České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Manipulace s logickými funkcemi pomocí grafů

Bc. Petr Vála

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, navazující magisterský

Obor: Výpočetní technika

leden 2009

Poděkování

Děkuji všem, co mě jakýmkoliv způsobem podpořili a svým přičiněním tak napomohli vzniku této práce. Jmenovitě především panu Fišerovi za jeho odborné rady a připomínky.

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 21.1. 2009 .....

Abstract

This Thesis discusses a graph as a mean of a representation of logic functions. Every logic function in its disjunctive normal form is simply represented as an undirected cyclic graph where nodes act as terms and edges act as conjunctive variables of these terms. This structure can be used in computer processing of many tasks and some of them like a *tautology question* or an *orthogonalization* are described and confronted with existing methods in this document.

Abstrakt

Tato práce pojednává o grafu jako nástroji pro reprezentaci logických funkcí. Každá logická funkce v disjunktivní normální formě je jednoduše reprezentována jako neorientovaný cyklický graf, kde vrcholy představují jednotlivé termy a hrany slouží jako společné proměnné těchto termů. Tuto strukturu lze použít pro počítačové zpracování mnohých úloh a některé z nich, jako například *otázka tautologie* nebo *ortogonalizace funkce*, jsou v tomto dokumentu popsány a srovnány se stávajícími přístupy.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

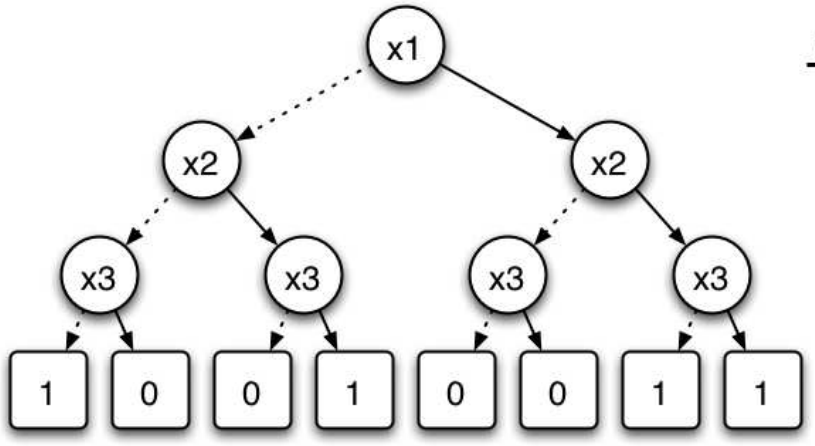
Logické funkce a Booleova algebra se v dnešním světě informačních technologií používají neustále a mají pro velké množství jejich oblastí naprosto zásadní význam. Potřeba je používat vyplývá ze způsobu, jakým dnešní výpočetní technika pracuje. Některé z dřívějších počítačů zpracovávaly informace ukládané v desítkové soustavě, avšak s příchodem tranzistorů se časem ukázalo, že takto složité mechanismy nejsou příliš výhodné, neboť jsou pomalejší a vedou k větší poruchovosti, než elektronické obvody pracující pouze se dvěma stavy. V dnešní době téměř všechny počítače pracují pouze s binárními hodnotami, tedy takovými, které mohou nabývat pouze dvou stavů 0,1 nebo-li nepravda, pravda. Takto ustálené vnímání počítačového světa vede k obrovskému nárůstu významu celé Booleovy algebry, kterou definoval již v polovině 19. století britský matematik George Boole. Až do počátků výpočetní techniky neměla příliš praktické využití.
Používání tohoto aparátu při výrobě počítačů a logických obvodů přirozeně vede k tomu, že je vynakládáno značné množství úsilí k tomu, aby k veškerým výpočtům v této oblasti docházelo co nejrychleji a nejefektivněji. Logickou funkci je možné prezentovat mnoha různými způsoby (např. kanonickou tabulkou pravdivostních hodnot, Karnaughovou mapou, zápisem ve formě DNF, apod.). Avšak nezávisle na tom, jakým způsobem je funkce prezentována, platí, že většina operací, které lze s danou logickou funkcí provést, je výpočetně velmi náročná. Doba řešení narůstá exponenciálně s množstvím vstupů logické funkce a tudíž hledání exaktních řešení některých problémů může trvat řádově i měsíce nebo roky výpočetního času.
Celkově strávený čas řešením úlohy pak nepodléhá až tolik samotnému výkonu počítače, který není tolik rozhodující, ale je daleko více ovlivněn zvoleným heuristickým algoritmem. Platí, že nelze vytvořit takový algoritmus, který by vždy výpočet znatelně urychlil, neboť jeho efektivita je přímo závislá na datech, která dostává ke zpracování. Proto se tento vědní obor neustále zabývá novými způsoby, jakými k problémům přistupovat, a hledají se nové algoritmy, které mohou dávat lepší výsledky pro některé typy dat.
Jedním z těchto nových způsobů, jakými lze logické funkce chápat a řešit, je za pomoci grafů, kdy jednotlivé uzly v grafu představují termy a hrany mezi nimi jsou vytvořeny na základě společných proměnných, které se vyskytují v určité podmnožině termů. Účelem této práce je zjistit, zda tento způsob řešení má využití a zda je konkurenceschopný oproti současným konvenčním způsobům řešení. Součástí práce je i implementace řešení některých problémů a srovnání rychlosti s ostatními programy, které se dnes běžně používají.

1.1 Existující řešení

V současné době není znám žádný projekt, který by logické funkce řešil za pomoci grafů takovým způsobem, jaký je použit v této práci. Existují však různé projekty a práce, které tyto problémy řeší svou vlastní cestou.

1.1.1 BDD

Binární rozhodovací diagramy [2] (Binary decision diagrams) jsou jedním z konvenčních způsobů reprezentace logické funkce. Jedná se o orientovaný acyklický graf, jehož uzly odpovídají osazení určité logické proměnné konkrétní hodnotou. Každý neterminální uzel v tomto grafu má dva následníky, z nichž jeden odpovídá hodnotě jedna a druhý hodnotě nula u dané proměnné. K listům tohoto grafu se dostáváme tak, že u všech logických proměnných určíme jejich hodnotu (rozhodujeme se, zda jít grafem ve směru levého či pravého potomka) a tyto listy už nabývají konkrétní hodnoty pro tuto konfiguraci vstupních proměnných. Listů je pochopitelně 2^n, kde n je počet vstupů logické funkce. Na rozdíl od naší struktury však BDD popisují chování funkce, zatímco náš graf popisuje její strukturu. Příklad takového diagramu je na obrázku 1.1.

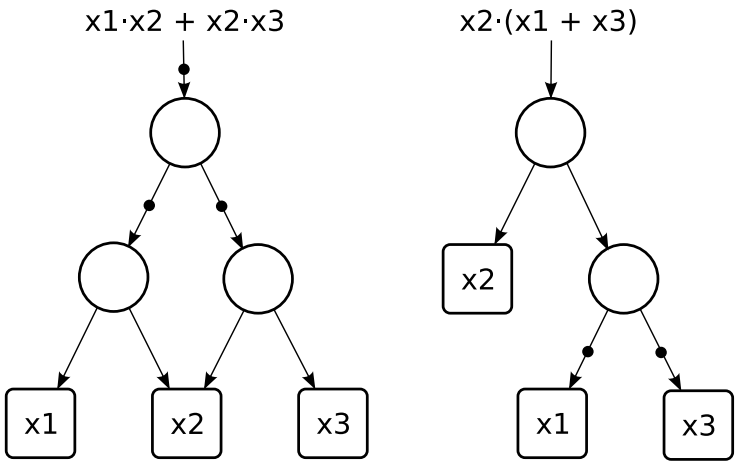


x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Obrázek 1.1: Binární rozhodovací diagram

1.1.2 AIG

AIG (And-Inverter Graph) [5] je dalším typem reprezentace logických funkcí. Podobně jako BDD je i AIG orientovaný acyklický graf. Jeho význam spočívá v tom, že definuje funkci za pomoci invertorů a hradel AND. Každý uzel, který není listem, představuje logický součin se dvěma vstupy a jedním výstupem. Pokud chceme vstup do hradla či jeho výstup negovat, označíme příslušnou hranu značkou. Listy v grafu jsou jednotlivé proměnné funkce. AIG už skutečně popisují strukturu funkce (podobně jako náš graf) a ne pouze její chování. Příklad AIG je vyobrazen na obrázku 1.2.



Obrázek 1.2: And-Inverter Graph

1.2 Struktura dokumentu

Následující text je rozdělen do několika kapitol. Po této úvodní kapitole následuje kapitola druhá, která konkrétněji pojednává o jednotlivých problémech, je v ní vysvětlena problematika složitosti úloh, v krátkosti se v ní rovněž pojednává o grafech a logických funkcích a také zde je přesněji deklarován záměr celé práce. Třetí kapitola se již věnuje analýze problémů a návrhu řešení. Ve čtvrté kapitole je popsána celá implementace a jsou v ní úryvky zdrojových kódů pro

snazší pochopení práce programu. Pátá kapitola popisuje testování implementovaného řešení a je v ní vyobrazeno srovnání s ostatními programy. Závěrečná kapitola pak hodnotí přínos celé práce, dosažené výsledky a jsou zde také uvedeny návrhy pro případné budoucí rozšíření.

1.3 Slovníček pojmů

Celá tato práce se týká především logických funkcí a manipulací s nimi. V souvislosti s tím se v práci používají ustálené pojmy, které jsou vysvětleny v této sekci.
logická proměnná – proměnná, která nabývá hodnot {0,1} (dále jen proměnná)
literál – proměnná nebo její negace
term – množina literálů, které jsou vzájemně vázané logickým operátorem

- **součinový** – vázán logickým součinem
- **součtový** – vázán logickým součtem

minterm/maxterm – součinový/součtový term obsahující všechny proměnné
logická funkce – funkce, jejíž vstupy nabývají pouze hodnot {0,1}, výstupy nabývají pouze hodnot {0,1,neurčený stav} a která jednoznačně definuje svůj výstup pro všechny variace ohodnocení svých vstupů
tautologie – funkce, která má výstupní hodnotu vždy 1
kontradikce – funkce, která má výstupní hodnou vždy 0
logická formule – výrok, který jednoznačně určuje logickou funkci, pro jednu logickou funkci existuje nekonečné množství logických formulí
krychle – logický součin množiny literálů (= součinový term)
pokrytí – množina krychlí reprezentující celou funkci
implikant – krychle, pro kterou platí, že implikuje danou funkci, což znamená, že pokud tato krychle pro dané ohodnocení proměnných nabývá hodnoty 1, nabývá celá funkce pro toto ohodnocení rovněž hodnoty 1

- **redundantní implikant** – takový implikant, po jehož odstranění z množiny všech implikantů nabývá funkce stále stejných výstupních hodnot pro všechna ohodnocení vstupů
- **přímý implikant** – takový implikant, ze kterého již nelze vypustit žádný literál tak, aby stále zůstal implikantem
- **podstatný implikant** – takový přímý implikant, který obsahuje minterm, který není obsažen v žádném jiném přímém implikantu

disjunktivní normální forma – reprezentace logické funkce za pomoci součtu krychlí (součinů), často zkracována na DNF (případně anglickou zkratkou SOP), aby DNF definovala funkci, musí být pokrytý každý onset minterm dané funkce a nesmí být pokrytý žádný offset minterm
ON-set – množina termů implikující výslednou funkci rovnou 1
OFF-set – množina termů implikující výslednou funkci rovnou 0
DC-set – množina termů implikující výslednou funkci jako nespecifikovanou
DC – (don't care) je označení nezávislosti na hodnotě dané proměnné
PLA – formát vstupních souborů pro Espresso (bude popsán dále)

2 Popis problému a vymezení cíle

Následující text vysvětluje, proč jsou některé úlohy prováděné s logickými funkcemi tak časově náročné a proč může být vhodná volba heuristického algoritmu dobrá cesta, jak celý průběh operace signifikantně urychlit. Dále je zde uvedeno krátké přiblížení do problematiky logických funkcí, grafů a je zde popsán záměr práce.

2.1 Asymptotická složitost algoritmů, problémy a jejich kategorie

Při řešení úloh ve výpočetní technice je zapotřebí nějakým způsobem ohodnotit a srovnat rychlost dvou různých algoritmů. Není možné toto srovnání provádět z hlediska toku reálného času, neboť skutečný čas věnovaný úloze je do jisté míry závislý na hardwarové konfiguraci testovacího prostředí, na konkrétním zadání úlohy a dalších parametrech. Proto byl zaveden pojem asymptotická složitost, který udává náročnost algoritmu jako funkci závislou pouze na jednom parametru a tím je velikost vstupních dat. Za pomoci tohoto aparátu je možné hrubě odhadnout, jak dlouho bude asi úloha trvat a rovněž jej lze použít pro srovnání dvou algoritmů a prokázat, že jeden z nich má nižší složitost a bude tedy pro většinu zadání rychlejší. Více se o asymptotických složitostech lze dočíst v [11].

V souvislosti se složitostmi, hovoříme často také o složitosti problému. Složitost problému udává spodní hranici pro algoritmy řešící tyto problémy. Jinými slovy vyjadřuje nejlepší možnou složitost algoritmu, které lze dosáhnout. Složitost problému vyplývá z logické úvahy nad způsobem, jakým lze problém řešit. Z principu není možné vymyslet algoritmus řešící tento problém s menší složitostí, než je složitost tohoto problému. Pokud by k něčemu takovému došlo, byl by to důkaz, že problém je jednodušší a jeho složitost byla určena nesprávně.
Díky tomuto rozdělení složitosti problémů vznikly jednotlivé třídy složitosti, které říkají, jak moc je problém složitý. Tyto třídy jsou podmnožiny všech problémů a každá z nich je charakterizována určitou vlastností, kterou mají všechny algoritmy, které do ní patří. Jednou z nejzákladnějších tříd složitosti je třída P. Tato třída deterministicky polynomiálních problémů je množinou takových problémů, které jsme schopni řešit v polynomiálně omezeném čase na deterministickém Turingově stroji. Problémy v této třídě se obvykle považují za efektivně řešitelné a v praxi bývá časová náročnost řešících algoritmů únosná.
Avšak problémy, kterými se budeme v této práci zabývat, do třídy P nepatří. Budeme se zde zabývat výhradně takovými problémy, které nejsme schopni řešit v polynomiálně omezeném čase na deterministickém Turingově stroji. Příkladem takového problému je řešení otázky tautologie. Jestliže máme zjistit, zda je logická funkce tautologická, musíme projít všechny variace vstupních hodnot a ověřit, že pro všechny tyto vstupy je jejím logickým výstupem pravda. Tento problém nejde řešit v polynomiálně omezeném čase a má exponenciální složitost. Doba výpočtu asymptoticky odpovídá výrazu 2^n, kde n je počet vstupních proměnných. Pokud bychom počet vstupních proměnných zdvojnásobili, doba výpočtu vzroste na druhou mocninu doby původní. Tabulka 2.1 ukazuje, jak dlouho trvá řešení problémů této složitosti hrubou silou. Časový údaj v tabulce je třeba brát jako platný pouze řádově, přesný čas se liší v závislosti na parametrech počítače, který problém řeší.

Je vidět, že již pro relativně nízký počet vstupních hodnot je doba trvání výpočtu prakticky nepoužitelná. Proto tyto problémy řešíme často za pomoci heuristik, které nemusí dávat vždy exaktní řešení, ale dávají řešení dostatečně blízké tomu nejlepšímu v polynomiálně omezeném čase.

velikost vstupních dat n	čas výpočtu
10	1s
20	17 minut
30	12 dní
40	25 let
50	40000 let

Tabulka 2.1: Doba trvání výpočtu při asymptotické složitosti $O(2^n)$

2.2 Základní definice Booleovy algebry a logických funkcí

Následující text velice stručně popisuje Booleovu algebru a logické funkce. Jeho účelem není čtenáři vysvětlit tyto pojmy, slouží pouze pro připomenutí některých základních principů. Více se o této problematice lze dočíst v [9].

2.2.1 Booleova algebra

Definice: Booleova algebra je struktura $<B, +, *, \bar{}, 0, 1>$, kde platí:

- B je nosičem algebry
- $+$ a $*$ jsou binární operace na B
- $\bar{}$ je unární operace na B
- 0,1 jsou nulární operace na B, kde pro všechna $a, b, c \in B$ vždy platí:
  1. komutativní zákon: $a + b = b + a, a * b = b * a$
  2. asociativní zákon: $a + (b + c) = (a + b) + c, a * (b * c) = (a * b) * c$
  3. distributivní zákon: $a + (b * c) = (a + b)(a + c), a * (b + c) = (ab) + (ac)$
  4. algebra má neutrální prvky 0 a 1: $a + 0 = a, a * 1 = a$
  5. unární operace $\bar{}$ vytváří komplement: $a + \overline{a} = 1, a * \overline{a} = 0$

Takto je Booleova algebra plně definována. Za pomoci těchto pěti zákonů je možné vyvodit i všechny další odvozené zákony algebry. Mezi ně například patří:

1. agresivita: 0 a 1: $a * 0 = 0, a + 1 = 1$
2. idempotence: $a * a = a, a + a = a$
3. dvojí negace: $\overline{\overline{a}} = a$
4. absorbce: $a + a * b = a$
5. absorbce negace: $a + \overline{a} * b = a + b$
6. de Morganův zákon: $\overline{(a + b)} = \overline{a} * \overline{b}, \overline{(a * b)} = \overline{a} + \overline{b}$

2.2.2 Logické funkce

Booleovy funkce n argumentů jsou funkce definované na n-ticích, které jsou vytvořeny z prvků množiny $\{0,1\}$ a jsou zobrazené do množiny $\{0,1\}$. Tyto funkce se rovněž nazývají logickými funkcemi.
Libovolná proměnná z množiny n sama o sobě tvoří Booleovský výraz. Jsou-li proměnné a, b Booleovskými výrazy, pak také $\overline{a}$, $(a+b)$ a $(a*b)$ jsou Booleovskými výrazy. Každý Booleovský výraz představuje nějakou funkci $f : B^n \rightarrow B$ (Booleovu nebo-li logickou funkci) n proměnných nad $<B, +, *, \bar{}, 0, 1>$, pokud proměnné $x_1, x_2, ...x_n$ chápeme jako proměnné, které nabývají hodnot nosiče B.
Jednu funkci je možné vyjádřit mnoha způsoby, proto se používá standardní tvar – normální forma. Jedná se o takové vyjádření funkce, v němž se vyskytují ve dvou úrovních operace součtu a součinu (jde tedy o součet součinů, nebo součin součtů).
V případě disjunktivní normální formy se jedná o součet součinů, kdy každý term ve výrazu se nazývá implikantem, neboť implikuje celou funkci – tedy pokud tento term nabývá pro určité ohodnocení vstupů výsledku 1, nabývá celá funkce rovněž výsledku 1. S funkcemi zadanými v DNF pracuje i tento program, který využívá pro svou práci formátu PLA (popsán v 2.2.4).
V logických obvodech se rovněž běžně používají vícevýstupové funkce. Vícevýstupová funkce je pouze rozšířena o možnost mít větší množství výstupů, které jsou definovány nad stejnou algebrou, ale jsou na sobě vzájemně nezávislé, pouze pro svou funkci využívají stejných vstupů. Obvykle se snažíme o to vyjádřit tyto různé výstupy za pomoci podobných funkcí, aby bylo možné později v logickém obvodu využívat společných prvků pro větší množství výstupů.

2.2.3 Grafy

Grafem v *teorii grafů* rozumíme objekt, který je popsaný množinou vrcholů a hran. *Prostý graf* je uspořádaná dvojice (V, H), kde V je množina vrcholů grafu G a H je množina hran téhož grafu. Takto definovaný graf však nedovede postihnout více hran téhož typu, proto v obecnějším případě hovoříme o tzv. *Obecném grafu*, který je tvořen uspořádanou trojicí (V, H, ε), kde ε je zobrazení incidence grafu.

Grafy lze dále rozdělit dle několika dalších parametrů. Uvedeme zde ty nejzákladnější parametry a pojmy zmiňované v této práci:

- **Orientace hran**
  - Orientované grafy - hrany orientovaného grafu jsou uspořádané dvojice vrcholů
  - Neorientované grafy - hrana je pouze dvouprvková množina vrcholů
- **Existence kružnic**
  - Cyklické grafy – grafy, které obsahují aspoň jednu kružnici
  - Acyklické grafy – grafy, které neobsahují žádnou kružnici
- **podgraf** - graf $F = (V_1, H_1, \varepsilon_1)$ je podgrafem grafu $G = (V_2, H_2, \varepsilon_2)$, jestliže $V_1 \subseteq V_2, H_1 \subseteq H_2$ a ε_1 je zúžením zobrazení ε_2
- **úplný podgraf** - graf F je úplným podgrafem G, je-li jeho podgrafem a zároveň obsahuje všechny hrany grafu G, jejichž oba krajní uzly patří do $V(F)$

Existuje více parametrů, kterými se grafy odlišují. Pro naše potřeby však stačí tyto, neboť budeme využívat pouze malou část z celé grafové teorie. Grafy jsou popsány v [4].
Grafy obecně slouží jako abstrakce pro nějaký problém. Mohou se používat například jako

zjednodušený model sítí, kde klíčovým bodem celého modelu jsou topologické vlastnosti jednotlivých objektů. V našem případě budou použity jako nástroj pro zaznamenávání společných proměnných, kdy uzly grafu budou představovat jednotlivé termy z DNF a hrany budou mezi takovými dvěma uzly, které mají společné proměnné. Více o způsobu konstrukce grafu v kapitole 3.1.

