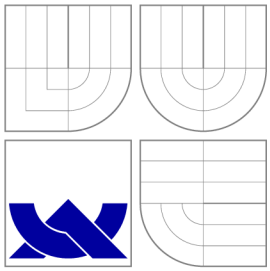
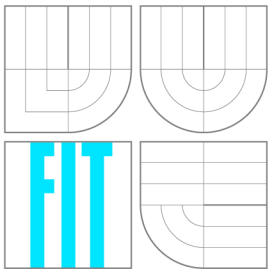


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ZPRACOVÁNÍ GRAFU VOLÁNÍ ZALOŽENÉ NA DOTAZOVACÍM JAZYKU

CALL GRAPH PROCESSING BASED ON QUERY LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. KAMIL DUDKA
AUTHOR

VEDOUCÍ PRÁCE — Dr. Ing. PETR PERINGER
SUPERVISOR

BRNO 2009

Zpracování grafu volání založené na dotazovacím jazyku

Zadání diplomové práce

1. Prostudujte současné grafové dotazovací jazyky a nástroje na jejich zpracování. Analyzujte dostupné implementace těchto nástrojů.
2. Navrhněte nástroj pro zpracování grafu volání založený na prostudovaném dotazovacím jazyku (nebo jeho podmnožině).
3. Implementujte navržený nástroj pro zpracování grafu volání (graf znázorňující vztahy mezi funkcemi/proměnnými).
4. Ověřte funkčnost nástroje na grafech volání vygenerovaných ze skutečných programů a zhodnoťte přínos.

Abstrakt

V této práci jsou analyzovány dostupné nástroje pro získávání grafů volání z programů a jejich následné zpracování a vizualizaci. Na základě získaných poznatků je potom navržen nástroj, který s grafy volání pracuje. Tento nástroj je následně implementován a testován na grafech volání vygenerovaných z různých programů včetně linuxového jádra.

Klíčová slova

graf volání, dotazovací jazyk, teorie grafů, bgl, graphviz, cgt

Abstract

In this thesis, available tools for call graph generation, processing and visualization are analyzed. Based on this analysis, a call-graph processing tool is designed. The tool is then implemented and tested on call graphs generated from various real-world programs, including the Linux kernel.

Keywords

call graph, query language, graph theory, bgl, graphviz, cgt

Citace

Kamil Dudka: Zpracování grafu volání založené na dotazovacím jazyku, diplomová práce, Brno, FIT VUT v Brně, 2009

Zpracování grafu volání založené na dotazovacím jazyku

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Dr. Ing. Petra Peringera a technickým dozorem Ing. Petra Machaty. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Kamil Dudka
25. května 2009

Poděkování

Děkuji svému vedoucímu Dr. Ing. Petru Peringerovi za pedagogické vedení této práce a konzultantovi firmy Red Hat, Ing. Petru Machatovi, který na vývoji nástroje pracoval jako první a následně mi pomohl na jeho práci navázat.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá návrhem nástroje na zapracování grafu volání[1]. *Graf volání* [12] je orientovaný graf, který reprezentuje vztahy volání mezi podprogramy (funkcemi, metodami) počítačového programu. Grafy volání lze rozdělit na statické a dynamické. *Statický* graf volání je generovaný na základě analýzy programu, který neběží. *Dynamický* graf volání je naopak výsledkem běhu programu. Tato práce se zabývá výhradně statickými grafy volání.

Grafy volání představují jednu z možností, jak jednoduše nahlédnout do struktury složitých programů. Jediným pohledem na vhodně zvolený graf volání získáme zajímavé informace bez toho, abychom museli studovat zdrojový kód programu. Grafy volání lze snadno vizualizovat a pro vývojáře jsou tak mnohem stravitelnější než čtení rozsáhlé dokumentace. Dnes vyvíjené programy jsou však příliš složité na to, aby jejich grafy volání mohly být vizualizovány jako celek. Je tedy potřeba nalézt nějaký způsob, jak z grafu dostat rychle a jednoduše informace, které nás zajímají.

Cílem této práce je návrh a implementace nástroje, který umožňuje s grafy volání rychle a efektivně pracovat. Pomocí vytvořeného nástroje je možné nad grafem volání provádět různé dotazovací operace a tím potřebné informace získat. Nástroj je možné použít buď v interaktivním režimu, nebo v dávkovém režimu. V interaktivním režimu může uživatel zadávat dotazy a okamžitě sledovat výsledky. Dávkový režim je určen pro použití nástroje uvnitř jiných nástrojů pro analýzu programů. V tomto režimu se nástroj chová jako tzv. *filtr* – tedy jako většina nástrojů, na které jsou uživatelé unixových systémů zvyklí.

V kapitole 2 jsou zavedeny základní pojmy z teorie grafů, se kterými tato práce pracuje. Souhrn poznatků o existujících nástrojích pro práci s grafy volání a s grafy obecně je v kapitole 3. Nejrozsáhlejší částí této práce je kapitola 4, která popisuje jednotlivé kroky návrhu dotazovacího nástroje a jeho komponent. V kapitole 5 jsou velmi stručně popsány implementační detaily. A konečně kapitola 6 prezentuje výsledky, kterých bylo dosaženo vytvořeným dotazovacím nástrojem.

[1] Práce vznikla na základě externího zadání firmy Red Hat Czech, s.r.o.

Kapitola 2

Teorie grafů

Teorie grafů představuje jeden z možných přístupů ke zpracování grafu volání. V této kapitole jsou zavedeny základní pojmy z teorie grafů, které jsou v této práci dále používány. Následující informace jsou výtahem z [4] (kapitola Základní pojmy). Všechny definice jsou citovány doslovně. Výjimečně jsou doplněny definice z jiné literatury, které jsou upraveny tak, aby používaly stejnou notaci jako [4].

