Úvod

(V této části práce budu užívat citace z předlohy od pana Vály)
Logické funkce a Booleova algebra se v dnešním světě informačních technologii 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 tranzistoru se časem ukázalo, Že takto složité mechanismy nejsou příliš výhodné, neboť jsou pomalejší a vedou k větší poruchovosti, než elektronický obvody pracující pouze se dvěma stavy. V dnešní době téměř všechny počítače pracují pouze s binárními hodnotami, tedy takovými, které mohou nabývat pouze dvou stavu 0,1 nebo-li nepravda, pravda. Takto ustálené vnímaní počítačového světa vede k obrovskému nárůstu významu celé Booleovy algebry, kterou definoval již v polovině 19. století britský matematik George Boole. Až do počátku výpočetní techniky neměla příliš praktické využití.
Používaní tohoto aparátu při výrobě počítačů a logických obvodu 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 hledaný exaktních řešení některých problémů muže 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 grafu, 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 konkurenceschopná oproti současným konvenčním způsobům řešení. Součástí práce je i implementace řešení některých problémů a srovnání rychlosti s ostatními programy, které se dnes běžně používají.

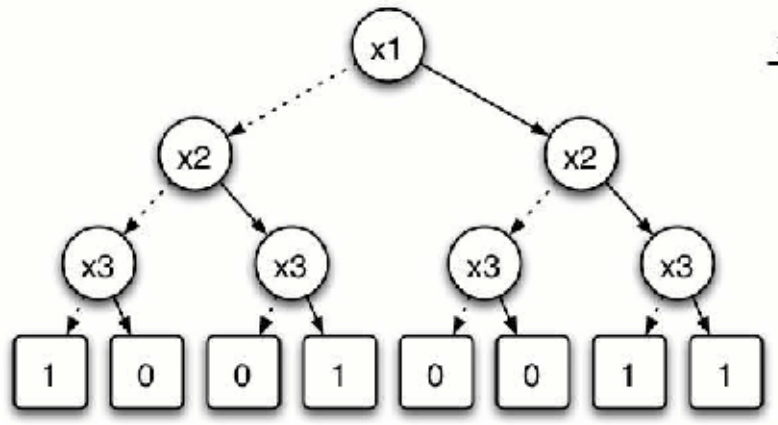
Existující řešení

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

BDD

Binární rozhodovací diagramy (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

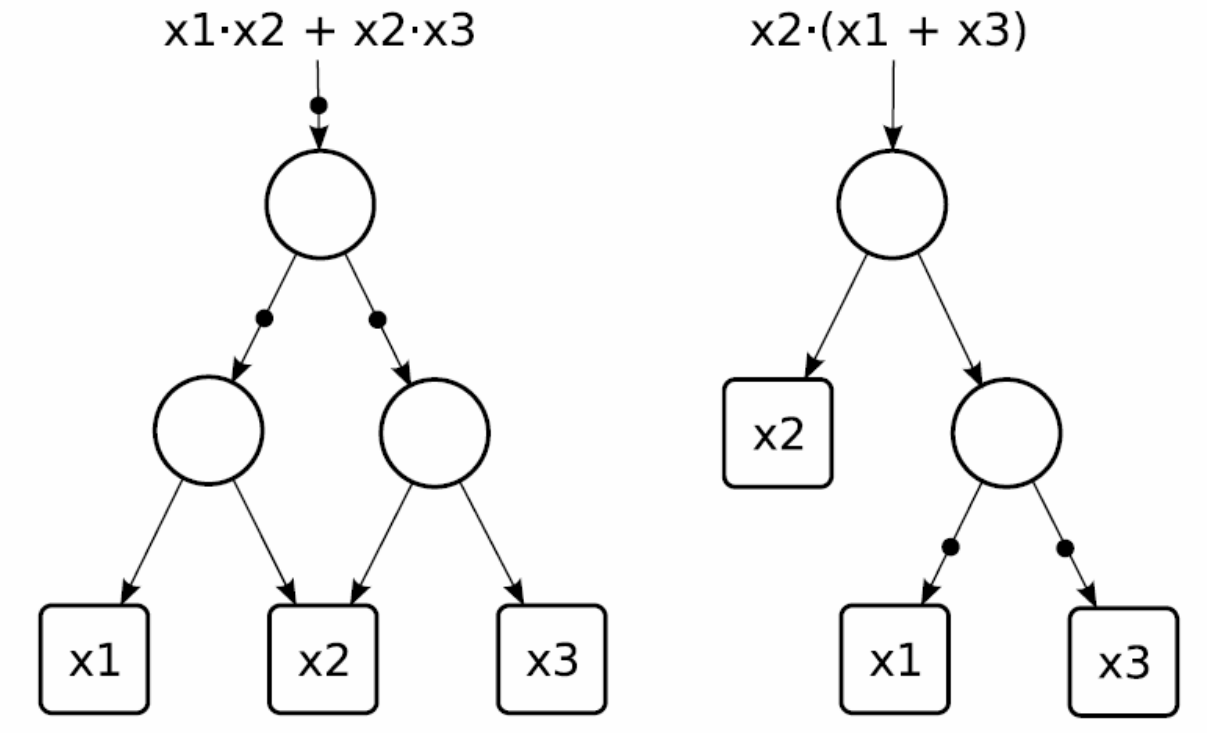
pochopitelně 2^n, kde n je počet vstupu logické funkce. Na rozdíl od naší struktury však BDD popisují chování funkce, zatímco náš graf popisuje její strukturu. Příklad takového diagramu je na následujícím obrázku:



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

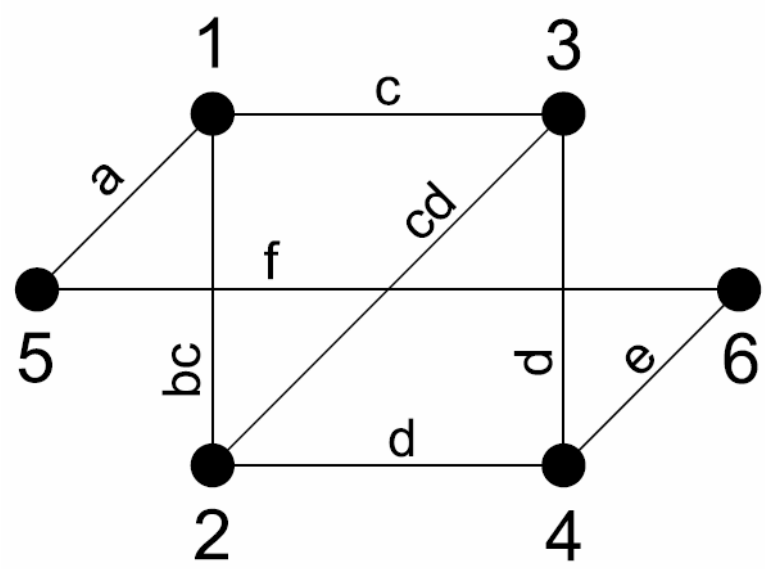
AIG

AIG (And-Inverter Graph) 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 invertoru 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 jednotlivě proměnný funkce. AIG už skutečně popisují strukturu funkce (podobně jako náš graf) a ne pouze její chování. Příklad AIG je vyobrazen na obrázku:



Návrh pana Vály

Tento návrh se využívá grafy tím způsobem, že pro každý term vytváří v grafu uzel. Uzly jsou po té mezi sebou propojeny hranou, pokud dané termy mají nějakou společnou proměnnou a nezáleží při tom na tom, jestli je proměnná v přímém nebo negovaném stavu. Toto je jen zjednodušený způsob, jak informace ukládat, ve skutečnosti tento návrh ukládá data trochu jiným způsobem, jak bude popsáno dále.



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

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

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 {O,1} (dále jen proměnná) literál proměnná nebo její negace
- **literál** proměnná nebo její negace
- **term** množina literálu, 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 {O,1}, výstupy nabývají pouze hodnot {O,1,neurčený stav} a který 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 O
- **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
- **disjunktivní normální forma** reprezentace logické funkce za pomoci součtu krychlí (součinu), často zkracována na DNF (případně anglickou zkratkou SOP), aby DNF definovala funkci, musí být pokrytý každý onset minterm dané funkce a nesmí být pokryt žádný offset minterm
- **ON-set** množina termu implikující výslednou funkci rovnou 1
- **OFF-set** množina termu implikující výslednou funkci rovnou O
- **DC-set** množina termu implikující výslednou funkci jako nespecifikovanou
- **DC** (don?t care) je označení nezávislosti na hodnotě dané proměnné
- **PLA** formát vstupních souboru pro Espresso (bude popsán dále)
- **původní návrh** návrh pana Vály

Popis problému a vymezení cíle

Nás1edující text vysvětluje, proč jsou některé úlohy prováděné s logickými funkcemi tak časově náročné a proč muže být vhodná volba heuristického algoritmu dobrá cesta, jak celý průběh operace signifikantně urychlit. Dále je zde uvedeno krátké přiblížení do problematiky logických funkcí, grafu a je zde popsán záměr práce.