2.2.4 Formát PLA

Tento formát byl původně navržen pro program Espresso [10] a slouží jako nosič informací o logické funkci zadané v DNF. Jedná se o jednoduchý způsob dvouúrovňového popisu logické funkce. Formát PLA je uložen v textovém souboru s koncovkou *pla*. Tento textový soubor je pak načítán po jednotlivých řádcích, kde každý řádek reprezentuje parametr logické funkce. Po hlavičce, která obsahuje základní parametry funkce, následuje matice znaků, kde jednotlivé řádky odpovídají termům z DNF a sloupce představují jednotlivé vstupní proměnné. Specifikace celého formátu je možné najít v [8]. Zde uvedeme pouze částečnou specifikaci s ohledem na její možné využití pro naše účely:

- veškeré bílé znaky v souboru jsou ignorovány, pokud neslouží jako oddělovače při určování parametrů
- znak # uvozuje komentář
- všechna klíčová slova začínají znakem . (tečkou)
- povinná klíčová slova: (tato slova musí obsahovat každý PLA)
  - **.i** - udává počet vstupních proměnných
  - **.o** - udává počet výstupních proměnných
- nepovinná klíčová slova:
  - **.type** – udává typ souboru, podporované typy jsou *f, fd, fr a fdr*
    typ F – popisuje ONSET funkce, DCSET se ve funkci nevyskytuje a OFFSET je komplementem ONSETu
    typ FD – popisuje ONSET a DCSET funkce, OFFSET je komplementem sjednocení těchto dvou množin
    typ FR – popisuje ONSET a OFFSET funkce, DCSET je komplementem sjednocení těchto dvou množin
    typ FDR – plně popisuje funkci, tzn. její ONSET, OFFSET i DCSET, musí zcela pokrývat celý stavový prostor
  - **.p** - udává celkový počet termů v PLA
  - **.ilb** - definuje názvy vstupních proměnných
  - **.ob** - definuje názvy výstupních proměnných
  - **.e** - ukončuje vstup PLA, cokoliv za touto značkou je překladači ignorováno

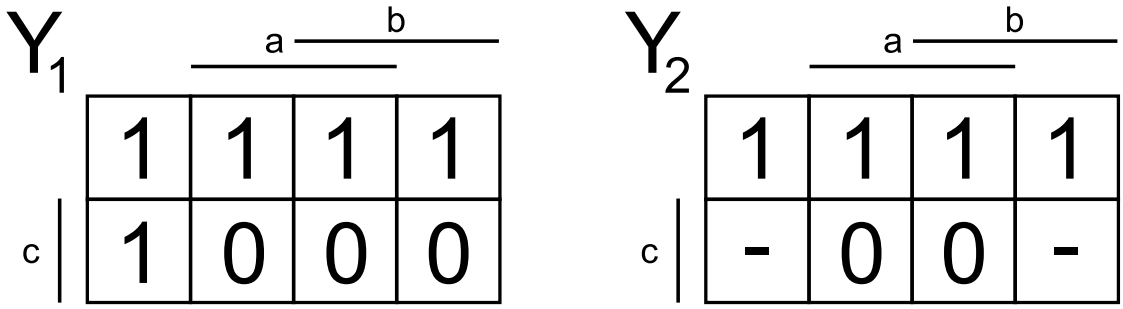
Po hlavičce obsahující tato klíčová slova začíná matice termů. Tato matice se načítá po řádcích, kde v každém řádku jsou po řadě uloženy stavy všech n vstupů, pak následuje oddělovač a další řada znaků pro výstupy. Příklad takového PLA souboru je na obrázku 2.1.

Na příkladu je vidět, že výsledná funkce má v tomto případě 3 vstupní a 2 výstupní proměnné. Jedná se o typ FD, tedy že je jednoznačně označen ONSET a DCSET, zatímco OFFSET je komplementem sjednocení těchto dvou množin. Celkem obsahuje funkce 3 termy, jejichž výpis

```
.i 3
.o 2
.type fd
.p 3

--0 11
00- 10
0-1 0-
.e
```

Obrázek 2.1: Příklad vstupního souboru ve formátu PLA

je od hlavičky oddělen prázdným řádkem.
V přední čtvercové části matice se nachází vstupní proměnné, které jsou od výstupní části odděleny bílými znaky. Ve vstupní části se mohou vyskytovat znaky z množiny $\{1, 0, -\}$ a vyjadřují, že proměnná na dané pozici je obsažena v termu v přímé formě, negované formě, nebo není v termu obsažena vůbec. Výstupní znaky mají rozdílné významy v závislosti na typu PLA. V tomto případě, kdy se jedná o FD, představují jedničky ONSET, pomlčky DCSET a nuly nemají žádný význam. Na obrázku 2.2 jsou zobrazeny Karnaughovy mapy pro tento PLA.



Obrázek 2.2: Karnaughovy mapy

2.2.5 Deklarace záměru

Cílem celého projektu je zanalyzovat možnosti a přínosy grafového přístupu k řešení logických funkcí především z hlediska otázek tautologie a ortogonalizace. Dále pak naimplementovat program, který dovede načíst a zanalyzovat datový vstup ve formátu PLA, vytvořit na základě tohoto vstupu graf odpovídající dané struktuře logické funkce a tento graf pak aplikovat při řešení otázky tautologie. V závěru pak porovnat rychlost tohoto způsobu řešení s rychlostí ostatních stávajících řešení.

3 Analýza a návrh řešení

Tato kapitola se již podrobně zabývá celou grafovou strukturou, její tvorbou i dalším využitím. Jsou zde podrobně rozebrány některé problémy i návrhy, jakým způsobem postupovat při jejich řešení a jak využívat zkonstruovaný graf. Rovněž jsou zde popsány jednotlivé algoritmy a jejich přesný postup.

3.1 Konstrukce grafu

V této části je vysvětleno, jakým způsobem se graf utváří, a jsou zde rovněž popsány vlastnosti tohoto grafu. Jedná se pouze o analýzu a teoretický postup. O samotné implementaci je pojednáno v kapitole 4.

3.1.1 Konstrukce na základě vstupních proměnných

Jednotlivé termy logické funkce nechť jsou reprezentovány v grafu jako uzly. Počet uzlů tedy jednoznačně odpovídá počtu krychlí funkce zadané v DNF a rovněž počtu řádků matice ve formátu PLA.
Pozn.: Od této chvíle budeme pojmy *term* a *uzel v grafu* považovat za ekvivalentní.

Mezi těmito uzly vytvoříme hrany na základě jejich shody ve vstupních proměnných. Za shodu považujeme libovolnou proměnnou, kterou mají dva termy společnou - nezávisle na tom, zda v přímé či negované formě. Pokud tedy libovolná dvojice termů má v sobě obsaženu aspoň jednu společnou proměnnou, existuje mezi takovými dvěma termy (uzly) hrana. Viz 3.1.
Jako příklad vezměme funkci, jejíž zápis v DNF vypadá takto:

$$y = abc + bc\overline{d} + \overline{c}d + de + af + \overline{e}f$$

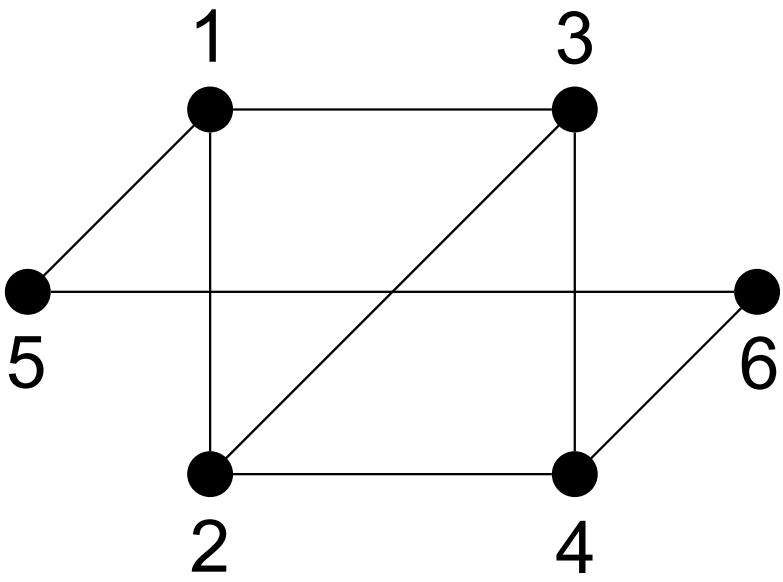
Pro názornost a snazší orientaci v dalších zobrazeních grafu označme jednotlivé termy čísly 1 až 6, jak je naznačeno v tabulce 3.1.

Zkonstruujeme-li z takto zadaného vstupu prostý graf (obrázek 3.1), vznikne nám graf o šesti uzlech odpovídajících jednotlivým termům. Hrany budou mezi takovými dvěma uzly, jejichž termy obsahují aspoň jednu stejnou proměnnou. Tedy například dvojice termů 1-3 má mezi sebou hranu, neboť oba termy obsahují proměnnou c. To, že *term1* ji obsahuje v přímé a *term3* v negované formě, nerozhoduje. Naopak dvojice termů 1-4 mezi sebou mít hranu nebude, neboť žádnou společnou proměnnou nemají.

Avšak takto definovaný graf není pro naše účely dostačující. Tento graf sice obsahuje informaci o tom, které dva termy jsou svázány nějakou společnou proměnnou, ale již nedává žádné informace o tom, která proměnná to je, či zda je jich více, apod. Proto zavedeme ohodnocení

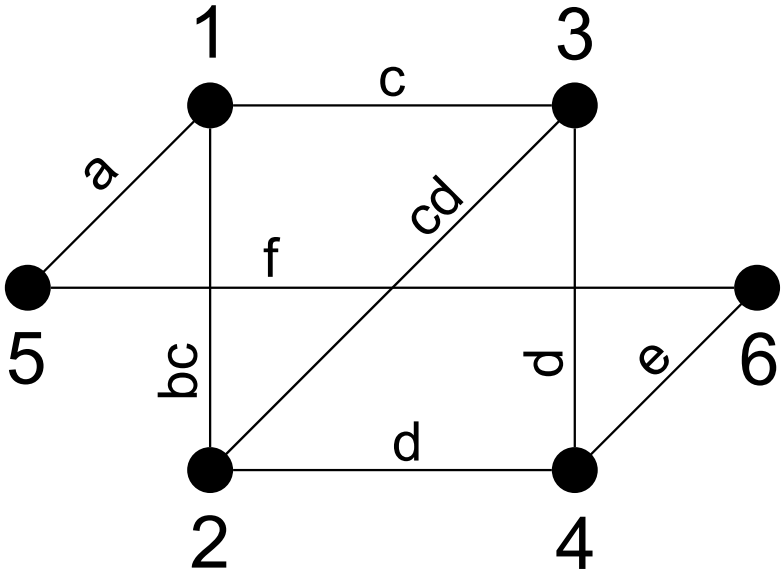
číslo termu/uzlu	term
1	abc
2	$bc\overline{d}$
3	$\overline{c}d$
4	de
5	af
6	$\overline{e}f$

Tabulka 3.1: Číselné označení termů



Obrázek 3.1: Ukázka grafu s neohodnocenými hranami

hran. Každou hranu ohodnotíme znakem reprezentujícím, která proměnná tuto dvojici uzlů spojuje. V případě, že jedna dvojice termů má více než jednu společnou proměnnou, budou uzly reprezentující tyto termy spojeny rovněž více hranami - vzniká tak tedy multigraf (obrázek 3.2). Na obrázku budeme pro přehlednost i nadále kreslit hranu jako v prostém grafu, pouze jednotlivé znaky s ohodnoceními hran sloučíme do řetězce.



Obrázek 3.2: Ukázka grafu s ohodnocenými hranami

3.1.2 Konstrukce na základě výstupních proměnných

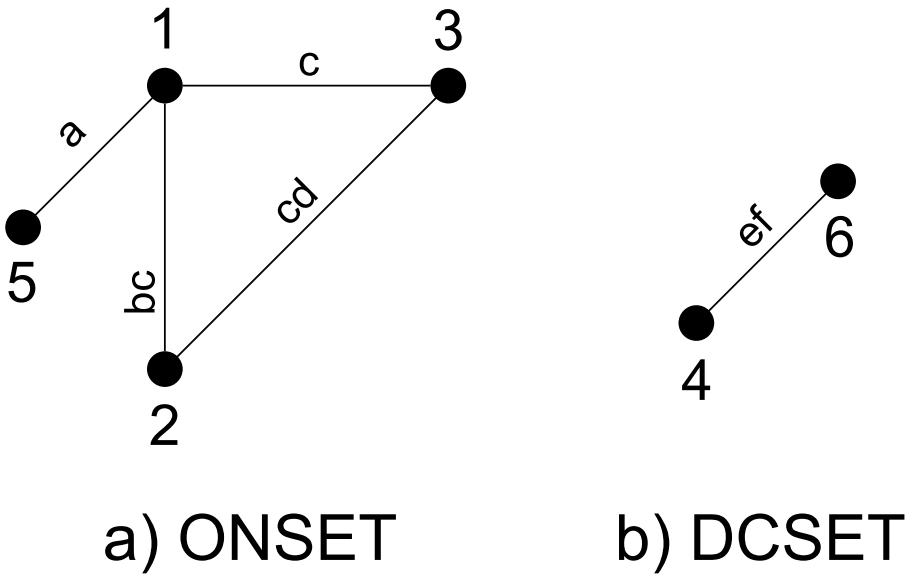
V tuto chvíli máme zaveden graf, který respektuje vztahy mezi termy z hlediska vstupních proměnných, jež obsahují. Dále jej již budeme nazývat *Graf závislostí*. Tento graf však lze ještě v mnoha případech zjednodušit, respektive rozdělit dle výstupů jednotlivých termů.
Při řešení mnohých problémů nemáme vždy pro všechny tři možné výstupy {1,0,dc} využití a je zbytečné pak do grafu a všech výpočtů zahrnovat všechny termy, pokud některé z nich nebudeme potřebovat. Například při řešení problému minimalizace jsme se rozhodli využít pouze ONSETu a DCSETu. Celý OFFSET tak můžeme zanedbat a urychlit tak celý výpočet díky větší jednoduchosti grafu.
Proto jsme se rozhodli, že veškeré termy budou rozděleny do tří skupin, které odpovídají každá jednomu ze tří možných typů výstupu, a vzniknou tak tři menší podgrafy. Tyto podgrafy budou pochopitelně úplnými podgrafy původního grafu závislostí a vzniknou tak, že jednotlivé uzly (termy) budou rozděleny do tří disjunktních skupin dle svého výstupu. U vícevýstupních funkcí s n nezávislými výstupy bude toto provedeno pro výstup 1 až n zvlášť. Tedy vznikne celkem $3n$ podgrafů.

Tuto vlastnost demonstruje příklad zadaný formou PLA na obrázku 3.3.

```
.i 6
.o 2
.type fd
.p 6

111--- 11
-110-- 11
--11-- 10
-0-010 -1
1----1 10
0-0-01 --
.e
```

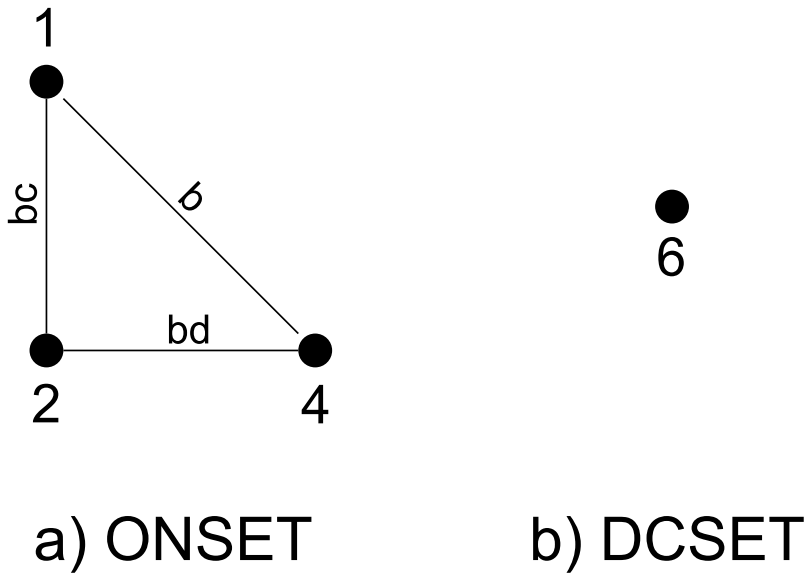
Obrázek 3.3: Ukázkové PLA s vícevýstupovou funkcí

Tato logická funkce má 3 vstupní a 2 nezávislé výstupní proměnné. Je definována za pomoci šesti termů a již na první pohled je patrné, že graf je možné rozdělit na podgrafy z hlediska rozdílných výstupů. Na obrázku 3.2 je vidět, jak vypadá původní graf. Dva jeho podgrafy pro ONSET a DCSET z hlediska prvního výstupu jsou zobrazeny na obrázku 3.4. Podgraf pro OFFSET je prázdný, neboť PLA typu FD v sobě OFFSET neobsahuje, definuje pouze ONSET a DCSET.



Obrázek 3.4: Grafy pro první výstup funkce

Některé termy se ve svých výstupech liší. Z toho vyplývá, že podgrafy pro druhý výstup budou vypadat jinak. Viz obrázek 3.5.



Obrázek 3.5: Grafy pro druhý výstup funkce

Z předchozího příkladu je patrné, jakým způsobem podgrafy vznikají. Toto zjednodušení může vést v nejlepším případě až na tři podgrafy (pokud je zadáno PLA typu fdr) a pokud je podíl všech tří typů výstupů v PLA stejný, obsahují tyto tři podgrafy každý třetinu z celkového počtu uzlů, což znatelně snižuje dobu výpočtu oproti situaci, kdy bychom uvažovali uzly všechny v rámci jednoho globálního grafu. Tato časová úspora je k-násobná, kdy k odpovídá počtu výstupních hodnot termů, neboť veškeré úpravy a výpočty je třeba provádět s každým výstupem zvlášť, a proto dojde k úspoře postupně u každého z nich.

3.1.3 Kliky v grafu

Způsob vytváření grafu byl již popsán v předchozích bodech. V tomto bodě bych se rád zmínil o jedné specifické vlastnosti takto vytvořeného grafu. Graf má několik zjevných vlastností:

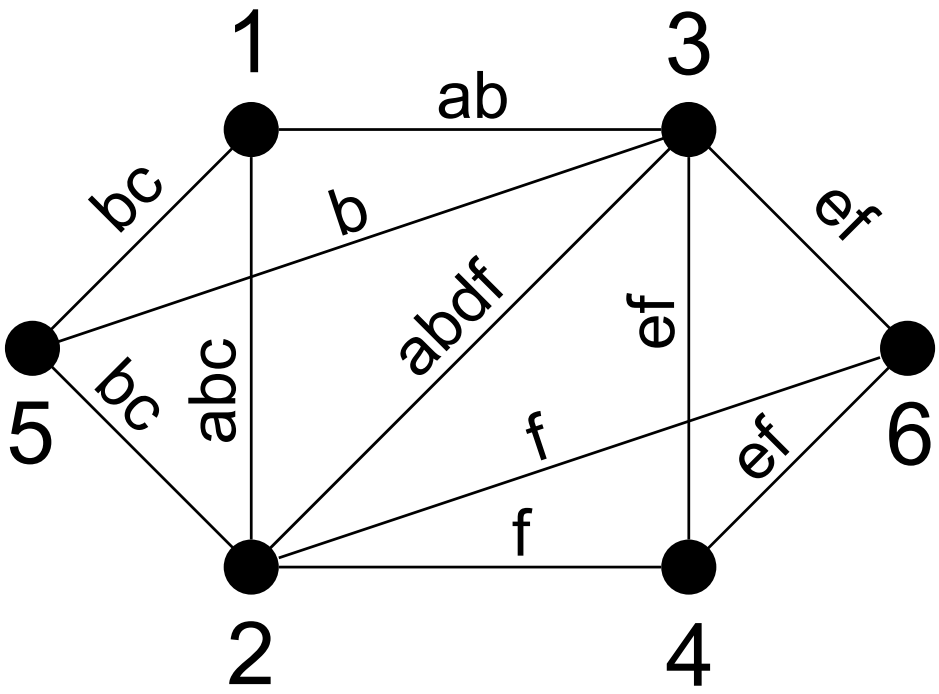
- je neorientovaný
- s rostoucím počtem don't care ve vstupních proměnných jednotlivých termů je řidší a výpočty na něm tudíž budou probíhat rychleji
- je cyklický
- je multigrafem

Kromě těchto vlastností má ještě jednu latentní vlastnost, která vyplývá ze způsobu, jakým je konstruován a kterou budeme nadále využívat později při implementaci. Pokud se zaměříme na graf z hlediska jednotlivých vstupních proměnných, zjistíme, že podgraf, který tak vznikne, je úplným grafem. Jinak řečeno, termy jsou z hlediska jedné konkrétní proměnné vždy v klice.