2.1 Základní pojmy

V matematice se pojmu graf používá nejčastěji ve smyslu grafického znázornění nějaké funkce [9]. Kreslení grafu funkcí je důležitou pomůckou při zkoumání funkcí. Tyto grafy je však potřeba odlišit od grafů popisovaných dále v této sekci, neboť oba pojmy vedou ke zcela odlišným matematickým oblastem.

Orientovaný graf – definice převzatá z [4]

Orientovaný graf je trojice $G = (V, E, \varepsilon)$ tvořená neprázdnou konečnou množinou V, jejíž prvky nazýváme *vrcholy*, konečnou množinou E, jejíž prvky nazýváme *orientovanými hranami*, a zobrazením $\varepsilon : E \to V^2$, které nazýváme *vztahem incidence*. Toto zobrazení přiřazuje každé hraně $e \in E$ uspořádanou dvojici vrcholů (x, y). Prvý z nich, x, nazýváme *počátečním vrcholem hrany* a značíme jej $Pv(e)$. Druhý nazýváme *koncovým vrcholem hrany* a značíme jej $Kv(e)$.

O hraně e říkáme, že *vede z vrcholu x do vrcholu y* a také, že *spojuje* vrcholy x a y. O vrcholech x, y pak říkáme, že jsou *incidentní* (nebo že *incidují*) s hranou e a také naopak hrana e je *incidentní* s vrcholy x, y. Oba vrcholy x, y také souhrnně nazýváme *krajními vrcholy hrany* e.

Jestliže $Pv(e) = Kv(e)$, pak hranu e nazýváme (orientovanou) *smyčkou*. Vrchol, který není incidentní s žádnou hranou, nazýváme *izolovaným vrcholem*.

Je možné, aby několik hran mělo stejné počáteční a koncové vrcholy, tj. aby pro různé hrany e_1, e_2 platilo $Pv(e_1) = Pv(e_2)$ a $Kv(e_1) = Kv(e_2)$ nebo, zapsáno jinak, $\varepsilon(e_1) = \varepsilon(e_2)$. O takových hranách říkáme, že jsou *rovnoběžné* nebo též *násobné*.

Množina hran grafu může být prázdná.

Množiny hran a vrcholů v orientovaných grafech – definice převzatá z [4]

Nechť $G = (V, E, \varepsilon)$ je orientovaný graf, nechť x a y jsou jeho libovolné vrcholy a $A \subseteq V$ nechť je libovolná podmnožina jeho vrcholů. Pak zavedeme následující pojmy a značení:

$V_G^+(x) = \{z \in V | (x, z) \in \varepsilon(E)\}$,
tj. množina *následníků vrcholu* x

$V_G^-(x) = \{z \in V | (z, x) \in \varepsilon(E)\}$,
tj. množina *předchůdců vrcholu* x

$V_G(x) = V_G^+(x) \cup V_G^-(x)$,
tj. množina *sousedů vrcholu* x

$V_G(A) = \bigcup_{x \in A} V_G(x)$,
tj. množina vrcholů spojených hranou s některým vrcholem z A

$E_G^+(x) = \{e \in E | Pv(e) = x\}$,
tj. *výstupní okolí vrcholu* x

$E_G^-(x) = \{e \in E | Kv(e) = x\}$,
tj. *vstupní okolí vrcholu* x

$E_G(x) = E_G^+(x) \cup E_G^-(x)$,
tj. *okolí vrcholu* x

$m_G^+(x, y) = |E_G^+(x) \cap E_G^-(y)|$,
tj. *násobnost hrany* (s ohledem na orientaci)

$m_G(x, y) = |E_G(x) \cap E_G(y)|$,
tj. *násobnost hrany* (bez ohledu na orientaci)

$d_G^+(x) = |E_G^+(x)|$,
tj. *výstupní stupeň vrcholu* x

$d_G^-(x) = |E_G^-(x)|$,
tj. *vstupní stupeň vrcholu* x

$d_G(x) = d_G^+(x) + d_G^-(x)$,
tj. *stupeň vrcholu* x

Vždy, když bude z kontextu zřejmé, jaký graf máme na mysli, budeme vynechávat index $_G$ a budeme psát stručněji $V(x)$ namísto $V_G(x)$ apod.

Prostý graf a multigraf – definice převzatá z [4]

Prostý graf je graf, v němž násobnost každé hrany je nejvýše rovna jedné. *Multigraf* je graf, v němž násobnosti hran mohou být i větší než jedna.

Prosté grafy a relace – definice převzatá z [4]

V prostém orientovaném můžeme každou hranu e ztotožnit s uspořádanou dvojicí vrcholů $(Pv(e), Kv(e))$, neboť touto dvojicí vrcholů je hrana e jednoznačně určena (v prostém grafu nemohou být dvě takové hrany). Množinu hran prostého orientovaného grafu tedy můžeme pokládat za binární relaci na množině vrcholů.

Prostý obyčejný graf – definice převzatá z [9]

Jestliže prostý orientovaný graf neobsahuje žádné orientované smyčky, potom se nazývá *obyčejný orientovaný graf*. Relace incidence obyčejného orientovaného grafu je *antireflexivní*.

Rovnost grafů – definice převzatá z [4]

Řekneme, že dva grafy $G_1 = (V_1, E_1, \varepsilon_1)$ a $G_2 = (V_2, E_2, \varepsilon_2)$, jestliže $V_1 = V_2$, $E_1 = E_2$ a $\varepsilon_1 = \varepsilon_2$.

