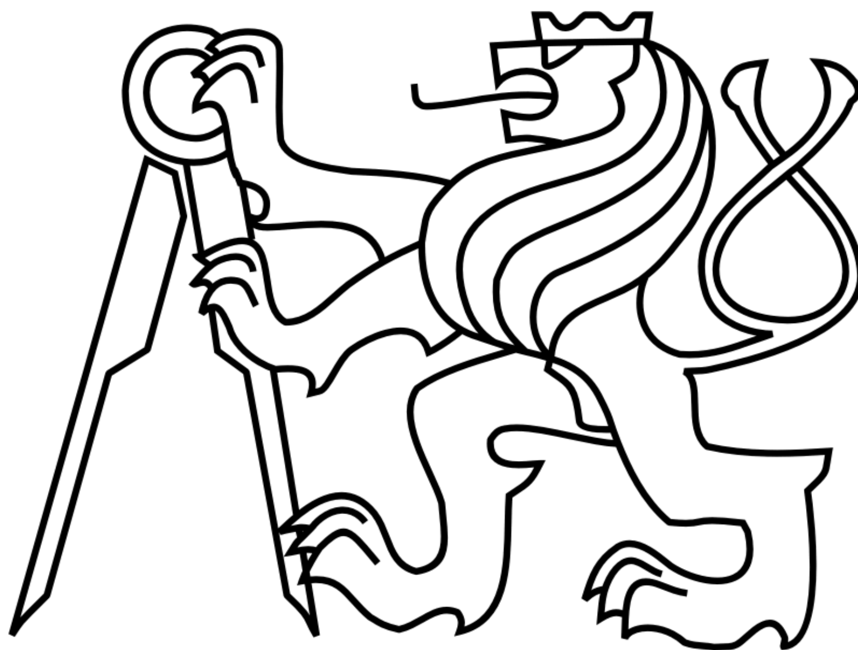
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ
FAKULTA ELEKTROTECHNICKÁ
KATEDRA POČÍTAČŮ

BAKALÁŘSKÁ PRÁCE



ESOP MINIMALIZACE LOGICKÝCH FUNKCÍ

LUKÁŠ PATERA
Studijní program : Elektrotechnika a informatika, obor výpočetní technika
Vedoucí bakalářské práce : Ing. Petr Fišer, Ph.D.
2009

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů.
Souhlasím se zapůjčování, práce a jejím zveřejňováním.

Lukáš Patera
V Praze dne: 31. 7. 2007

Název práce : ESOP MINIMALIZACE LOGICKÝCH FUNKCÍ
Autor : Lukáš Patera
Katedra : Katedra počítačů ČVUT FEL Praha
Vedoucí : Ing. Petr Fišer, Ph.D.
E-mail vedoucího: fiserp@fel.cvut.cz

ANOTACE : Cílem této práce je popsat dosavadní metody používané k minimalizaci logických funkcí ve tvaru ESOP, tedy využívajících zápis pomocí XORu logického součinu vstupních proměnných, popsat jejich princip a výhody a nevýhody. Metody a přístupy k řešení tohoto problému dále popsané jsou řazené chronologicky podle toho kdy se ten který přístup objevil. Poslední část práce tvoří minimalizátor ESOP funkcí který je navázaný na stávající jádro programu BOOM vyvíjeného Ing. Petrem Fišerem, Ph.D.

Klíčová slova : ESOP, minimalizace, BOOM

Title : minimalization of ESOP funcitons
Author : Lukáš Patera
Department : Department of computer science CTU Praha
Supervisor : Ing. Petr Fišer, Ph.D.
Supervisor’s e-mail : fiserp@fel.cvut.cz

ABSTRACT : The aim of this paper is to describe methods used to minimalisaton of ESOP functions, therefore ones that use XOR of products instead of normal OR and to describe their principles, advantages and disadvantages. Methods and approaches to solve this problem are ordered chronologicaly from older to new ones. Last part of this work is an ESOP minimizer based on existing BOOM engine developed by Ing. Petr Fišer, Ph.D.

Keywords : ESOP, minimalization, BOOM

OBSAH

1. Úvod 6
2. Základní pojmy 7
3. Zjednodušování výrazů pomocí přepisovacích pravidel 8
3.1. Algoritmus EXMIN 8
4. Zjednodušování pomocí iterované transformace krychlí 11
4.1 Algoritmus EXORCISM 11
4.2 Algoritmus DECENTUA 14
5. Typy souborů používané pro vyjádření funkcí ve výše uvedených algoritmech 17
5.1 PLA 17
5.2 BLIF 18
6. Porovnání jednotlivých algoritmů ve zkušebních obvodech 19
6.1 Zkušební obvody 19
6.1.1 ISCAS zkušební obvody 19
6.1.2 ITC zkušební obvody 19
6.2 Výsledky algoritmů 19
6.2.1 DECENTUA / EXORCISM 19
6.2.2 EXORCISM / EXMIN / MINT 20
7 . Popis vlastního algoritmu 21
7.1 Engine BOOM 21
7.2 Popis implementace 21
7.3 Dosažené výsledky 24
8. Závěr 25
Reference 26

1. ÚVOD

Obvody užívající XOR hradla, od svého objevení v r. 1927 I. I. Zhegalkinem v roce 1927 a svého pozdějšího znovuobjevení S. M. Reedem a D. E. Millerem nalezly několik zajímavých uplatnění. Ačkoliv nejsou tak moc rozšířené jako běžně užívané obvody AND a OR, například pro aritmetické funkce, funkce používané v telekomunikacích např. na korekci signálů jsou tyto obvody menší než jejich protějšky z AND-OR funkcí.

Dnes se tyto obvody masově nenasazují, a to hlavně kvůli praktické implementaci XOR hradla které je téměř dvakrát vetší než implementace NAND a NOR hradel a také kvůli větší odezvě. I přes tyto překážky jsou obvody realizované pomocí XOR hradel často rychlejší a menší než standardní typy obvodů. Historicky to ale také vede k tomu že relativně málo algoritmů skutečně počítá s použitím těchto obvodů ve své vnitřní reprezentaci.

Současný výzkum v této oblasti ovšem objevil nové oblasti možného využití těchto obvodů. Experimentuje se s nimi jako součást reverzibilních obvodů a na poli kvantových počítačů.

V mé práci bych se chtěl zaměřit na způsoby optimalizace těchto obvodů a funkcí které je vyjadřují, shrnout stávající přístupy k tomuto problému a navrhnout program, který by implementoval některý z uvedených postupů.

Prvním a nejstarším přístupem k optimalizaci je zjednodušování výrazu pomocí přepisovacích pravidel. Tento přístup má velké výhody ve své jednoduchosti na pochopení a snadnou modifikaci a optimalizaci na řešení konkrétních úloh.
Tento princip má ovšem i své velké nevýhody, které jsou důvodem proč se v praxi příliš nepoužívá. Největší nevýhodou toho mechanismu je to že vstupní funkce nesmí obsahovat žádné nedefinované stavy, pravidla s nimi potom nepracují tak jak by měla. Vzhledem k tomu ze valná většina prakticky řešených problémů má své don’t cares, tak tento přístup k optimalizaci zklamává a je třeba buď nasadit jiný algoritmus, nebo tento problém nějak ošetřit. Každopádně při nasazení na zjednodušování poskytuje vcelku efektivní výsledky za přijatelnou dobu. Problémem je hlavně doba výpočtu, nové algoritmy jako např. EXORCISM mají podobné a spíše lepší výsledky za kratší dobu. Zůstává tak jen výhoda jednoduchosti, ale tento přístup nachází své uplatnění v některých složitějších algoritmech na optimalizaci které ho používají na získání mezivýsledků. Asi nejpoužívanějším algoritmem z této kategorie je algoritmus EXMIN.

