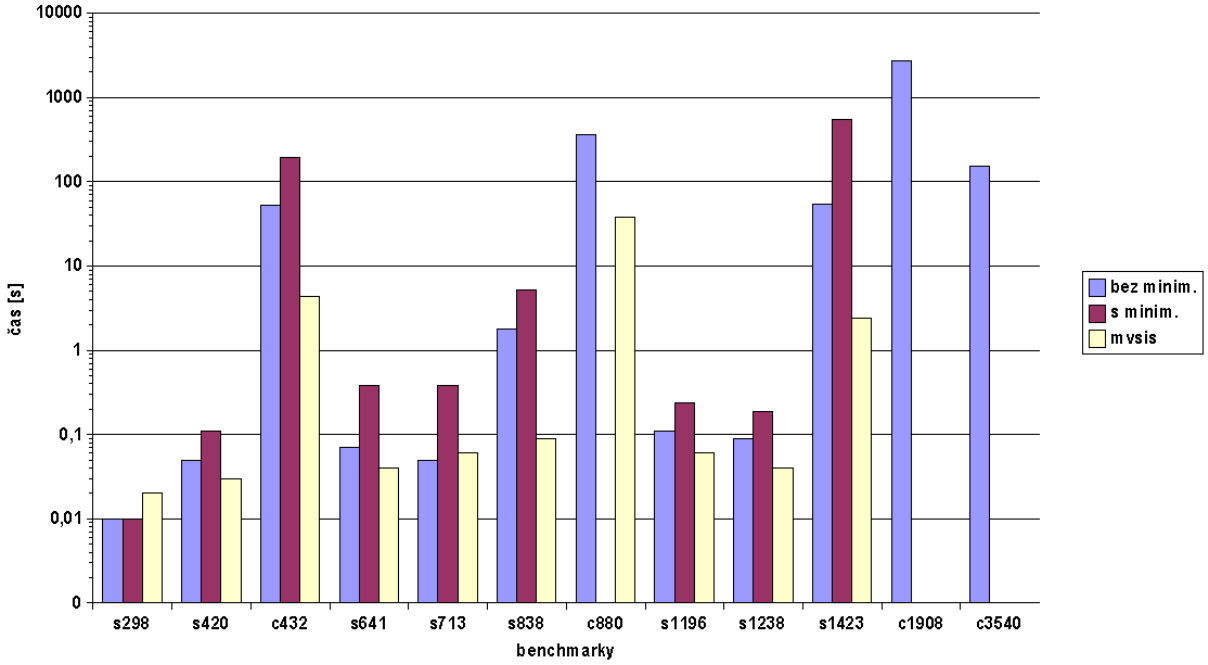
Tabulka 6.6: Naměřené hodnoty při srovnání minimalizačních algoritmů

benchmark	termy bez minim.	termy s minim.	termy mvsis	čas bez minim.	čas s minim.	čas mvsis
s298	74	73	69	0,01	0,01	0,02
s420	3020	106	58	0,05	0,11	0,03
c432	3922799	1790726	84242	52,85	194,05	4,38
s641	2294	1707	912	0,07	0,38	0,04
s713	2218	1762	912	0,05	0,38	0,06
s838	75743	1474	114	1,79	5,25	0,09
c880	16387376	N/A	116073	364,31	N/A	38,16
s1196	5396	2236	1118	0,11	0,24	0,06
s1238	5405	2236	1104	0,09	0,19	0,04
s1423	1392361	548435	43885	54,05	550,93	2,4
c1908	147923804	N/A	N/A	2745	N/A	N/A
c3540	7926401	N/A	N/A	151,25	N/A	N/A