Izomorfismus grafů – definice převzatá z [4]

Orientované grafy G, G' se nazývají vzájemně *izomorfní*, když existují dvě bijektivní zobrazení $f : V \to V'$ a $g : E \to E'$ taková, že zachovávají vztahy incidence ε a ε'. Přesněji, když pro každou hranu $e \in E$ platí:

$$\varepsilon(e) = (x, y) \Longleftrightarrow \varepsilon'(g(e)) = (f(x), f(y))$$

Vztah izomorfismu značíme $G \cong G'$.

Mnoho vlastností grafů se přenáší izomorfismem. Přesněji, mnoho vlastností ν je takových, že má-li graf G_1 vlastnost ν a platí-li $G_1 \cong G_2$, pak i graf G_2 má vlastnost ν. Teorie grafů se téměř výlučně zabývá právě takovýmito vlastnostmi.

Podgraf – definice převzatá z [4]

Graf G' je *podgrafem* grafu G, vznikne-li z grafu G vynecháním nějakých (nebo žádných) vrcholů a hran. Podstatné je, že podgraf musí být také grafem: spolu s každou hranou, která je v podgrafu, tam musí být i oba její krajní vrcholy. Poznamenejme, že každý graf pokládáme za podgraf sebe sama. Jsou dva speciální druhy podgrafů:

Graf G' nazýváme *faktorem* grafu G, vznikne-li z grafu G pouze vynecháním některých (nebo žádných) hran, tj. platí-li $V(G) = V(G')$.

Graf G' nazýváme *podgrafem indukovaným množinou vrcholů* $A \subseteq V(G)$ (též *úplným podgrafem na množině* A), jestliže podgraf G' má množinu vrcholů A a obsahuje všechny hrany grafu G, jejichž oba vrcholy leží v A. Indukovaný podgraf G' lze získat z grafu G tím, že vynecháme vrcholy, které neleží v množině A, a pak vynecháme všechny hrany, které byly incidentní s vynechanými vrcholy. (Tyto hrany je potřeba vynechat, aby to, co zbude, byl graf.)

Sled – definice převzatá z [4]

Posloupnost vrcholů a hran $v_0, e_1, v_1, e_2, v_2, ..., e_k, v_k$ nazýváme *orientovaným sledem*, jestliže pro každou hranu e_i z této posloupnosti platí $Pv(e_i) = V_{i-1}$ a $Kv(e_i) = v_i$.

Posloupnost vrcholů a hran $v_0, e_1, v_1, e_2, v_2, ..., e_k, v_k$ nazýváme *neorientovaným sledem*, jestliže každá hrana e_i z této posloupnosti spojuje vrcholy v_{i-1}, v_i.

Vrchol v_0 v obou případech nazýváme *počátečním* a vrchol v_k *koncovým vrcholem sledu*. O sledu říkáme, že *vede z vrcholu* v_0 *do vrcholu* v_k, nebo také, že *spojuje vrcholy* v_0, v_k.

V obecných sledech se mohou vrcholy i hrany opakovat. *Triviální sled* je sled, který obsahuje jediný vrchol a žádnou hranu. Triviální sled lze pokládat za orientovaný i neorientovaný.

Každý sled (kromě triviálního) je jednoznačně určen posloupností svých hran. V prostém grafu je sled určen i posloupností vrcholů.

Tah, cesta – definice převzatá z [4]

Orientovaný (neorientovaný) sled, v němž se žádná hrana neopakuje, nazýváme *orientovaným (neorientovaným) tahem*. Orientovaný (neorientovaný) sled, v němž se neopakuje žádný vrchol, nazýváme *orientovanou (neorientovanou) cestou*.

Z faktu, že se v cestě neopakují vrcholy, vyplývá, že se v ní neopakují ani hrany. Každá cesta je tedy zároveň tahem, zatímco tah není vždy cestou.

Uzavřené sledy – definice převzatá z [4]

Sled (orientovaný nebo neorientovaný), který má alespoň jednu hranu a jehož počáteční a koncový vrchol splývají, nazýváme *uzavřeným sledem*. Podobně mluvíme o *uzavřeném tahu*.

Uzavřená cesta je uzavřený sled, v němž se neopakují vrcholy (kromě toho, že $v_0 = v_k$) a navíc se neopakují ani hrany. Pro uzavřené cesty se používají speciální názvy: *kružnice* je neorientovaná uzavřená cesta a *cyklus* je orientovaná uzavřená cesta. Opět platí, že cyklus je zároveň i kružnicí, ale naopak to neplatí.

Kružnice, která má právě tři hrany, se nazývá *trojúhelník*.

Dostupnost vrcholu – definice převzatá z [4]

Řekneme, že vrchol y je *orientovaně (neorientovaně) dostupný z vrcholu* x, jestliže existuje orientovaný (neorientovaný) sled vedoucí z vrcholu x do vrcholu y. Sled spojující x a y existuje právě tehdy, když existuje cesta spojující tyto vrcholy.

2.2 Reprezentace grafu

Graf lze reprezentovat různými způsoby. Některé způsoby jsou lépe čitelné pro lidi (zpravidla různé formy vizualizace), jiné jsou zase vhodnější pro zpracování strojem (matice, seznamy). Vizualizací grafu se v této sekci zabývat nebudeme.