Druhým a asi nejrozšířenějším principem zjednodušovacích algoritmů je zjednodušování transformací krychlí v iterovaném cyklu. Tento princip se opírá o takzvané Karnaughovy mapy (dále K-mapy) spočívá v analýze stávajících krychlí ve funkci (krychlí je myšlen obrazec vystihující term v K-mapě) a rozhodnutí, zda krychle může či nemůže být sloučena s jinou, což by vedlo ke zmenšení velikosti termů, a k úplné eliminaci některých z nich. Nejrozšířenějším algoritmem využívající tento princip je EXORCISM. Jeho největší výhodou je to že si dokáže poradit s jakýmkoliv typem vstupu včetně případných don’t cares. Je také velmi rychlý a výsledky jím vytvořené jsou více než přijatelné.

Další záležitostí o které je nutno se zmínit jsou zkušební obvody. Na nich se testují výsledné velikosti funkcí, rychlosti odezvy, doba výpočtu zkrátka cokoliv relevantního aktuálnímu problému, ale z důvodů opakovatelnosti měření je důležité mít ustálené sady funkcí, kterými se bude dát změřit výsledek jednotlivých přístupů k problému.

2. Základní pojmy

BOOM - v dalším textu odkazuje na minimalizátor vyvíjený na ČVUT FEL Praha, více viz kapitola která ho popisuje a materiál [8]

XOR - z matematiky exklusivní disjunkce nebo nonekvivalence, je logická funkce jenž nabírá pravdivé hodnoty právě tehdy pokud se obě vstupní hodnoty navzájem liší. Odpovídá binárnímu sčítání bez přenosu.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Literál - jedna ze vstupních proměnných

Term - viz Krychle

Krychle - v této práci a v pracích souvisejících jde o logický součin literálů.

Vzdálenost krychlí - počet literálů ve kterých se dvě krychle liší.

ESOP - je XOR žádné nebo více krychlí. V případě že máme pravdivostní tabulku funkce vytváří se stejně jako se vytváří funkce vyjádřená běžnými ORy.

Inset - taková kombinace vstupních proměnných která v pravdivostní tabulce vede k výstupu hodnoty true

Offset - taková kombinace vstupních proměnných která v pravdivostní tabulce vede k výstupu hodnoty false

Don't care -3tí možný stav vstupní nebo výstupní proměnné, ve kterém proměnná nijak neovlivňuje výsledek nebo výsledek má nezajímavou hodnotu. Pro potřeby algoritmů se hodnota tohoto stavu může libovolně měnit podle aktuální potřeby.

Pokrytí funkce - označení přejaté z [5] – v dalším textu je tak označována funkce která již byla převedena do vnitřní reprezentace ve tvaru ESOP v chvílích kdy se bavíme o funkci jako celku

Podfunkce - Pro funkci N proměnných f a proměnnou x, podfunkce f jsou definovány jako funkce s $x = 0$ a $x = 1$ a značí se jako $f_{\{0\}}$ a $f_{\{1\}}$. Funkce $f_{x:\{0,1\}}$ je potom definována jako $f_{\{0\}} \oplus f_{\{1\}}$.

Shannonova exanze, Daviova pozitivní a negativní expanze
- matematické axiomy používané v následujícím textu opírající se o podfunkce, definice je následující :

- Shannonova expanze $f = \overline{x_1} f^0 \oplus x_1 f^1$
- Positivní Daviova expanze $f = f_0 \oplus x_2 f^2$
- Negativní Daviova expanze $f = f^1 \oplus \overline{x_1} f^2$

Kde : $f^0 = f(0, x_2, ...x_n)$

$f^1 = f(1, x_2, ...x_n)$

$f^2 = f^0 \oplus f^1$

A kde $\oplus$ je sčítání v modulo 2.

3. Zjednodušování pomocí přepisovacích pravidel

3.1 EXMIN II

Prvním z nich který popíši je zjednodušování založené na přepisování termů funkce pomocí matematických pravidel, jehož zástupcem je algoritmus EXMIN II [1] .
Tento algoritmus nezaručuje získání minimálního tvaru funkce, pouze produkuje rozumně velké výrazy které mohou, ale nemusí vést k absolutně minimálnímu výsledku. Vezmeme základní sestavu pravidel převzatou z [2] a doplněnou o některá další pravidla [1] :

$x \oplus \bar{x} \rightarrow 1$	S1
$x \oplus 1 \rightarrow \bar{x}$	S2
$xy \oplus \bar{y} \rightarrow 1 \oplus \bar{x}y$	S3
$xy \oplus \bar{x}\bar{y} \rightarrow \bar{x} \oplus y$	S4
$xy \oplus \bar{x}\bar{y} \rightarrow x \oplus \bar{y}$	S5
$1 \oplus 1 \rightarrow 0$	S6
$x \oplus 0 \rightarrow x$	S7

pozn. Pro aplikaci jakýchkoliv z těchto pravidel je též nutná platnost asociativního a distributivního zákona.

Výkon tohoto algoritmu je možno zvýšit zvětšením počtu pravidel, která je schopen použít. Tohoto lze dosáhnout buď experimentálně, nebo pomocí tzv. L-transformací.

Jejich princip je takový, že se použije stávající, často viděný řetězec, vyjmou se z něho ty termy, jenž obsahují x, které se z nich vytkne a $\bar{x}$, které se též vytkne. Dostaneme tedy funkci ve tvaru $x \cdot f_1 \oplus \bar{x} \cdot f_2 \oplus f_3$ kde f1 reprezentuje všechny termy z vytknutým x, f2 všechny termy s vytknutým $\bar{x}$ a f3 všechny termy které neobsahovaly ani x ani $\bar{x}$. Tento postup je blíže popsán i s teoretickým výkladem v [1], kde je popsán i algoritmus který tohoto využívá a který si popíšeme později. Potom se x jednoduše vymaže ze všech termů, jenž ho obsahují a zároveň se přidá do termů funkce f3 (tj. do těch, jenž neobsahovaly ani x ani $\bar{x}$.)

Toto otevírá cestu zmíněnému algoritmu na zjednodušování kde samotné zjednodušení je provedeno pomocí pravidel na menší podvýrazy vytvořené rekurzivním voláním sebe sama.

Navržený postup [1] je následující :

Funkce_min P (**funkce_vstup**, [x])
{
1) možná jdi na krok 6
2) vyber proměnnou x
3) **funkce_vstup1** = P (**funkce_vstup**, x) ; **funkce_vstup2** = P (**funkce_vstup**, xnon) ; **funkce_vstup** = g1 $\oplus$ g2;
4) aplikuj pravidlo S1 na 1 pár z **funkce_vstup1** a 1 pár z **funkce_vstup2** (tou dobou už spojené v **funkce_vstup**)
5) aplikuj pravidlo 2S na co nejvíce termů v **funkce_vstup**
6) pokud možno minimalizuj **funkce_vstup**
7) return (**funkce_vstup**)
}

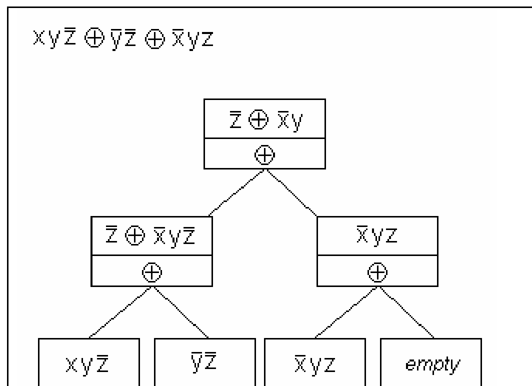
V tomto algoritmu je třeba specifikovat podmínky kdy se budou jednotlivé kroky vykonávat. Tento nedeterministický algoritmus by bylo možné alespoň částečně (neznáme konkrétní implementaci minimalizační funkce) determinizovat následujícím způsobem :

```
Funkce_min P (funkce_vstup,parametr_zanoření , [x] )
{
 if ( parametr_zanoření > požadovaný )
   {
   x = rand (funkce_vstup);
          // vybrání náhodného x z funkce
   funkce_vstup1 = P (funkce_vstup, parametr_zanoření-1, x ) ;
   funkce_vstup2 = P (funkce_vstup, parametr_zanoření-1,xnon);
   funkce_vstup = funkce_vstup1⊕ funkce_vstup2;
while (konec = false)
     {
     konec = S1 (funkce_vstup)
     // funkce vyhledá dvojice podle svého pravidla a to
použije
     }
while (konec = false)
     {
     konec = S2(funkce_vstup)
          // totéž co S1
     }
}
funkce_vstup  = minimalizuj (funkce_vstup)
pozn. ve zdroji je použit algoritmus exmin I [1]
return (funkce_vstup)}
```