Uvažujme jednoduchou jednovýstupovou funkci, která má všechny své termy v ONSETu a kterou definujeme za pomoci následující DNF:

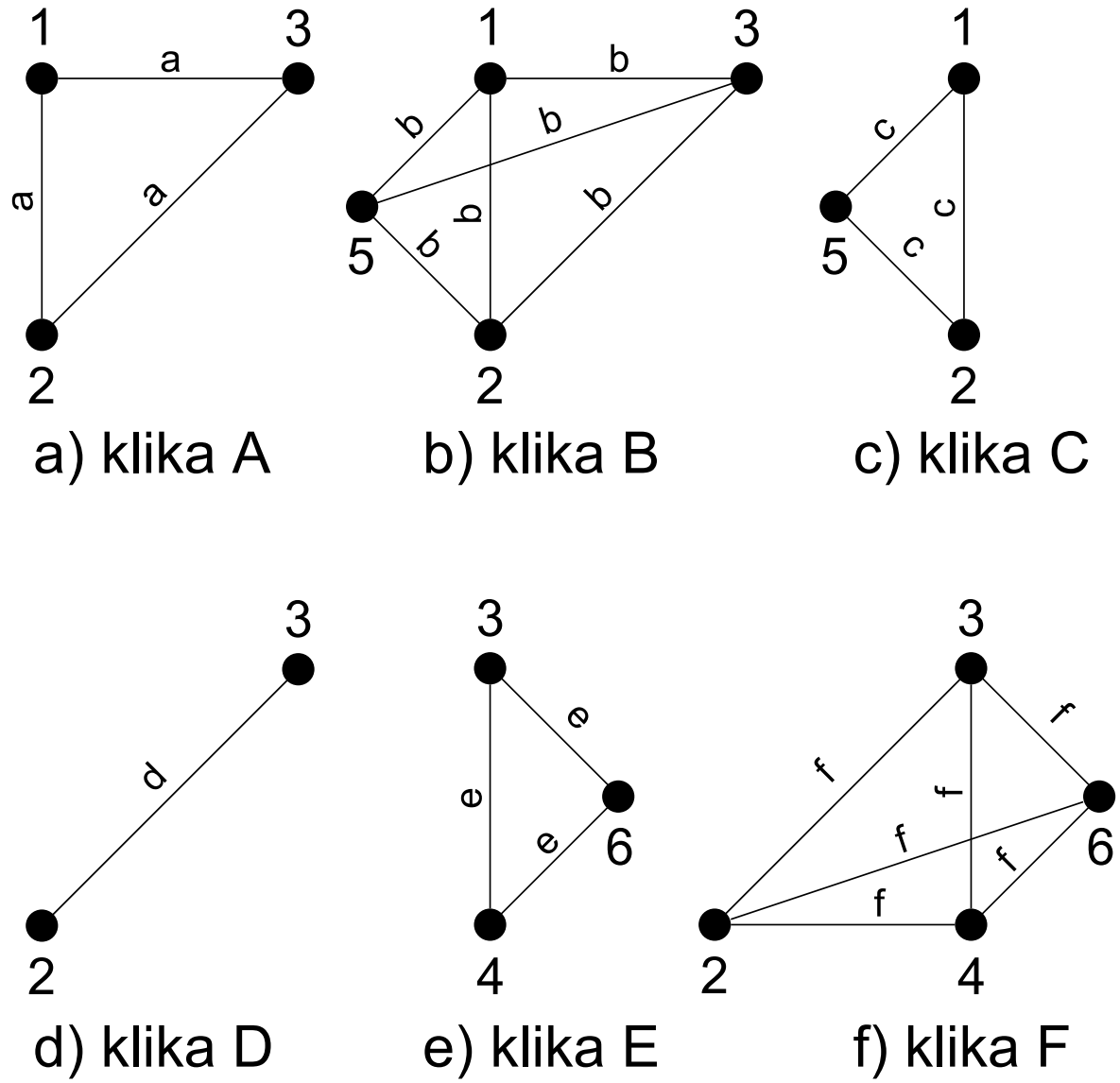
$$Y = ab\bar{c} + \bar{a}bc\bar{d}f + \bar{a}\bar{b}d\bar{e}\bar{f} + ef + \bar{b}c + \bar{e}f$$

Termy opět označíme v pořadí čísly 1-6. Tato funkce bude mít graf závislostí dle obrázku 3.6.



Obrázek 3.6: Tvorba klik v grafu - kompletní graf závislostí

Avšak zaměříme-li se na jednotlivé proměnné *a-f* a budeme se na graf dívat z pohledu těchto proměnných, zjistíme, že v grafu na obrázku 3.7 vznikají pouze kliky.



Obrázek 3.7: Tvorba klik v grafu - pohledy z hlediska jednotlivých proměnných

Tvrzení:
Vytvoříme-li podgraf F z grafu závislostí G tak, že zvolíme libovolnou vstupní proměnnou x a do grafu F přeneseme pouze takové uzly představující termy, které tuto proměnnou obsahují, a takové hrany, které jsou ohodnoceny touto proměnnou x, bude tento podgraf úplným grafem. Mezi každou dvojicí uzlů bude tedy existovat hrana a z hlediska původního grafu se jedná o kliku.

Důkaz:
Pokud by existoval v novém podgrafu F pouze jeden uzel, nemá smysl o klice uvažovat. V případě dvou uzlů budeme postupovat takto: vztah „mít stejnou proměnnou x ve svém termu“ označíme jako binární relaci R dvou uzlů. Je evidentní, že relace R je symetrická, protože pokud platí, že term t_1 obsahuje proměnnou z termu t_2, pak t_2 musí obsahovat i tu samou proměnnou z t_1, tedy formálně zapsáno platí, že:

$$\forall V_1, V_2 \in F; aRb \Rightarrow bRa$$

Zbytek důkazu provedeme tak, že dokážeme, že relace aRb je tranzitivní. Již jsme dokázali, že pokud je graf tvořen pouze dvěma uzly, tvrzení platí. Oba uzly v této relaci jsou - to vyplývá z definice způsobu, jakým je graf tvořen. Pro větší počet uzlů platí následující:
V grafu již je n uzlů označených $t_1...t_n$, které tvoří úplný graf ($n \geqq 2$) s hranami ohodnocenými proměnnou x. Přidáme-li nějaký další uzel představující term t_i do grafu F na základě společné proměnné x s termem t_j, který už v grafu je, musí v sobě t_i obsahovat proměnnou x, kterou již obsahuje t_j. To vyplývá z definice grafu F. A protože v grafu neexistují jiné hrany než ty s ohodnocením x, nemohou v něm existovat ani uzly, které v sobě nemají x, neboť graf je úplný, tedy je jasné, že proměnnou x v sobě neobsahují pouze termy t_i a t_j, ale rovněž všechny další termy z $t_1...t_n$. Naše relace je tudíž i tranzitivní:

$$\forall V_1, V_2, V_2 \in F; (aRb \wedge bRc) \Rightarrow aRc$$

3.1.4 Reprezentace grafu v paměti

Zjištění pravidla o vytváření klik v grafu nám pomůže při implementaci. V 3.1.3 jsme dokázali, že graf závislostí neobsahuje žádnou hranu, která by nebyla součástí kliky. Této specifické vlastnosti grafu můžeme využít tak, že jej není třeba implementovat jako obecný graf, kde každý objekt *Uzel* má spojový seznam svých sousedů, ale vystačíme se seznamy uzlů tvořících kliku. Pro prezentaci funkce, která má m termů a n proměnných, tak stačí vytvořit m objektů *Term* představujících uzly v grafu a n objektů *Variable* představující jednotlivé proměnné. Každý objekt *Variable* pak obsahuje spojový seznam termů, které mají tuto proměnnou v sobě obsaženu a tak tvoří kliku. Tímto způsobem postihneme všechny hrany v grafu a zjednodušíme celou reprezentaci grafu v paměti, čímž se i urychlí výpočty prováděné nad tímto grafem. Objekty *Term* obsahují spojové seznamy objektů *Variable*, které udávají, zda danou proměnnou mají ve svém vstupu obsaženu.
Tyto seznamy vzájemných vztahů objektů *Term* a *Variable* jsou ještě navíc rozděleny do tří podseznamů, které nesou informaci o tom, zda je proměnná v termu obsažena v přímé formě, negativní formě, či v ní není obsažena vůbec.
V rámci objektu *Variable* je tato informace ještě rozšířena o výstup daného termu, kdy pro každý ze tří možných typů výstupu existuje jeden seznam, kterým se rozdělují uzly na tři vzájemně disjunktní množiny pro ONSET, OFFSET a DCSET. Tyto tři seznamy jsou uloženy o-krát, kde o je počet výstupů funkce.

Pro lepší orientaci v problému vezměme následující příklad z obrázku 3.8, kde je zobrazeno vstupní PLA a Karnaughova mapa. Pokud by zadání vypadalo takto, vytvoříme 5 objektů *Term* a 3 objekty *Variable*, očíslujeme je Term1-5 a Var1-3.

```
.i 3
.o 2
.type fdr
.p 5

--0 1~
-01 01
-11 -0
1-0 ~1
0-0 ~1
.e
```

x_1	x_2	x_3	Y_1	Y_2
0	0	0	1	1
0	0	1	1	-
0	1	0	1	1
0	1	1	1	-
1	0	0	0	1
1	0	1	0	1
1	1	0	-	0
1	1	1	-	0

Y_1

x_1 \ x_3 x_2				
	1	1	1	1
x_1	0	0	-	-

Y_2

x_1 \ x_3 x_2				
	1	-	-	1
x_1	1	1	0	0

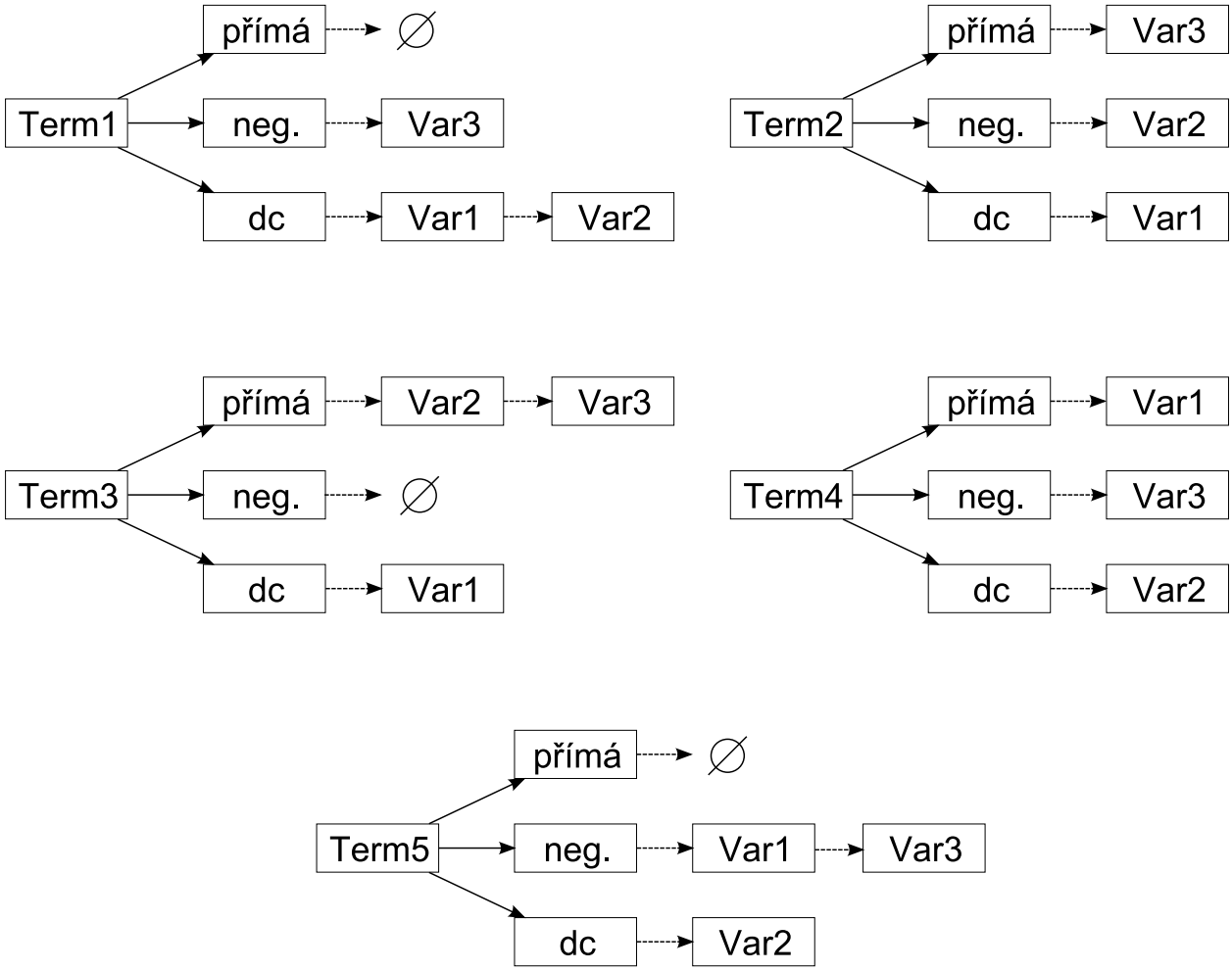
Obrázek 3.8: Příklad pro spojové seznamy

U objektů *Term* je situace jednoduchá. Každý z nich má tři seznamy, ve kterých jsou umístěny ukazatele na objekty *Variable* a tyto ukazatele jsou rozřazeny podle toho, zda je proměnná obsažena ve výrazu daného termu v přímé formě, negované formě, nebo tam není obsažena vůbec. Spojové seznamy ukazatelů by vypadaly tak, jak je zobrazeno na obrázku 3.9.

V případě objektů *Variable* již je spojových seznamů více. Opět zde jsou 3 seznamy ukazatelů na *Term*, aby *Variable* nesl informaci o tom, které *Termy* jej mají či nemají obsaženy a v jaké formě. Tyto seznamy jsou analogické s těmi v *Termech* a jsou konzistentní. Jak již bylo řečeno v 3.1.2, rozdělujeme dále tyto seznamy, které odpovídají klikám v grafu, podle výstupů jednotlivých *Termů*. Term, který má nějaký význam pro daný výstup, může nabývat 3 výstupních hodnot {0, 1, don't care}. Proto vzniknou 3 seznamy, které těmto jednotlivým výstupům odpovídají a v každém z nich budou původní 3 seznamy s významem vztahu dané proměnné k *Termu*. Celkově tedy má *Variable* již 9 seznamů. A protože jsou výstupy termů vzájemně nezávislé, je třeba těchto 9 seznamů udělat pro každý z nich, v našem případě, kdy máme 2 výstupy, vznikne 18 seznamů ukazatelů na objekt *Term*. Seznamy pro první dvě proměnné jsou zobrazeny na obrázku 3.10.

Tento způsob konstrukce přináší následující vlastnosti:

- díky rozdělení výstupů se konstruují menší grafy a algoritmy nad nimi probíhají rychleji
- pro velké množství algoritmů nepotřebujeme všechny tři množiny (ONSET, OFFSET a



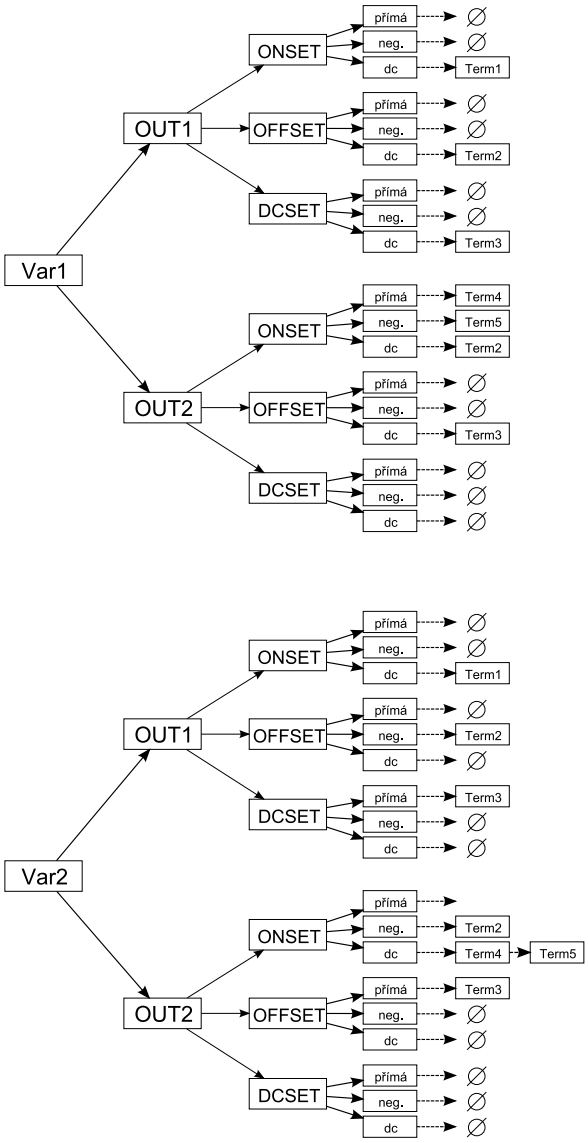
Obrázek 3.9: Spojové seznamy termů

DCSET), tudíž je graf opět menší a algoritmy rychlejší

- lze vyhledat všechny *termy*, které mají společnou určitou proměnnou s určitým *termem*, v čase O(1)
- vyhledat všechny *termy*, které naopak nějakou proměnnou neobsahují, lze rovněž v čase O(1)
- vyhledávání všech sousedů nějakého *termu* je možné v čase O(n*p) (n odpovídá počtu vstupů funkce, p je počet termů)
- vyhledat všechny *termy*, které sousední s daným termem nejsou, lze také v O(n*p)
- při vyhledání termu víme ihned, jaký má vztah k proměnné, za pomoci které jsme jej vyhledali, díky konstrukci seznamů

3.1.5 Časová náročnost konstrukce

Graf pro jednotlivé výstupy konstruujeme způsobem, jaký byl popsán v části 3.1.4, tedy vyplňujeme seznamy podle jednotlivých vstupů. Pro vytvoření grafu musíme projít celou vstupní matici, která má p řádků a i sloupců, kde p odpovídá celkovému počtu termů a i celkovému



Obrázek 3.10: Spojové seznamy proměnných

počtu vstupních proměnných. Matici procházíme znak po znaku a každý znak znamená zařazení do jednoho seznamu objektů Variable a jednoho seznamu objektů Term. Toto musíme udělat o-krát, kde o je počet výstupů funkce. Je tomu tak proto, že jsme v části 3.1.2 řekli, že budeme uchovávat uzly a hrany pro každý výstup zvlášť.

Celá operace konstrukce tedy bude trvat $i * o * p * k$ cyklů, kde k je konstanta.

3.2 Problém tautologie

Tautologie představuje první problém, který bychom rádi vyřešili. Pokud v logice je nějaký výraz tautologie, znamená to, že nabývá vždy pravdivé hodnoty nezávisle na ohodnocení vstupních proměnných. Příkladem takové tautologie může být následující vzorec:

$$Y = a + \bar{a}$$

Hodnota Y bude vždy jedna, ať se rozhodneme proměnnou a ohodnotit jakoukoliv hodnotou z množiny {0,1}.

3.2.1 Rozbor problému

Zjistit, zda funkce je tautologie, je problém třídy co-NP, kdy pro ověření tohoto tvrzení nezbývá, než skutečně projít všech 2^n různých možných ohodnocení vstupů a ověřit, že výstup je vždy jedna. Avšak ne vždy je třeba procházet celý stavový prostor hrubou silou. Funkce, která obsahuje termy t_1 až t_n, je splněná právě tehdy, pokud je splněn aspoň jeden z jejích termů - to je základní vlastnost implikantů, které jednotlivé termy představují, a je to jedna ze dvou vlastností DNF, které budeme využívat, protože stačí, abychom během ohodnocování proměnných splnili jediný term a přestože není ohodnocení proměnných úplné, nemusíme v ohodnocování dále pokračovat, neboť je jisté, že funkce již bude pro toto ohodnocení vždy splněna. To nám ušetří značné množství stavů, které nemusíme dále testovat.

Druhou vlastností je nezávislost termů na některých vstupních proměnných. Pokud je term splněn pro konkrétní ohodnocení proměnných, kde neobsahuje don't care, tak na ostatních proměnných, kde don't care obsahuje, už nezáleží a zůstává v daném případě splněn vždy.

Příkladem takového termu zadaného formou PLA může být:
`1-01011011101--- 1`

Tento term je z funkce, která obsahuje 16 vstupních proměnných. Nicméně jeho výsledek je závislý pouze na ohodnocení 12 z nich. Pokud označíme vstupy abecedně zleva doprava, tak proměnné *b, n, o* a *p* vůbec neovlivňují výsledek tohoto termu. To znamená, že pokud jsou ostatní proměnné v termu „správně" ohodnoceny, tak zbylé čtyři proměnné už nemusíme ohodnocovat a víme, že výsledek termu je vždy jedna. Tyto čtyři proměnné je možné ohodnotit dalšími 2^4 možnými způsoby. Pokud toto nyní dělat nemusíme, tak ve srovnání s procházením hrubou silou jsme ušetřili 2^4 variací ohodnocení, které už nemusíme testovat, neboť víme, že by byly už vždy splněny. Obecně tedy term, který obsahuje k proměnných a implikuje funkci o n vstupech, implikuje $2^{(n-k)}$ stavů. A nám při procházení prostorem stačí „vkročit" do libovolného z těchto stavů a víme, že další ohodnocování proměnných již v tomto případě nemá smysl.