Matice sousednosti – definice převzatá z [4]

Nechť G je orientovaný graf. Zvolíme-li (libovolně, ale pevně) pořadí jeho vrcholů $v_1, ..., v_n$, můžeme grafu G přiřadit *matici sousednosti* M_G^+ řádu n předpisem:

$$m_{ij}^+ = m^+(v_i, v_j)$$

Matice incidence – definice převzatá z [4]

Nechť G je orientovaný graf bez smyček. Zvolíme-li (libovolně, ale pevně) nejen pořadí vrcholů $v_1, ..., v_n$, ale i pořadí hran $e_1, ...e_m$, můžeme grafu G přiřadit *matici incidence* B_G typu (m, n) předpisem

$$b_{ij} = \begin{cases} 1, & \text{jestliže } v_i \text{ je počátečním vrcholem hrany } e_j, \\ -1, & \text{jestliže } v_i \text{ je koncovým vrcholem hrany } e_j, \\ 0 & \text{v ostatních případech.} \end{cases}$$

Popis grafu pomocí matic je matematicky elegantní, ale pro praxi méně vhodný, zejména pro grafy s relativně málo hranami, neboť matice pak obsahuje značný počet nul [4].

Seznamy vrcholů a hran – definice převzatá z [4]

Množina vrcholů je popsána prostým výčtem (seznamem) prvků, množina hran je popsána seznamem uspořádaných trojic tvořených jménem hrany a jejím počátečním a koncovým vrcholem. V podstatě se jedná o úplný popis grafu podle definice. Pokud nám nezáleží na jménech hran, můžeme je vypustit a hrany popisovat pouze uspořádanými dvojicemi vrcholů.

K výhodám tohoto popisu grafu patří univerzálnost a relativní úspornost. Navíc lze tímto způsobem snadno popisovat i ohodnocené grafy: ohodnocení prostě připíšeme k hranám či vrcholům. Díky těmto výhodám je tento způsob v praxi velmi často používán, třebaže detaily provedení bývají někdy poněkud odlišné.

Seznamy vrcholů a seznamy okolí vrcholů – definice převzatá z [4]

Tento způsob je vlastně úspornější variantou předchozího způsobu. Množina vrcholů je opět popsána seznamem prvků, ale hrany jsou popisovány po skupinách: pro každý vrchol x je vždy uvedeno jeho výstupní okolí (množina $E^+(x)$). Každá hrana je pak popsána pouze svým jménem a koncovým vrcholem, neboť počáteční vrchol x je pro celou skupinu hran společný. Je samozřejmé, že bychom místo výstupního okolí mohli uvádět též vstupní okolí (množinu $E^-(x)$).

Kapitola 3

Analýza existujících řešení

Spolu se zadáním diplomové práce jsem dostal také hotový prototyp řešení, který se skládá ze dvou částí. První část má za úkol generování grafu volání ze zdrojových kódů. Druhá část provádí dotazování nad vygenerovaným grafem volání. Tento prototyp budu dále v textu označovat *cgt*. Následující podkapitoly shrnují poznatky o existujících řešeních podle jednotlivých kategorií.

3.1 Grafové dotazovací jazyky

Součástí cgt je také program napsaný v jazyce Python, který umožňuje dotazování nad grafem volání pomocí dotazovacího jazyka. Samotný dotazovací jazyk je také založen na jazyce Python (viz. kapitola 5.1). Kromě základních množinových operací (průnik, sjednocení, rozdíl) jsou k dispozici operace pro zjištění přímo/nepřímo volajících/volaných funkcí a operace pro hledání cesty mezi dvěma funkcemi. V kapitole 6.2 je dotazování předvedeno na příkladech. Detailní popis dotazovacího jazyka je k dispozici na přiloženém CD.

Graph Query Language

Jazyk *GOQL* (Graph Object Query Language) vychází z jazyka *OQL* (Object Query Language) [10]. Je navržen pro dotazy nad modelem objektově orientovaného modelu grafu.

Jazyk rozlišuje základní typy objektů (celá čísla, řetězce, ...) a strukturované typy objektů (typy složené z dříve definovaných typů). Skládat objekty je možné pomocí množiny, n-tice a posloupnosti. Předdefinovány jsou typy *uzel* (node; v terminologii [4] vrchol), *hrana* (edge), *cesta* (path) a *graf* (graph).

Pro dotazování jsou k dispozici například tyto operace:

- dotaz na všechny cesty z jednoho vrcholu do druhého
- dotaz na všechny podsledy nějakého sledu
- konkatenace cest
- dotaz na množinu hran/vrcholů nějaké cesty
- obvyklé množinové operace

Dotaz zapsaný v objektově orientovaném jazyku GOQL je před jeho zpracováním převeden do jazyka založeného na operátorech a zpracován pomocí tzv. *O-Algebry*.

3.2 Generování a zpracování grafu volání

Generování grafu volání v cgt zajišťuje zásuvný modul pro překladač gcc. Pro použití zásuvného modulu je nutné mít větev gcc s názvem *Dehydra*[1]. Zásuvný modul se stará o generování grafů volání během překladu a jejich ukládání do odpovídajících (textových) souborů. Při sestavování jsou pak kromě samotných modulů sestavovány také jejich grafy volání. Pro vygenerování grafu volání celého projektu tedy stačí použít upravený překladač a sestavovací program a sestavit projekt obvyklým způsobem. Dále v této kapitole bude stručně popsáno několik běžně dostupných nástrojů pro generování grafů volání ze zdrojových kódů a jejich následné zpracování:

Doxygen

Doxygen je nástroj pro generování dokumentace ze zdrojových kódů. Generování dokumentace je založené na analýze zdrojových kódů a zpracování speciálních komentářů ve zdrojových kódech, kterým tento nástroj rozumí.

Grafy volání (stejně jako jiné užitečné informace) jsou získávány přímo ze zdrojových kódů bez použití překladače odpovídajícího jazyka. Kromě generování grafu volání umí Doxygen také generovat grafy závislostí, grafy dědičnosti a grafy spolupráce. Grafy jsou vizualizovány pomocí nástroje Graphviz a ve formě obrázků vkládány do vygenerované dokumentace.