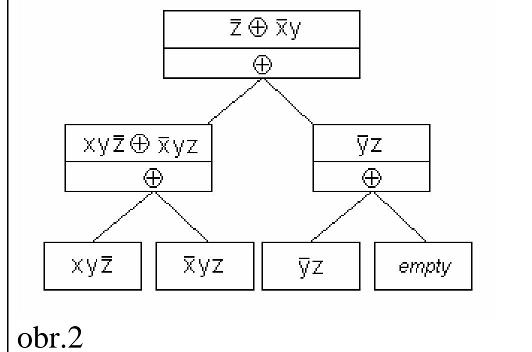
Jak je vidět algoritmus pomoci pravidel rekurzivním voláním sebe sama rozloží výraz na co nejjednodušší součásti na které se snaží aplikovat minimalizační pravidla která jsou definována ve funkci minimalizuj.

Tento algoritmus kopíruje princip který je použitý v algoritmu EXMIN II [1]. V originálním znění jako samotná minimalizační funkce používá algoritmus EXMIN I, což je důvod proč bych se do jeho implementace nepouštěl, protože by pravděpodobně bylo velmi náročné vytvořit minimalizátor stejných nebo podobných kvalit jako je ten používaný v originále.

Jak je vidět na rozdíl od níže uvedeného algoritmu Exorcism se vstupní ESOP postupně přepisuje na minimalizovanou podobu a nezkouší se možnosti které by mohly vzniknout pokud by se daný výraz přepsal jinak než je v pravidlech a nebo třeba dočasně zhoršit parametry výrazu aby se v dalším kroku dosáhlo výraznějšího zjednodušení.



obr.1



obr.2

Jak je vidět, původní výraz se rozloží na tolik bloků, kolik je termů. Ty se pak dále zakládají do binárního stromu, kde dochází k jejich postupnému zjednodušení. Zde na obrázku je uveden příklad, kde je na první dvojici aplikováno pravidlo S3, druhá „dvojice" je přepsána výše a výsledek je přepsán do kořene stromu, a pokud budeme korektně používat daná pravidla, lze se dobrat optimálnějšímu tvaru funkce.

Na obrázku 1 je ukázán příklad funkce a je zobrazena její vnitřní reprezentace. Zde je funkce naskládána do stromu jednoduše tak jak přišla.Jak je ovšem vidět na obrázku č.2 pokud by funkce byla zapsána jiným způsobem, v nižších větvích stromu se nezjednoduší a bude kontrolována ve své objemné formě dokud nepřijde jiný výraz z druhé strany větve který umožní její zjednodušení.

Je otázkou pro experiment, zda je výhodnější funkce nechávat v neoptimalizovaném tvaru (dejme tomu v každém kroku použít max. 1 optimalizující pravidlo) nebo při příchodu funkci optimalizovat dokud existuje pravidlo které se dá použít. Výhodou prvního přístupu by byla daleko menší časová náročnost pro špatně rozložitelné a přitom rozsáhlé funkce protože místo kontrol při kterých by se zjistilo ze nelze nic by se něco aktivně provádělo a by se prováděné operace více rozložily mezi jednotlivá patra stromu, čímž by se omezilo množství přístupů do paměti a tím i čas.

Samozřejmě při tomto přístupu by, pokud by byla funkce dobře rozložitelná, došlo nárůstu počtu oprav a kontrol tou dobou již rozsáhlého kořene, kam by se přesunuly veškeré předchozí neprovedené ale známé modifikace které by na sobě vzájemně závisely – viz obr 2 kde je k použiti pravidla S1 nejprve nutno použít pravidlo S3. To by ve výsledku zvýšilo výpočetní čas u tohoto druhu funkcí (pokud by šlo o modifikace typu úprav úplné formy výrazu tak by došlo paradoxně k jeho snížení protože jednotlivé modifikace by na sobě nebyly závislé).

Moje implementace tohoto řešení by se pravděpodobně opírala o právě tento algoritmus který by rekurzí vygeneroval příslušný strom a zkoušel ho pomocí výše uvedených pravidel zjednodušovat. Celková míra zjednodušení by byla velmi závislá na kvalitě základního zjednodušovacího algoritmu. Není ani třeba aby produkoval naprosté minimum proměnných v termu, ale hlavně aby dával dobrý poměr zjednodušení/čas, protože se jeho služeb bude využívat velmi často a pokud nedokáže některé neefektivnosti zachytit v nižších úrovních stromu, je možné že je zachytí výše. K tomu by přispělo co největší množství pravidel které by se v tomto algoritmu vyskytovaly. Pro tento způsob implementace jsem se však nakonec nerozhodl vzhledem k tomu že by bylo velmi obtížné vyrobit dostatečně efektivní minimalizátor.

4. Zjednodušování pomocí iterované transformace krychlí

Než začnu popisovat princip algoritmů založených na principu transformace krychlí, je zapotřebí definovat několik termínů, které budu v textu potřebovat.

Krychlí je v textu myšlen výraz, který se v K-mapě zakresluje jako obrazec velikosti 2^n a ohraničuje jedno spojení proměnných ANDy které je od dalších podobných odděleno XORy.

Vzdáleností krychlí označuji číslo, v kolika proměnných se dvě krychle od sebe liší. Tzn pokud v jedné krychli je proměnná A v přímé podobě, v další krychli v negované podobě, zvětší se vzdálenost krychlí o 1. Totéž nastává v případě že je proměnná A v první krychli v přímé podobě a ve druhé není vůbec (don't care) atd.

4.1 EXORCISM

Exorcism je první z algoritmů na minimalizaci XOR logických funkcí využívající princip transformace krychlí, který zde zmíním. Tento algoritmus nezaručuje nalezení absolutního minima funkce, ať už z pohledu minimálního počtu literálů nebo krychlí, ale lze s vysokou pravděpodobností říci, že se bude v alespoň jednom z kritérií minimální vzhledem k ostatní používaným algoritmům.

Algoritmus na minimalizaci EXORCIMus [3] poté využívá následujících definic k ověření minimálnosti funkce : Funkce je minimální právě tehdy, pokud všechny krychle mají vzájemné vzdálenosti 2 a větší. Absolutním minimem se funkce stává tehdy, pokud obsahuje nejmenší množství krychlí ze všech minimálních funkcí.

Dále se postupuje podle následujícího algoritmu [3] :

Popis – proměnná pro uchovávání vzorce funkce
F – funkce jako vstupní proměnná, předpokládá se že bude zadána v upraveném tvaru jako ESOP ne jako čisté PLA

```
esop Heuristická_Minimalizace ( esop f, int počet_iterací )
{
esop popis = Generuj_startovní_pokryti( F );
while ( Q > 0 ) {
     Resetuj_páry(popis);
     Do
     {Do
          {
          popis = Agresivní_Minimalizace(popis);
//tato funkce se snaží zmenšit počet krychlí v popisu funkce
          }
          while (je co zlepšovat);
          popis = poslední_pokus(popis);
// tato funkce právě obdržela lokální minimum získané
agresivní optimalizací. Tato mohla zvýšit počet literálů a
proto je vhodné před další smyčkou znovu projet popis a zkusit
tento počet zmenšit.
     }

     while (je co zlepšovat);
     počet_iterací = počet_iterací -1;
     }
popis = čistící_ Minimalizace (popis); //
return popis;}
```