3.2.2 Popis algoritmu

Náš algoritmus pracuje tak, že se snaží dokázat, že funkce tautologií není. Tedy hledá takové ohodnocení proměnných, které vede k tomu, že výsledek žádného z termů není jedna.
Dělá to tak, že vhodně postupně v iteracích ohodnocuje jednotlivé proměnné tak, aby tímto krokem vždy vyřadil nějaký ze zbývajících termů coby kandidátů na splnění aktuálního ohodnocení. V počátečním stavu jsou kandidáty všechny termy. Algoritmus se vždy zaměří na některý z nich a nastaví některé z proměnných tohoto termu takovou polaritu, aby výsledkem tohoto termu již nemohla být pravda. Takto „znehodnocuje" jednotlivé termy až do chvíle, kdy buď

nezbývá žádný platný term a v takovém případě končí, neboť nalezl ohodnocení, jehož výsledkem není jedna, nebo do chvíle, kdy zbývající kandidáty není schopen již vyřadit ze seznamu kandidátů. Pak je tato větev již skutečně tautologická a v takovém případě vrací některá rozhodnutí o ohodnoceních zpět (backtracking) a volí jiné způsoby, jakými proměnné ohodnocovat. Jestliže algoritmus prošel všechny větve stavového prostoru, ve kterých se mohlo nacházet řešení a nezdařilo se mu najít ohodnocení, které hledal, prohlásí funkci za tautologii.

Přesný postup algoritmu:

1. Seřaď sestupně termy podle množství don't care ve vstupech. To děláme proto, aby naše ohodnocování proměnných seřízlo co největší část stavového prostoru.
2. Dej každému termu příznak, zda může být ještě splněn (v počátku všichni mohou), vytvoř globální proměnnou, do které se bude ukládat aktuální ohodnocení všech proměnných (zpočátku jsou všechny neohodnocené) a zkonstruuj stavový zásobník, pro předchozí stavy.
3. Vezmi ze seznamu první term, který ještě stále může být splněn, najdi první proměnnou tohoto termu, která ještě nebyla ohodnocena a ohodnoť ji tak, aby výsledek tohoto termu byl nula. Příznak splnitelnosti termu nastav na nesplnitelný.
4. Proveď distribuci tohoto ohodnocení proměnné všem termům, které jsou přes tuto proměnou s tímto termem v klice. Pokud jim nastavení této polarity proměnné nevyhovuje, nastav jejich příznaky také na nesplnitelné.
5. Zaznamenej nastavení nové proměnné do globální proměnné s ohodnocením a ulož všechny změny, které toto ohodnocení přineslo, na stavový zásobník.
6. Opakuj body 3-5, dokud
   (a) nezbývá už žádný term, který by byl splnitelný, v takovém případě přeruš algoritmus a navrať aktuální ohodnocení globální proměnné. Pro toto ohodnocení není výsledek funkce 1 a tudíž se nejedná o tautologii.
   (b) pro term, který ještě stále může být splněn, neexistuje taková neohodnocená proměnná, která by způsobila, že jeho výsledek bude nula. V takovém případě pokračuj bodem 7.
7. Odeber ze zásobníku poslední stav a proveď návrat k předchozímu termu.
   (a) Zásobník nebyl prázdný a bylo možné poslední stav odebrat, pokračuj bodem 8.
   (b) Zásobník už byl prázdný, funkce je tautologie, ukonči algoritmus.
8. Najdi v pořadí další proměnnou tohoto termu, která zajistí jeho nesplnitelnost.
   (a) Pokud taková proměnná existuje, použij ji pro ohodnocení v bodu 3 a pokračuj tímto bodem.
   (b) Pokud již neexistuje, pokračuj v návratu bodem 7.

Příklad:
Mějme funkci zadanou dle PLA z obrázku 3.11.

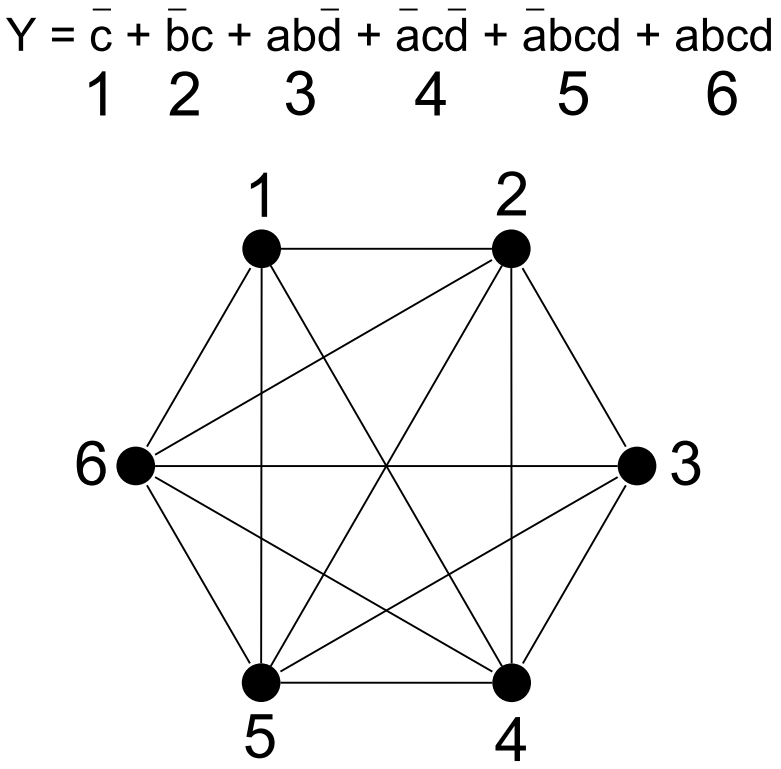
```
.i 4
.o 1
.p 5
.type f

11-0 1
0111 1
-01- 1
0-10 1
--0- 1
1111 1
```

Obrázek 3.11: Ukázkové PLA pro příklad tautologie

Pokud proměnné opět označíme abecedně, dá se příklad převést na jednoduché DNF takto:

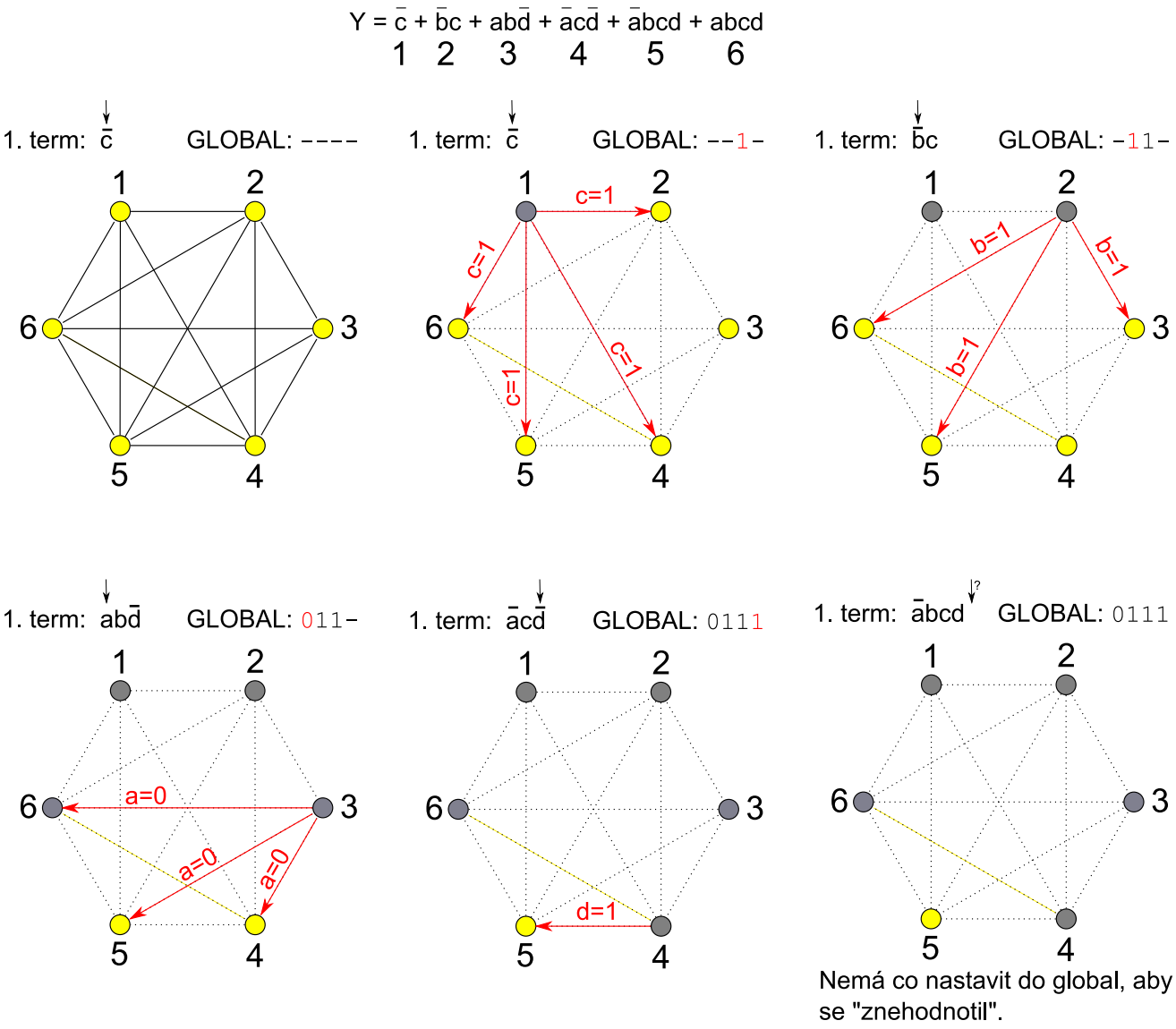
$$y = ab\overline{d} + \overline{a}bcd + \overline{b}c + \overline{a}c\overline{d} + \overline{c} + abcd$$

Vytvoříme *graf závislostí* těchto termů. Termy pak seřadíme dle bodu 1 v postupu algoritmu a číselně je označíme čísly 1-6, jak je znázorněno na obrázku 3.12. Pro přehlednost již nezobrazujeme v obrázcích ohodnocení hran.



Obrázek 3.12: Graf závislostí pro příklad tautologie

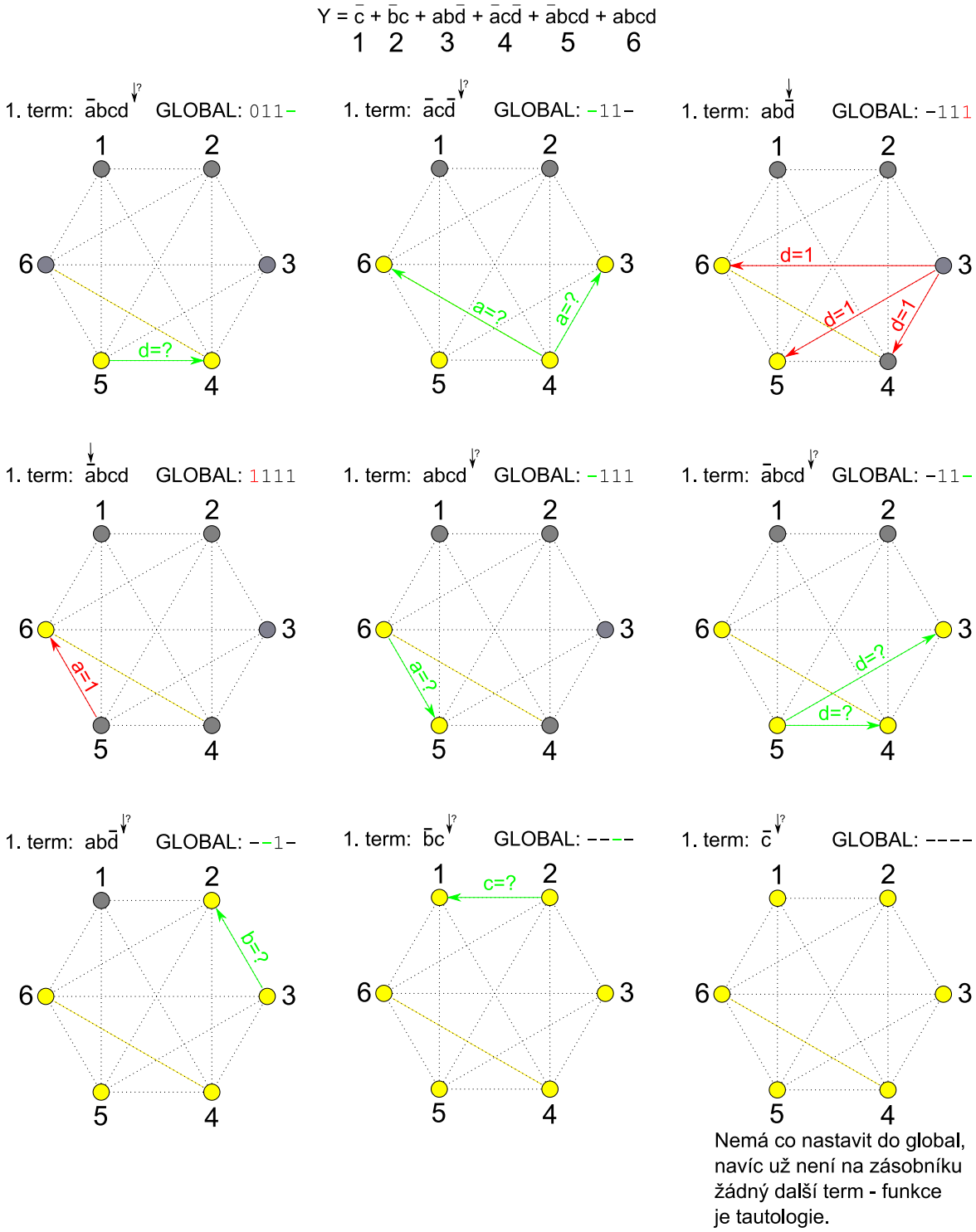
Nyní budeme postupovat dále dle algoritmu. Na obrázku 3.13 je znázorněna sekvence jednotlivých kroků algoritmu, jakými budeme postupovat. Jedná se o první část algoritmu (kroky 3-6), kdy se pouze ohodnocují proměnné.
Žlutě jsou zobrazeny termy, které jsou pro aktuální konfiguraci zatím stále splnitelné, vyřazené termy jsou šedé. Červenou barvou je vyznačena změna oproti předchozímu stavu v GLOBAL a distribuce nového rozhodnutí o ohodnocení proměnné ostatním termům.



Obrázek 3.13: Algoritmus řešící tautologii - 1. část

V tuto chvíli se algoritmus dostal do stavu, kdy už nemá způsob, jakým by zabránil splnění termu číslo 5. V globální stavové proměnné není žádná nastavitelná hodnota, kterou by se tento term vyřadil. V tomto případě je dokonce globální proměnná plně osazena, ale obecně tomu tak být nemusí.
Je tedy jasné, že pro toto ohodnocení bude funkce již vždy splněna, protože ji tento term číslo 5 už vždy splňuje a nelze tomu zabránit vhodným ohodnocením zbývajících neoohodnocených proměnných. A proto nyní algoritmus pokračuje další fází, kdy provádí backtracking, vrací se zpět a volí jiné cesty, jak zneplatnit jednotlivé termy. Postup backtrackingu i dalšího přiřa-

zování hodnot proměnným je na obrázku 3.14, kde je zobrazený celý další postup algoritmu až do úplného vyřešení této demonstrační úlohy.



Obrázek 3.14: Algoritmus řešící tautologii - 2. část s backtrackingem

Jak je vidět, po backtrackingu následují další pokusy o ohodnocení proměnných dle popisu algoritmu. Tímto způsobem algoritmus postupuje, dokud nenastane jeden ze dvou případů.

První, který skutečně v tomto případě nastal, je, že by chtěl provést backtracking z úplně prvního iniciačního termu, kdy na zásobníku nemá žádný předešlý stav. V takovém případě prošel již všechny možnosti a funkce je tautologie. Druhý způsob ukončení algoritmu nastává, pokud ohodnotil další proměnnou a tímto ohodnocením způsobil, že žádný term již není schopen toto ohodnocení splnit (na obrázku by všechny termy byly šedé). V takovém případě funkce tautologie není, protože nesplňuje ohodnocení, které aktuálně je v proměnné Global.
U vícevýstupových funkcí probíhá tentýž algoritmus sekvenčně pro každý výstup zvlášť.

3.2.3 Rychlost algoritmu v grafové struktuře

Tento algoritmus by pochopitelně šel použít i v původní reprezentaci za pomoci matice ve formátu PLA. Je tedy otázkou, zda a jak grafová struktura napomáhá rychlosti řešení. V rámci algoritmu se provádí a trvá u obou datových struktur stejně dlouho:

- řazení termů podle jejich množství don't care ve vstupech - je třeba projít všech n znaků u každého termu v obou případech
- tvorba příznaku termu, globální proměnné a zásobníku
- vyhledávání termu s příznakem „splnitelný“ - postupným procházením pole příznaků v obou případech
- vyhledávání proměnné, kterou je třeba nastavit - probíhá v rámci termu
- zaznamenávání nového stavu na zásobník a jeho návrat

Všechny tyto operace se provádí buď v rámci jednoho termu, nebo se samotných termů přímo netýkají a tudíž nemají se strukturou zajišťující propojení termů nic společného. Jediná operace, kde se může grafová struktura projevit, je vzájemná „komunikace“ mezi termy a k té dochází pouze v bodě 4 postupu algoritmu. Bod 4 zní „Proveď distribuci tohoto ohodnocení proměnné všem termům, které jsou přes tuto proměnou s tímto termem v klice. Pokud jim nastavení této polarity proměnné nevyhovuje, nastav jejich příznaky také na nesplnitelné.“
Tato distribuce informace se v tabulkové struktuře musí provádět za pomoci procházení sloupce s danou proměnnou. V grafové struktuře toto není třeba, neboť máme k dispozici spojový seznam takových termů, které mají v této proměnné opačnou polaritu.
Při každé distribuci tedy dojde k úspoře. Pokud budeme uvažovat termy, v jejichž vstupech budou zastoupeny možné hodnoty {0,1,-} stejným podílem, bude v klasické tabulkové prezentaci trvat každá distribuce trojnásobek času, neboť se tato informace bude šířit v sloupci do všech termů, zatímco pouze pro třetinu z nich je relevantní.
Celkový počet těchto distribucí je různý v závislosti na možnostech ořezávání stavového prostoru při běhu algoritmu. Pokud by algoritmus musel stavový prostor projít celý, musel by provést 2^n přiřazení, aby vyzkoušel všechny možnosti ohodnocení proměnných. V takovém případě by byl rozdíl mezi strukturami největší. V případě seřezávání stavového prostoru by se při snižování podílu projitého stavového prostoru snižoval rozdíl v časech ve stejném poměru, jde tedy o lineární závislost.

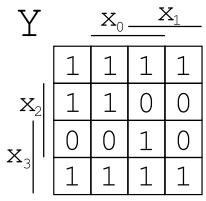
3.3 Výpočet komplementu funkce

V obecné matematice se pojmem *komplement množiny* (či *doplněk množiny*) označuje množina všech prvků, které nejsou v jiné množině obsaženy v rámci nějakého prostoru. Je-li dána nějaká libovolná množina H a její podmnožina G, označíme za doplněk množiny G (značíme $\bar{G}$) takovou množinu, jejíž prvky jsou obsaženy v H a zároveň nejsou obsaženy v G. Například doplňkem množiny {1,2,4,5,6,8,9,} v rámci celých kladných čísel menších než 10 je množina {3,7}.

3.3.1 Rozbor problému

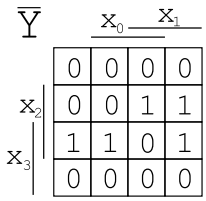
V logických výrazech hovoříme o doplňku či komplementu funkce. Je dána logická funkce s n vstupy, která nabývá pro určitá ohodnocení vstupních proměnných výstupní hodnoty 0. Naší úlohou je najít takovou funkci, která pro všechna tato ohodnocení nabývá výstupní hodnoty 1 a pro všechna ostatní nabývá hodnoty 0. Na obrázku 3.15 je zobrazen příklad takové funkce Y i jejího komplementu $\bar{Y}$.

x_3	x_2	x_1	x_0	Y	$\bar{Y}$
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	0




```
.i 4
.o 1
.type f
.p 3

-1-0 1
--0- 1
11-1 1
.e
```

$$Y = \bar{b}\bar{d} + \bar{c} + abd$$




```
.i 4
.o 1
.type f
.p 3

-110 1
-011 1
011- 1
.e
```

$$Y = b\bar{c}d + \bar{b}cd + abc$$

Obrázek 3.15: Komplement funkce

3.3.2 Popis algoritmu

Pro vyhledávání komplementu používáme modifikovaný algoritmus zjišťování tautologie. Náš algoritmus pro tautologii pracuje tak, že vyhledává první konfiguraci, pro kterou není výsledek výrazu roven jedné. Pokud ji nenajde, skončí a funkci prohlásí za tautologii. Najde-li ji, okamžitě se ukončí a jako výsledek vrací tuto konfiguraci proměnných.
Vyhledávání komplementu je obdobný problém, kdy však nás nezajímá pouze jedno takové ohodnocení, ale zajímají nás všechny. Tudíž algoritmus modifikujeme tak, aby si ve chvíli, kdy najde konfiguraci, pro niž je výsledek roven nule, tuto konfiguraci pouze poznamenal do seznamu offsetů a dál pokračoval v běhu. Tímto způsobem algoritmus poběží až do chvíle, kdy projde všechny možné konfigurace. Po jeho skončení pokrývá seznam offsetů svými termy všechny nuly v kaurnaghově mapě a tudíž se skutečně jedná o komplement funkce. Z tohoto seznamu tak můžeme sestrojit nový graf, použít minimalizační algoritmus a vygenerovat výsledné PLA,

které obsahuje komplement původní funkce.
Při vícevýstupové funkci probíhá vyhledávání komplementu sekvenčně pro každý výstup zvlášť a na jeho konci se používá hromadný vícevýstupový minimalizační algoritmus.