Egypt

Egypt je na rozdíl od Doxygenu nástroj určený přímo pro zpracování grafu volání. Zdrojové kódy neanalyzuje přímo, ale využívá mezikód překladače gcc – tzv. *RTL (Register Transfer Language)* [5].

Ukládání RTL do souboru se zapíná volbou překladače `-dr`, mezikód je potom ukládán do (textových) souborů s příponou `.00.expand`. Pomocí programu napsaného v jazyce Perl jsou v souborech s mezikódem vyhledána volání funkcí. Výstupem programu je opět textový soubor, který lze předat nástroji Graphviz pro vykreslení grafu. Pomocí parametru `--omit` je možné zadat seznam funkcí, které nemají být do grafu zaneseny. Nástroj tedy disponuje jednoduchou operací nad grafem volání.

Hlavní výhoda Egyptu spočívá v jeho jednoduchosti. Pro generování grafu volání není potřeba používat speciálně upravený překladač. Na distribuci Gentoo Linux je Egypt k dispozici jako balíček. Tento nástroj jsem úspěšně testoval na jednoduchých zdrojových kódech v jazyce C a C++.

[1] *Dehydra* je nástroj pro statickou analýzu, který patří k projektu Mozilla. Více informací na stránce projektu https://developer.mozilla.org/en/Dehydra.

CodeViz

CodeViz je podobný nástroji cgt, který byl popsán výše. Graf volání je generován pomocí upraveného překladače gcc. Na rozdíl od nástroje cgt není generování grafu volání zapouzdřeno do podoby zásuvného modulu. Formát generovaných souborů je podobný jako v případě cgt. Výhodou je, že není potřeba provádět *demangling* [13] (překlad do lidmi čitelné podoby) pro získané identifikátory.

Součástí nástroje CodeViz jsou také dva skripty napsané v jazyce Python. První z nich (`genfull`) zajišťuje spojení grafů volání jednotlivých modulů do jednoho souboru. Druhý skript (`gengraph`) z nashromážděných dat vybere požadovanou část grafu volání a vizualizuje ji pomocí Graphvizu. K dispozici jsou jen základní operace nad grafem volání, které lze zadat pomocí parametrů skriptu: výběr „top-level“ funkcí, seznam ignorovaných funkcí, maximální hloubka zanoření apod. Ale například hledání množiny cest nebo sledů z jedné funkce do druhé nástroj neumožňuje.

Bohužel poslední dostupná verze záplaty je pro překladač gcc verze 3.4.6, zatímco aktuálně vyvíjená verze gcc je 4.4.0. Překladač gcc verze 3.4.6 se záplatou pro CodeViz se mi (s drobnými úpravami) podařilo přeložit na školním serveru `eva` a otestovat na jednoduchých projektech v jazyku C a C++.

Graphviz

Graphviz je otevřený software pro vizualizaci grafů, který disponuje širokým výběrem rozložení, ale také různými grafickými rozhraními pro webovou a interaktivní vizualizaci. K dispozici jsou také nějaké přídavné nástroje, knihovny a napojení na různé programovací jazyky. Graphviz je používán pro vizualizaci výše zmíněnými nástroji Doxygen, Egypt a CodeViz a plánuji ho také využít pro vizualizaci výsledků navrhovaného nástroje pro zpracování grafu volání.

Prefuse

Prefuse visualization toolkit je framework napsaný v jazyce Java. Oproti Graphvizu je lépe připravený pro interaktivní vizualizaci, která je k dispozici již v základní instalaci. Navíc disponuje širokým výběrem animací a vizuálních efektů. Nevýhodou oproti Graphvizu je závislost na prostředí Java a slabší podpora dávkového zpracování.

3.3 Statická analýza

Jedním z dlouhodobých cílů vývoje dotazovacího nástroje je jeho nasazení pro statickou analýzu zdrojového kódu. K tomu je však potřeba rozšířit zásuvný modul pro překladač gcc. Zásuvným modulem pro překladač se tato práce nezabývá. Kromě výše zmíněného nástroje Dehydra jsem experimentoval také s nástrojem Sparse, který je běžně dostupný v linuxových distribucích.

Sparse[2] je sémantický parser pro jazyk C, který vyvinuli vývojáři linuxového jádra. Ve zdrojových kódech dokáže odhalit některé chyby, které nevidí samotný překladač, jako je například míchání ukazatelů do uživatelského adresového prostoru s ukazateli do adresového

[2] http://www.kernel.org/pub/software/devel/sparse/

prostoru jádra. Dodatečné (sémantické) informace o typech jsou definovány pomocí tzv. *anotací*. Tyto anotace z pohledu překladače jazyka C nemají žádný význam. Sparse lze spustit samostatně pomocí příkazu `sparse`, detekované chyby jsou potom vypisovány jako varování na standardní chybový výstup. Pomocí různých parametrů `-W` lze nastavit, které sémantické chyby mají být detekovány nebo naopak potlačeny (pomocí `-Wno-`). K dispozici je také wrapper pro překladač gcc (`cgcc`), který přijímá parametry `gcc` i parametry `sparse`. Tento wrapper zajistí spuštění sparse po dokončení překladu. Na distribuci Gentoo Linux je Sparse dostupný jako balíček.

3.4 Knihovna BGL

BGL (*Boost Graph Library*) [11] je jedna z knihoven patřících do projektu Boost C++ Libraries. Knihovna je založena na šablonách jazyka C++ a s výjimkou parseru souborového formátu Graphvizu není potřeba ji kompilovat, stačí importovat potřebné hlavičkové soubory.