V prvním kroku algoritmus vygeneruje základní reprezentaci funkce z jejího popisu. K této reprezentaci používá BDD (Binary Decision Diagram) do kterého se vloží startovní pokrytí. Během toho se na nalezené krychle používají následující z 3 možné vzorce pro rozšíření výrazu (pro přehlednost uvádím jestě jednou) :

- Shannon - $f = \overline{x_1} f^0 \oplus x_1 f^1$ Kde : $f^0 = f(0, x_2, ...x_n)$
- Positive Davio $f = f_0 \oplus x_2 f^2$ $f^1 = f(1, x_2, ...x_n)$
- Negative Davio $f = f^1 \oplus \overline{x}_1 f^2$ $f^2 = f^1 \oplus f^2$

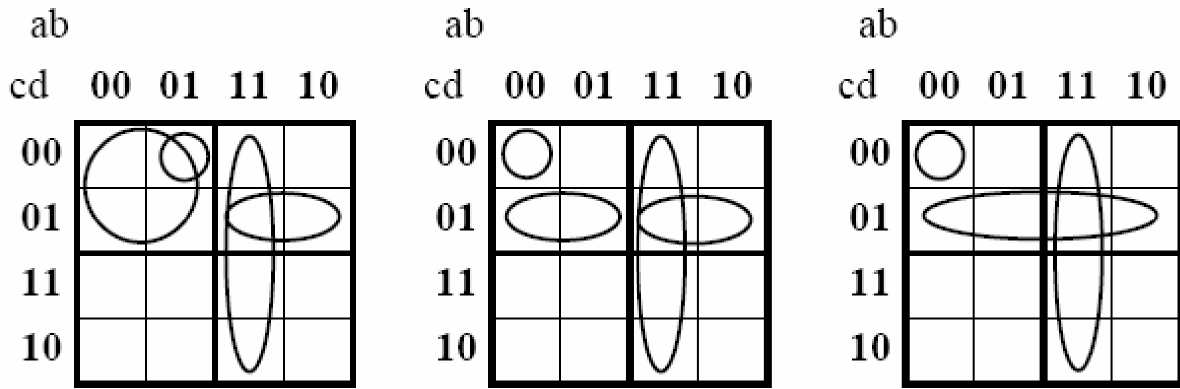
Kde $\oplus$ je sčítání v modulo 2.

A poté vyhodnotí, zda použití vzorce vede ke snížení počtu krychlí a nebo ne. Snížení počtu krychlí v této fázi vede zároveň i ke snížení počtu literálů. Toto ohodnocování se zadává do BDD diagramu který funkci reprezentuje a ve druhé fázi se tento diagram prochází a zjišťuje se, které možnosti (reprezentované větvemi stromu diagramu) vedou ke snížení celkového počtu krychlí. Přesnější popis algoritmu se nachází v [3].

V dalším kroku se provádějí samotné minimalizační operace. Algoritmus potom využívá Exorlink operací uvedených poprvé v [4] . Hlavní idea těchto operací je nahrazení jedné sadou jiných, ale tak, aby došlo ke zmenšení jejich velikosti a zvýšení úspornosti i za cenu dočasného zvýšení počtu literálů nebo krychlí. Na základě vzdálenosti jednotlivých krychlí od sebe potom algoritmus vybírá jednu z následujících strategií :

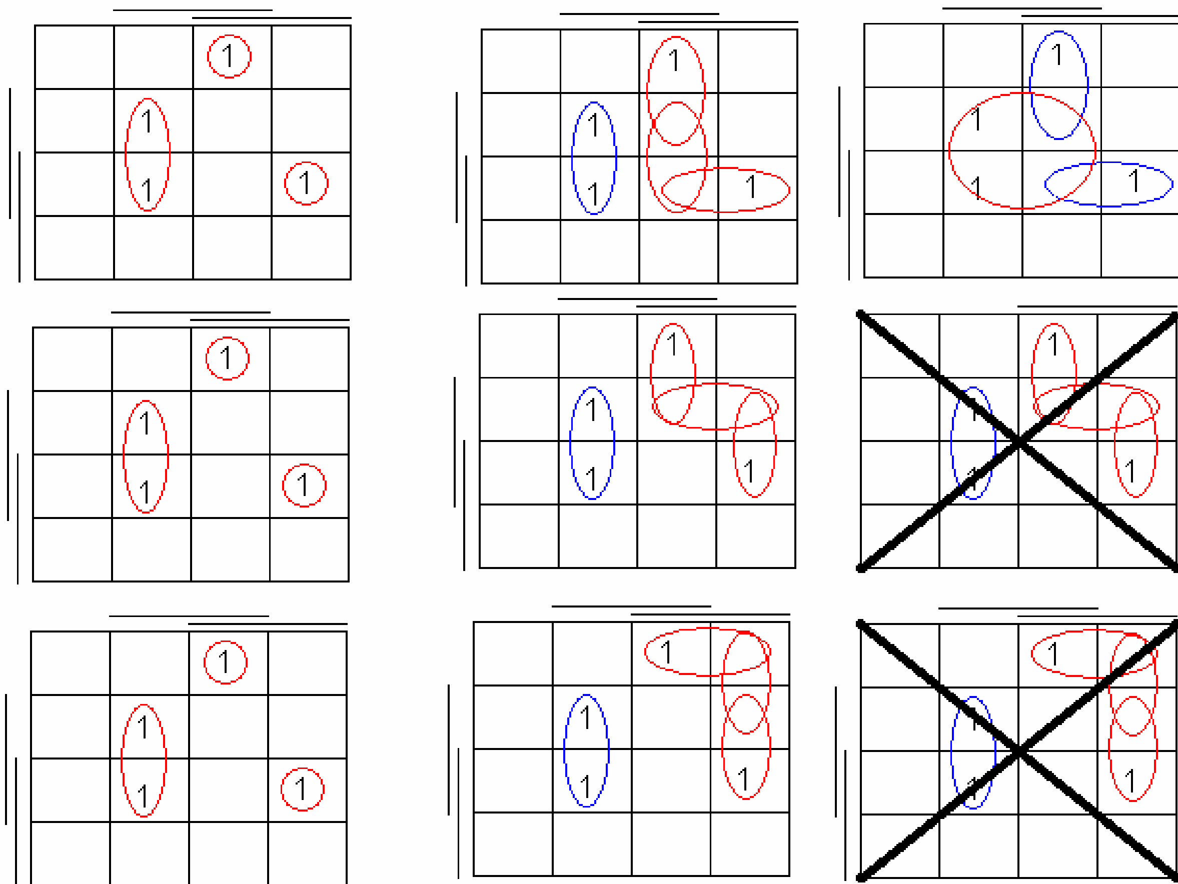
Vzdálenost Krychlí	Kriteria pro provedení transformací	
	Agresivní strategie	Vyčišťovací strategie
2	Vede k redukci počtu krychlí	Nezvyšuje počet krychlí, snižuje počet literálů.
3	Nezvyšuje počet krychlí	Nezvyšuje počet krychlí, snižuje počet literálů.
4	Nezvyšuje počet krychlí	nepoužito

Algoritmus rovněž používá několik strategií pro předvídání, zda se nějaká operace neprojeví výhodnou někdy v budoucnu. V tomto jsou v zásadě 2 přístupy. Buď se nejdříve vypočte celý strom a poté se prohledává celý při každé úpravě, což je výpočetně náročné a jak autoři 3 a 4 zmiňují, „Nevede většinou k o mnoho lepším výsledkům, i když někdy má své opodstatnění“. Místo toho zkouší prohledat jen několik nejbližších větví stromu, zda by nebylo výhodné provést rozkladnou operaci která dočasně povede ke zhoršení počtu krychlí nebo literálů a poté povede k většímu zjednodušení.



tento obrázek přejatý z [3] ilustruje princip funkce předvídacích strategií , kde zhorší počet literálů jedné z krychlí aby mohl být v dalším kroku podstatně redukován.

Dalším principem který EXORCISM využívá je snaha o spojování krychlí. Tato metoda slouží primárně ke snížení počtu literálů, a jak autoři přiznávají může při ní dojít k dočasnému počtu zvýšení počtu krychlí, proto je implementována citlivě s možností zahození učiněných kroků. Způsob spojování krychlí ukáži na dalším obrázku.