3.3.3 Rychlost algoritmu v grafové struktuře

Použijeme-li grafovou strukturu, dochází oproti tabulkové struktuře k téže úspoře jako u problému *tautologie*. To bylo již vysvětleno v 3.2.3. Pokud chceme získat komplement v jeho minimální formě, následuje poté ještě minimalizační algoritmus nad nově sestrojeným grafem komplementu a rozdíl mezi strukturami se tak projeví ještě jednou. Rozdíl v rychlostech algoritmu minimalizace je popsán v 3.4.3.

3.4 Minimalizace

Každá logická funkce může být vyjádřena nekonečně mnoha způsoby, které jsou funkčně ekvivalentní. Při návrhu a realizaci logických obvodů se vždy snažíme o to, aby toto vyjádření bylo co nejjednodušší. Vyhledávání tohoto co nejjednoduššího (minimalizovaného) tvaru zadané funkce se nazývá minimalizací této logické funkce.

3.4.1 Rozbor problému

Jak již bylo řečeno v 3.4, účelem minimalizace je zjednodušit logickou funkci na její minimalizovaný tvar. Je třeba jej tedy nějakým způsobem definovat.

3.4.1.1 Minimalizovaný tvar

Abychom nějaký tvar funkce mohli prohlásit za minimalizovaný, musí být splněna následující jednoznačně definovaná kritéria:

1. počet termů, kterými je funkce vyjádřena, je co nejmenší
2. počet proměnných v jednotlivých termech je rovněž co nejmenší, tedy jde o termy s co největší dimenzí - přímé implikanty funkce

Jinými slovy, minimalizovaný tvar funkce lze získat tak, že vytvoříme množinu všech přímých implikantů funkce a z této množiny vybereme takovou podmnožinu, aby funkce vytvořená pouze z těchto implikantů zachovávala stejnou hodnotu výstupů jako funkce původní a zároveň aby velikost této množiny byla minimální. Na tomto principu pracuje většina dnešních exaktních minimalizačních algoritmů.

3.4.1.2 Exaktní přístup a heuristiky

V 3.4.1.1 byl ukázán minimalizovaný tvar funkce, tedy výsledek, kterého chceme v ideálním případě dosáhnout. Metody, které vždy ve 100% případů vedou neomylně k tomuto výsledku, ať už je zadání jakékoliv, se nazývají exaktní. Avšak tyto exaktní metody jsou časově velice náročné. Jde o NP-těžký problém, který pro vyšší počet vstupů logické funkce není možné vyřešit v rozumném časovém úseku.
Z tohoto důvodu se pro vyřešení tohoto problému často používají heuristiky. Heuristika si neklade za cíl dosáhnout nejlepšího možného řešení, ale najít takové řešení, které je *dostatečně dobré*. V podstatě jde o kompromis mezi kvalitou řešení a časem věnovaným na vyhledání tohoto řešení.
Naše grafová struktura bude v této práci použita pro řešení za pomoci heuristiky.

3.4.2 Popis algoritmu

Náš algoritmus v prvním kroku odstraňuje z grafu všechny OFFSETy, protože pro ně nemá žádné další využití. Následně nasadí prvotní optimalizaci, kdy využívá tří zákonů Booleovy algebry, za pomoci kterých zjednodušuje termy. Jsou to:

- absorbce $a + ab = a$
- absorbce negace $a + \bar{a}b = a + b$
- idempotence $a + a = a$

Tyto tři zákony uplatní v celém rozsahu PLA tak, že vzájemně srovnává termy každý s každým. Začíná opět termy s vyšší dimenzí, které mají větší pravděpodobnost, že pohltí nějaký jiný term za pomoci absorbce či absorbce negace. Za pomoci těchto tří zákonů dojde ke zjednodušení termů a některé z nich mohou samozřejmě i zcela zaniknout, což opět zjednodušuje jakékoliv další výpočty nad grafem.

V tuto chvíli rozvětvíme algoritmus na dvě části, budou tedy ve výsledku algoritmy dva. Jediný rozdíl mezi nimi bude spočívat v tom, že rychlejší algoritmus tuto fázi přeskočí.

Pomalejší algoritmus v tuto chvíli začne vytvářet nové termy z takových dvojic termů, které jsou ve vzdálenosti 1. Vzdálenost v tomto případě odpovídá počtu neshod při porovnání řetězců vstupních proměnných, kdy za neshodu považujeme pouze má-li jeden term na dané pozici logickou jedna a druhý logickou nulu. Například termy s těmito vstupy `111-` a `-011` mají jen jednu neshodu v proměnné uprostřed a společně tak vytvoří nový term `11-11`. Pokud by jich měly více, jako například dvojice termů `111-` a `00--`, nevytvoří term žádný.
Po tomto vytvoření nových termů za pomoci konsenzu, necháme ještě jednou proběhnout předešlou část algoritmu s absorbcemi a idempotencí pro případ, že nově vytvořené termy by vedly na další zjednodušení. Poté, co proběhne i tato část, máme jednoznačně určeny všechny existující přímé implikanty naší funkce.
Konsenzus zároveň slouží i pro určení nesporných termů. Ve chvíli, kdy za pomoci termu y vytváříme další nové termy $x_0..x_i$, použijeme tuto množinu nových termů a sjednotíme ji se všemi již existujícími termy, které mají s y společný minterm. V případě, že tato množina sjednocených termů pokrývá celý y, není y nesporný, v opačném případě je.

Následující text už opět platí pro obě varianty algoritmu:
Při všech operacích až do této chvíle popsaných využíváme ONSETové i DCSETové termy. Pokud dojde během běhu algoritmu k tomu, že DCSETový term napomůže za pomoci absorbce či absorbce negace nějakému ONSETovému termu, stává se z tohoto DCSETového termu automaticky ONSETový. Tím dojde k tomu, že všechny použité DCSETové termy se stanou ONSETové a o zbývajících víme, že pro ně nemáme při minimalizaci využití a tudíž je v tuto chvíli odstraníme.

Další postup algoritmu:

1. vytvoř seznam představující množinu termů použitých ve finálním řešení (v případě algoritmu tvořícího všechny přímé implikanty do něj v tuto chvíli umístíme všechny nesporné implikanty, pokud jde o rychlejší algoritmus, tak seznam zůstane prázdný)
2. ověř, zda termy z tohoto seznamu již nepokrývají některý ze zbývajících termů, pokud ano, ihned jej odstraň - term si můžeme dovolit odstranit, neboť je beze zbytku pokryt disjunktivní množinou termů ve finálním řešení, je tedy z našeho hlediska již redundantní

3. v rámci seznamu termů, které nejsou zařazeny do finálního řešení, hledej další nesporné implikanty (bez využití implikantů ze seznamu termů ve finálním řešení)

   - najdeš-li další nesporné implikanty, přesuň je do seznamu finálních termů a pokračuj bodem 3
   - pokud žádný nenajdeš, znamená to, že se zbývající termy vzájemně všechny překrývají, vezmi libovolný term z termů s nejnižší dimenzí a definitivně jej odstraň, pokračuj bodem 3 a znovu hledej další nesporné implikanty - tento term smíme odstranit, neboť z hlediska termů, které ještě nejsou zařazeny do finálního řešení, je redundantní a je pokryt jejich disjunktivní množinou
   - je-li seznam termů nezařazených do finální množiny prázdný, skonči algoritmus

Ve chvíli, kdy algoritmus skončí, zbývají v rámci našeho PLA pouze takové termy, které jsme přesunuli do seznamu finálních termů, všechny ostatní jsou smazány. Ani jedna varianta není exaktní, neboť i v případě, že v první části určujeme všechny přímé implikanty, ve druhé části, kdy exaktní algoritmy vybírají z přímých implikantů co nejmenší množinu, která pokrývá celou funkci, náš algoritmus pouze na základě heuristiky odstraňuje termy a nemusí to být nutně správná cesta k exaktnímu řešení.

3.4.3 Rychlost algoritmu v grafové struktuře

Při algoritmu minimalizace se neustále hledají sousední či vzájemně se překrývající termy. Ve srovnání s rychlostí u tabulkové struktury dochází opět až k trojnásobnému zrychlení (podobně jako v 3.2.3) u každé proměnné, kterou daný term prohledává své okolí. K tomuto prohledávání dochází ve všech iteracích, kdy je porovnáván každý term s každým.
V rámci jedné iterace je to tedy $(t-1)*(t-2)*...*2*1$ prohledávání, což je aritmetická řada, jejíž součet se dá vyjádřit následujícím vzorcem: $s = n * \frac{(t-1)+1}{2}$
Počet iterací je proměnlivý v závislosti na mezivýsledcích. Avšak bylo otestováno, že pro většinu případů se pohybuje mezi 1-5. K jedné iteraci dojde vždy, což znamená, že větší rychlost grafové struktury se zcela jistě nejméně jednou projeví.

3.5 Ortogonalizace

Pod pojmem ortogonalizace logické funkce rozumíme její rozklad na součet vzájemně se nepřekrývajících termů. Dva termy nazveme vzájemně se nepřekrývající právě tehdy, nemají-li společný žádný minterm.

3.5.1 Rozbor problému

Primitivním řešením ortogonalizace funkce je rozklad všech jejích termů na mintermy a odstranění duplicitních mintermů. Pro praktické účely se ovšem opět (podobně jako u minimalizace) snažíme o co nejjednodušší výraz, který by definoval naši funkci a zároveň využíval pouze ortogonálních termů.
Náš algoritmus vychází z jednoduché myšlenky, že pokud se termy nemají vzájemně překrývat, dáme v konfliktních situacích přednost termům s větší dimenzí, které pokrývají větší množství stavů a mají tak větší pravděpodobnost, že v konečném důsledku zapříčiní menší počet všech termů ve výrazu.

3.5.2 Popis algoritmu

Velká část algoritmu probíhá stejným způsobem jako minimalizační algoritmus popsaný v 3.4.2, tedy tak, že uplatňuje tyto tři základní zákony:

- absorbce $a + ab = a$
- absorbce negace $a + \bar{a}b = a + b$
- idempotence $a + a = a$

Děláme to ze stejných důvodů jako v předešlém případě, tedy abychom měli termy co nejjednodušší a práce s nimi byla rychlejší. Dále se opět algoritmus dělí na dva rozdílné algoritmy, první z nich generuje pouze přímé implikanty a u toho zjišťuje nesporné způsobem, jaký byl popsán v 3.4.2. Druhý tuto část přeskakuje a značně tak zkracuje celkovou dobu úlohy na úkor kvality řešení.
Stejně jako v minimalizaci i zde se snažíme využít termy z DCSETu a podaří-li se nám to, předěláváme jejich výstup tak, že se z nich stávají termy ONSETové. Ostatní DCSETové termy zahazujeme, OFFSETové termy rovněž.

Další postup algoritmu se již od minimalizace liší:

1. vytvoř prázdný seznam termů, které budou použity ve finálním řešení ortogonalizace
2. seřaď termy do fronty sestupně podle velikosti, termy s největší dimenzí umístíme na začátek fronty (pokud používáme pomalejší algoritmus a máme definovány nesporné termy, umísti tyto nesporné termy před všechny ostatní, rovněž v sestupném pořadí)
3. vezmi první term z fronty
4. zjisti, zda se překrývá s některými z termů již zařazených do finální množiny termů
   - nepřekrývá - celý tento term zařaď do výsledného seznamu
   - celý je pokryt disjunktní množinou termů ze seznamu seznamu finálních termů - zahoď celý term, nebude použit (je z našeho pohledu již redundantní)
   - je částečně pokryt termy v seznamu - v rámci dimenze tohoto termu zjisti komplement množiny sjednocení všech ostatních termů v seznamu, které jej překrývají, tento komplement pak zařaď do výsledného seznamu, je vůči ostatním členům seznamu ortogonální
5. pokud fronta není prázdná, pokračuj bodem 3

Po skončení tohoto algoritmu získáváme ekvivalentní PLA, které je však plně ortogonalizované. Protože je možné, že některé termy jsou zbytečně malé, necháme nad touto strukturou proběhnout ještě jeden velmi jednoduchý minimalizační algoritmus, který bude spojovat dva sousední termy ve vzdálenosti jedna v jeden term, bude-li to možné. Tedy například z termů $a\bar{b}c + abc$ vytvoří jediný term ac. Tuto operaci je třeba provádět i opakovaně, je-li to možné.

3.5.3 Rychlost algoritmu v grafové struktuře

Výhoda grafové struktury oproti tabulkové opět spočívá v rychlosti vyhledávání sousedních termů. Toto vyhledávání je použito v počátku při uplatňování zákonů, které vedou na zjednodušení termů a následně při vyhledávání konfliktních termů, které je nutno ortogonálně rozložit.
Je velmi obtížné určit skutečné zrychlení, neboť to velmi záleží na vstupních datech. Pokud budou zadané termy už v ortogonální podobě, bude rozdíl v rychlostech nejmenší a i přesto

nebude zanedbatelný, neboť jen ono počáteční uplatňování zákonů pro zjednodušení znamená porovnávat každý term s každým, tudíž se jedná opět o $s = n * \frac{(t-1)+1}{2}$ porovnávání a to i v případě rychlejšího algoritmu.

4 Implementace

V této kapitole je rozebrán způsob implementace a některé její detaily. Lze zde také nalézt úryvky ze zdrojového kódu programu. Tyto části zdrojového kódu nemusí být vždy přesné, v rámci lepší přehlednosti mohou být zkráceny či jinak upraveny. Kompletní plnohodnotná verze je umístěna na CD přiloženém k této publikaci.

4.1 Programovací jazyk

Protože výsledkem implementační části této práce měl být konzolový program, jehož vstupem i výstupem bude nějaký soubor ve formátu PLA a jehož stěžejní vlastností má být především rychlost, nebylo třeba volit jazyky vyšší úrovně (jako například JAVA) a byl zvolen jazyk C++, který je zcela univerzální a stále se považuje za jeden z nejrozšířenějších programovacích jazyků. Druhým důvodem pro volbu tohoto programovacího jazyku byla možnost některé jeho části později naimplementovat do minimalizátoru Boom [3], který je na katedře počítačů vyvíjen.

4.2 Obecný popis implementace

Program byl od počátku vytvářen s myšlenkou na jeho budoucí rozšiřitelnost a snadnou modifikaci. Tudíž je plně objektový a výsledkem je jedna třída *Graph*, která obsahuje veřejné funkce, jimiž je možno provádět všechny operace. Kromě základních prvků jazyka C++ je v implementaci použita standardní knihovna šablon STL.
Pro zajištění kompatibility bylo rovněž třeba použít některé funkce použité v programu Boom, nicméně tyto funkce jsou použity pouze pro import a export datových struktur, aby nebylo nutné generovat a opětovně načítat celý PLA. Třída *Graph* je plně použitelná i bez jádra Boom a nic dalšího než knihovnu STL ke své činnosti nepotřebuje.
Na CD u této práce jsou umístěny rovněž i spustitelné soubory, které demonstrují funkčnost celé aplikace.

4.3 Popis datových struktur

Jak již bylo řečeno, hlavní třídou celé implementace je třída *Graph*. Kromě této třídy se v práci nachází také pomocné třídy *GTerm* a *GVariable*, ve kterých je fyzicky uchován celý graf. V následujícím textu budou tyto jednotlivé třídy popsány z hlediska datových struktur a budou zde vysvětleny také hlavní funkce těchto tříd.

4.3.1 Třída GTerm

Třída *GTerm* představuje základní třídu celého programu. Je to základní stavební kámen grafu, neboť každý tento vytvořený objekt odpovídá jednomu uzlu v naší grafové struktuře. V této struktuře je v podstatě uchován celý zápis PLA a k vygenerování PLA z aktuálního grafu nám stačí pouze tyto objekty. Obě zbývající třídy nenesou fakticky žádná data a slouží pouze pro virtualizaci celého problému. Z toho plyne, že je dokonce možné z těchto *GTermů* vytvořit celý graf. Na obrázku 4.1 je zobrazen zdrojový kód této třídy.

Každý *GTerm* obsahuje dva řetězce se svými vstupy a výstupy v podobě, jakou známe z formátu PLA. Dále zde jsou uloženy tři seznamy ukazatelů na proměnné. Členství proměnné v seznamu určuje, zda je nastavena v přímé formě, negativní formě či jde o dc. Počty *count* udávají délku těchto seznamů.
Všechny tyto privátní proměnné jsou vyplněny v konstruktoru, který dostává na vstup celý

```
class GTerm
{
private:
 string inputs;
 string outputs;
 int count_on, count_off, count_dc;
 list<GVariable*> vars_on;
 list<GVariable*> vars_off;
 list<GVariable*> vars_dc;
public:
 GTerm::GTerm(string &_inputs, string &_outputs,
   vector<GVariable*> &variables);
 friend class Graph;
 string GetIn() const;
 char GetIn(const int numinput) const;
 string GetOut() const;
 char GetOut(const int numoutput) const;
 void SetOut(const int numoutput, char set);
 int GetCount_dc() const;
 list<GVariable*> GetVars(const char set) const;
 bool Satisfy(const string &_variables) const;
 bool operator>(const GTerm &druhy);
 bool operator<(const GTerm &druhy);
 bool operator>=(const GTerm &druhy);
 bool operator<=(const GTerm &druhy);
 friend ostream& operator<< (ostream &os, const GTerm &t);
};
```

Obrázek 4.1: Třída GTerm

řádek *termu* z PLA, který postupně parsuje a tak vyplňuje všechny seznamy dle vztahů, které *term* k daným proměnným má.
Veřejné funkce slouží převážně pro manipulaci se vstupními a výstupními řetězci. Obsah většiny z nich se dá snadno odhadnout z názvu funkce. Za zmínku stojí funkce *Satisfy*, která dostává jako vstupní hodnotu řetězec s ohodnocením všech proměnných a určuje, zda tento *term* má v sobě takto určený minterm obsažen. Funkce je použita u algoritmu tautologie hrubou silou, který byl rovněž naprogramován pro srovnání rychlosti s naší tautologií nad grafem.
Ve funkci jsou rovněž přetíženy operátory $>, \geq, <, \leq$. *Term* t_1 je považován za větší než *term* t_2 právě tehdy, pokud dimenze t_1 je vyšší než dimenze t_2. Jinými slovy *velikost termu* je dána počtem mintermů, které pokrývá.

4.3.2 Třída GVariable

Tato třída (viz obrázek 4.2) zastupuje jednotlivé proměnné, které se nacházejí v logické funkci. Narozdíl od třídy *GTerm* je počet těchto objektů zcela fixní po celou dobu programu, neboť

počet vstupních proměnných se v rámci našeho PLA nemění.

```
class GVariable
{
private:
 const int number;
 struct Outputs
 {
  struct ONset
  {
   list<GTerm*> term_on;
   list<GTerm*> term_off;
   list<GTerm*> term_dc;
  } onset;
  struct OFFset
  {
   list<GTerm*> term_on;
   list<GTerm*> term_off;
   list<GTerm*> term_dc;
  } offset;
  struct DCset
  {
   list<GTerm*> term_on;
   list<GTerm*> term_off;
   list<GTerm*> term_dc;
  } dcset;
  int count_on, count_off, count_dc;
 }* outputs;
public:
 GVariable::GVariable(const int numoutputs, const int counter);
 GVariable::~GVariable();
 friend class Graph;
 int GetNum() const;
 void AddTermFromPLALine(GTerm *term, const char &relation);
 void RemoveTerm(GTerm *term, const char &relation);
 void RemoveTermOutput(GTerm *term, const char &relation,
   const int output, const char old_output);
 void AddTermOutput(GTerm *term, const char &relation,
   const int num_output, const char new_output);
 list<GTerm*> GetTerms(const int output, const char set,
   const char list) const;
};
```

Obrázek 4.2: Třída GVariable

Jak již bylo podrobně popsáno v 3.1.4, obsahuje každá z těchto tříd 9 seznamů pro každou výstupní proměnnou, jak je zachyceno na obrázku 3.10. V této kapitole bylo i popsáno, k čemu tyto seznamy slouží, nebudeme se zde tedy již rozborem struktury *Outputs* zabývat. Kromě

toho obsahuje každá proměnná své pořadové číslo pro snazší orientaci při běhu programu. Veřejné funkce jsou:

- Konstruktor - pouze zakládá proměnnou jako objekt, žádné termy nevkládá
- Destruktor - je zapotřebí, neboť *Outputs* je dynamické pole, které je třeba při zániku objektu *GVariable* uvolnit
- GetNum - vrací pořadové číslo proměnné
- AddTermFromPLALine - tato funkce se spouští pokaždé, když se konstruuje nový objekt *GTerm*, zajišťuje zařazení *GTermu* do správných seznamů a vytváří tak informaci o hranách v grafu
- RemoveTerm - funkce odstraňuje *GTerm* ze všech struktur, používá se při definitivním odstraňování *GTermu*
- RemoveTermOutput - funkce odstraňuje vazby na tento *GTerm*, ale pouze v rámci jednoho výstupu, nejčastější použití má při minimalizaci vícevýstupových funkcí
- AddTermOutput - přidává *GTerm* do dalších seznamů z hlediska nějakého výstupu, kde předtím neměl *GTerm* žádný význam
- GetTerms - vrací jeden ze spojových seznamů

Celá třída především udržuje informaci o hranách (respektive klikách) v grafu. Situace je mírně nepřehledná právě díky rozhodnutí mít těchto množin hran více rozdělených dle pořadí výstupu, typu výstupu a vztahu proměnné k termu jako takovému.