Knihovna vychází z STL, základní myšlenkou je oddělení datových struktur (kontejnerů) od algoritmů, které s nimi pracují. Pokud je m počet algoritmů a n počet kontejnerů, tento přístup dokáže redukovat velikost kódu z $O(m \cdot n)$ na $O(m + n)$. Nezávislost přístupu k datům je zajišťována pomocí *iterátorů* [6], které tvoří rozhraní pro průchod datovými strukturami. Knihovna dokáže pracovat také s externími datovými strukturami pomocí *adaptérů* [6], přičemž není nutné stávající data kopírovat.

Grafové algoritmy jsou snadno rozšiřitelné pomocí abstrakce tzv. *návštěvníka* (*visitor* [6]), což je funkční objekt s více metodami. Při průchodu grafem pomocí zvoleného grafového algoritmu jsou pak při různých událostech volány odpovídající metody návštěvníka. Pro vrcholy i hrany lze samozřejmě definovat libovolné aplikačně specifické vlastnosti.

Na nejnižší vrstvě jsou implementovány algoritmy prohledávání grafu: prohledávání do šířky (*Breadth First Search*), prohledávání do hloubky (*Depth First Search*) a prohledávání podle ceny (*Uniform Cost Search*). Nad těmito algoritmy jsou pak postaveny algoritmy vyšší úrovně, jako například vyhledávání nejkratší cesty (*Dijkstra*, *Bellman-Ford*, ...) nebo vyhledávání minimální kostry grafu (*Kruskal*, *Prim*).

Kapitola 4

Návrh dotazovacího nástroje

Tato kapitola popisuje jednotlivé kroky návrhu dotazovacího nástroje. Nejprve jsou navrženy abstraktní datové typy pro práci s grafem a operace nad nimi. Většina zde uvedeného platí pro grafy obecně, navíc jsou zmíněny souvislosti s aplikací na graf volání. Potom jsou představeny základní metody pro zpracování navržených dotazovacích operací a jejich optimalizace. Dále je vysvětleno, k čemu je sestavovací program grafů volání a jak pracuje. V poslední části návrhu je stručně představen objektový model dotazovacího nástroje a sestavovacího programu.

4.1 Datové typy

V této kapitole jsou navrženy abstraktní datové typy, se kterými pracují navazující kapitoly návrhu. Jednotlivé datové typy jsou pak na úrovni implementace přímo/nepřímo nahrazeny datovými typy z knihovny BGL, STL kontejnery, bitovými poli apod.

Jako abstrakce pro graf volání byl použit *orientovaný graf*, který byl definován v kapitole 2. Vrcholy grafu odpovídají *funkcím*, hrany grafu odpovídají *voláním* funkcí. S ohledem na zmíněnou definici budeme označovat množinu všech funkcí symbolem V, množinu všech volání symbolem E a graf volání symbolem G. Množinu všech grafů volání (univerzum) označme U_G. Dále je potřeba navrhnout datové struktury pro výsledky (a případně mezivýsledky) dotazování nad grafem volání:

Podmnožina vrcholů *(vertex subset)*
: Oborem hodnot datového typu *podmnožina vrcholů* je množina 2^V. Typickým příkladem operace, jejíž výsledkem je podmnožina vrcholů (funkcí), je dotaz na přímé/nepřímé předchůdce/následníky vrcholu (přímo/nepřímo volající/volané funkce). Datový typ však nemusí být implementován jako množina. Možným přístupem k reprezentaci podmnožiny je bitové pole, které funguje jako tzv. *charakteristická funkce množiny* [8].

Orientovaná cesta *(directed path)*
: Dalším datovým typem je orientovaná cesta. Podle kapitoly 2 lze cestu v grafu jednoznačně definovat jak posloupností hran, tak posloupností vrcholů. S ohledem na následující datový typ (orientovaný tah) byla pro reprezentaci zvolena posloupnost hran.

Díky tomu je možné na vstup některých operací dát cestu i tah, aniž by to zpracovávající algoritmus musel nějak rozlišovat. Oborem hodnot datového typu *orientovaná cesta* (dále označovaným jako P) je množina všech orientovaných cest nad dotazovaným grafem:

$$\begin{aligned} P := \{(e_0, e_1, ..., e_n) | \quad & \forall i = 0, ..., n \;\; e_i \in E \;\; \wedge \\ & \forall i = 1, ..., n \;\; (Pv(e_i) = Kv(e_{i-1})) \;\; \wedge \\ & \forall i, j = 0, ..., n \;\; (Pv(e_i) = Kv(e_j) \Rightarrow i = j + 1) \;\; \} \end{aligned} \tag{4.1}$$

Délka cesty (hloubka zanoření volání) se rovná počtu prvků uspořádané n-tice (n). Protože množina hran grafu (E) je konečná, bude množina P také konečná a délka cesty omezená. Nad tímto datovým typem lze definovat množinu. Oborem hodnot datového typu *množina orientovaných cest* je 2^P.

Orientovaný tah *(directed trail)*

Orientovaný tah je zobecněným datovým typem předchozího. Jak bylo uvedeno v kapitole 2, orientovaný tah je jednoznačně určen posloupností svých hran. Oborem hodnot datového typu *orientovaný tah* (dále označovaným jako T) je množina všech orientovaných tahů nad dotazovaným grafem:

$$\begin{aligned} T := \{(e_0, e_1, ..., e_n) | \quad & \forall i = 0, ..., n \;\; e_i \in E \;\; \wedge \\ & \forall i = 1, ..., n \;\; (Pv(e_i) = Kv(e_{i-1})) \;\; \wedge \\ & \forall i, j = 0, ..., n \;\; (e_i = e_j \Rightarrow i = j) \;\; \} \end{aligned} \tag{4.2}$$

Stejně jako u předchozího datového typu je délka tahu omezená. Nad orientovaným tahem lze také definovat množinu. Oborem hodnot datového typu *množina orientovaných tahů* je 2^T.