Asymptotická složitost algoritmu, 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 algoritmu a prokázat, že jeden z nich má nižší složitost a bude tedy pro většinu zadaní rychlejší.
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 jednodužší a jeho složitost byla určena nesprávně.
Díky tomuto rozdělení složitosti problému 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 algoritmu 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. Následující tabulka ukazuje, jak dlouho trvá řešení problému 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 trvaní výpočtu prakticky nepoužitelný Proto tyto problémy řešíme často za pomoci heuristik, které nemusí dávat vždy exaktní řečení, 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

Booleova algebra

Definice: Booleova algebra je struktura <B, +, *,—,0, 1> kde platí:

- B je nosičem algebry
- + a * jsou binární operace na B
- - je unární operace
- 0,1 jsou nulární operace na B, kde pro všechna a, b, r ∈ 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 - vvtváří komplement: $a + \overline{a} = 1$, $a * \overline{a} = 0$

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

1. agresivita: 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 + a* b = a + b
6. de Morganů v zákon: $\overline{(a+b)} = a*\overline{b}$, $\overline{(a*b)} = \overline{a} + \overline{b}$

Logická funkce

Booleovy funkce n argumentů jsou funkce definované na n-ticích, které jsou vytvořeny z prvku množiny {O, 1} a jsou zobrazené do množiny {O, 1}. Tyto funkce se rovněž nazývají logickými funkcemi.

Libovolná proměnná z množiny n sama o sobě tvoří Booleovský výraz. Jednu funkci je možné vyjádřit mnoha způsoby, proto se používá standardní tvar normální forma. Jedná se o takový vyjádření funkce, v němž se vyskytují ve dvou úrovních operace součtu a součinu (jde tedy o součet součinu, nebo součin součtu).

V případě disjunktivní normální formy se jedná o součet součinu, kdy každá term ve výrazu se nazývá implikantem, neboť implikuje celou funkci tedy pokud tento term nabývá pro určité ohodnocení vstupu výsledku 1, nabývá celá funkce rovněž výsledku 1. S funkcemi zadanými v DNF pracuje i program BOOM, který využívá pro svou práci formátu PLA. V logických obvodech se rovná běžně používá vícevýstupová funkce. Vícevýstupová funkce je pouze rozšířena o možnost mít větší množství výstupu, které jsou definovány nad stejnou algebrou, ale jsou na sobě vzájemně nezávislé, pouze pro svou funkci využívají stejných vstupu.

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 výstupu.

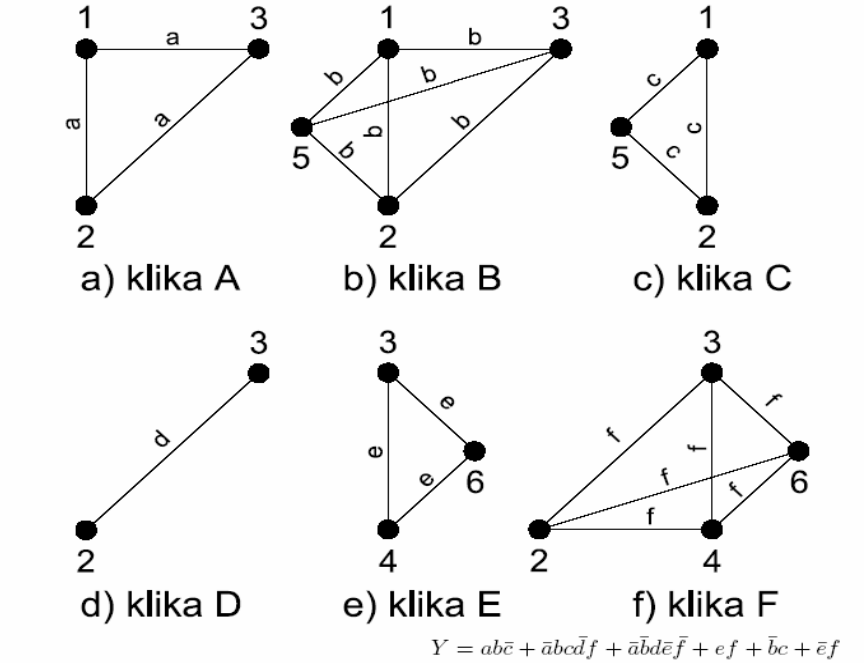
Grafy

Grafem v teorii grafu rozumíme objekt, který je popsaný množinou vrcholů a hran. Prostý graf je uspořádaná dvojice (V, H), kde V je množina vrcholu 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. Obecným grafu, který je tvořen uspořádanou trojicí (V, H, E), kde E je zobrazení incidence grafu.
Grafy lze dále rozdělit dle několika dalších parametrů.
Uvedeme zde ty nejzákladnější:
Orientované grafy - hrany orientovaného grafu jsou uspořádané dvojice vrcholu
Neorientované grafy - hrana je pouze dvouprvková množina vrcholu
Cyklické grafy - grafy, které obsahují aspoň jednu kružnici
Acyklické grafy - grafy, které neobsahují žádnou kružnici

Existuje více parametru, kterými se grafy odlišují. Pro naše potřeby však stačí tyto, neboť budeme využívat pouze malou část z celé grafové teorie. 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 nastroj pro zaznamenávání společných proměnných, kdy uzly grafu budou představovat jednotlivé termy z DNF a hrany budou mezi takovými dvěma uzly, který mají společné proměnné.

Grafová reprezentace logických funkcí navržená panem Válou

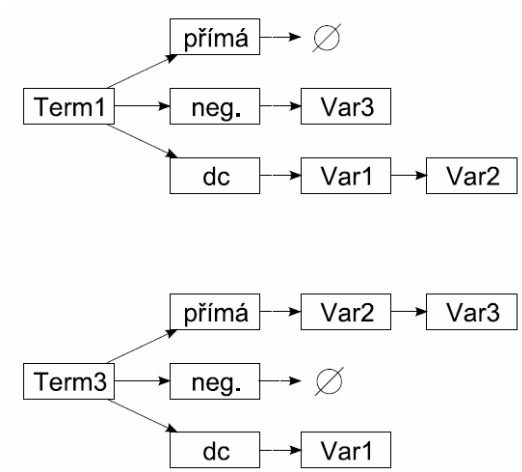
Již v úvodu jsem uvedl jednoduchý příklad návrhu pana Vály.
Zde uvedu jen základní vlastnosti, na které pan Vála přišel, podrobně to rozebírá ve své diplomové práci, kterou přikládám na CD.
Zjednodušeně řečeno si všiml toho, že graf lze rozdělit na kliky grafu (úplné podgrafy), kde každý podgraf je vždy pro jednu konkrétní proměnnou a každý term má v něm spoj s každým termem.

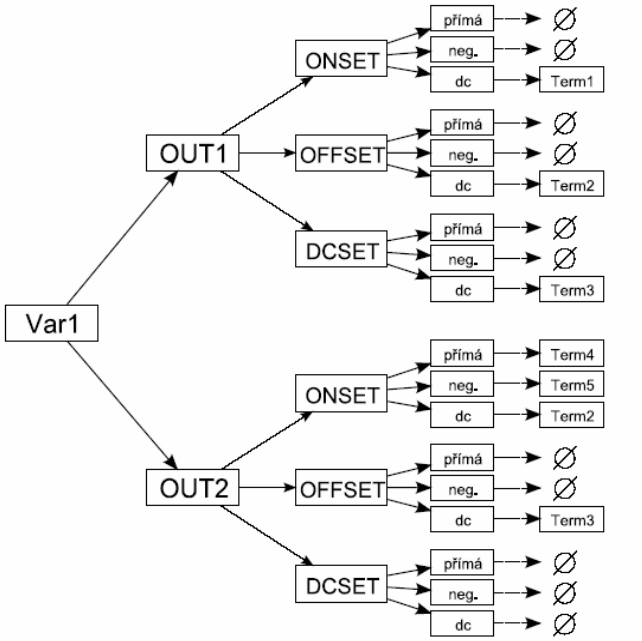


Toto zjištění ho vedlo k tomu, že není třeba vytvářet složitou grafovou strukturu, kde by každý uzel musel mít spoj na každý jiný uzel, který má s ním společnou proměnnou.
Ve skutečnosti stačí jen udělat seznam všech proměnných a ke každé proměnné seznam všech

termů, které tuto proměnnou obsahují.
Rozdělil proto implementaci na dva problémy, z nich každý v této práci probereme a zjistíme, jestli je vhodné jej implementovat do programu BOOM.
Tyto problémy jsou:

a) reprezentace vstupních proměnných termu pomocí tří seznamů a to pro přímou formu, negovanou a proměnné, které v termu nejsou obsaženy (obrázek vlevo)
b) pro každou proměnnou kompletní seznam všech termů, ve které je daná proměnná obsažena, tento seznam je dále rozdělen podle toho, v jakém stavu jsou výstupy a v jaké formě je proměnná v termu obsažena (obrázek vpravo)





Formát PLA

Tento formát byl původně navržen pro program Espresso 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 textovým 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 znaku, kde jednotlivé řádky odpovídají termům z DNF a sloupce představují jednotlivé vstupní proměnné. Specifikace celého formátu je možné najít v dokumentaci na přiloženém CD. Zde uvedeme pouze částečnou specifikaci s ohledem na její možné využití pro naše účely:

- veškeré bílé znaky v souboru jsou ignorovánv, pokud neslouží jako oddělovače při určování parametru
- znak # uvozuje komentář
- všechna klíčová slova začínají znakem . (tečkou)
- povinná klíčová slova: (tato slova musí obsahovat každý PLA)
  1 - udává počet vstupních proměnných
  0 - udává počet vstupních proměnných

Po hlavičce obsahující tato klíčová slova začíná matice termu. Tato matice se načítá po řádcích, kde v každém řádku jsou po řadě uloženy stavy všech n vstupů, pak následuje oddělovač a další řada znaku pro výstupy. Příklad takového PLA:

```
.i 3
.o2
.type fd
.p3
--0 11
00- 10
0-1 0-
.e
```

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 je od hlavičky oddělen prázdným řádkem.
V přední čtvercové časti matice se nachází vstupní proměnné, které jsou od výstupní časti odděleny bílými znaky. Ve vstupní časti se mohou vyskytovat znaky z množiny {1, 0, -} a vyjadřují, že proměnné 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 jedna o FD, představují jedničky ONSET, pomlčky DCSET a nuly nemají žádný význam.