4.3.3 Třída Graph

Poslední a zároveň nejdůležitější třídou je třída *Graph*. Předchozí dvě třídy slouží pouze pro vnitřní reprezentaci grafu, ale tuto konzistentní strukturu je ještě třeba nějakým způsobem zastřešit a zpřístupnit běžnému uživateli. Právě tento problém třída *Graph* řeší.

Začneme popisem privátních proměnných:

- terms - seznam ukazatelů na všechny aktuálně existující *GTermy* v grafu
- outputs - další seznamy ukazatelů na *GTermy*, tentokrát rozdělené podle čísla výstupu a typu výstupu, co *GTerm* dává
- countery - počty vstupů, výstupů a *GTermů*
- graphtype - určuje typ grafu, typ je definován takto:
  typedef enum {undefined,f,fd,fr,fdr} graphtype;
- names - formát PLA umožňuje jednotlivé vstupní a výstupní proměnné pojmenovávat, pokud ve vstupním PLA byly nějak pojmenovány, zůstávají pojmenovány i ve výstupním

Nyní k veřejným funkcím, které zpřístupňují celý graf k běžnému použití:

- konstruktor - konstruktor pouze nastavuje inicializační hodnoty
- destruktor - fyzicky odstraňuje všechny *GTerm*, *GVariable* a své dynamické pole seznamů

```
class Graph
{
 list<GTerm*> terms;
 struct Outputs
 {
  list<GTerm*> terms_on;   //onset
  list<GTerm*> terms_off;  //offset
  list<GTerm*> terms_dc;   //dcset
  list<GTerm*> terms_nm; //no meaning
 }* outputs;

 vector<GVariable*> variables;
 int count_terms, int count_inputs, int count_outputs;
 graphtype type;
 string input_names, output_names;

public:
 Graph::Graph();
 Graph::~Graph();
 void ConstructFromPLALines(list<string> &buffer);
 void ConstructFromPLAFile(char *filename);
 void ConstructFromBoomPLA(PLA *pla);
 string GenerateStreamForPLA();

 void Simplification();
 string BruteForceTautology() const;
 string BranchAndBoundTautology() const;
 Graph* Complement();
 void Minimization(const bool deep);
 void Orthogonalization(const bool deep);
};
```

Obrázek 4.3: Třída Graph

- ConstructFromPLALines - vytváří graf na základě PLA, které je této funkci předloženo jako seznam řetězců - každý prvek v seznamu odpovídá jednomu řádku v PLA, v případě nesprávného vstupního formátu vyhodí výjimku
- ConstructFromPLAFile - má stejné chování jako předešlý construct, ale PLA načítá ze souboru
- ConstructFromBoomPLA - konstruuje graf na základě třídy PLA, která je součástí programu Boom, při nezdaru opět vyhazuje výjimku
- GenerateStreamForPLA - generuje řetězec s PLA, který je možné například uložit do souboru

- Simplification - zjednodušuje vyjádření grafu tak, že na *termy* aplikuje zákony absorbce, absorbce negace a idempotence, funkce zachovává ONSETy, DCSETy i OFFSETy, tudíž nově vygenerovaný PLA bude funkčně ekvivalentní původnímu před spuštěním funkce
- BruteForceTautology - řeší problém tautologie hrubou silou, umístěno zde pouze pro srovnání časů s naší tautologií, vrací buď řetězec „tautology“, pokud se jedná o tautologii, nebo řetězec s ohodnocením proměnných, pro které výstup není jedna, v případě, že se o tautologii nejedná
- BranchAndBoundTautology - náš algoritmus tautologie popsaný v 3.2, výstupní hodnoty jsou stejné jako u tautologie hrubou silou
- Complement - algoritmus pro tvorbu komplementu, který byl popsán v 3.3, návratová hodnota je ukazatel na zcela nový objekt graf, ve kterém je komplement uložen
- Minimization - minimalizuje funkci v grafu dle našeho algoritmu z 3.4, konstanta deep udává, zda má předtím generovat všechny přímé implikanty, jak je popsáno v 3.4.2
- Orthogonalization - provádí ortogonalizaci dle 3.5, konstanta deep opět udává, zda generovat všechny přímé implikanty, případně tuto část přeskočit

Pozn.: Třída *Graph* obsahuje velké množství dalších pomocných privátních funkcí, které jsou využívány při manipulaci s celou strukturou.
Minimalizace a ortogonalizace byly vytvořeny s předpokladem, že se ve vývoji těchto dvou funkcí bude později ještě pokračovat a rozdíly mezi hlubokou a mělkou verzí algoritmů se budou více prohlubovat.

5 Testování

Náplní této kapitoly je testování všech našich grafových algoritmů a jejich srovnání s jinými programy, které řeší stejnou problematiku. Jsou zde zobrazeny tabulky naměřených hodnot, přehledné grafy s výsledky a rovněž jsou v rámci této kapitoly učiněny některé závěry, které lze z pozorovaných výsledků vyvozovat.

5.1 Generátor PLA

Pro testovací účely bude zapotřebí použít nějaký parametrizovatelný generátor, aby bylo možné provádět testy a sledovat efektivitu v závislosti na typu předložených instancí. Jako vyhovující byl určen generátor, který vznikl jako bakalářská práce kolegy Měchury [6].
Generátor vytváří booleovské funkce ve formátu PLA. Má mnoho parametrů, kterými lze ovlivňovat podobu výsledného PLA. Pro nás nejdůležitější jsou počty vstupů, výstupů, termů a procentuální specifikace množství don't care ve vstupech.
Pro všechny další testy bude vždy použit tento generátor s jednoznačně určenými parametry. Všechny použité testovací instance jsou uloženy na CD přiloženém k této práci.

5.2 Tautologie

Pro řešení tautologie máme naimplementovány vlastní dva algoritmy. Stěžejní je ten, který pracuje tak, jak bylo popsáno v 3.2.2, tedy ořezáváním stavového prostoru na základě jednoznačnosti výsledku v některé z větví prostoru. Tento algoritmus je ten, který plně využívá grafovou strukturu a ten budeme srovnávat s ostatními algoritmy. Těmi jsou:

- Algoritmus využívající hrubou sílu - tedy takový, který prochází všech 2^n variací ohodnocení a hledá vždy první term, který toto ohodnocení splňuje.
- Algoritmus, který detekuje tautologii za pomoci BDD - tento algoritmus vytvořil v rámci své bakalářské práce kolega Navrkal a lze se o něm dočíst v [7].
- Náš algoritmus tautologie s předpřípravou PLA - tato předpříprava spočívá ve spuštění funkce Simplification před samotnou detekcí tautologie. Tato funkce, jak již bylo řečeno v 4.3.3, efektivně zjednodušuje PLA na základě zákonu absorbce a idempotence.

5.2.1 Instance

Aby se dala nějakým způsobem porovnat efektivita algoritmů, je třeba vybírat pouze takové instance, které budou tautologie. Jinak by se řešení nacházelo někde uprostřed stavového prostoru a výsledky by nebyly zcela relevantní, neboť by byly závislé na způsobu, jakým jednotlivé algoritmy stavový prostor procházejí. Pokud vstupem bude vždy tautologie, musí všechny algoritmy projít stavový prostor beze zbytku, respektive musí mít jistotu, že se ve zbývající neprojité části nenachází řešení.
Dále je třeba určit základní 4 parametry generátoru.

- počet vstupů - počet vstupů budeme postupně navyšovat a zkoumat vliv tohoto parametru na výsledné časy algoritmů, počáteční a koncové hodnoty určíme experimentálně podle referenčního algoritmu, jímž bude náš grafový algoritmus pro tautologii
- počet výstupů - pro vícevýstupové funkce se musí proces tautologie opakovat pro každý vstup a výpočet běží sekvenčně, není tedy důvod nastavovat vyšší hodnoty, počet výstupů bude roven 1

- počet termů - toto číslo bude opět určeno experimentálně a to tak, že budeme číslo stále navyšovat až do chvíle, kdy pro daný počet vstupů bude dostatečně vysoká pravděpodobnost, že náhodně vygenerované termy budou tvořit tautologii
- procentuální množství don't care na vstupu - toto je nejdůležitější ze všech parametrů, domníváme se, že efektivita algoritmu má velkou závislost na tomto parametru, z počátku nastavíme tento parametr na 0% a budeme jej postupně navyšovat po 10% až do výše 90%

S takto definovanými hodnotami generátoru vygenerujeme vždy pět instancí pro každé jeho nastavení, abychom předešli zaznamenávání extrémních případů. Z těchto pěti měření odstraníme první a poslední z hlediska rychlosti, zbylé tři budeme průměrovat.

5.2.2 Naměřené výsledky

V tabulkách 5.1 až 5.10 jsou zobrazeny již zprůměrované hodnoty, jak bylo popsáno v 5.2.1. Ve sloupcích jsou časy všech čtyř algoritmů pro dané nastavení generátoru. V záhlaví je nastavení parametrů generátoru ve tvaru [počet vstupů]x[počet termů].
V jednotlivých řádcích jsou řešící algoritmy. BF je tautologie řešená hrubou silou. Graf je náš algoritmus, Graf* je tentýž algoritmus, jemuž ale předtím předchází úprava vstupního PLA (čas se počítá pochopitelně jako součet obou operací) a BDD je řešení za pomoci binárních rozhodovacích algoritmů od pana Navrkala.
V případě, že celkový čas úlohy přesáhl 3 minuty a zároveň tuto úlohu zvládl jiný algoritmus řádově rychleji, bylo měření tohoto algoritmu po těchto 3 minutách ukončeno.

DC = 0%	**06x64**	**07x128**	**08x256**	**09x512**	**10x1024**
BF	<10 ms	<10 ms	16 ms	47 ms	0,2 s
Graf	94 ms	688 ms	7,6 s	125 s	>180 s
Graf*	31 ms	94 ms	0,33 s	1,13 s	7,3 s
BDD	15 ms	47 ms	141 ms	391 ms	766 ms

Tabulka 5.1: Srovnání rychlostí algoritmů řešících tautologii (0% DC)

DC = 10%	**06x150**	**07x250**	**08x450**	**09x1100**
BF	<10 ms	<10 ms	16 ms	98 ms
Graf	57 ms	0,34 s	2,82 s	33 s
Graf*	32 ms	0,13 s	0,37 s	3,2 s
BDD	22 ms	47 ms	163 ms	0,34 s

Tabulka 5.2: Srovnání rychlostí algoritmů řešících tautologii (10% DC)

DC = 20%	**07x150**	**08x250**	**09x500**	**10x1200**
BF	<10 ms	16 ms	24 ms	147 ms
Graf	57 ms	44 ms	3,94 s	48 s
Graf*	39 ms	170 ms	0,67 s	5 s
BDD	34 ms	84 ms	0,23 s	0,37 s

Tabulka 5.3: Srovnání rychlostí algoritmů řešících tautologii (20% DC)

DC = 30%	**08x200**	**09x300**	**10x450**	**11x750**
BF	16 ms	20 ms	24 ms	97 ms
Graf	135 ms	627 ms	1,97 s	23,3 s
Graf*	63 ms	192 ms	0,89 s	8,6 s
BDD	72 ms	120 ms	0,25 s	0,37 s

Tabulka 5.4: Srovnání rychlostí algoritmů řešících tautologii (30% DC)

DC = 40%	**10x400**	**11x600**	**12x800**	**13x1000**	**14x1500**
BF	33 ms	87 ms	85 ms	0,22 s	0,4 s
Graf	190 ms	894 ms	2,7 s	97 s	542 s
Graf*	165 ms	368 ms	1,4 s	31 s	147 s
BDD	172 ms	0,25 s	0,4 s	0,78 s	1,85 s

Tabulka 5.5: Srovnání rychlostí algoritmů řešících tautologii (40% DC)

DC = 50%	**12x200**	**13x300**	**14x500**	**15x700**	**16x1200**
BF	35 ms	77 ms	0,18 s	0,3 s	0,72 s
Graf	34 ms	207 ms	2,59 s	3,2 s	79 s
Graf*	33 ms	110 ms	0,82 s	1,3 s	52 s
BDD	170 ms	302 ms	0,45 s	1,1 s	2,4 s

Tabulka 5.6: Srovnání rychlostí algoritmů řešících tautologii (50% DC)

DC = 60%	**15x500**	**16x600**	**17x700**	**18x900**	**19x1100**
BF	0,23 s	0,34 s	0,68 s	1,3 s	3,3 s
Graf	0,32 s	0,52 s	4,1 s	9,1 s	61 s
Graf*	0,27 s	0,34 s	0,81 s	4,6 s	27 s
BDD	0,42 s	0,62 s	1,3 s	3,3 s	6,5 s

Tabulka 5.7: Srovnání rychlostí algoritmů řešících tautologii (60% DC)

DC = 70%	**22x400**	**24x600**	**26x700**	**28x900**	**29x1200**
BF	15 s	78 s	>180 s	>180 s	>180 s
Graf	722 ms	50 ms	62 ms	93 ms	128 ms
Graf*	240 ms	285 ms	372 ms	545 ms	1,03 s
BDD	7,2 s	26,5 s	130 s	>180 s	>180 s

Tabulka 5.8: Srovnání rychlostí algoritmů řešících tautologii (70% DC)

DC = 80%	**30x250**	**35x400**	**40x700**	**45x1200**	**50x1800**
BF	-	-	-	-	-
Graf	25 ms	33 ms	52 ms	0,18 s	0,45 s
Graf*	75 ms	196 ms	534 ms	1,65 s	6 s
BDD	94 s	154 s	>180 s	>180 s	>180 s

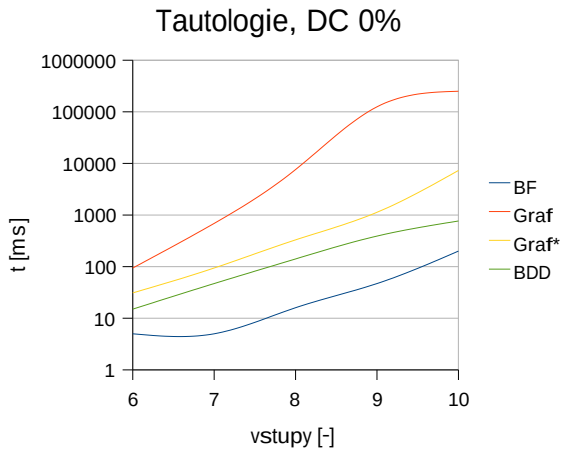
Tabulka 5.9: Srovnání rychlostí algoritmů řešících tautologii (80% DC)

DC = 90%	**70x300**	**80x400**	**90x600**	**100x1500**	**110x2500**
BF	-	-	-	-	-
Graf	31 ms	75 ms	0,14 s	0,44 s	0,87 s
Graf*	240 ms	0,42 s	0,81 s	5,87 s	22 s
BDD	-	-	-	-	-

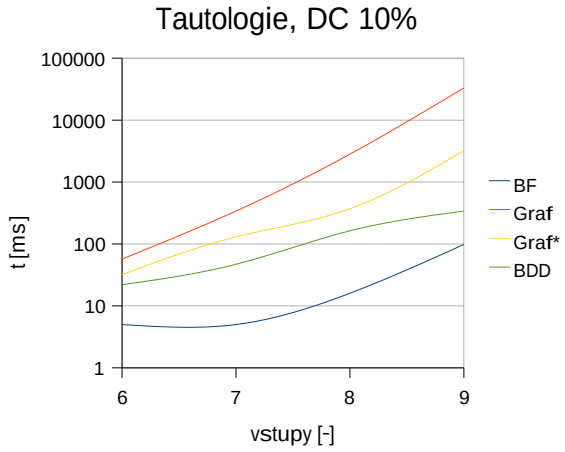
Tabulka 5.10: Srovnání rychlostí algoritmů řešících tautologii (90% DC)

5.2.3 Grafy a zhodnocení

Zde jsou vyobrazeny grafy z předešlých měření. Vzhledem k tomu, že se jedná o exponenciálně složitou úlohu, utváří grafy exponenciály a díky tomu je možné z nich zobrazit pouze pár bodů, neboť při příliš malém n je doba úlohy velmi krátká a neměřitelná, při příliš vysokém naopak je doba trvání úlohy příliš dlouhá. Ve většině grafů bylo zvoleno pět bodů, ostatní části grafu jsou vytvořeny interpolací. Stupnice na ose Y je logaritmická kvůli rozdílu několika řádů v časech.

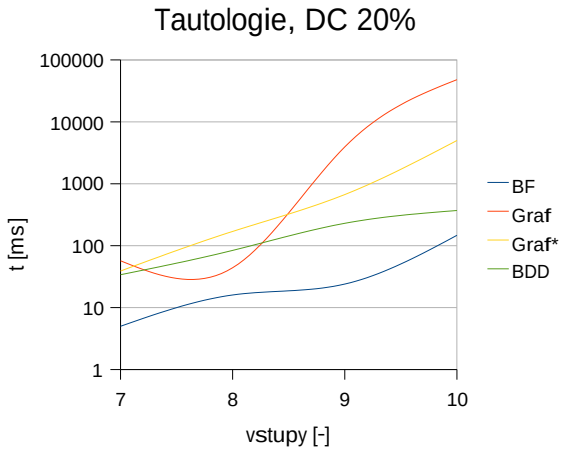


Obrázek 5.1: Graf s časy algoritmů při detekci tautologie (DC 0%)

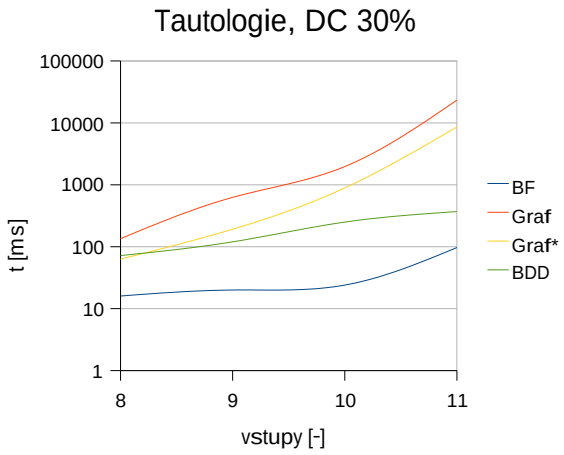


Obrázek 5.2: Graf s časy algoritmů při detekci tautologie (DC 10%)

Na počátečních grafech 5.1 až 5.5 můžeme vidět jednoznačné prvenství algoritmu, který řeší problém hrubou silou. Je tomu tak proto, že vstupní n jsou velmi malá, čili je možné projít všechny variace ohodnocení poměrně rychle a dále mají vstupy příliš malé obsazení don't care, tudíž výhody inteligentního procházení stavového prostoru u ostatních algoritmů nejsou zatím tolik zřetelné.
Jako druhý algoritmus se umisťuje BDD, obě varianty našeho grafu jsou poslední. Je tomu tak z důvodů popsaných výše, navíc je třeba brát v úvahu, že graf není dělaný pouze pro tautologii,

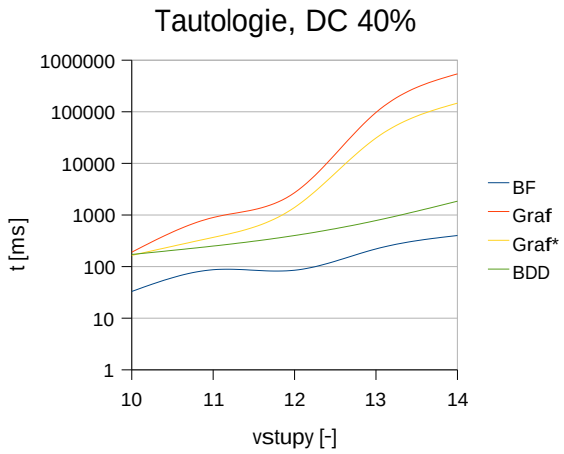


Obrázek 5.3: Graf s časy algoritmů při detekci tautologie (DC 20%)

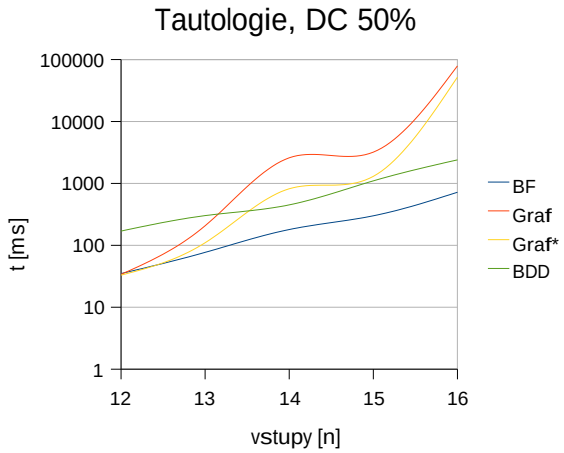


Obrázek 5.4: Graf s časy algoritmů při detekci tautologie (DC 30%)

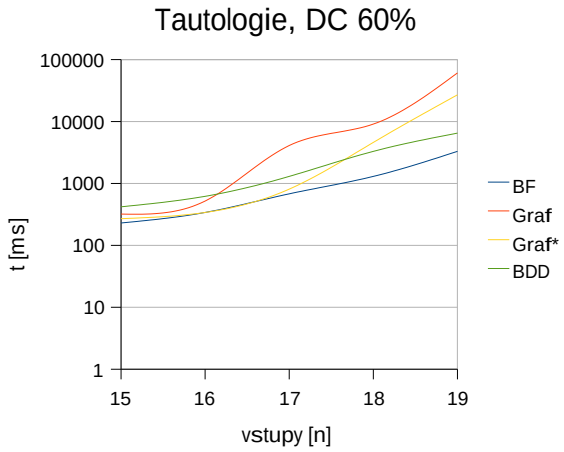
řeší i složitější úkoly a je tedy zbytečně robustní ve srovnání s ostatními algoritmy, které jsou na tautologii přímo specifikovány.
Zaměříme-li se na grafy 5.6 a 5.7, zjistíme, že časy začínají být srovnatelnější. Pořadí výkonnosti algoritmů se nemění, avšak již zde nejsou vidět propastné rozdíly jako u prvních pěti grafů.