Zde je vidět, jak se algoritmus v jednom kroku dočasným zvýšením počtu krychlí snaží zjednodušit výraz. Vybere 2 krychle o vzdálenosti menší než jeho zadaný činitel kvality, a vypočte všechny možnosti jejich propojení. Poté zkoumá, zda nově vzniklá krychle nemůže být nijak spojena s některou stávající. Na obrázku jsou ilustrovány 3 pokusy, z toho dva neúspěšné kdy výsledek měl více krychlí i po snaze zjednodušit ho a jeden úspěšný. Tento je pak vybrán a začleněn do řešení.

Jak je vidět v tabulce 1 v úvodu, je EXORCISM velmi dobrý algoritmus hlavně z ohledu poměr čas/zjednodušení, proto se také často používá na zjednodušování XOR výrazu v praxi. S dostatkem času je podle jeho autorů schopen najít minimální tvar jakéhokoliv výrazu.

5.2 DECENTUA algoritmus

EXORCISM ovšem není jediným algoritmem, který používá metodu transformace krychlí ke zmenšení ESOP funkcí. Department of Electrical and Computer Engineering National Technical University of Athéna též přišla s vlastním algoritmem podle výsledků srovnatelným s algoritmem EXORCISM a o něco jednodušším. Teoretická analýza problému se nachází v originálním dokumentu, [5].

Základní pokrytí funkce F je generováno přímo z ní beze změny a neprovádí se jakákoliv příprava funkce před jejím použitím a zjednodušení funkce se nechává přímo na algoritmu samotném. Ten provádí postupnou selekci krychlí a snaží se je opět korektně transformovat na menší tvar. Pokrytí je poté uchováváno jako seznam krychlí.

Dále se dělá následující : pro přesně daný počet iterací se projdou všechny krychle ve funkci a hledají se takové se vzdáleností 0 nebo 1, které se pak vymažou nebo sloučí. Následně se vyberou 2 krychle a získají se všechny jejich možné náhražky pro každou z nich. Poté se pro každou z těchto náhražek vypočte jejich vzájemná vzdálenost a pokud je menší než 2, potom se tyto dvě funkce nahradí těmito vypočtenými podfunkcemi.Návrh tohoto algoritmu jak je popsaný v originálním dokumentu [5] :

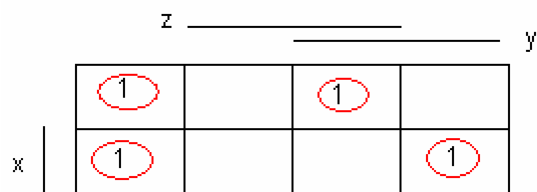
Algoritmus 1:

```
Minimizer (f; počet_iterací)
begin
F=Generuj_Pokryti (f);
loop=počet_iterací;
while loop > 0 do
     s=Size (F);
     zruš_nebo_sluč_krychle_vzdálenosti_0_nebo_1 (F);
     2to2 (F);
     proházej (F);
     if s = Size(F) then
     loop=loop-1;
     else
     loop= počet_iterací;
     endif
endw
end
```

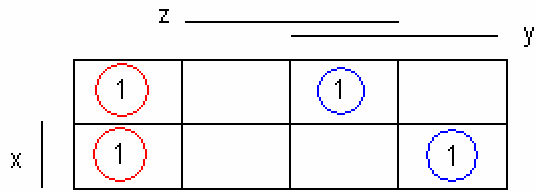
x	y	z	Q
			1
1			1
	1		
1	1		1
		1	
1		1	
	1	1	1
1	1	1	

Princip tohoto algoritmu nyní popíšu. Za první příklad si vezměme funkci $xy\bar{z} \oplus \bar{y}\bar{z} \oplus \bar{x}yz$. Tato funkce má následující pravdivostní tabulku a z ní vyplývající K-mapu (nuly nejsou zapisovány) . Algoritmus tuto pravdivostní tabulku buď rovnou získá ze vstupu (např. soubor PLA) nebo ji vypočte ze zadané funkce. Následně sestaví odpovídající K-mapu, ve které vyhledá všechny jedničky a označí je za samostatné krychle, jak je znázorněno na dalším obr. Tyto krychle by bylo možno stavět i větší pokud by byla funkce zadá textovým řetězcem, ale pro ilustraci použijme tento příklad.

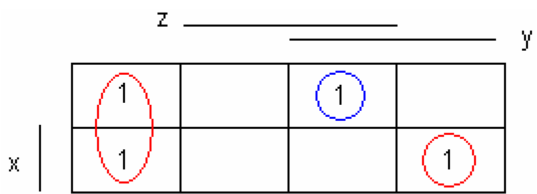
Následuje samotná zjednodušovací smyčka. Počtem iterací se dá regulovat maximální délka výpočtu a tím samozřejmě míra zjednodušení funkce. Pro ukázku budeme považovat parametr počet_iterací za rovný 4.



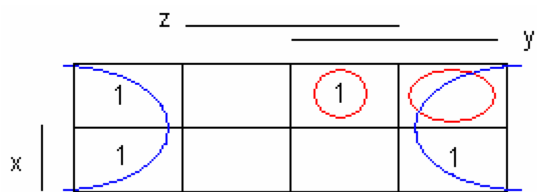
První instrukcí v cyklu asi není třeba se zabývat, slouží pro kontrolu jestli operace provedené nad výrazem ho nezvětšily místo požadovaného zmenšení, ale druhá instrukce zruš_nebo_sluč_krychle_vzdálenosti_0_nebo_1 již za pozornost stojí. Hledá totiž krychle, které mají vzdálenost 0 nebo 1 a následně je odstraní nebo sloučí.



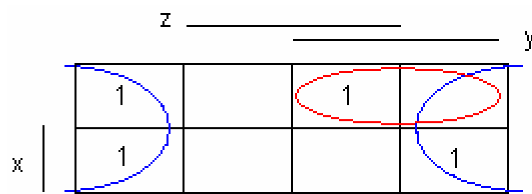
Zde byly nalezeny 2 možnosti sloučení krychlí. Krychle (= term) c1 a c2 se liší jen v proměnné x a c1 a c4 se liší jen v proměnné y. Pro minimalizaci vyberu term c1 a c2.



Nyní je vidět že po sloučení těchto 2 se naskýtá další příležitost ke slučování. Zde už ovšem přínos není takový jako v předchozím případě kde jsme snížili jak počet krychlí, tak počet literálů, ale tím že optimalizaci provedeme snížíme pouze počet literálů. Toto má smysl pouze ve chvíli, kdy tímto krokem nezvýšíme počet krychlí ve výrazu.



Zde vidíme výsledek. Je vidět že nedošlo ke snížení počtu krychlí, leč optimalizace snížila počet literálů o jeden. K našemu štěstí nově vzniklá krychle se liší jen v proměnné z s jinou stávající krychlí a je tudíž možné provést další optimalizaci, nyní již finální na tvar na dalším obrázku.



No a zde je vidět finální tvar funkce, který již dále nelze zjednodušit. V tomto případě by bylo optimálního tvaru dosaženo při čtvrtém volání funkce zrus_nebo_sluc.

Funkce se optimalizuje postupně, tudíž pokud bychom zadali jako maximální počet iterací menší počet iterací než by byl potřeba na minimalizaci, získali bychom alespoň částečně zjednodušený výsledek.

Tato funkce ovšem pracuje jen s krychlemi vzdálenosti 0 a 1. Pokud bychom měli jiné případy, kde je vzdálenost větší než 1 tak bohužel selže. Proto nastupuje 2 fáze, která má za

účel najít jiný alternativní způsob jak krychle vyjádřit aby se mohl proces zjednodušení opakovat.