Deklarace záměru

Cílem tohoto projektu je prověřit, jestli lze grafovou reprezentaci logických funkcí implementovat do programu BOOM a při tom zjistit, jestli bude mít lepší časovou složitost než původní implementace pomocí bitových polí. Budeme tedy muset rozdělit grafovou reprezentaci na jednotlivé problémy a prozkoumat, jestli má smysl je implementovat a jakým způsobem.

Analýza a návrh řešení

Analýza původního návrhu a jeho implementace

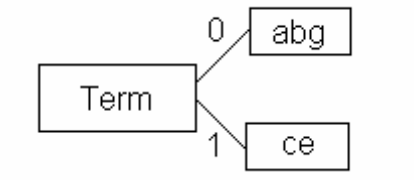
V původním návrhu bylo použito spojových seznamů, jako kontejneru, který držel pro každý term seznam proměnných. Přesně řečeno byly to tři seznamy a to první pro proměnné v přímé formě, druhý pro proměnné v negované formě a třetí seznam pro proměnné, které se v termu nevyskytovaly (DC-set). Tato implementace má v některých případech nevýhody. Např. pokud budeme porovnávat dva termy, a bude na nich zjišťovat, které proměnné mají shodné, potom budeme muset projít seznamy prvního termu lineárně jednu proměnnou po druhé a k tomu, abychom zjistili, jestli je tato proměnná i ve druhém termu, budeme muset projít alespoň dva z jeho seznamů také lineárně. To je složitost $O(n^2)$, kde n je počet proměnných. Toto lze ale provést s lepší složitostí. Prohledávání prvního termu budeme muset vždy projít lineárně, jinak se seznam všech proměnných nedozvíme, zde se nic vylepšit nedá. Ale požitím vhodného kontejneru můžeme ušetřit vyhledávání ve druhém termu. My jen potřebujeme vědět, jestli ve druhém termu proměnná je, nebo ne a v jaké formě, jestli v negované nebo přímé.
Pokud namísto spojových seznamů použijeme množinu (set), můžeme se potom tohoto kontejneru ptát na to, jestli proměnnou obsahuje či nikoliv, v konstantním čase. Celá operace na zjištění toho, jestli mají dva termy společnou proměnnou potom bude znamenat, že se musí projít množiny vstupních proměnných u jednoho termu a na každou proměnnou se zeptat (tentokrát s konstantní časovou složitostí) na druhém termu. Dohromady to dává složitost O(n), kde n je počet vstupních proměnných, což je složitost lepší než u původního návrhu.

Výběr vhodného kontejneru

Nyní se musíme rozhodnou, jaký druh množiny budeme používat a jak do ní budeme ukládat naše proměnné.

Způsob využití množiny

Doposud je v BOOMu uložen term jako pole znaků, kde číslo indexu znamená pořadí proměnné a hodnota pole na daném indexu určovala, jestli tam proměnná je a v jaké je formě, jestli v příme nebo negované.
Př: 001-1-0
Tento příklad odpovídá termu $\overline{ab}c\overline{eg}$
Množinu budeme používat tak, že do ní vložíme číslo proměnné a tím dáme najevo, že proměnná je v daném termu.
Pro každý term budou 2 množiny a to pro přímé a negované proměnné. To jestli je tam proměnná v přímé nebo negované formě dáme najevo tím, že jí vložíme do příslušné množiny.
Na rozdíl od původního návrhu se použijí jen dvě množiny, třetí množina pro DC nebude existovat. To že na proměnné nezáleží se dá najevo tím, že nebude ani v přímé ani v negované množině.
Náš příklad tedy bude vypadat takto:



Zvolení množiny

Obecně existují dva druhy množin:

a) množiny, které udržují svoje prvky seřazené
b) množiny, které udržují svoje prvky neseřazené

Implementace těchto množin v c++ je std::map a std::tr1::unordered_map.
std::map ukládá data v binárním stromě, zatímco std::tr1::unordered_map ukládá data podobně jako hash tabulku, jen bez hodnoty (uloží jen klíč)
Pojďme se nyní zabývat tím jestli vůbec potřebujeme mít nějakým způsobem prvky v množině seřazené.
Pokud se podíváme do zdrojových souborů aplikace BOOM, tak zjistíme, že program sice vyhledává vždy od první proměnné k poslední, ale to jen proto, že je to nejjednodušší způsob implementace. Ve skutečnosti na pořadí, ve kterém se proměnné testují nezáleží.
Můžeme si tedy zvolit tu mapu, která nám bude více vyhovovat a to z hlediska paměťové nebo výkonové náročnosti.
Vzhledem k tomu, že se snažím optimalizovat především výkon, zvolím k implementaci neseřazenou množinu std::tr1::unordered_map, která je rychlejší jak při výběru tak i při ukládání dat, protože nemusí procházet celý strom a zjištovat kde je daný prvek umístěný.
To se mně podařilo orientačním měřením ověřit
std::map je při vkládání i výběru dat zhruba o 10% pomalejší než std::tr1::unordered_map.

Rozdělení problémů

Cílem je ověřit, jestli je grafová struktura vhodná pro implementaci do BOOM.
Grafová struktura se nám ale dělí na dvě části a to:

a) uložení termů jako seznamů proměnných
b) vytvoření seznamů proměnných a termů, které je obsahují

Rozhodl jsem se zabývat především bodem a), což je uložení termů jako seznamů proměnných v přímé nebo negované formě. Stav DC bude dán najevo tím, že se nebude vyskytovat ani v jednom seznamu, jak jsem již vysvětlil výše.
Co se týká bodu b) tak pro něj vytvořím orientační výsledky, podle kterých se budeme moci rozhodnout, zda implementace této struktury má nebo nemá smysl, ale to vysvětlím později.

Přínos grafové struktury pro specifické operace nad termy

Jak později dokážu měřením, nejvíce náročné na čas jsou při řešení minimalizace dvě operace.
Tyto operace jsou:

a) test na průnik – intersects
b) test na obsaženost jednoho termu ve druhém – contains

Obě tyto operace procházejí jeden term a hledají společné proměnné v jiném termu.
Jak jsem již popsal v analýze původního návrhu, tyto operace lze udělat s časovou složitostí O(n). Současná implementace v BOOM je se stejnou složitostí, pokud za složitost bereme počet iterací, které se musí provést.
Ovšem současná implementace prochází všechny proměnné v obou termech a to i pokud tam daná proměnná není (je ve stavu DC).
Moje implementace projde jen ty proměnné, které v termu jsou obsaženy a pro ty, které jsou ve stavu DC se nebude dělat žádná operace. To by znamenalo, že pro řidké grafy (grafy, kde je větší množství DC v termech) bude tato implementace rychlejší.
Pojďme se nyní blíže podívat na tyto dvě operace

Intersects

Tato operace zjišťuje, zdali vůbec existuje průsečík mezi dvěma termy.
Dva termy mají průsečík právě tehdy, pokud všechny proměnné jednoho termu jsou ve stejném stavu ve druhém termu anebo jsou v jednom nebo druhém termu ve stavu DC.
Na tom jestli testujeme první se druhým anebo druhý s prvním termem nesejde, zde platí komutativní zákon.
Příklad průsečíku:

--10110--	0-10010--
10-01-0--	10-01-0--
Tyto dva termy se protínají	Tyto termy se neprotínají

V naší implementaci to znamená, že na každé proměnné stačí jen otestovat, jestli není ve druhém termu v jiné množině. Pokud je v jiné množině, pak průsečík neexistuje, pokud je ve stejné nebo není vůbec, tak průsečík existuje.
Zde je ještě prostor pro další optimalizaci. Vzhledem k tomu, že je jedno jestli testujeme první se druhým nebo druhý s prvním termem, můžeme si libovolně vybrat. My si z těch dvou daných termů vybereme ten, který má méně proměnných, abychom na druhém termu volali méně testů.
Zde je pro názornost implementace této operace v c++:

```
bool CubeGraph::intersects( const CubeGraph &t ) const {
      bool thisSmaller = (this->on.size() + this->off.size())
                              < (t.on.size() + t.off.size());
      const CubeGraph& first = thisSmaller ? *this : t;
      const CubeGraph& second = thisSmaller ? t : *this;

      for (Container::const_iterator it = first.on.begin();
            it != first.on.end(); ++it)
      {
            if (second.off.find(*it) != second.off.end())
                  return false;
      }

      for (Container::const_iterator it = first.off.begin();
            it != first.off.end(); ++it)
      {
            if (second.on.find(*it) != second.on.end())
                  return false;
      }

      return true;
}
```

Contains

Tato operace zjišťuje, zdali jeden term je obsažen ve druhém termu.
Jeden term obsahuje druhý právě tehdy, pokud všechny proměnné, které nejsou ve stavu DC jednoho termu, jsou ve stejném stavu ve druhém termu.
Na rozdíl od průsečíku, zde neplatí komutativní zákon, a tak to, jestli jeden term obasuje druhý neznamená, že druhý obsahuje také první.
Příklad obsaženosti:
1-00-
1000-
První term obsahuje druhý, ale druhý neobsahuje první

V naší implementaci to znamená, že na každé proměnné stačí jen otestovat, jestli je ve druhém termu ve stejné množině. Pokud není, tak daný term neobsahuje druhý term. Pokud všechny proměnné jsou ve stejných množinách ve druhém termu, tak term obsahuje druhý term.
Zde prostor pro optimalizaci ve smyslu prohození termů není, operace se musí dělat v daném pořadí.
Zde je pro názornost implementace této operace v c++:

```
bool CubeGraph::contains( const CubeGraph &t ) const {
      for (Container::const_iterator it = on.begin();
            it != on.end(); ++it)
      {
            if (t.on.find(*it) == t.on.end())
                  return false;
      }

      for (Container::const_iterator it = off.begin();
            it != off.end(); ++it)
      {
            if (t.off.find(*it) == t.off.end())
                  return false;
      }
      return true;
}
```

Druhá část rozdělení

Jak jsem již psal v bodě rozdělení problémů, grafová reprezentace se dělí na dvě části. Tou druhou je generování seznamu termů pro každou proměnnou. Tuto část jsem se rozhodl prozkoumat jen okrajově na té úrovni, zdali se má smysl zabývat implementací či nikoliv. Implementace této struktury je velmi složitá a z původní práce pana Vály není možné vycházet, protože grafy neimplementoval přímo do BOOM, ale vytvořil si na to celý svůj vlastní program, kde použit i svoje algoritmy pracující nad touto strukturou.
Nepoužil tedy standardní řešení BOOMu.