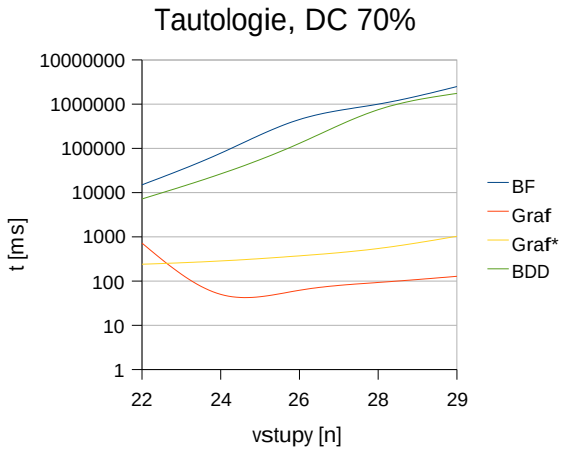
Obrázek 5.5: Graf s časy algoritmů při detekci tautologie (DC 40%)



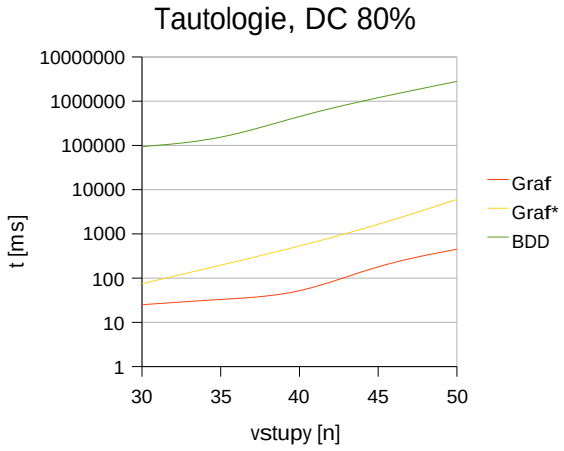
Obrázek 5.6: Graf s časy algoritmů při detekci tautologie (DC 50%)



Obrázek 5.7: Graf s časy algoritmů při detekci tautologie (DC 60%)



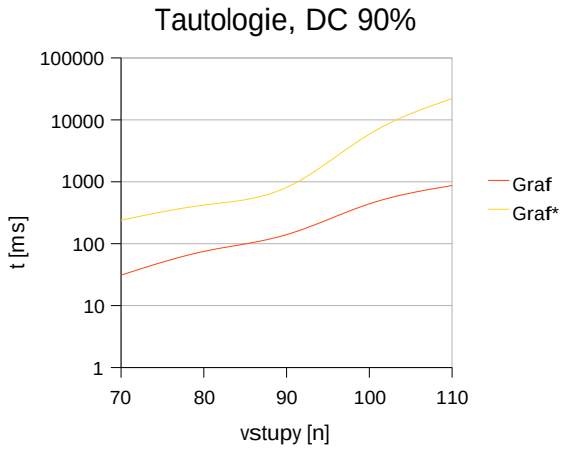
Obrázek 5.8: Graf s časy algoritmů při detekci tautologie (DC 70%)



Obrázek 5.9: Graf s časy algoritmů při detekci tautologie (DC 80%)

Při zhodnocování posledních tří grafů 5.8 až 5.10 je na první pohled patrné, že se do popředí dostávají grafové algoritmy. Za povšimnutí rovněž stojí fakt, že se pořadí našich dvou variant grafových algoritmů v tuto chvíli změnilo. Dříve byl rychlejší algoritmus upravující strukturu před samotnou detekcí, tady tomu již tak není a je výhodnější spouštět detekci tautologie přímo.
Příčina velké efektivity našich grafů je v jejich řídkosti a z toho vyplývající extrémně velké rychlosti zpracování. V posledních dvou grafech chybí algoritmus hrubé síly, neboť jeho časy byly již příliš vysoké, ze stejného důvodu není v posledním grafu algoritmus BDD.

Celkové zhodnocení:
V našich testovacích podmínkách se jako nejpoužitelnější algoritmus pro malá n jeví algoritmus hrubé síly. S rostoucím počtem DC byl tento algoritmus schopen udržet své prvenství až do velikosti vstupu $n < 20$ při DC $< 60\%$.
Počínaje DC $= 70\%$ se v testech ukázaly jako nejrychlejší naše grafové algoritmy, konkrétně klasický algoritmus tautologie bez úprav. Druhý náš grafový algoritmus pomáhající si předchozími úpravami je dobře použitelný pro složitější grafy, kde je DC nízké, ale při DC $> 70\%$ už tak



Obrázek 5.10: Graf s časy algoritmů při detekci tautologie (DC 90%)

dobré výsledky nevykazuje.
Je jasné, že někde mezi 60-70% se nachází předěl, kde grafový algoritmus získává své prvenství. Pro jeho podrobnější vyhledání by bylo zapotřebí ještě značné množství dalších testů. Jedním z přínosů této práce je tedy grafový algoritmus řešící tautologii velmi rychle pro DC větší než 60-70%.

5.3 Výpočet komplementu

Pro výpočet komplementu máme jedinou grafovou funkci. Tuto funkci budeme srovnávat s programem Espresso [10], který, přestože primárně slouží k minimalizaci funkcí, je možné parametrizovat tak, že bude vyhledávat komplement námi zadané funkce. (`espresso.exe -ofr vstup.pla`)
Výsledky budeme srovnávat pouze z hlediska množství času věnovanému vyřešení úlohy. Kvalitu řešení v tomto případě srovnávat již nebudeme, neboť by se jednalo o testování kvality závěrečné minimalizace, kterou implicitně espresso v tomto případě nespouští a tudíž není nutné ji provádět. Efektivita minimalizace je otestována v 5.4. Zde v tomto testu budeme pouze vypočítávat komplement, který se nebudeme pokoušet dále zjednodušovat.

5.3.1 Instance

Využijeme instance použité již dříve v tautologii. Z těchto instancí vybereme náhodných 10 s DC 0-90% ve vstupních proměnných. Tyto instance patřičně zkrátíme o určitý počet termů tak, aby se nejednalo o tautologii. Takto vytvořené instance jsou pochopitelně jen velmi hrubý nástroj pro otestování komplementu. V případě srovnatelných časů, bychom nasadili jemnější nástroj k testování podobně, jako například u tautologie. Důvodem, proč takto postupujeme, je naše domněnka, že výpočet komplementu zavedený na původním algoritmu pro detekci tautologie nebude příliš efektivní. Pokud se potvrdí tato naše domněnka, není třeba jej nadále ještě testovat.

5.3.2 Naměřené výsledky

V tabulce 5.11 jsou zobrazeny výsledky pro námi vybraných 10 instancí. Název souboru je ve tvaru [počet vstupů]x[počet výstupů]_[procento DC].pla. Některé úlohy, které během řešení přesáhly dobu pěti minut, nebyly dopočítány až do konce. Převaha algoritmu Espresso je jasně

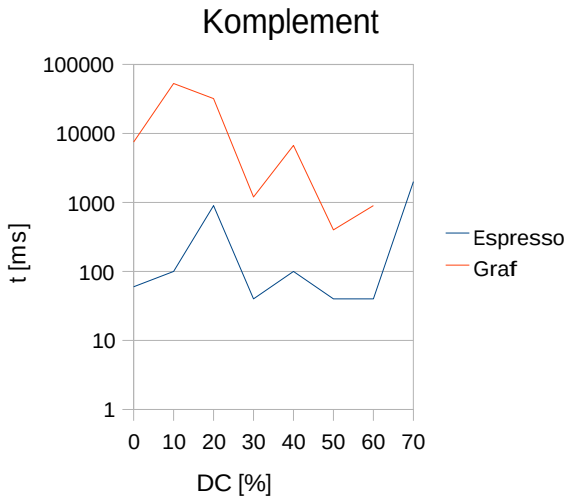
patrná již z několika málo hodnot.

	celkový čas úlohy	
vstupní soubor	Espresso	Graf
08x251_0.pla	60 ms	7,5 s
09x925_10.pla	0,1 s	53 s
10x1025_20.pla	0,9 s	32 s
11x650_30.pla	40 ms	1,2 s
12x576_40.pla	0,1 s	6,7 s
13x251_50.pla	40 ms	0,4 s
15x275_60.pla	40 ms	0,9 s
26x76_70.pla	2 s	>300 s
40x476_80.pla	>300 s	>300 s
90x426_90.pla	>300 s	>300 s

Tabulka 5.11: Srovnání rychlostí algoritmu pro výpočet komplementu funkce

5.3.3 Grafy a zhodnocení

Jak je jasně patrné z tabulky 5.11 i grafu 5.11, potvrdila se naše hypotéza o neefektivnosti grafového algoritmu pro výpočet komplementu. Je to způsobeno především několikanásobným zkoušením podobných variací ohodnocení, které vyplývá z logiky postupu algoritmu. S větším množstvím termů funkce roste i doba zpoždění za Espressem.



Obrázek 5.11: Graf s časy algoritmů při výpočtu komplementu

5.4 Minimalizace funkce

Obě verze našeho minimalizačního algoritmu (rychlou i pomalou) tak, jak byly popsány v 3.4.2, budeme testovat s minimalizačním programem Espresso [10]. Po proběhnutí těchto algoritmů nejprve zjistíme, zda námi vytvořené PLA je z hlediska výstupů identické s předcházejícím rovněž za pomoci Espressa a pak obě řešení srovnáme z hlediska časové náročnosti a kvality. Kvalitativní srovnání proběhne na základě počtu termů v minimalizovaných PLA, v případě, že tento počet bude pro dva z algoritmů stejný, bude druhé měřítko kvality celkové množství DC ve vstupních proměnných.

5.4.1 Instance

Opět budeme parametrizovat generátor použitý v tautologii. Tentokrát budeme chtít sledovat změnu efektivity v závislosti na více parametrech. Proto navolíme implicitní konfiguraci a budeme v ní vždy měnit pouze jednu proměnnou, abychom dobře viděli, jaký má její změna vliv na efektivitu algoritmu. Implicitní konfigurace:

- počet vstupů = 40
- počet výstupů = 1/5 (budeme testovat pouze jednovýstupovou a pětivýstupovou funkci)
- počet termů = 200
- don't care ve vstupech = 50%

V této implicitní konfiguraci budeme vždy měnit pouze jeden parametr. Nejprve počet vstupů, pak termů a nakonec don't care. Příkaz pro tvorbu našich souborů PLA bude implicitně vypadat takto:

```
genpla.exe -i 40 -o 1 -t 200 -idc 50 -odc 30 -odt 100
```

5.4.2 Naměřené výsledky

Tabulka 5.12 obsahuje výsledky pro jednovýstupovou funkci s pohyblivým parametrem počtu vstupů, počtu termů a procentuálního množství DC ve vstupech. Jména vstupních souborů jsou ve tvaru [počet vstupů]x[počet výstupů]x[množství termů].pla.
Pozn.: Od testování pomalejší verze algoritmu bylo nakonec upuštěno z důvodu velké efektivity rychlé verze algoritmu.

	Espresso			**Graf**		
	čas	počet termů	počet DC	čas	počet termů	počet DC
40x1x10.pla	21 ms	7	156	35 ms	7	156
40x1x20.pla	31 ms	15	314	40 ms	15	314
40x1x50.pla	75 ms	39	805	78 ms	39	805
40x1x75.pla	136 ms	55	1115	120 ms	55	1115
40x1x100.pla	270 ms	69	1382	185 ms	69	1382
40x1x200.pla	1,7 s	142	2861	0,55 s	142	2861
40x1x250.pla	98 s	172	3446	0,79 s	172	3446
40x1x400.pla	>300 s	-	-	1,97 s	294	-
10x1x200.pla	20 ms	1	10	29 ms	1	10
20x1x200.pla	0,45 s	145	-	0,36 s	148	-
40x1x200.pla	1,6 s	148	-	0,6 s	142	-
60x1x200.pla	1,8 s	141	4265	0,8 s	141	4265
80x1x200.pla	2,4 s	135	5424	1 s	135	5424
100x1x200.pla	2,9 s	134	6761	1,2 s	134	6761
125x1x200.pla	265 s	155	9684	1,5 s	125	9684
150x1x200.pla	289 s	132	9946	1,9 s	132	9946
200x1x200.pla	8 s	137	13762	2,6 s	137	13762
40x1x200dc0.pla	0,13 s	144	0	0,6 s	144	0
40x1x200dc10.pla	0,14 s	141	553	0,6 s	141	553
40x1x200dc20.pla	0,17 s	146	1165	0,59 s	146	1165
40x1x200dc30.pla	0,21 s	134	1543	0,58 s	134	1543
40x1x200dc40.pla	0,43 s	142	2206	0,58 s	142	2206
40x1x200dc50.pla	1,35 s	136	2693	0,5 s	136	2693
40x1x200dc60.pla	>300 s	-	-	0,21 s	147	-
40x1x200dc70.pla	>300 s	-	-	0,68 s	149	-
40x1x200dc80.pla	>300 s	-	-	0,7 s	151	-

Tabulka 5.12: Srovnání rychlostí algoritmů v minimalizaci funkcí s jedním výstupem

Dále byly srovnány algoritmy v minimalizaci pětivýstupové funkce. Výsledky jsou zobrazeny v tabulce 5.13. Jména vstupních souborů jsou zde opět ve tvaru [počet vstupů]x[počet výstupů]x[množství termů].pla.

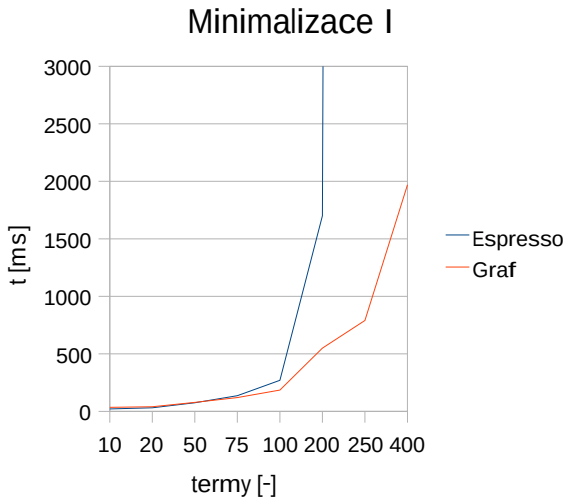
	Espresso			**Graf**		
	čas	počet termů	počet DC	čas	počet termů	počet DC
40x5x10.pla	30 ms	10	198	50 ms	10	198
40x5x20.pla	50 ms	20	416	80 ms	20	416
40x5x50.pla	0,3 s	49	985	0,3 s	49	985
40x5x75.pla	0,6 s	74	1505	0,6 s	74	1505
40x5x100.pla	1,3 s	98	1990	1 s	98	1990
40x5x200.pla	9,6 s	198	4005	3,8 s	198	4005
40x5x250.pla	>300 s	-	-	7 s	249	-
40x5x400.pla	>300 s	-	-	17 s	398	-
20x5x200.pla	3 s	200	-	3 s	196	-
40x5x200.pla	11 s	198	4005	4 s	198	4005
60x5x200.pla	11 s	199	5985	6 s	199	5985
80x5x200.pla	>300 s	-	-	8 s	200	8065
40x5x200dc0.pla	8 s	200	0	3 s	200	0
40x5x200dc10.pla	0,9 s	199	821	3 s	199	821
40x5x200dc20.pla	1,1 s	200	1658	3 s	200	1658
40x5x200dc30.pla	2 s	200	2443	2,5 s	200	2443
40x5x200dc40.pla	4 s	200	3249	3 s	200	3249
40x5x200dc50.pla	12 s	198	4005	2,2 s	198	4005
40x5x200dc60.pla	>300 s	-	-	2,8 s	199	4854

Tabulka 5.13: Srovnání rychlostí algoritmů v minimalizaci funkcí s pěti výstupy

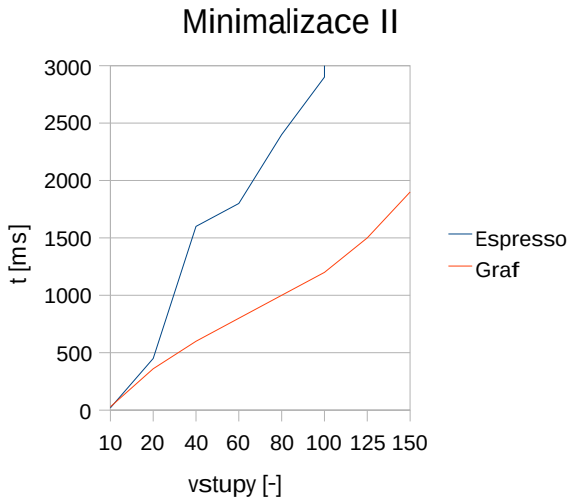
5.4.3 Grafy a zhodnocení

V této části jsou graficky znázorněny výsledky měření z hlediska časové náročnosti algoritmů. Pokud křivka v grafu končí dříve než na jeho konci, je to tím, že doba v tomto bodě přesáhla 300 s. Vykreslovat kvalitu řešení grafem v tomto případě nebudeme, neboť oba algoritmy se téměř vždy shodují a graf by nebyl příliš názorný.

Na prvním grafu 5.12 je vidět, že u Espressa je křivka daleko strmější a doba řešení problému u něj roste s rostoucím počtem termů rychleji.

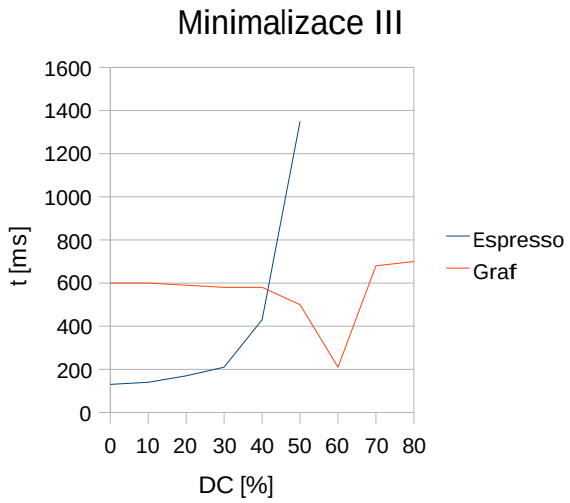


Obrázek 5.12: Rychlosti minimalizačních algoritmů u jednovýstupových funkcí v závislosti na počtu termů



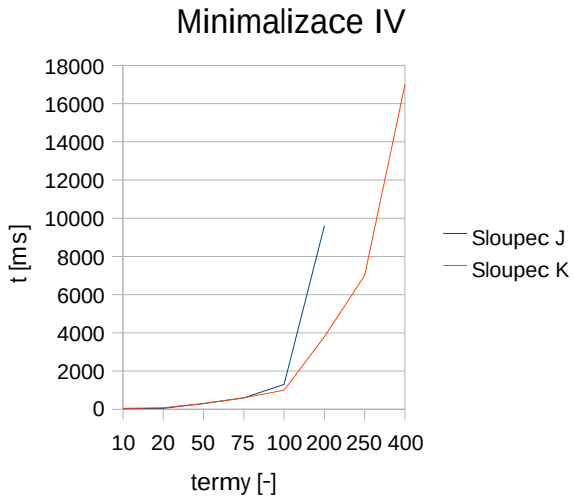
Obrázek 5.13: Rychlosti minimalizačních algoritmů u jednovýstupových funkcí v závislosti na počtu vstupů

Obrázek 5.13 zobrazuje graf závislosti na počtu vstupních proměnných. Jak je vidět, u obou algoritmů jde v tomto případě o téměř lineární závislost. Grafový algoritmus je opět rychlejší než Espresso.



Obrázek 5.14: Rychlosti minimalizačních algoritmů u jednovýstupových funkcí v závislosti na množství DC ve vstupech

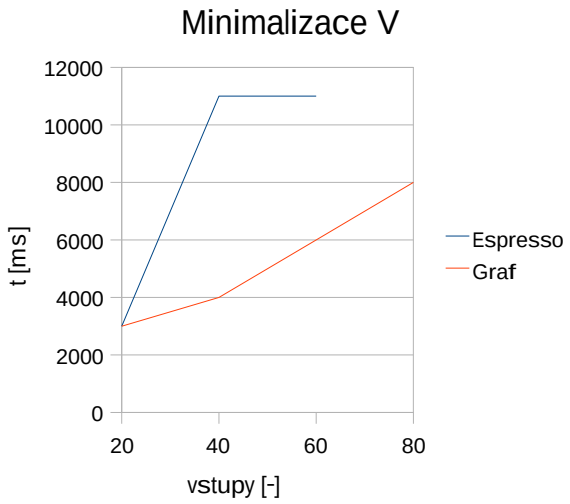
Na grafu 5.14 můžeme vidět, že pro nižší DC byl Espresso zhruba trojnásobně rychlejší oproti grafovému algoritmu. Zlom nastává mezi 40-50%, kdy se ukazuje, že Espresso nabírá strmý exponenciální tvar pro vyšší procento DC, zatímco grafový algoritmus je vůči změně DC inertní.