V zásadě jde o následující : z množiny existujících krychlí se vyberou 2 z nich (pravděpodobně první 2) a získají se všechna jejich alternativní vyjádření (teoretické vysvětlení viz [5]). Pro každou z těchto dvojic se vypočte jejich vzdálenost a pokud je menší nežli 2 vzhledem k jakékoliv ze zbylých krychlí, nahradí se vybrané krychle alternativním vyjádřením a poté se pomocí funkce zruš_nebo_sluč sloučí do jedné.

Následuje zcela náhodné proházení pořadí jednotlivých krychlí mezi sebou, aby se příště vybrala jiná dvojice krychlí. Poté se smyčka spustí znovu a na této jiné kombinaci krychlí se snaží opět dosáhnout nějakého zlepšení. Pokud se nenajde zlepšení takový počet cyklů, jako je proměnná počet iterací, algoritmus skončí a odevzdá výsledek. Ten ovšem nemusí být nutně minimální, záleží na volbě proměnné počet iterací která určuje kvalitu zjednodušení. Hodnota této proměnné určuje, kolikrát se algoritmus provede znovu bez toho ze by v předchozí iteraci našel nějaké zjednodušení.

Algoritmus 2: `Find 2to2 (Formula c1; Formula c2)`

```
Begin
Oprav řazení x1,x2…… proměnných v c1, c2 tak aby byly stejné;
Formula P = společná část c1, c2;

Formula g = c1' ⊕ c2'
// kde c1 a c2 jsou Cka s částí P odstraněnou;

najdi g1, g0, g2;
//g1, g0, g2 jsou podfunkce g kde :
//g1 = ( 1, x2, x3…… xn )- všechna x1 se nahradí 1
//g0 = ( 0, x2, x3…… xn )- všechna x1 se nahradí 0
//g2 = g0 + g1 přicemž + je dvojkové sčítání bez prenosu (XOR)

najdi weight(g1);weight(g0);weight(g2);
// popis funkce v původním dokumentu, jedná se o velikost
jednotlivých položek co do počtu krychlí

if max(size(g1);size(g0);size(g2)) = 2 then
return [(c1 ⊕ c2)];
endif //nedosáhlo se zlepšení, vrací se původní hodnoty

return
[
(Pxk + 1g1 ⊕ P¹xk+1g0);
(Pxk+1g2   ⊕Pg0);
(P ¹xk+1g2 ⊕ Pg1);
];
end
// dosáhlo se zlepšení
```

Jak je vidět, algoritmus je jednodušší a podle měření provedených [5] je srovnatelný s algoritmem EXORCISM popsaným výše. Tabulka 1 (níže) je vytvořena pro algoritmus v jeho rozšířené podobě, kdy obsahuje navíc i algoritmus hledání náhrady pro krychle se vzdáleností 3. Hrubý popis tohoto algoritmu je uveden v [5].

6. TYPY SOUBORŮ POUŽÍVANÉ PRO VÝRAZY ESOP :

6.1 PLA

Tento typ souborů popisuje funkci pomocí její pravdivostní tabulky. Klíčová slova pak dále popisují jak funkce vypadají. Většina těchto klíčových slov musí být v PLA před vlastním vyjmenováváním hodnot pravdivostní tabulky (z následující tabulky všechny kromě .e) Navíc BOOM neakceptuje všechna klíčová slova definovaná ve standartu, ta která nezná ignoruje.

Klíčová slova akceptovaná enginem BOOM :

.i *d*	Určuje počet vstupních hodnot - POVINNÉ
.o *d*	Určuje počet výstupních hodnot - POVINNÉ
.ilb *x1,x2…xd*	Pojmenovává vstupní binární hodnoty které se nacházejí ve vstupu v pořadí takovém jakém jsou v souboru. Počet těchto hodnot musí být stejný jako je počet vstupních proměnných.
.ob *y1,y2…yd*	Pojmenovává výstupní binární hodnoty na výstupu v pořadí takovém jakém jsou v souboru. Počet těchto hodnot musí být stejný jako je počet vstupních proměnných.
.type *s*	Nastavuje logické uspořádání matice znaků. Může nabývat hodnot fr a fd, Fd je defaultní hodnota, ale pro jádro BOOMu, je zapotřebí tuto hodnotu nastavit na fr.
.p *d*	Specifikuje počet termů produktu
.e (.end)	Ukončuje PLA

Příklad 1 :

```
# 2-bitová sčítačka zapsaná v souboru PLA
 .i 4
 .o 3
 .p 16
 .type fr
 0000 000
 0001 001
 0010 010
 0011 011
 0100 001
 0101 010
 0110 011
 0111 100
 1000 010
 1001 011
 1010 100
 1011 101
 1100 011
 1101 100
 1110 101
 1111 110
 .e
```

Tento jednoduchý příklad ilustruje zápis logické funkce tak aby ji BOOM dokázal zpracovat – tedy nejjednodušší zápis kombinační logické funkce bez pojmenovávání proměnných a s označením typu souboru fr.

Příklad 2 :

```
# Kombinační logika resetovatelného čítače
.i 5
.o 4
.p 10
.ilb RESET q0 q1 q2 q3
.ob d0 d1 d2 d3
.type fr
00000 0001
00001 0010
00010 0011
00011 0100
00100 0101
00101 0110
00110 0111
00111 1000
01000 1001
01001 0000
1---- 0000
.e
```

Tento příklad ilustruje složitější funkci, sekvenční a navíc s nedefinovanými stavy opět ve tvaru aby BOOM byl schopen soubor zpracovat.

6.2 BLIF

Pravdivostní tabulka v souboru PLA sice výborně popisuje funkci kterou máme, ale nedá se nijak zjednodušit ani nic neříká o obvodu jako takovém, čili na vyjádření výsledku minimalizace používám soubor formátu BLIF. Je velmi podobný souboru PLA, s tím rozdílem ze nepopisuje celou pravdivostní tabulku, ale jen její část.

Příklad :

```
.model traffic_cl
.inputs a b c d e
.outputs f
.names h f
0 1
.names d a b c e h
000-- 1
00-0- 1
0-00- 1
00--0 1
0-0-0 1
0--00 1
.end
```

Jak je vidět klíčová slova jsou velmi podobná PLA, inputs, outputs plni stejnou funkci jako v i, o u PLA. Klíčové slovo names slouží k znázornění vlastních operací, případně zavedení jakýchsi „lokálních proměnných“. Mezi všemi parametry klíčového slova names se totiž provede operace která je v pravdivostní tabulce pod klíčovým slovem a výsledek se vždy uloží do „proměnné“ jejíž jméno je posledním parametrem klíčového slova names.

7. POROVNÁNÍ ALGORITMŮ VE ZKUŠEBNÍCH OBVODECH

7.1. Zkušební obvody

7.1.1. ISCAS zkušební obvody

Bohužel nejsou k dispozici obecné charakteristiky všech obvodů, liší se obvod od obvodu a jsou popsány blíže u jednotlivých obvodů. Existují aktuálně se používají 2 verze, ISCAS 85 a ISCAS 89. Tyto sady pro testování obsahuje velké množství různých obvodů, namátkou například :

- c432 : 27mi kanálový řadič přerušení
- c880, c3540 : 8-bit ALU
- c7552 : 32-bit sčítačka/komparátor

Bližší informace k těmto obvodům lze najít například na stránkách www.eecs.umich.edu/~jhayes/iscas/ , nebo jejich oficiálních homepage www.cbl.ncsu.edu/.

7.1.2. ITC zkušební obvody

Tyto zkušební obvody, ve verzi z roku 99 patří též mezi prakticky používané. Obecnými vlastnostmi všech obvodů v těchto testech je že neobsahují žádné instrukce specifické jen pro určitý kompilátor, vnitřní paměti (krom registrů) a žádné 3-stavové sběrnice. Dále používají jen logické a aritmetické obvody standartu IEEE a jsou kompletně synchronní. Počítají s vždy přístupným globální signálem reset a clock signálem přivedeným přímo na jednotlivé registry.