Jak měření prokazují, nejvíce času trvá programu metoda isImplikant.
Zkusme tedy generovat „falešnou“ strukturu pro všechny proměnné a termy, které je obsahují. Tato struktura se bude jen generovat, ale nikdo z ní nebude číst.
Tím zjistíme, jestli samotné generování nevezme tak velkou část výkonu, že i kdyby metoda isImlikant po té trvala 0 času, tak by již nebylo možné dosáhnout původního výkonu.

K tomuto problému se tedy v našich měřeních budeme jen zabývat tím, jestli generování této velké struktury nezabere většinu času.
Z Toho potom budeme moci udělat závěr, že se tato implementace pro BOOM nehodí a to i přes to, že původní práce pana Vály došla k závěru, že lze dosáhnout lepších výsledků. Jak jsem již uvedl, předchozí práce dosáhla lepších výsledků, ale ne v implementaci pro BOOM, všechny algoritmy tam byly napsány znovu jiným způsobem vhodným ke grafové reprezentaci.

Implementaci této části je možné najít v souboru cubegraph.cpp:

```
class FullGraph {
public:
    typedef std::tr1::unordered_set<CubeGraph*> VarTerms;
```

```
    struct Sets {
        VarTerms one;
        VarTerms zero;
        VarTerms dc;
    };
    // all terms with given variable
    typedef vector<Sets> Variables;

    void setVariableCount(int count) {
        vars.resize(count);
    }

    void setTerm(int variable, CubeGraph* cube) {
        removeTerm(variable, cube);
        switch (cube->getLit(variable)) {
            case '1': vars[variable].one.insert(cube); break;
            case '0': vars[variable].zero.insert(cube); break;
            case '-': vars[variable].dc.insert(cube); break;
        };
    }
    void removeTerm(int variable, CubeGraph* cube) {
        vars[variable].one.erase(cube);
        vars[variable].zero.erase(cube);
        vars[variable].dc.erase(cube);
    }
    void setAllVars(CubeGraph* cube) {
        for (int i = 0; i<cube->size(); i++)
            switch (cube->getLit(i)) {
                case '1': vars[i].one.insert(cube); break;
                case '0': vars[i].zero.insert(cube); break;
                case '-': vars[i].dc.insert(cube); break;
            };
    }
    void removeAllVars(CubeGraph* cube) {
        for (int i = 0; i<vars.size(); i++) {
            vars[i].one.erase(cube);
            vars[i].zero.erase(cube);
            vars[i].dc.erase(cube);
        }
    }

protected:
    Variables vars;

} g_fullGraph;
```

Na zdrojovém kódu je vidět, že jsou vytvořené jen metody pro ukládání, a pro čtení těchto dat žádná metoda není. Stejně tak proměnná Variable vars je protected a tak k ní nikdo nemá přístup. Navíc je celá tato struktura implementována v .cpp souboru a tak k ní jiný .cpp soubor nemá přístup. Je to prostě jen pro zápis.
Všechny tyto metody jsou volány pomocí maker:

```
    #define FULL_SET_COUNT(count) g_fullGraph.setVariableCount(count)
    #define FULL_SET_TERM(variable, cube) \
               g_fullGraph.setTerm(variable, cube)
    #define FULL_REMOVE_TERM(variable, cube) \
               g_fullGraph.removeTerm(variable, cube)
    #define FULL_SET_ALL(cube) g_fullGraph.setAllVars(cube)
    #define FULL_REMOVE_ALL(cube) g_fullGraph.removeAllVars(cube)
```

Takže lze jednoduše tuto funkčnost zakázat a nechat tak jen graf s první částí implementace.

Program s makry zachází tak, že při každé změně nějaké proměnné v termu ihned mění hodnoty i ve struktuře. Jakákoliv změna v proměnných se tedy ihned promítne i do struktury. Mezi tyto operace patří především kopírovací konstruktor, inicializace, setLit, destruktor.

Jak zjistit, jestli by se vyplatila kompletní implementace grafů

Jako rezervu bereme funkci isImplicant, která trvá většinu doby běhu programu a u ní můžeme předpokládat, že se doba zpracování zmenší po úplné implementaci.
Nejprve spustíme program BOOM s implementací bez kompletních struktur boomgraph.exe.
Naměříme jak dlouho trvá celý program a kolik z toho trvá isImplicant.
Po té provedeme stejné měření ovšem i s implementací všech struktur boomfull.exe.
Rozdílem doby trvání celého programu zjistíme, kolik vlastně trvalo samotné generování struktur.
Od main odečteme dobu isImplicant a přičteme dobu generování struktur.
Toto číslo podělíme původním main a zjískáme tí tak % rezervy, která nám dává prostor k využití kompletních struktur namísto volání funkce isImplicant.

Generace = MainFull – MainGraph
NoIsImplicant = MainGraph – IsImplicantGraph + Generace
Rezerva = NoIsImplicant / MainGraph * 100 [%]

Realizace

Techniky ke zjištění, které metody a proměnné se využívají

Jedním z hlavních úkolů bylo implementovat třídu Cube pomocí grafů.
Vznikla tak nová třída CubeGraph, původní třída Cube byla přejmenována na CubeStd a vznikla další nová třída Cube, která je odvozeninou CubeStd anebo CubeGraph podle nastavení.
Jeden z problémů byl, že na několika místech v programu BOOM se přistupovalo přímo na proměnné třídy Cube. Tyto proměnné ale z důvodu odlišné implementace v nové CubeGraph již nebyly obsaženy, takže se musely z celého BOOMu zrušit.
Požadavek ale byl takový, aby šlo jednoduše přepnout mezi původní implementací a novou grafovou implementací.
Proto jsem zvolil celkem běžný postup a to, že všechny interní proměnné třídy Cube jsem z oprávnění public změnil na protected.
Překladač při překladu zobrazil všechny místa, na kterých byly tyto proměnné použity a tak bylo možné jednoduše změnit v původní implementaci, aby tyto proměnné nepoužívala přímo, ale zprostředkovaně pomocí metod.

Dalším problémem bylo, že původní implementace využívala duplicitní informace (bitové pole a pole znaků) a tak podle toho, jak to bylo vhodné využívala k volání jedním nebo druhým způsobem.
Metody se totiž nevolaly např. inersects(cube &c), ale intersects(string s), kde se jako s předávala vnitřní proměnná.
I tyto volání pomohl odstranit překladač.

Dalším příjemným zjištěním bylo, že program BOOM měl implementováno spoustu metod pro třídu Cube, které BOOM nikdy nevolá.
To také pomohl odhalit překladač. Postupně jsem zakomentoval některé metody a zjistil, že program lze přeloži a spustit.

Toto jsou jedny z velkých výhod typového jazyka jako je c++. V některých jiných netypových jazycích by toto bylo jen velmi těžké a na chyby by se přicházelo až za chodu programu.

Jako velmi výhodné se ukázalo i implementovat všechny metody v .cpp souboru a nenechávat nic v .h. Soubor Cube.h je includován prakticky do všech souborů a tak jakákoliv změna vyžadovala přeložit celý program. Když byly změny jen v .cpp, překládal se pokaždé jen tento jeden soubor.

Jak spouštět BOOM a kontrolovat jeho výsledky

BOOM je nástroj, který je z velké míry závislý na náhodě.
V některých případech se rozhoduje náhodně, a tak často dosahuje různých výsledků optimalizace. Jak již bylo v úvodu řečeno, logickou funkci lze optimalizovat různě a je nekonečně mnoho možností jejího zadání.
Tato funkčnost bránila srovnání jednotlivých výsledků, proto byla náhoda v BOOM pro účely měření výkonu dočasně vypnuta.
K tomu slouží jednoduché makro v souboru stdafx.h:

```
#define rand() 0
```

Přepínání mezi různými verzemi implementace

V současnosti jsou k dispozici 3 implementace

a) původní implementace
b) grafová implementace
c) grafová implementace s generováním seznamů pro proměnné

Původní implementaci je možné zprovoznit, pokud se v souboru cube.h přepne CubeBase na:

```
typedef CubeStd CubeBase;
```

Grafová implementace se pak zapíná pomocí stejné definice ovšem na:

```
typedef CubeGraph CubeBase;
```

Pokud se mají generovat i struktury pro proměnné, musíme v souboru CubeGraph.cpp nadefinovat makro:

```
#define FULLGRAPH
```

Měření výkonu

Dlouhou dobu se mě nedařilo naměřit očekávané výsledky.
Jednu dobu jsem dokonce zkoušel kombinaci implementace pomocí pole i grafu současně, což se na spousta operacích jevilo jako dobrá kombinace.
Současná implementace v BOOM také využívá dvě implementace a to pomocí pole znaků a současně bitového pole.
Po odladění jsem ale chtěl vygenerovat výsledný binární tvar, a tak jsem přepnul z debug do release verze.
Výsledek byl téměř neuvěřitelný. V debug verzi bylo zrychlení okolo 25%, ale v release to je pro některá data i 50%.
Rozhodl jsem se tedy odstranit duplicitní pole z původní implementace a zjistil jsem, že se vše ještě urychlilo. Měřením jsem zjistil, že vůbec nemá smysl duplicity uchovávat.