Obrázek 5.15: Rychlosti minimalizačních algoritmů u vícevýstupových funkcí v závislosti na počtu termů

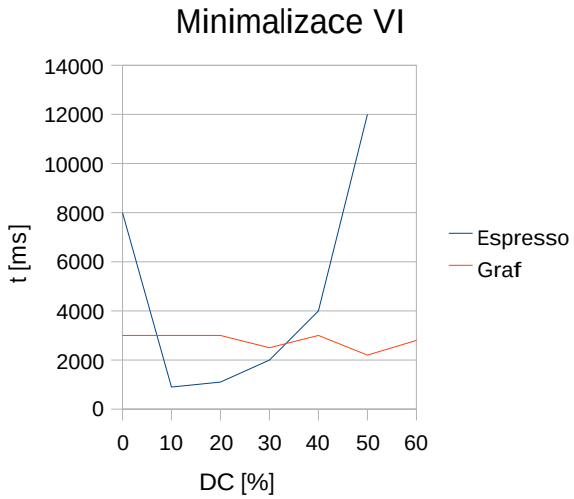
Na grafu 5.15 můžeme vidět podobnou závislost, jako jsme již pozorovali u grafu 5.12. Množství výstupních hodnot tedy nemá vliv, stále zůstávají na grafu dvě podobné křivky, z nichž ta

patřící algoritmu Espresso je strmější.



Obrázek 5.16: Rychlosti minimalizačních algoritmů u vícevýstupových funkcí v závislosti na počtu vstupů

Graf 5.16 vyobrazuje (podobně jako graf 5.13) téměř lineární závislost na počtu vstupních proměnných. Espresso je opět pomalejší a má horší časy než grafový algoritmus.



Obrázek 5.17: Rychlosti minimalizačních algoritmů u jednovýstupových funkcí v závislosti na množství DC ve vstupech

Na obrázku 5.17 vidíme zase podobnou závislost jako u jednovýstupové funkce zobrazené na obrázku 5.14. Grafový algoritmus je opět zhruba 3x pomalejší než algoritmus Espresso až do zlomu, který nastává mezi 40-50% DC. V počátečním bodě byl sice Espresso pomalejší než grafový algoritmus, ale zřejmě jde o výjimku, která by při větším množství testů byla statisticky odstraněna.

Celkové zhodnocení:
Pro většinu našich instancí se ukázal být jako rychlejší grafový algoritmus než Espresso. Nicméně toto mohlo být částečně zapříčiněno nevhodnou volbou parametrů PLA. Na jednotlivých grafech je možné vidět závislosti algoritmů a je evidentní, že náš grafový algoritmus získává nejvíce na parametru DC. Zatímco počet termů ovlivňuje oba algoritmy exponenciálně a počet vstupů lineárně, u DC je vidět markantní rozdíl ve zpracování oběma různými algoritmy.
Dá se hypoteticky předpokládat, že při DC 0-30% by dosahoval Espresso lepších výsledků než grafový algoritmus nezávisle na ostatních parametrech. Naopak při DC nad 50% se jeví jako rychlejší grafový algoritmus.

5.5 Ortogonalizace funkce

Výsledky obou variant našeho grafového algoritmu pro ortogonalizaci budou srovnány s programem DSOP [1]. Podobně jako u minimalizace provedeme kvalitativní srovnání na základě počtu termů v novém vyjádření funkce a v případě, že bude stejný, zaměříme se na množství DC ve vstupních proměnných.
U všech výstupních hodnot budeme za pomoci Espressa ověřovat, zda se skutečně jedná o vyjádření stejné funkce, jako jsme použili na vstupu jednotlivých algoritmů.

5.5.1 Instance

Pro účely testování ortogonalizace jsme se rozhodli použít stejné instance problému, jako jsme měli u problému minimalizace.

5.5.2 Naměřené výsledky

V tabulce 5.14 jsou výsledky ortogonalizací u jednovýstupové funkce s proměnlivými parametry. Jména vstupních souborů jsou ve tvaru [počet vstupů]x[počet výstupů]x[množství termů].pla.

	DSOP			**Graf-fast**			**Graf-slow**		
	čas	termy	DC	čas	termy	DC	čas	termy	DC
40x1x10.pla	16 ms	7	156	12 ms	7	156	15 s	7	156
40x1x20.pla	0,1 s	25	503	16 ms	25	503	62 ms	25	503
40x1x50.pla	0,3 s	141	-	0,6 s	168	-	0,5 s	168	-
40x1x75.pla	0,3 s	178	-	0,7 s	205	-	1,1 s	205	-
40x1x100.pla	0,4 s	215	-	0,9 s	242	-	13 s	242	-
40x1x200.pla	2,2 s	580	-	22 s	1044	-	>300 s	-	-
40x1x250.pla	2 s	604	-	16,2 s	920	-	>300 s	920	-
40x1x400.pla	>300 s	0	-	>300 s	-	-	>300 s	-	-
10x1x200.pla	32 ms	1	10	15 ms	1	10	15 ms	1	10
20x1x200.pla	77 s	2590	-	>300 s	-	-	>300 s	-	-
40x1x200.pla	2,2 s	580	-	22 s	1044	-	>300 s	-	-
60x1x200.pla	430 ms	245	6776	1,9 s	245	7517	26 s	245	7517
80x1x200.pla	63 ms	135	5424	1,3 s	135	5424	17 s	135	5424
100x1x200.pla	140 ms	134	6761	1,6 s	134	6761	3,4 s	134	6761
125x1x200.pla	125 ms	155	9684	2,1 s	155	9684	4,3 s	155	9684
150x1x200.pla	95 ms	132	9946	2,36	132	9946	5 s	132	9946
200x1x200.pla	94 ms	137	13762	3,1 s	137	13762	6,9 s	137	13762
40x1x200dc0.pla	47 ms	143	-	0,9 s	144	-	1,8 s	144	-
40x1x200dc10.pla	47 ms	141	553	0,9 s	200	-	1,8 s	141	553
40x1x200dc20.pla	47 ms	146	1165	0,9 s	146	1165	1,7 s	146	1165
40x1x200dc30.pla	47 ms	134	1543	0,8 s	134	1543	1,8 s	134	1543
40x1x200dc40.pla	170 ms	163	2420	0,9 s	163	2541	19 s	163	2541
40x1x200dc50.pla	0,9 s	396	-	7 s	619	-	>300 s	-	-
40x1x200dc60.pla	>300 s	-	-	>300 s	-	-	>300 s	-	-
40x1x200dc70.pla	>300 s	-	-	>300 s	-	-	>300 s	-	-
40x1x200dc80.pla	>300 s	-	-	>300 s	-	-	>300 s	-	-

Tabulka 5.14: Srovnání rychlostí algoritmů v ortogonalizaci funkcí s jedním výstupem

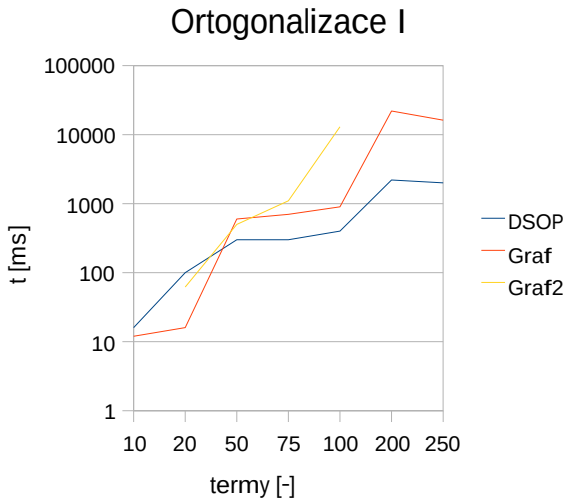
U vícevýstupových funkcí se ukázala pomalejší verze algoritmu jako příliš časově náročná. Veškeré výpočty trvaly déle než 300 s, a proto bylo od srovnání tohoto algoritmu upuštěno. V tabulce 5.15 jsou tedy srovnány pouze DSOP s rychlejší verzí grafové ortogonalizace.

	DSOP			**Graf-fast**		
	čas	počet termů	počet DC	čas	počet termů	počet DC
40x5x10.pla	31 ms	10	198	31 ms	10	198
40x5x20.pla	180 ms	39	682	200 ms	39	682
40x5x50.pla	0,4 s	159	-	1,1 s	199	-
40x5x75.pla	1 s	290	-	4,5 s	478	-
40x5x100.pla	1,4 s	393	-	6,2 s	629	-
40x5x200.pla	32 s	22	-	228 s	3248	-
40x5x250.pla	>300 s	-	-	>300 s	-	-
40x5x400.pla	>300 s	-	-	>300 s	-	-
20x5x200.pla	>300 s	-	-	>300 s	-	-
40x5x200.pla	32 s	22	-	228 s	3248	-
60x5x200.pla	1,1 s	21	-	6,8 s	313	-
80x5x200.pla	1 s	21	-	7,2 s	242	-
100x5x200.pla	170 ms	200	10066	8,5 s	200	10066
125x5x200.pla	200 ms	200	12499	9,9 s	200	12499
150x5x200.pla	250 ms	200	15011	12 s	200	15011
200x5x200.pla	330 ms	199	19953	16 s	199	19953
40x5x200dc0.pla	100 ms	200	0	4,1 s	200	0
40x5x200dc10.pla	100 ms	199	821	4,2 s	199	821
40x5x200dc20.pla	100 ms	200	1658	4,1 s	200	1658
40x5x200dc30.pla	230 ms	213	2586	4,2 s	213	2664
40x5x200dc40.pla	450 ms	325	-	5 s	389	-
40x5x200dc50.pla	27 s	1319	-	222 s	3248	-
40x5x200dc60.pla	>300 s	-	-	>300 s	-	-

Tabulka 5.15: Srovnání rychlostí algoritmů v ortogonalizaci funkcí s pěti výstupy

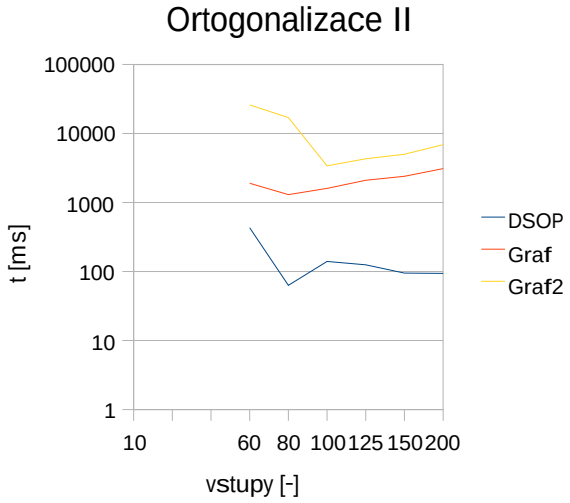
5.5.3 Grafy a zhodnocení

Následující grafy zobrazují srovnání časů jednotlivých algoritmů. Srovnání z hlediska kvality je názornější z tabulek 5.14 a 5.15. V žádné použité instanci neměl žádný grafový algoritmus vyšší efektivitu než DSOP. Asi ve 40% případů měly algoritmy efektivitu stejnou, v ostatních případech byl DSOP až několikanásobně efektivnější.



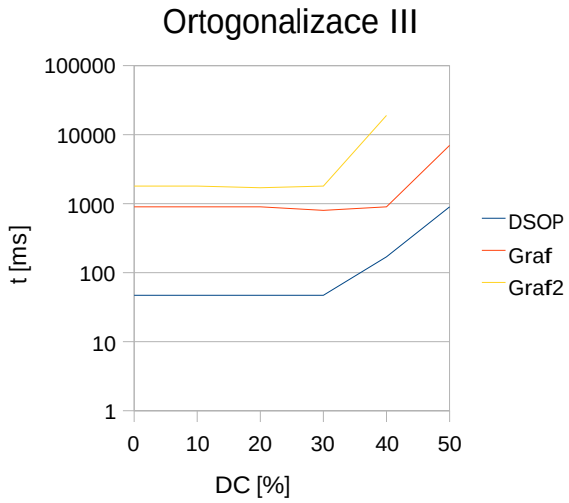
Obrázek 5.18: Rychlosti algoritmů ortogonalizace u jednovýstupových funkcí v závislosti na počtu termů

Graf 5.18 zobrazuje závislost doby výpočtu na množství termů. Všechny algoritmy vykazují podobný průběh, avšak DSOP je řádově rychlejší oproti grafovým algoritmům.



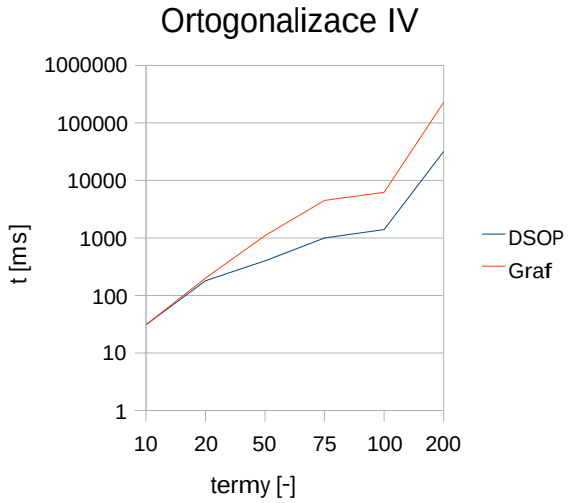
Obrázek 5.19: Rychlosti algoritmů ortogonalizace u jednovýstupových funkcí v závislosti na počtu vstupů

Na obrázku 5.19 je graf závislosti doby výpočtu na počtu proměnných. Je vidět, že s rostoucím počtem vstupních proměnných se doba výpočtu příliš nezhoršuje, v případě DSOP dokonce pro naše instance tato doba klesá.



Obrázek 5.20: Rychlosti algoritmů ortogonalizace u jednovýstupových funkcí v závislosti na DC

Z grafu 5.20 je patrné, že procentuální množství DC ve vstupních hodnotách nemá pro interval 0-30% žádný vliv, při vyšších hodnotách je nárůst dosti strmý (od 60% výše již žádný algoritmus neskončil v čase 300 s a méně).

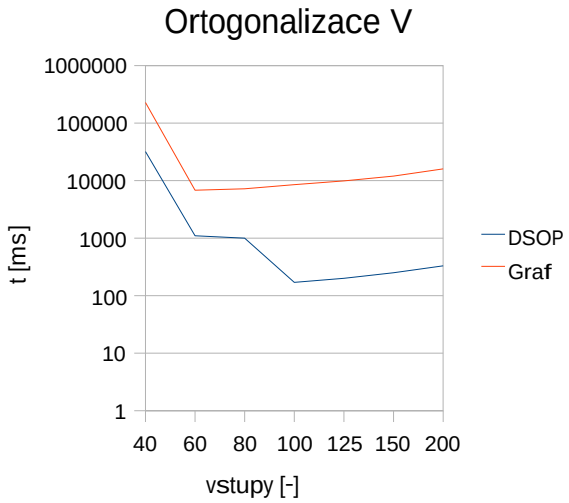


Obrázek 5.21: Rychlosti algoritmů ortogonalizace u funkcí s pěti výstupy v závislosti na počtu termů

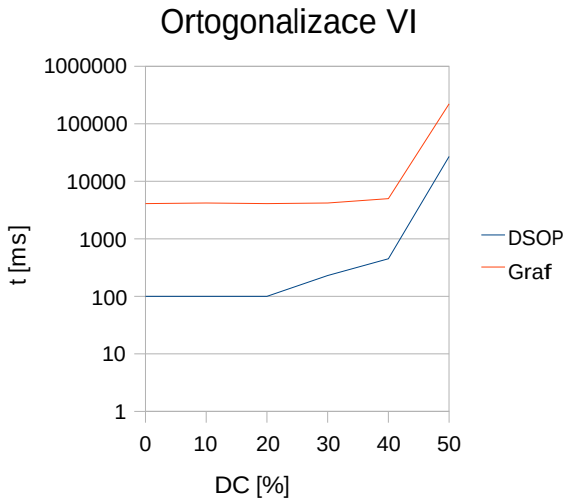
Graf 5.21 vykazuje podobné vlastnosti jako graf 5.18. Opět je DSOP řádově rychlejší než náš grafový algoritmus.

Z grafu 5.22 lze vyčíst podobné chování jako u jednovýstupové funkce zobrazené v grafu 5.19. Množství termů nemá zásadní vliv na dobu výpočtu ani v jednom z algoritmů.

Na grafu 5.23 se opět ukazuje, že pro nízké hodnoty DC platí, že neovlivňují dobu výpočtu. Až při vyšších hodnotách (v tomto případě 40% a více) dochází k prudkému nárůstu doby výpočtu.



Obrázek 5.22: Rychlosti algoritmů ortogonalizace u funkcí s pěti výstupy v závislosti na počtu vstupů



Obrázek 5.23: Rychlosti algoritmů ortogonalizace u funkcí s pěti výstupy v závislosti na DC

Celkové zhodnocení:
Ve všech případech našich instancí se ukázal být DSOP nejefektivnější. Naše grafové algoritmy se mu dokázaly vyrovnat v asi 40% případů z hlediska efektivity. Ve srovnání času opět nejlépe obstál DSOP, kdy v porovnání s rychlejším grafovým algoritmem byl až 10x rychlejší.
Rozdíl mezi pomalejší a rychlejší verzí našeho algoritmu v časech je rovněž mnohdy řádový, přičemž pomalejší verze dosáhla vyšší efektivity oproti verzi rychlejší pouze jedenkrát.

6 Závěr

Cílem této práce bylo naimplementovat grafovou strukturu pro logické funkce zadané v PLA řešící problém tautologie a zanalyzovat, zda by se dala tato reprezentace použít při ortogonalizaci funkce. Tento cíl byl splněn v celém rozsahu. Práce navíc obsahuje i samotnou implementaci ortogonalizace a kromě toho také implementaci funkce pro výpočet komplementu a funkce pro minimalizaci.

Bylo zjištěno, že *detekce tautologie* nad touto grafovou strukturou je velice efektivní, pokud množství DC ve vstupních hodnotách přesahuje 70%. V tomto pásmu dosáhla výsledků v kratším čase než všechny ostatní testované algoritmy pro detekci tautologie na všech použitých instancích. *Ortogonalizace* nad grafem dosahovala kvalitativně srovnatelných výsledků s referenčním algoritmem asi ve 40% případů, v ostatních případech nebyla tak efektivní. Z hlediska toku času byla 2-50x pomalejší v závislosti na konkrétních instancích. Pravděpodobně největším přínosem a úspěchem této práce je implementace *minimalizace* funkce, která dosahovala srovnatelných a v mnoha případech i několikanásobně lepších časových výsledků než jeden z nejrozšířenějších současných minimalizačních algoritmů. Z hlediska kvality byly výsledky minimalizací vždy srovnatelné a ve více než 90% případů zcela totožné. Naopak *výpočet komplementu* implementovaný za pomoci grafu v této práci se ukázal jako velice neefektivní algoritmus, který pravděpodobně nebude mít žádné další využití.

V případném budoucím rozšíření by bylo možné se podrobněji zabývat otázkou ortogonalizace a pokusit se vytvořit konkurenceschopnější algoritmus, než byl naprogramován v rámci této práce. Dále by rovněž bylo možné naimplementovat lepší algoritmus pro výpočet komplementu a umožnit tak provádět konverze jednotlivých typů grafu, například z FR do FDR v krátkém výpočetním čase.

7 Literatura

[1] F. L. A. Bernasconi, V. Ciriani and L. Pagli. A new heuristic for DSOP minimization. In *Proc. 8th Int. Workshop on Boolean Problems (IWSBP'08)*, pages 169–174, Freiberg, Germany, 2008.

[2] T. T. Christopher Meinel. *Algorithms and Data Structures in VLSI Design : OBDD - Foundations and Applications*, volume 1. Springer, New York, 1st edition, 1998.

[3] P. Fišer and H. Kubátová. Flexible two-level boolean minimizer BOOM-II and tts applications. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 369–376, Washington, DC, USA, 2006. IEEE Computer Society.

[4] J. Kolář. *Teoretická informatika*. Česká informatická společnost, Praha, 1st edition, 2000.

[5] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 532–535, New York, NY, USA, 2006. ACM.

[6] T. Měchura. Parametrizovaný generátor náhodných booleovských funkcí, 2006. Bakalářská práce, FEL ČVUT v Praze.

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

[8] Input file format for espresso(1OCTTOOLS). `http://service.felk.cvut.cz/vlsi/prj/BoomBench/pla_c.html`.

[9] A. I. Provotar, V. A. Kondratenko, and T. N. Dudka. Boolean algebra as a fragment of the theory of boolean toposes. *Cybernetics and Sys. Anal.*, 37(1):131–137, 2001.

[10] R. L. Rudell. Multiple-valued logic minimization for PLA synthesis. Technical Report UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.

[11] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

A Obsah přiloženého CD

- `data/complement` - instance pro komplement
- `data/minimization` - instance pro minimalizaci
- `data/orthogonalization` - instance pro ortogonalizaci
- `data/tautology` - instance pro tautologii
- `exe/complement` - spustitelné soubory použité při testování komplementu
- `exe/minimization` - spustitelné soubory použité při testování minimalizace
- `exe/orthogonalization` - spustitelné soubory použité při testování ortogonalizace
- `exe/tautology` - spustitelné soubory použité při testování tautologie
- `src/boolgraph` - zdrojový text programu
- `src/boom_graph` - zdrojový text programu zakomponovaný do jádra
- `text/dp.pdf` - text této diplomové práce ve formátu PDF programu Boom
- `readme.txt` - základní popis struktury adresářů