Jako příklady obvodů jenž jsou součástí těchto lze uvést :

- b04 : výpočet minima a maxima
- b06 : řadič přerušení
- b13 : interface k meteorologickým senzorům
- b15 : část procesoru 80386

bližší informace lze najít na stránkách http://www.cad.polito.it/tools/itc99.html

7.2. Výsledky popsaných algoritmů ve zkušebních obvodech :

7.1.1 DECENTUA / EXORCISM

Jméno funkce	Velikost řešení [termy]		Doba výpočtu [ms]	
	DECENTUA	EXOR4	DECENTUA	EXOR4
9sym	51	51	481	
Clip:1	16	16	29	105
Clip:2	18	18	30	91
Clip:3	21	21	32	43
Clip:4	27	27	67	65
Clip:5	15	5	31	41
T481	13	13	34	46
Life	46	47	4962	
Xor5	5	5	30	34
Ryy6	40	40	29	30

Tab. 1 : srovnání algoritmu DECENTUA s algoritmem exor[4] pro vybrané funkce

Tabulka ukazuje srovnání výše uvedeného algoritmu s algoritmem exor[4] pro vybrané funkce, v případě že hodnota nebyla uvedena pro EXOR4 byla tak malá že byla neměřitelná.

7.1.2 EXORCISM / EXMIN / MINT

Tab. 2 Srovnání algoritmů EXMIN2, EXORCISM4 a MINT

Z

Benchmark			Krychlí			Literálů			Čas výpočtu		
Jméno	Vstupů	Výstupů	EXM2	MINT	Exor4	EXM2	MINT	Exor4	EXM2	MINT	Exor4
5xp1	7	10	34	32	31	186	181	175	13	25	3
9sym	9	1	53	51	51	433	427	426	25	52	13
add6	12	7	127	127	127	872	936	859	430	320	31
addm4	9	8	91	90/89	90/89	654	651	624	129	355	39
b12	15	9	28	28	28	164	167	166	4	17	1
clip	9	5	68	64	64/63	517	492	479	55	140	11
ex7	16	5	81	81	81	601	592	584	46	166	13
f51m	8	8	32	31	31	161	185	162	10	22	5
in7	26	10	35	35	35	333	352	343	12	56	3
intb	15	7	307	267	268	3036	2519	2527	1353	2476	134
life	9	1	54	52/51	50/48	415	391	370	23	32	9
m181	15	9	29	29	29	169	172	171	5	18	1
m4	8	16	84	83	77/76	783	897	714	189	178	120
max512	9	6	89	88/83	84/82	696	723	672	71	187	64
rd53	5	3	15	16/15	14	60	69	57	2	2	1
rd73	7	3	42	36	36	221	194	197	20	36	9
rd84	8	4	59	55/54	59/58	330	303	333	45	11	17
ryy6	16	1	40	40	40	368	368	368	13	18	2
sao2	10	4	29	29/27	28	308	311	288	8	12	1
seq	41	35	259	249/248	246	5305	5187	5048	2797	15182	378
sym10	10	1	84	82	79	751	735	702	154	176	20
t3	12	8	25	25	24	209	214	216	5	10	1
t481	16	1	13	13	13	53	53	53	677	377	1
vg2	25	8	184	184	184	1992	2033	2010	163	1655	42
z4	7	4	29	29	29	145	148	133	4	9	3
Celkem			1891	1804	1791	18762	18300	17677	6253	21532	922
Zisk			0	-87	-100	0	-462	-1085			
				-4.6%	-5.3%		-2.5%	-5.8%			

n

ání jednotlivých existujících algoritmů v ohledu na počet výstupních krychli, počtu literálů a doba která byla k výpočtu použita.

Je vidět, že Exorcism ve verzi 4 vítězí téměř na všech frontách – výsledky jím produkované jsou v průměru o něco lepší než u zbylých 2 algoritmů ale co je důležité, výpočetní doba je 6x kratší. O tomto algoritmu je pojednaní na dalších stranách níže.

Jsou zahrnuté typové algoritmy, EXMIN2, jako vylepšená verze původního EXMINu od jednoho z autorů [1], MINT, algoritmus trochu staršího data, který ovšem produkuje řetězce s menším počtem literálů a krychlí než EXMIN, je uveden spíše pro srovnání protože výpočetní čas je více než 20x větší než u EXORCISMu při dosažení srovnatelných (2x lepší, 14x stejný, 4x horší) výsledků a asi 4x větší ve srovnání s EXMINem při lepších (15/9/1) výsledcích.

8. POPIS VLASTNÍ IMPLEMENTACE

8.1 Engine BOOM

Boom je engine poskytující zázemí pro addony implementující jeho jednotlivé funkce. Jeho výhodou jsou nízké paměťové nároky a jednoduchost na pochopení kterou jsem velmi ocenil. Verze kterou jsem dostal a pro kterou jsem vytvářel svůj addon obsahovala funkce pro práci se soubory PLA. Vstupy se mohou nacházet v jakémkoliv stavu (1, 0, don't care) aniž by bylo potřeba je nějak připravovat.

BOOM jako takový je dvojúrovňový heuristický minimalizátor určený k minimalizaci funkcí definovaných pomoci SOP, velmi rychlý, s nízkými paměťovými nároky. Podobně jako výše uvedené minimalizátory je iterativní čili opakovanýmy průchody zlepšuje své výsledky.

8.2 Popis implementace

Moje implementace minimalizátoru se opírá o algoritmus DECENTUA, s tím že na rozdíl od originálu jsem zvolil jinou vnitřní strukturu – místo používání LONGu jako vektoru je v mé implementaci přímo třída literál, která se pak vkládá do spojového seznamu v třídě Cube. Vše zastřešuje třída ESOP, která obsahuje spojový seznam krychlí. Vzhledem k navázání na jádro programu BOOM je jako vstupní formát využíván soubor typu PLA, a jako výstupní formát byl zvolen BLIF.

Engine BOOMu sice obsahoval třídy jako Term a Tyrše které jsem zprvu zamýšlel využít pro své účely, ale nakonec jsem zjistil že by byly zapotřebí velmi rozsáhlé modifikace těchto tříd a tak jsem od toho záměru upustil.

Rozhodl jsem se proto vytvořit vlastní hierarchii tříd, kterou dále popíšu :

Nejmenší jednotkou v mém hierarchii tříd je třída literál.

```
class Literal
     {
     public :
     Literal();
     Literal (char value);
     Literal (bool value);
     char GetValue();
     void SetValue(char value_new);
     void SetValue(bool value_new);
     bool isEqual(bool value_new);
     bool isEqual(char value_new);

     private :
     bool value; // 0 or 1
     bool flag;  // 0 - not present, 1 - dont care
     }
```

Tato třída reprezentuje elementární prvek každého logického výrazu. V enginu BOOMu neexistovala sama o sobě, byla součástí třídy term, a přístup k ní byl složitější než jsem chtěl. Nová třída na rozdíl od originálu nereprezentuje literál znakem ale jen 2 booly, proto existují i přetížené verze přístupových metod se vstupní hodnotou typu string.