V současné době je tedy implementace pomocí grafu čistě grafová bez původních objektů.
Tyto informace jsem uvedl proto, pokud budete využívat a zkoušet mojí implementaci, aby jste ji provozovali v režimu release.

Pomocné nástroje

Jeden z problémů, jak vůbec dosáhnou nějakých výsledků, je měření doby trvání jednotlivých operací. Ze začátku jsem se potýkal s tím, že ve windows není nějaký vhodný nástroj, jak změřit dobu trvání elementárních operací, jako je například uložení proměnné do množiny (set). Tato a spoustu dalších operaci, je totiž na vykonání času velmi krátká a téměř žádný nástroj nedokázal čas nad těmito operacemi změřit.
Nakonec jsem našel funkci `QueryPerformanceCounter`. Tato funkce vrací counter, který je s přesností na jeden tik procesoru.
Měření výkonu pomocí tiků dnes již není příliš přesné. Přepínání kontextů, swapování paměti a další způsobují to, že skučený čas je odlišný od naměřeného.
Přesto tato metoda dává v průměru jen asi 5% chybu, takže je ji možné použít pro orientační měření.
Pro měření funkcí, které trvají delší dobu, by bylo možné použít jiné přesnější nástroje, ale vzhledem k tomu, že už jsem měl tento nástroj hotový, rozhodl jsem se ho používat pro měření všech výsledků této práce
Podívejme se nyní na objekt, který poměrně jednoduše měří dobu trvání

```
#define MESURE TimeCntr mesureObj( __FILE__ " - " __FUNCTION__ );

class Delays {
public:
	~Delays();
	void add(const std::string & name, ULONG64 count);
protected:
	map<string, ULONG64> delays;
};

class TimeCntr {
public:
	TimeCntr(const char* name);
	~TimeCntr();
protected:
	LARGE_INTEGER cnt;
	string name;

};


Delays g_delays;

Delays::~Delays() {
	for (map<string, ULONG64>::const_iterator it = delays.begin();
		it != delays.end(); ++it)
			cout << it->first << " " << (int)it->second << "\n";
}
void Delays::add(const std::string & name, ULONG64 count) {
	if (delays.find(name) == delays.end())
		delays[name] = count;
	else
		delays[name] = delays[name] + count;
}

TimeCntr::TimeCntr(const char* name) {
	this->name = name;
	QueryPerformanceCounter(&cnt);
}
TimeCntr::~TimeCntr() {
	LARGE_INTEGER now;
	QueryPerformanceCounter(&now);

	g_delays.add(name, now.QuadPart - cnt.QuadPart);
```

Technika měření doby trvání využívá tzv. scoped object.
Princip je jednoduchý.
Vytvoří se objekt, který si v konstruktoru zapamatuje čas a v destruktoru ho odečte od aktuálního času. To způsobí, že známe dobu existence tohoto objektu.
V destruktoru dále uložíme nebo sečteme tento čas do nějakého globálního objektu.
Pokud tento objekt vložíme na začátek měřené metody, tak nám změří, jak dlouho tato metoda byla vykonávána. Tento objekt se jmenuje `TimeCntr`
Globální objekt nám tak drží všechny časy. Na konci programu všechny časy vypíše v destruktoru globálního objektu. Tento objekt se jmenuje `Delays` a je ho globální proměnná je `g_delays`.

Testování

Testování probíhalo za pomocí několika nástrojů:

Jeden z nástrojů je program pro generování náhodného zadání funkce.
K tomu slouží program genpla.exe, který přikládám i s návodem na doprovodném CD.
Tento program dokáže podle vstupu vygenerovat různé .pla soubory.
Já jsem se zaměřil na 5 parametrů, které jsem měnil a při tom měřil výkon programu.
Tyto parametry jsou:

a) počet vstupních proměnných
b) počet výstupů
c) počet termů
d) % DC ve vstupních proměnných
e) % 1 a 0 ve výstupech

Jak jsem již popsal k testování se používal objekt, který dokáže měřit čas běhu funkce.
Tyto objekty jsem umístil do metod, které trvají větší část doby běhu celého programu BOOM. Výsledky se vždy vytisknou na konci běhu programu.

Jako poslední jsou dva .bat skripty, které cyklickým voláním genpla a sbírání informací z měřicích objektů generovali tabulky závislostí. Tyto skripty jsou mesure.bat a mesureall.bat:

mesure.bat

```
echo %1 %2 %3 %4 %5 >>mesure.txt
\david\genpla -i %1 -o %2 -t %3 -idc %4 -odt %5 \david\in.pla
IF NOT ERRORLEVEL 0 EXIT 1
\david\boomstd \david\in.pla opt.pla >>mesure.txt
IF NOT ERRORLEVEL 0 EXIT 1
\david\boomgraph \david\in.pla opt.pla >>mesure.txt
IF NOT ERRORLEVEL 0 EXIT 1
\david\boomfull \david\in.pla opt.pla >>mesure.txt
IF NOT ERRORLEVEL 0 EXIT 1
```

mesureall.bat

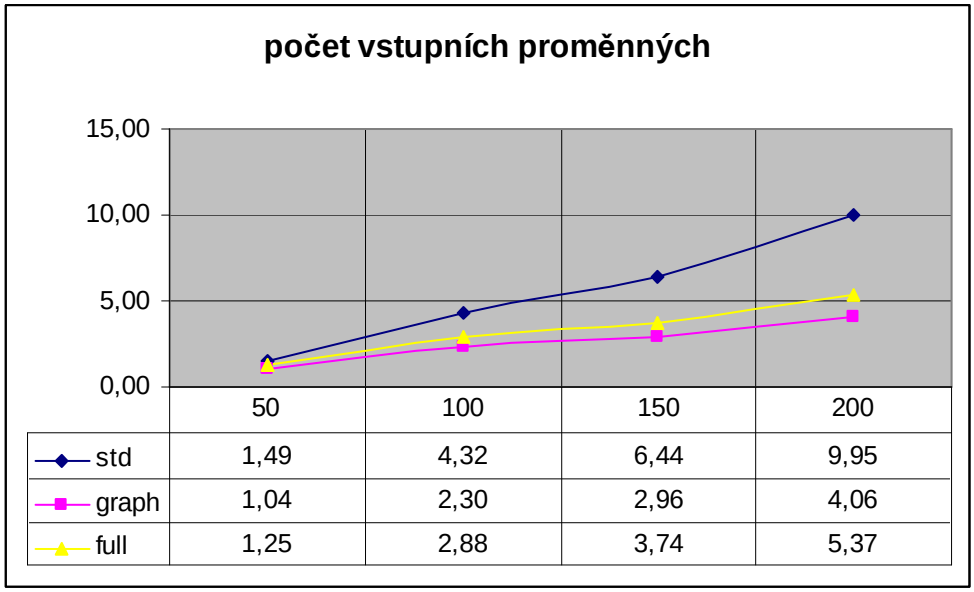
```
echo MESURE>mesure.txt
set VARS=50
set OUTS=5
set TERMS=25
set DC=50
set ONES=50
echo VARS
FOR /L %%I IN (50,50,200) DO call mesure %%I %OUTS% %TERMS% %DC% %ONES%
echo OUTS
FOR /L %%I IN (1,5,20) DO call mesure %VARS% %%I %TERMS% %DC% %ONES%
echo TERMS
FOR /L %%I IN (25,25,150) DO call mesure %VARS% %OUTS% %%I %DC% %ONES%
echo DC
FOR /L %%I IN (0,20,80) DO call mesure %VARS% %OUTS% %TERMS% %%I %ONES%
echo ONES
FOR /L %%I IN (10,20,90) DO call mesure %VARS% %OUTS% %TERMS% %DC% %%I
```

Takto nasbírané udaje se ještě za pomocí regulárních výrazů museli upravit, aby je bylo možné vložit do tabulkového kalkulátoru, kde se z nich vygenerovali grafy