V grafech 6.20 a 6.21 vidíme srovnání počtu termů výsledné pravdivostní tabulky a časy výpočtu vypočtené bez výstupní minimalizace, s výstupní minimalizací a `mvsisem`. Vidíme, že výsledky na méně rozsáhlých příkladech (s298, s641, s713, s1196, s1238) jsou poměrně vyrovnané, byť je `mvsis` vždy lepší. Na rozsáhlých příkladech (c432, c880, c1423) je však vidět již značný kvalitativní rozdíl, v případě obvodu c880 i několikařádový. Problém je zejména ve velikosti vytvořených ROBDD, kdy sice existují algoritmy pro jejich minimalizaci, nicméně, jak jsem



Obrázek 6.20: Porovnání collapsingu podle počtu termů



Obrázek 6.21: Porovnání collapsingu podle počtu termů

ukázal v kapitole 6.1, jejich primárním cílem je snížit na minimum počet uzlů v ROBDD, nikoli počet cest (termů). Ve vetších příkladech, kdy je počet generovaných termů již v řádu jednotek či desítek milionů, je již minimalizace velice obtížná, a to jak časově, tak paměťově. Proto nejsou v tabulce a grafu zaneseny počty termů generované po minimalizaci. Důležitý fakt je, že vzhledem k obrovskému množství termů, které je nutno uložit do souboru ve formátu PLA, zabere drtivou většinu času zápis na disk. Algoritmus samotný není zdaleka tak časově náročný, bohužel se však velikost výstupních souborů např. pro obvod c880 pohybuje v jednotkách GB, což je již příliš mnoho. Na opravdu velmi rozsáhlých obvodech s mnoha desítkami vstupů (c1908, c3540) se ovšem projevuje nevýhoda `mvsisu`: složitost je již tak vysoká, že časové nároky narostou nad únosnou mez. Např. výpočet pro obvod c3540 běžel déle než 70 hodin a výsledek stále nebyl vypočten. Pro rozsáhlé obvody tedy není `mvsis` použitelný.

7 Závěr

V průběhu řešení této práce se objevilo značné množství podnětů a problémů k řešení. Podařilo se navrhnout a implementovat algoritmus pro převod víceúrovňové logické sítě na dvouúrovňovou a vyhodnotit jeho vlastnosti. Pro řešení tohoto problému byly použity struktury známé jako binární rozhodovací diagramy. Bylo provedeno srovnání s jiným řešením stejného problému, systémem pro logickou syntézu `mvsis`. Jako vedlejší produkt, který nebyl původně součástí zadání, byl navrhnut a implementován zcela nový algoritmus pro minimalizaci logických funkcí. Tento algoritmus byl také důkladně analyzován, otestován a porovnán se systémem pro minimalizaci logických funkcí `espresso`.

Výsledky minimalizačního algoritmu jsou velmi povzbuzující; jeho předností je zejména rychlost, kdy překonává `espresso` ve všech případech. V některých případech je rozdíl dokonce řádový. Kvalita minimalizace sice kvalit `espressa` nedosahuje, nicméně v nejdůležitějším kritériu, efektivitě výpočtu, je ve většině případů lepší. Do budoucna lze ještě pracovat jak na rychlosti (naznačené vylepšení propagací sloučení listů), tak na kvalitě minimalizace. To by mohlo zahrnovat např. zakomponování absorpce negace a hlavně implementaci expanse, která je standardní součástí ostatních minimalizačních algoritmů. Jednou z oblastí využití minimalizačního algoritmu se může stát i tzv. preprocessing, kdy může algoritmus, díky své rychlosti, pomoci snížit výpočetní čas pokročilých minimalizačních nástrojů. Celkově jde o slibný nástroj, na kterém lze do budoucna postavit kvalitní systém pro minimalizaci logických funkcí.

Srovnání vlastního převodu víceúrovňové logické sítě na dvouúrovňovou předcházelo vyhodnocení nejlepších metod pro minimalizaci ROBDD. Jako nejlepší se ukázaly metody postavené na siftingu (sifting, symmetric sifting, group sifting), které poskytovaly velmi dobrou kvalitu řešení při nízké časové složitosti. S využitím těchto metod bylo potom provedeno finální srovnání našeho převodu s převodem poskytovaným systémem `mvsis`. V tomto srovnání se ukázal `mvsis` jako pokročilejší nástroj, kdy i pro středně rozsáhlé obvody dokázal najít v nízkém čase kvalitní řešení. Ve velmi rozsáhlých obvodech se ovšem `mvsis` ukázal jako nepoužitelný a na rozdíl od mého převodu, nedošel v akceptovatelném čase k žádnému řešení. V méně rozsáhlých obvodech jsou oba přístupy srovnatelné jak časově, tak kvalitou řešení.