Další třídou v hierarchii, o stupeň výše postavenou je třída Cube.

```
class Cube
      {
      protected :
      int literal_count;
      int size;
      vector<Literal> literals;

      public :
      Cube(int size_init);
      Cube(const Cube & Copied_Cube);
      ~Cube();
      void Set_x_on       (const int X);
      void Set_x_off      (const int X);
      void Set_x_dontcare (const int X);
      void Remove_X       (const int X);
      void Add_X         (const int X, string status);
      void Contains_X     (const int X);
      bool empty ();
      bool operator == ( const Cube & Compared_Cube )
      const;
      Cube operator  = ( const Cube & Original_Cube );
      Cube createF0 (int x);
      Cube createF1 (int x);
      Cube createF2 (int x);
      Int size ();     // size of cube, in #literals
      Int distance (); // returns distance to another
      cube ( number of different literals )
      }
```

Tato třída je ekvivalent třídy term, ale její struktura je velmi odlišná. V mém programu je často třeba dělat úpravy jednotlivých kostek podle vzorců uvedených výše v textu (nejčastěji výpočet podfunkcí) a k tomu jsou zapotřebí operace s jednotlivými literály které nebyla předchozí třída schopna zajistit. Dále je tu přetížený operátor „=“ který vytváří hlubokou kopii krychle a „==“ který nepřekvapivě slouží k porovnávání obsahu dvou krychlí.

```
class ESOP
      {
      protected :
      vector<Cube> terms;
      vector<Cube> terms_temp;

      int iterations_without_improvement;
      int literal_count;
      int cube_count;
      int Quality;

      public :
      ESOP (PLA *src);
      // default constructor, will create default
      cover from PLA file and check it

      ESOP (PLA *src,int output);
```

```
// same as previous one, but for multi output
files, can select function Y1, Y2 etc.

~ESOP (void); // destructor

void Minimize ();
void Recalculate_Size();
void Shuffle();
void SetQuality();
void Copy_temp_to_original();
void Restore_temp_from_original();

void Remove_Distance_0_cubes ();
void Remove_Distance_1_cubes ();
void Remove_Distance_2_cubes ();


bool contains (const int X );

void Add_Cube (Cube * NewCube );
void Remove_Cube (Cube * NewCube );

void Remove_Empty_Cubes ();
}
```

Poslední a nejvýše postavenou třídou v hierarchii je třída ESOP která pokrývá zbytek funkcí kromě jediné (finální tisk do souboru, o tom dále) Dva konstruktory jsou jen pro přehlednost, `ESOP(PLA *src);` je tentýž co `ESOP (PLA*src,1);` a slouží pro funkce s jedním výstupem.

Funkce Minimize je jádro vlastní minimalizace, zde se řídí a rozhoduje co se program (ne)pokusí minimalizovat v závislosti na požadované kvalitě výstupu, kde kvalita je počet pokusů o zmenšení bez toho aby se našlo zmenšení.

```
ESOP :: minimize ()
      {
      void Remove_Empty_Cubes ();
      size == Recalculate_Size();

      while (iterations_without_improvement < quality)
            {
            Remove_Distance_0_cubes ();
            Remove_Distance_1_cubes ();
            Remove_Distance_2_cubes ();
            Shuffle(); // move cubes for next iteration
            If (size == Recalculate_Size())
                  iterations_without_improvement++;
            }
      }
```

Funkce Remove_distance_0_cubes() je velmi jednoduchá, hledá stejné krychle a pak odstraňuje tu dále ve spojovém seznamu, stejně jako funkce Remove_distance_1_cubes() která hledá dvojice krychlí, které se liší jen v jediném literálu a ten potom odmaže z jedné z nich a druhou odmaže celou.

Funkce Remove_distance_2_cubes() je složitější ale odpovídá funkci popsané výše v textu u popisu algoritmu DECENTUA. Úplná znění zdrojových kódů se nachází v příloze.

Funkce Shuffle prohazuje krychle mezi sebou aby se v příštím průchodu zkoušely srovnat jiné kombinace krychlí. Je realizována posunem hlavy spojového seznamu krychlí o náhodný počet míst který není roven počtu krychlí funkce aby nedocházelo k rotaci krychlí ve funkci.

V originálním algoritmu se nachází ještě funkce Remove_distance_3_cubes() kterou jsem ovšem nezařadil do předkládaného řešení protože obsahovala chyby které se mi nepodařilo nalézt a tedy dávala nesprávné výsledky.

7.3 Dosažené výsledky

Jako testovací obvody jsem zvolil následující :

	Vstupů	Výstupů
9sym.pla	9	1
clip.pla	9	5
xor5.pla	5	1
apex1.pla	45	45
apex5.pla	117	88

Algorit. / Obvod	Moje implementace		EXOR4	
	Počet termů	Čas výpočtu [ms]	Počet termů	Čas výpočtu [ms]
9sym.pla	66	302	51	13
clip.pla	95	41	64	11
xor5.pla	5	57	5	30
apex1.pla	361	5607	287	51
apex5.pla	930	12904	398	544

Pro srovnání jsem použil algoritmus EXORCISM, protože moje implementace podporuje větší rozsah vstupů než původní algoritmus a výsledky původního algoritmu jsou jen minimálně rozdílné od EXORCISMu.

Je vidět že pro obvody s menším počtem vstupů a hlavně výstupů je algoritmus horší nicméně srovnatelný ale pro větší soubory je bohužel pomalý a nedokáže si poradit s velkou vzdáleností mezi krychlemi a tudíž nenajde žádná zjednodušení.

Další věcí co se negativně odráží na výpočetním čase je fakt že program není napsán pomocí vláken a tedy veškeré výpočty probíhají sériově za sebou.

Výsledky jsou tedy bohužel horší než u EXORCISMu, ale to je očekávaná věc neboť vzhledem k délce vývoje jde již o velmi komplexní algoritmus který si dokáže poradit i s velkým počtem vstupů a výstupů.

9. ZÁVĚR

Shrnul jsem všechna mě dostupná fakta současných algoritmech na zjednodušování funkcí. Žádný z nich nedává 100 procentní záruku absolutního minima funkce, ale to nebyl ani jejich cíl, kterým bylo nalezení co nejmenšího výsledku v omezené době. Všechny jmenované algoritmy proto obsahují nějaký způsob kontroly času který se věnuje zpracování výrazu, buď ve formě maximálního stupně zanoření do stromu funkce nebo maximálního počtu iterací provedených nad funkcí. Z jejich srovnání vychází nejlépe algoritmus EXORCISM [3] . Jeho poměr kvalita výsledku / čas je zdaleka nejlepší ve všech sledovaných obvodech až na jeden.

Moje implementace algoritmu DECENTUA dosahuje podobných výsledků jako původní algoritmus s tím že je zde stále velký prostor ke zlepšení. Původní algoritmus nepracoval s velkým počtem vstupů takže jsem zjistil jak si vede když je nasazen na větší funkce, kde výsledky nebyly již tak dobré jak jsem předpokládal. Částečně to je tím že jsem do předkládaného řešení nezahrnul část co hledá krychle vzdálenosti 3 (nepodařilo se ho odladit tak aby dával bezchybné výsledky). Podle mého názoru tento algoritmus ještě může podat lepší výsledky pokud se provedou vylepšení v jeho kódu. Jmenovitě například paralelizování minimalizace pro jednotlivé výstupy například pomocí vláken by velmi pravděpodobně výrazně snížila stávající výpočetní časy pro funkce s více výstupy. Vymyšlení a napsání metod které by vyhledávaly a slučovaly krychle větších vzdáleností než dva, resp. 3 by zase pomohlo výsledků v počtu termů ale toto vylepšení by bylo nejspíše velmi složité.

REFERENCE

[1] „Minimization of AND-EXOR Expressions Using Rewrite Rules“ Daniel Brand and Tsutomu Sasao, IEEE TRANSACTIONS ON COMPUTERS, VOL. 42, NO. 5, MAY 1993
[2] S. Even, I. Kohavi, and A. Paz, “On minimal modulo-2 sum of products for switching functions,” IEEE Trans. Electron. Compuf., vol. EC-16, pp. 671-674, Oct. 1967.
[3] Fast Heuristic Minimization of Exclusive-Sums-of-Products Alan Mishchenko and Marek Perkowski, Department of Electrical and Computer Engineering, Portland State University
[4] N. Song. Minimization of Exclusive Sum of ProductExpressions for Multi-Valued Input Incompletely Specified Functions. M.S. Thesis. EE Dept. Portland State University.Portland, OR, 1992.
[5] S. Stergiou, K. Daskalakis, G. Papakonstantinou, A Fast and Efficient HeuristicESOP Minimization Algorithm
[6] http://www.cad.polito.it/tools/itc99.html
[7] www.eecs.umich.edu/~jhayes/iscas/
[8] http://service.felk.cvut.cz/vlsi/prj/BOOM/
[9] http://www.sgi.com/tech/stl/

PŘÍLOHA : 1 DVD se úplným zdrojovým kódem aplikace