Výsledky měření

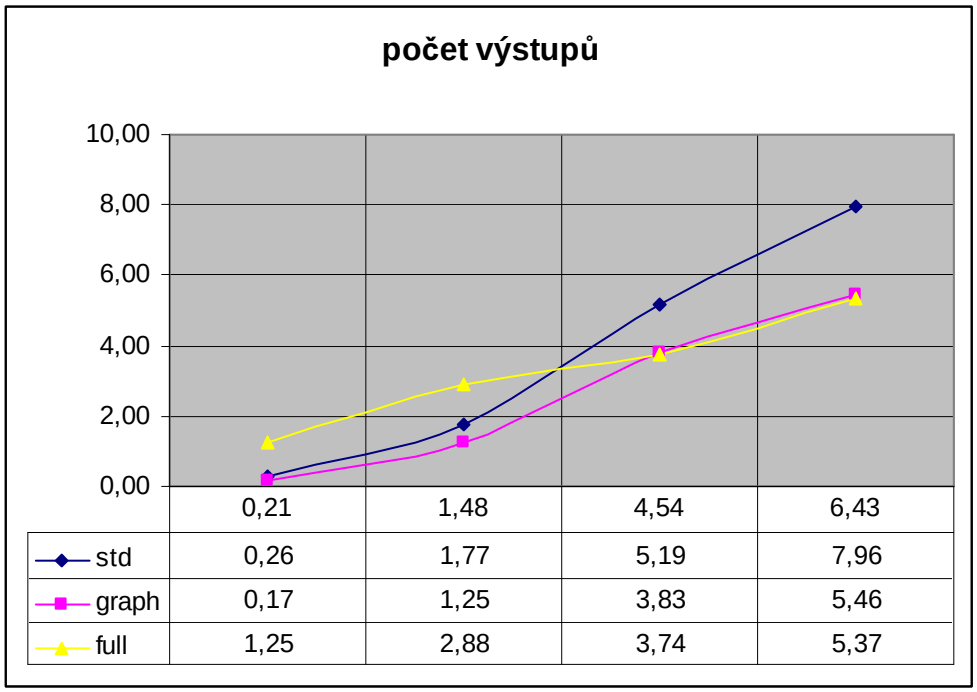
Výchozí vstupní parametry měření jsou vždy:
počet vstupních proměnných = 50
počet výstupů = 5
počet termů = 25
% DC ve vstupních proměnných = 50
% 1 a 0 ve výstupech = 50
Takto jsou nastaveny parametry, které daný graf nesleduje, tzn. každý graf je nějakou závislostí, a to co v grafu není, je vždy ve výchozí hodnotě.

1) závislost doby trvání běhu programu na počtu vstupních proměnných



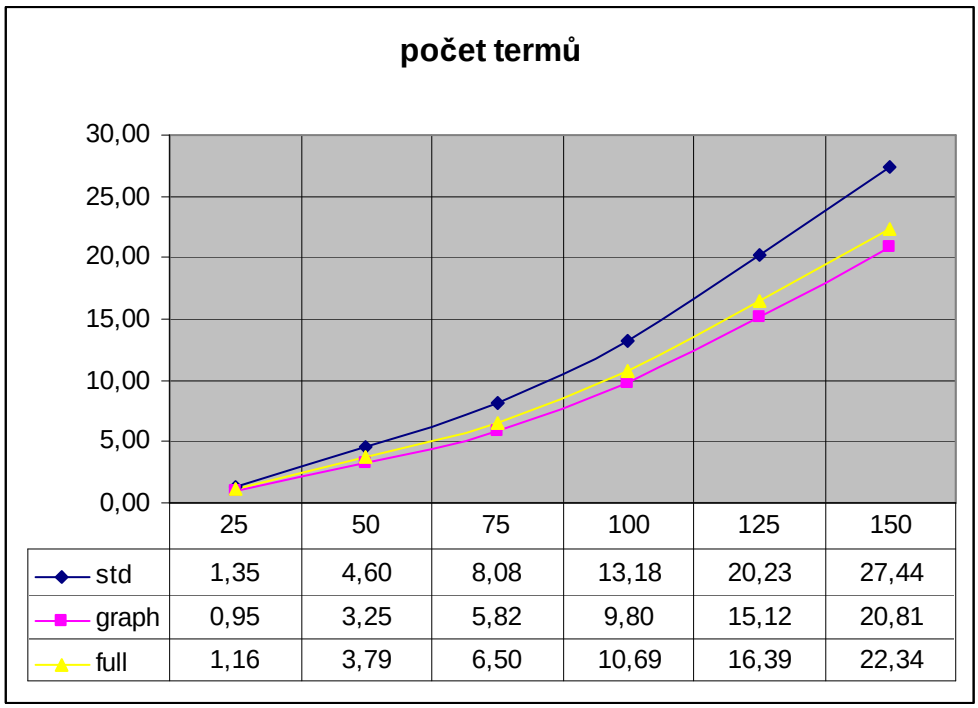
	50	100	150	200
std	1,49	4,32	6,44	9,95
graph	1,04	2,30	2,96	4,06
full	1,25	2,88	3,74	5,37

2) závislost doby trvání běhu programu na počtu výstupů



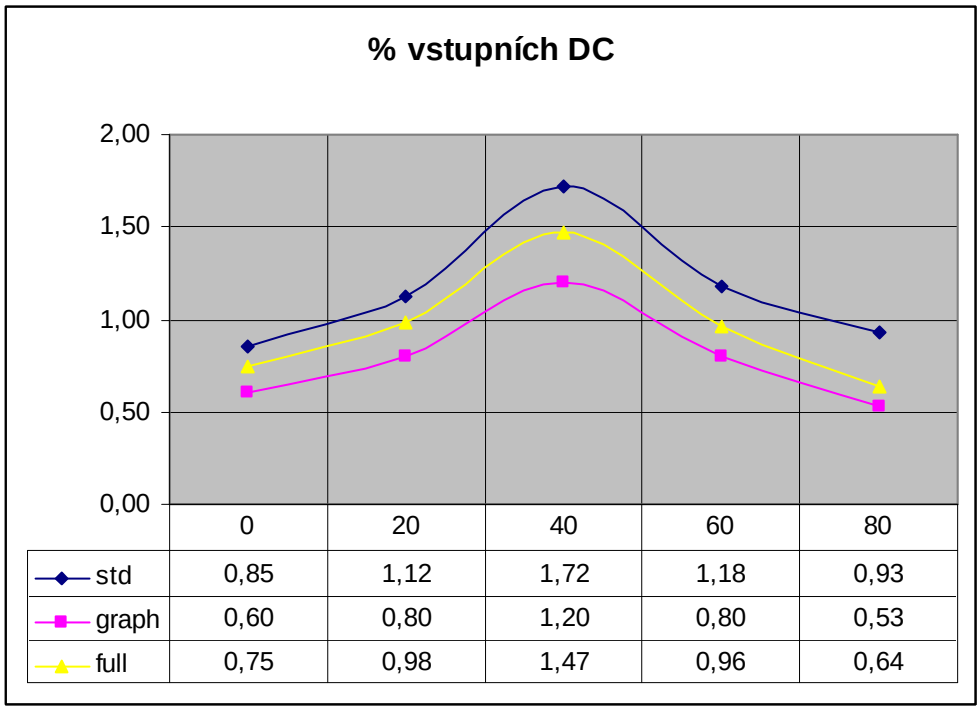
	0,21	1,48	4,54	6,43
std	0,26	1,77	5,19	7,96
graph	0,17	1,25	3,83	5,46
full	1,25	2,88	3,74	5,37

3) závislost doby trvání běhu programu na počtu termů



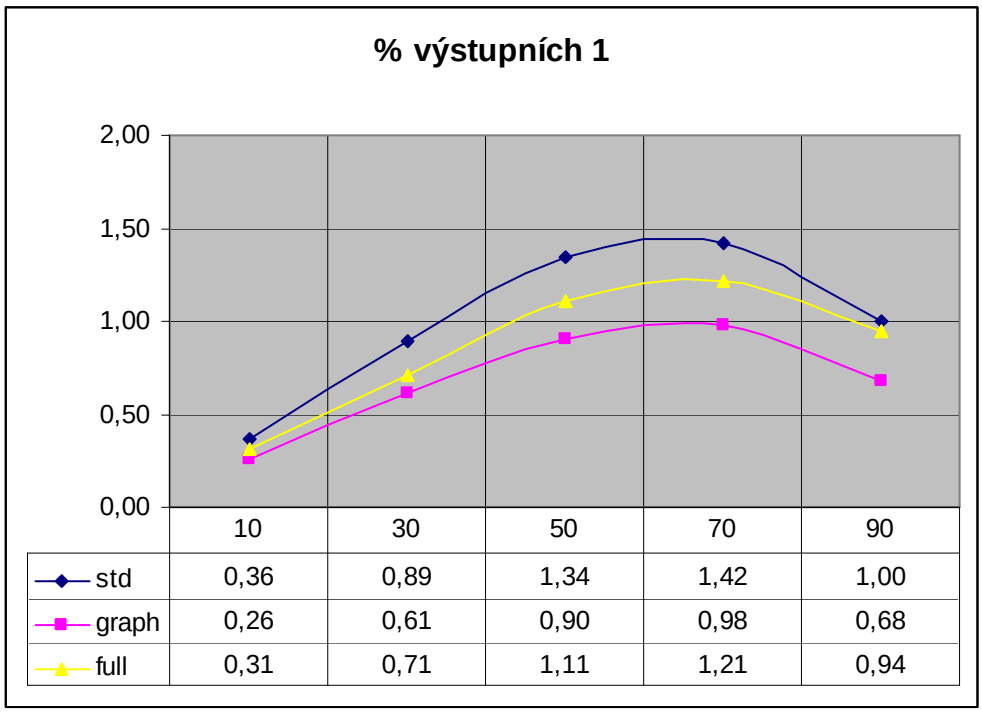
	25	50	75	100	125	150
std	1,35	4,60	8,08	13,18	20,23	27,44
graph	0,95	3,25	5,82	9,80	15,12	20,81
full	1,16	3,79	6,50	10,69	16,39	22,34