Orientovaný sled *(directed walk)*

Orientovaný sled je dalším zobecněním předchozích datových typů. Na rozdíl od orientovaného tahu není požadována podmínka, aby položky uspořádané n-tice byly navzájem různé. Oborem hodnot orientovaného sledu (dále označovaného jako W) je tedy:

$$\begin{aligned} W := \{(e_0, e_1, ..., e_n) | \quad & \forall i = 0, ..., n \;\; e_i \in E \;\; \wedge \\ & \forall i = 1, ..., n \;\; (Pv(e_i) = Kv(e_{i-1})) \;\; \} \end{aligned} \tag{4.3}$$

Podgraf indukovaný množinou sledů *(induced subgraph on a walk set)*

Pro datové typy orientovaná cesta a orientovaný tah existovaly odpovídající datové typy množina orientovaných cest a množina orientovaných tahů. Stejnou situaci bychom proto očekávali u datového typu orientovaný sled. Problém je, že (na rozdíl od dvou předchozích typů) může být množina orientovaných sledů potenciálně nekonečná i pro konečný graf. S takovým datovým typem je tedy potřeba pracovat symbolicky.

Snažil jsem se vyhnout vymýšlení nestandardního způsobu konečné reprezentace potenciálně nekonečné množiny orientovaných sledů a jako datový typ pro její reprezentaci jsem zvolil opět graf. Konkrétně se jedná o *podgraf indukovaný množinou vrcholů*,

který byl definován v kapitole 2. Množina vrcholů je přitom definována následovně:

$$A = \{v \in V(G) | v \text{ je součástí aspoň jednoho sledu z množiny sledů}\} \tag{4.4}$$

Výhodou takového přístupu je, že datový typ je opět graf. Je tedy možné jej použít jako vstup dotazovacích operací, které očekávají graf na svém vstupu. Tento datový typ se objevuje na vstupu tzv. *prohledávače cest* (viz. dále).

4.2 Dotazovací operace

Pro dotazování nad grafem volání byly navrženy operace, které pracují s výše uvedenými datovými typy. S výjimkou obvyklých množinových operací (průnik, sjednocení, rozdíl) se nejedná o binární operace na množině [8]. Nebudou tedy zkoumány vlastnosti operací jako je asociativita, komutativita apod.

Na všechny navrhované operace je možné se dívat jako na *zobrazení z A do B* [8]. Pro každou navrhovanou operaci bude uvedeno odpovídající zobrazení ve tvaru $A \to B$. Tím je definována množina přípustných vstupních hodnot a omezen (pro surjektivní zobrazení definován) obor výstupních hodnot. Takto definované operace usnadní návrh gramatiky (případně výběr vhodné existující gramatiky) dotazovacího jazyka, kterým budou tyto operace zadávány.

Hledání předchůdců/následníků

Dotaz na množinu předchůdců/následníků vrcholu je jedna ze základních operací nad grafem, která může být použita jako základ složitějších dotazů. Obě operace jsou navrženy jako zobrazení:

$$U_G \times V \to 2^V \tag{4.5}$$

Podle značení uvedeného v kapitole 2 budou tyto operace označovány jako $V^-(x)$, resp. $V^+(x)$ a budou mít také stejný význam, tj. vracet množinu přímých předchůdců, resp. přímých následníků.

Nad takto definovanými operacemi lze vytvořit tranzitivní a tranzitivní-reflexivní uzávěr. Tím se výsledná množina rozšíří o nepřímé předchůdce, resp. nepřímé následníky; v případě tranzitivního-reflexivního uzávěru také o dotazovaný vrchol samotný. Jako rozšíření lze navrhnout omezení tranzitivity na určitou hloubku – např. přidáním hloubky jako parametru:

$$U_G \times V \times \mathbb{N} \to 2^V \tag{4.6}$$

Hledání cest a tahů

Operace *hledání cest* je definována jako zobrazení:

$$U_G \times V^2 \to 2^P \tag{4.7}$$

Na vstupu operace je graf volání, zdrojový vrchol a cílový vrchol (zdrojová a cílová funkce). Výsledkem operace je množina všech (orientovaných) cest ze zdrojového vrcholu do cílového. Pokud ze zdrojového vrcholu do cílového neexistuje cesta, výsledná

množina je prázdná. Podobně lze definovat operaci hledání tahů:

$$U_G \times V^2 \to 2^T \tag{4.8}$$

Pro výsledek operace hledání tahů platí vždy, že je nadmnožinou (připouštíme rovnost) výsledku operace hledání cest. V případě, že graf volání neobsahuje přímou ani nepřímou rekurzi (acyklický graf), budou výsledky těchto operací stejné.

Dotaz na existenci cesty

Přestože lze k tomuto účelu využít operaci hledání cest, z důvodu efektivity je vhodné pro tyto dotazy vytvořit samostatnou operaci. Pro zjištění „jednobitové informace“ o existenci cesty není potřeba shromažďovat informace o všech možných cestách a pak je bez užitku zahodit. Operace *dotaz na existenci cesty* je definována jako zobrazení:

$$U_G \times V^2 \to \{0, 1\} \tag{4.9}$$

Hledání sledů

Výsledkem této operace je podgraf indukovaný množinou vrcholů, který byl definován jako datový typ v předchozí kapitole. Stejně jako u předešlých dotazovacích operací je na vstupu graf a dvojice vrcholů. Zobrazení má tvar:

$$U_G \times V^2 \to U_G \tag{4.10}$$

Tato operace je využívána prohledávačem cest a tahů k ořezání grafu a zvýšení efektivity prohledávání. Ale také lze pomocí této operace například jednoduše definovat podgraf grafu volání pro vizualizaci.