Z hlediska druhého cíle práce, experimentu s novým přístupem pro řešení převodu mezi logickými sítěmi, byla práce úspěšná, jelikož se mi podařilo pojmenovat hlavní problémy, které tento přístup implikuje a jejich možná řešení. Do budoucna z této práce tedy plyne nutnost vytvoření algoritmů pro minimalizaci ROBDD, které se, spíše než na minimalizaci počtu uzlů, budou soustředit na minimalizaci počtu termů jimi generovaných. Druhou cestou, kterou by mohla pokračovat následná práce, je způsob generování termů od listů ke kořeni ve spojení s booleovskou minimalizací, která by mohla razantně snížit jejich počet. K tomuto je ovšem třeba prozkoumat možnosti minimalizace a vyhodnotit, zda je možné v akceptovatelné časové složitosti dospět k řešení. Dále je třeba nalézt nebo vyvinout vhodný nástroj pro tento způsob generování termů, jelikož zde použitý balík CUDD tento přístup neumožňuje.

8 Seznam literatury

[1] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for obdds. *International Workshop on Logic Synthesis*, 1995.

[2] R. K. Brayton. Symbolic boolean manipulation with ordered binary decision diagrams. 1992.

[3] R. K. Brayton and et al. Mvsis. `http://www-cad.eecs.berkeley.edu/mvsis/`.

[4] R. K. Brayton and et al. Logic minimization algorithms for vlsi synthesis, 1984. Kluwer Academic Publishers.

[5] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortan. *Proc. of International Symposium on Circuits and Systems*, pages 663–698, 1985.

[6] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of obdds. *International Workshop on Logic Synthesis*, 1995.

[7] P. Fišer and J. Hlavička. Boom - a heuristic boolean minimizer. *Computers and Informatics, Vol. 22*, pages 19–51, 2003.

[8] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. *In Proceedings of the European Conference on Design Automation*, pages 50–54, 1991.

[9] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. *In Proceedings of the International Conference on Computer-Aided Design*, pages 472–475, 1991.

[10] V. Kozák. Převaděč formátů pro specifikaci logických obvodů. *DP ČVUT, FEL*, 2005.

[11] P. Kubalík, P. Fišer, and H. Kubátová. Minimization of the hamming code generator in self checking circuits. 2004.

[12] T. Měchura. Parametrizovaný generátor náhodných booleovských funkcí. *BP ČVUT, FEL*, 2006.

[13] S. Panda and F. Somenzi. Who are the variables in your neighborhood. *In Proceedings of the International Conference on Computer-Aided Design*, pages 74–77, 1995.

[14] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. *In Proceedings of the International Conference on Computer-Aided Design*, pages 628–631, 1994.

[15] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. *In Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, 1993.

9 Obsah přiloženého CD

Přiložené CD obsahuje v kořenovém adresáři následující podadresáře:

- text - obsahuje text diplomové práce v pdf formátu
- collapsing - obsahuje program na převod víceúrovňové sítě na dvouúrovňovou ve spustitelné podobě pro 32-bitový OS GNU/Linux i v podobě zdrojových kódů; navíc obsahuje několik příkladů obvodů v obou reprezentacích (před i po převodu)
- pupik - obsahuje minimalizační algoritmus v samostaném programu schopném načítat soubory ve formátu PLA; také je na CD uložen v podobě spustitelného souboru a v podobě zdrojových kódů spolu s několika příklady
- cudd - obsahuje balík CUDD ve verzi 2.4.1 ve formě zdrojových kódů