4) závislost doby trvání běhu programu na % vstupních DC



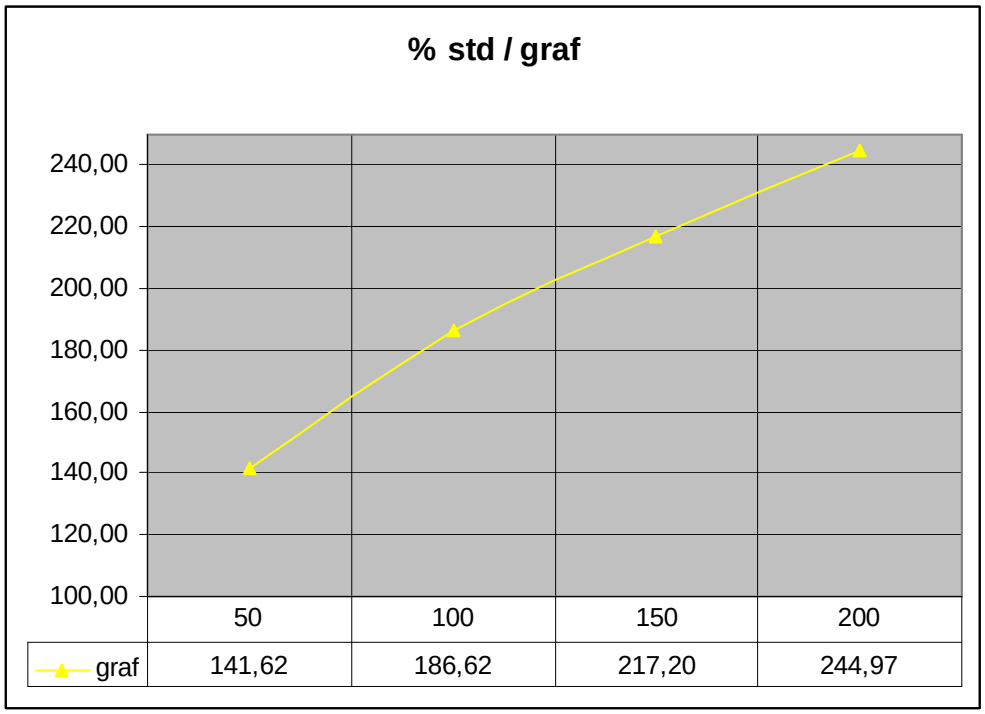
	0	20	40	60	80
std	0,85	1,12	1,72	1,18	0,93
graph	0,60	0,80	1,20	0,80	0,53
full	0,75	0,98	1,47	0,96	0,64

5) závislost doby trvání běhu programu na % výstupních 0 a 1



	10	30	50	70	90
std	0,36	0,89	1,34	1,42	1,00
graph	0,26	0,61	0,90	0,98	0,68
full	0,31	0,71	1,11	1,21	0,94

6) závislost doby trvání běhu programu na % běhu boomstd/boomgraph



Všechny grafy 1-5 potvrzují to, co bylo analyzováno a to, že grafová struktura (nebo alespoň její první část) nemá na asymptotickou složitost vůči složitosti předchozí implementace vliv. Všechny grafy mají stejný průběh, jediná změna je v měřítku, což indikuje to, že grafová struktura je jen úměrně rychlejší.

Velkým překvapením je graf 6. Na první pohled v něm není nic zajímavého, pojďme ho ale prozkoumat.
Na ose X je počet vstupních proměnných a na ose Y je poměr rychlosti standardního řešení vůči grafovému řešení. Z analýzy vyplývá, že grafy by měli být úměrně rychlejší, tedy poměr rychlosti původního řešení vůči grafovému by měl být konstantní. Z tohoto grafu je ale vidět, že konstantní není. Čím více je vstupních proměnných, tím je poměr starého vůči grafovému řešení pomalejší. Grafy tedy pro tento případ nejsou úměrně rychlejší, ale kvadraticky.

Význam kompletní implementace grafů

MainGraph	IsImplicant	MainFull	Generovani	NoIsImplikant	Rezerva [%]
3 783 399	2 540 597	4 606 088	822 689	2 065 491	54,5935282
8 305 655	5 853 894	10 455 387	2 149 732	4 601 493	55,40192796
10 659 956	8 560 233	13 518 198	2 858 242	4 957 965	46,51018259
14 628 124	12 083 787	19 504 693	4 876 569	7 420 906	50,73040125

Z měření vyplývá, že výkonová rezerva pro implementaci kompletních struktur grafové reprezentace je v průměru 50%. To dokládá jednak tato tabulka, ale také kompletní měření, které se nachází na CD jako bp.xls.
Z celého měření je vidět, že v podstatě na žádném parametru není rezerva závislá a je stále okolo 50%.
Můžeme tedy prohlásit, že implementace celé grafové struktury by se pravděpodobně vyplatila.

Závěr

Srovnání vzhledem k původnímu řešení

Bohužel není možné přímo porovnat výsledky této práce a předchozí práce pana Vály a to z několika důvodů:

a) implementace pana Vály je celá řešená novými algoritmy a tak není možné přímo porovnat tuto práci s mojí implementací do programu BOOM.
b) Implementace pana Vály dává dost odlišné výsledky minimalizace, v podstatě na složitějším zadání není možné docílit stejného výsledku a to až s takovým rozdílem, že např. výsledek pana Vály má 2x více termů než výsledek z BOOM
c) Zrychlení (zpomalení) mé implementace nemá zcela implementovanou druhou část problému (seznam termů pro proměnné) a tak to není možné porovnat s řešením, které tuto strukturu má implementovanou

Závěr k další implementaci

Je zapotřebí mít na paměti, že druhá část problému je implementována jen z části generování struktur, jejich použití zde již není a je velmi pravděpodobné, že pokud by se začali používat, zpomalilo by to program ještě o další část.
Naměřené hodnoty, které říkají, že je ve vykonávání ještě 50% rezerva napovídají, že by se vyplatilo zkusit implementovat celou grafovou strukturu tak, jak ji navrhl pan Vála.

Závěr k programu BOOM

Při implementaci do programu BOOM jsem si často všimnul několika nedokonalosti, které by se dali vylepšit a tím celý běh urychlit. Často to bylo způsobené tím, že původně byl BOOM navržen pro odlišné struktury.

Závěr k dané implementaci

V mojí implementaci, kde z grafů využívám jen seznamy proměnných pro termy, které jsem ještě vylepšil o použití množiny namísto zřetězeného seznamu se průměrné zrychlení pohybuje okolo 40%. Grafy jsem tedy nejen implementoval, ale ještě i optimalizoval.

Seznam obrázků

Seznam tabulek a grafů

Zdroje a literatura

[1] diplomová práce Ing. Petra Vály
[2] zdrojové kódy BOOM

Zdrojové kódy

```
class CubeGraph {
public:
      typedef std::tr1::unordered_set<int> Container;
private:
      Container one, zero;
      int litCount;

      unsigned int index;  // index of the term
      bool unwanted;          // this one is to be deleted by Purge();


      ...


};


//// MACROS
////////////
//#define FULLGRAPH

#ifdef FULLGRAPH
    #define FULL_SET_COUNT(count) g_fullGraph.setVariableCount(count)
    #define FULL_SET_TERM(variable, cube) \
       g_fullGraph.setTerm(variable, cube)
    #define FULL_REMOVE_TERM(variable, cube) \
       g_fullGraph.removeTerm(variable, cube)
    #define FULL_SET_ALL(cube) g_fullGraph.setAllVars(cube)
    #define FULL_REMOVE_ALL(cube) g_fullGraph.removeAllVars(cube)
#else
    #define FULL_SET_COUNT(count)
    #define FULL_SET_TERM(variable, cube)
    #define FULL_REMOVE_TERM(variable, cube)
    #define FULL_SET_ALL(cube)
    #define FULL_REMOVE_ALL(cube)
#endif


//// FULL GRAPH CLASS
////////////////////
class FullGraph {
public:
    typedef std::tr1::unordered_set<CubeGraph*> VarTerms;
    struct Sets {
        VarTerms one;
        VarTerms zero;
        VarTerms dc;
    };
    // all terms with given variable
    typedef vector<Sets> Variables;

    void setVariableCount(int count) {
        vars.resize(count);
    }
```

```
void setTerm(int variable, CubeGraph* cube) {
        removeTerm(variable, cube);
        switch (cube->getLit(variable)) {
            case '1': vars[variable].one.insert(cube); break;
            case '0': vars[variable].zero.insert(cube); break;
            case '-': vars[variable].dc.insert(cube); break;
        };
    }

    void removeTerm(int variable, CubeGraph* cube) {
        vars[variable].one.erase(cube);
        vars[variable].zero.erase(cube);
        vars[variable].dc.erase(cube);
    }

    void setAllVars(CubeGraph* cube) {
        for (int i = 0; i<cube->size(); i++)
            switch (cube->getLit(i)) {
                case '1': vars[i].one.insert(cube); break;
                case '0': vars[i].zero.insert(cube); break;
                case '-': vars[i].dc.insert(cube); break;
            };
    }
    void removeAllVars(CubeGraph* cube) {
        for (int i = 0; i<vars.size(); i++) {
            vars[i].one.erase(cube);
            vars[i].zero.erase(cube);
            vars[i].dc.erase(cube);
        }
    }

protected:
    Variables vars;

} g_fullGraph;


//// INITIALIZATION METHODS
/////////////////////////

CubeGraph::CubeGraph(void) {
    unwanted = false;
    litCount = 0;
}

CubeGraph::CubeGraph( const unsigned int iv ) {
    init(iv);
}

void CubeGraph::operator= ( const CubeGraph &src ) {
    FULL_REMOVE_ALL(this);
    one = src.one;
    zero = src.zero;
    litCount = src.litCount;
    index = src.index;
    unwanted = src.unwanted;
    SO_CoverVector = src.SO_CoverVector;
    CoverVector = src.CoverVector;
    tmp = src.tmp;
    FULL_SET_ALL(this);
}
```

```
CubeGraph::CubeGraph( const CubeGraph &src) {
    operator=(src);
}

CubeGraph::~CubeGraph(void)
{
    FULL_SET_ALL(this);
}

void CubeGraph::clear(void) {
    SO_CoverVector.clear();
    CoverVector.clear();
    one.clear();
    zero.clear();
}

void CubeGraph::init( const unsigned int iv ) {
    FULL_SET_COUNT(iv);
    FULL_REMOVE_ALL(this);
    one.clear();
    zero.clear();
    unwanted = false;
    SO_CoverVector.clear();
    CoverVector.clear();
    litCount = iv;
}


void CubeGraph::setCube( const string val ) {
    FULL_SET_COUNT(val.size());
    FULL_REMOVE_ALL(this);
     unsigned int i;

    if ( val.size() != this->size() && this->size() != 0 ) throw
            RuntimeException( "CubeGraph::setCube: Wrong IM size" );
    one.clear();
    zero.clear();
    litCount = val.size();
    for ( i = 0; i < val.size(); i++ )
        if (val[i] != '-')
            this->setLit( i, val[i] );
}

void CubeGraph::setCube( const vector<Literal> val ) {
    FULL_SET_COUNT(val.size());
    FULL_REMOVE_ALL(this);
     unsigned int i;

    if ( val.size() != this->size() && this->size() != 0 ) throw
            RuntimeException( "CubeGraph::setCube: Wrong IM size" );
    litCount = val.size();
    one.clear();
    zero.clear();
    for ( i = 0; i < val.size(); i++ )
        if (val[i] != DC)
            this->setLit( i, val[i] );
}
```

```
void CubeGraph::assign( Literal val ) {
    FULL_REMOVE_ALL(this);
     unsigned int n;

    one.clear();
    zero.clear();

    n = litCount;
    switch ( val ) {
        case ONE: for (int i = 0; i<litCount ; i++) one.insert(i);
                  FULL_SET_ALL(this);
                  break;
        case ZERO:for (int i = 0; i<litCount ; i++) zero.insert(i);
                  FULL_SET_ALL(this);
                  break;
        case DC:  break;
        default: throw RuntimeException(
             "CubeGraph::assign: incorrect literal value" );
    }
}

void CubeGraph::assign( unsigned int size, Literal val ) {
    litCount = size;
    this->assign(val);
}

void CubeGraph::CopyFrom(const CubeGraph* src) {
    FULL_REMOVE_ALL(this);
    one = src->one;
    zero = src->zero;
    litCount = src->litCount;
    FULL_SET_ALL(this);
}



//// GET/GET METHODS
////////////////////

string CubeGraph::getCube(void) const {
    string s(size(), '-');
    for (Container::const_iterator it = one.begin();
        it != one.end(); ++it)
            s[*it] = '1';
    for (Container::const_iterator it = zero.begin();
        it != zero.end(); ++it)
            s[*it] = '0';
    return s;
}

vector<Literal> CubeGraph::getCube2(void) const {
    vector<Literal> res;
    unsigned int i;

    for ( i = 0; i < size(); i++ )
        res.push_back( getLit2(i) );
    return res;
}
```

```
CubeGraph CubeGraph::getCube3() const {
    CubeGraph c;
    c.CopyFrom(this);
    return c;
}

bool CubeGraph::isEmpty() const {
    return one.empty() && zero.empty();
}

unsigned int CubeGraph::size(void) const { return litCount; }
unsigned int CubeGraph::inputs(void) const { return litCount; }

void CubeGraph::setLit_0( const unsigned int pos ) {
    if ( pos >= this->size() ) throw RuntimeException(
            "CubeGraph::setLit_0: Index out of range" );
    FULL_SET_TERM(pos, this);
    one.erase(pos);
    zero.insert(pos);
}

void CubeGraph::setLit_1( const unsigned int pos ) {
    if ( pos >= this->size() ) throw RuntimeException(
            "CubeGraph::setLit_1: Index out of range" );
    FULL_SET_TERM(pos, this);
    one.insert(pos);
    zero.erase(pos);
}

void CubeGraph::setLit_dc( const unsigned int pos ) {
    if ( pos >= this->size() ) throw RuntimeException(
            "CubeGraph::setLit_dc: Index out of range" );
    FULL_SET_TERM(pos, this);
    one.erase(pos);
    zero.erase(pos);
}

bool CubeGraph::isLit_0( const unsigned int pos ) const {
    return zero.find(pos) != zero.end();
}

bool CubeGraph::isLit_1( const unsigned int pos ) const {
    return one.find(pos) != one.end();
}

bool CubeGraph::isLit_dc( const unsigned int pos ) const {
    return (!isLit_0(pos)) && (!isLit_1(pos));
}
```

```
void CubeGraph::setLit( const unsigned int pos, const Literal val  ) {
     if ( pos >= this->size() ) throw RuntimeException(
             "CubeGraph::setLit: Index out of range" );
    switch ( val ) {
        case ZERO:
            setLit_0(pos);
            break;
        case ONE:
            setLit_1(pos);
            break;
        case DC:
            setLit_dc(pos);
            break;
        default: throw RuntimeException(
             "CubeGraph::setLit: Wrong IM value" );
            break;
    }
}

void CubeGraph::setLit( const unsigned int pos, const char val ) {
     if ( pos >= this->size() ) throw RuntimeException(
             "CubeGraph::setLit: Index out of range" );
    switch ( val ) {
        case '0':
            this->setLit_0(pos);
            break;
        case '1':
            this->setLit_1(pos);
            break;
        case '-':
            this->setLit_dc(pos);
            break;
        default:
            throw RuntimeException( "CubeGraph::setLit: Invalid value" );
    }
}


char CubeGraph::getLit( const unsigned int pos ) const {
    if (isLit_0(pos))
        return '0';
    else if (isLit_1(pos))
        return '1';
    else return '-';
};

Literal CubeGraph::getLit2( const unsigned int pos ) const {
    if ( this->isLit_0(pos) ) return ZERO;
    else if ( this->isLit_1(pos) ) return ONE;
    else return DC;
}

unsigned int CubeGraph::getDim(void){
     return litCount - one.size() - zero.size();
}
```

```
//// CUBE OPERATIONS
//////////////////

bool CubeGraph::intersects( const CubeGraph &t ) const {
MESURE
    unsigned int i;

    if ( t.size() != this->size() ) throw RuntimeException(
            "CubeGraph::intersects: Comparing incompatible terms");

    bool thisSmaller = (this->one.size() + this->zero.size()) <
                       (t.one.size() + t.zero.size());
    const CubeGraph& first = thisSmaller ? *this : t;
    const CubeGraph& second = thisSmaller ? t : *this;

    for (Container::const_iterator it = first.one.begin();
        it != first.one.end(); ++it)
    {
        if (second.zero.find(*it) != second.zero.end())
            return false;
    }

    for (Container::const_iterator it = first.zero.begin();
        it != first.zero.end(); ++it)
    {
        if (second.one.find(*it) != second.one.end())
            return false;
    }

    return true;
}

int CubeGraph::intersectSize( const CubeGraph &t ) const
{
    throw RuntimeException(
      "this method is not tested, may return invalid result");

    int ret = 0;

    for (Container::const_iterator it = one.begin(); it != one.end(); ++it)
        if (t.isLit_dc(*it))
            ret++;
    for (Container::const_iterator it = zero.begin(); it != zero.end();
            ++it)
        if (t.isLit_dc(*it))
            ret++;
    for (Container::const_iterator it = t.one.begin(); it != t.one.end();
            ++it)
        if (this->isLit_dc(*it))
            ret++;
    for (Container::const_iterator it = t.zero.begin(); it != t.zero.end();
            ++it)
        if (this->isLit_dc(*it))
            ret++;

    return ret;
}
```

```
bool CubeGraph::contains( const CubeGraph &t ) {
MESURE
    unsigned int i;

    if ( t.size() != this->size() )
        throw RuntimeException(
            "CubeGraph::contains: Comparing incompatible terms");

    for (Container::const_iterator it = one.begin();
        it != one.end(); ++it)
    {
        if (t.one.find(*it) == t.one.end())
            return false;
    }

    for (Container::const_iterator it = zero.begin();
        it != zero.end(); ++it)
    {
        if (t.zero.find(*it) == t.zero.end())
            return false;
    }
    return true;
}

bool CubeGraph::operator== ( const string &s ) const {
    return getCube() == s;
};


bool CubeGraph::operator== ( const CubeGraph &t ) const {
    if (litCount != t.litCount || one.size() != t.one.size() ||
        zero.size() != t.zero.size())
            return false;

    for (Container::const_iterator it = one.begin();
        it != one.end(); ++it)
            if (t.one.find(*it) == t.one.end())
                return false;
    for (Container::const_iterator it = zero.begin();
        it != zero.end(); ++it)
            if (t.zero.find(*it) == t.zero.end())
                return false;
    return true;
}
```

Obsah přiloženého CD

- Kompletní zdrojové kódy programu BOOM včetně mojí implementace CubeGraph
- Naměřené výsledky v .txt formátu
- Všechny tabulky ve formátu .xls
- Text této práce ve formátu .pdf i .doc
- Přeložený program BOOM ve třech variantách – boomstd, boomgraph, boomfull pro možnost porovnání mezi sebou
- Aplikace genpla.exe pro generování .pla souborů a její dokumentace
- Instalaci Visual Studia Express, na kterém jsem vyvíjel
- Skripty mesure.bat a mesureall.bat
- Diplomovou práci pana Vály včetně zdrojových textu
- Dokumentaci k formátu .pla