Ořezání grafu množinou vrcholů (funkcí)

Operace *ořezání grafu množinou vrcholů* je zobecněním předchozí operace. Výsledkem je opět graf indukovaný množinou vrcholů. Tentokrát však může být množina vrcholů volena libovolně, například jako výsledek jiné dotazovací operace. Zobrazení, které popisuje dotazovací operaci, má tvar:

$$U_G \times 2^V \to U_G \tag{4.11}$$

Dotaz na existenci cyklu

Operace *dotaz na existenci cyklu* je definována jako zobrazení:

$$U_G \times V \to \{0, 1\} \tag{4.12}$$

Na vstupu je orientovaný graf a vrchol. Výsledkem je 1 pokud je vrchol součástí nějakého cyklu v grafu a 0 v opačném případě. Tuto operaci lze jednoduše převést na dotaz na existenci cesty z daného vrcholu do vrcholu samého. Z pohledu grafu volání získáme informaci o tom, jestli je v dané části programu rekurze. Smyčka v grafu je brána jako triviální cyklus a odpovídá přímé rekurzi.

Transformace grafu na prostý graf

Transformace grafu na prostý graf je další operací, jejíž cílem je zjednodušení grafu. Je definována jako zobrazení:

$$U_G \rightarrow U_G \tag{4.13}$$

Vstupem operace je jakýkoliv graf a výsledkem je odpovídající prostý graf. Násobné hrany jsou po skupinách vždy nahrazeny jednou hranou, která je zastupuje.

Množinové operace

Množinové operace (průnik, sjednocení, rozdíl) má smysl definovat pro množiny prvků stejných typů. Tyto operace jako jediné odpovídají definici binární operace na množině a tvoří tedy algebry $(2^V, \cap, \cup, -)$, $(2^P, \cap, \cup, -)$, $(2^T, \cap, \cup, -)$ a $(2^W, \cap, \cup, -)$.

Jak bylo uvedeno výše, platí $2^P \subseteq 2^T$ a tedy algebra $(2^P, \cap, \cup, -)$ je *podalgebrou* algebry $(2^T, \cap, \cup, -)$. Pro úplnost (a snadnější volbu gramatiky) jsou uvedeny odpovídající zobrazení i pro množinové operace:

$$2^V \times 2^V \rightarrow 2^V \tag{4.14}$$

$$2^P \times 2^P \rightarrow 2^P \tag{4.15}$$

$$2^T \times 2^T \rightarrow 2^T \tag{4.16}$$

$$2^W \times 2^W \rightarrow 2^W \tag{4.17}$$

4.3 Prohledávání do hloubky

Základem většiny dotazovacích operací je prohledávání do hloubky (*Depth Search*). Pozor, nejedná se o algoritmus *Depth First Search*, protože cílem není najít první vrchol, který splňuje nějakou vlastnost. Cílem je najít všechny vrcholy, které splňují danou vlastnost, případně najít všechny cesty, tahy nebo sledy mezi vrcholy, apod.

Samotná knihovna BGL disponuje implementací algoritmu prohledávání do hloubky, která tvoří základ složitějších algoritmů. Původní verze dotazovacího nástroje pracovala s BGL implementací Depth Search a vlastní implementací návštěvníka (*visitor*). BGL implementace Depth Search však kromě zásobníku používá také barvení vrcholů a tím pádem vyžaduje další úložiště pro tyto barvy. Navrhované dotazovací operace barvení vrcholů nevyužívají. Navíc implementace návštěvníka byla objemnější než implementace samotného algoritmu Depth Search.

Pro dotazovací operace, které pracují s Depth Search byla zavedena abstrakce tzv. *bitmapového indexu* (kapitola 4.5), která umožňuje získat množinu vrcholů prohledávaných do hloubky v konstantním čase. Výhodou grafové reprezentace BGL je, že lze snadno otočit směr prohledávání – není tedy problém prohledávat graf ve směru i proti směru orientovaných hran.

4.4 Filtrování grafu

Filtrování grafu je jedna ze základních operací nad grafem. Používá se buď jako dotazovací operace přímo nebo jako součást nějaké dotazovací operace (pro ořezání grafu a zvýšení efektivity). Příkladem prvního může být výše navržená operace *dotaz na množinu sledů*, příkladem druhého mohou být operace hledání cest a tahů. Podle teorie grafů (kapitola 2) odpovídá filtrování grafu vytváření podgrafu. Filtrování grafu je možné provádět ve dvou rozměrech – filtrování vrcholů a filtrování hran. Při filtrování vrcholů je potřeba s každým odebraným vrcholem odebrat také incidující hrany.

Jako implementaci grafového filtru lze použít šablonu `filtered_graph` z BGL, která je velice efektivní. Šablona původní graf ani jeho data nijak nekopíruje. Pouze předefinuje vrcholové (resp. hranové) iterátory tak, aby přeskočily vrcholy (resp. hrany), které neodpovídají zvolenému vrcholovému (resp. hranovému) predikátu. Nevýhodou takového přístupu je, že čísla (v terminologii BGL deskriptory) vrcholů (resp. hran) nejdou za sebou. S tím je potřeba při zpracování filtrovaného grafu počítat a přistupovat k prvkům grafu výhradně pomocí iterátorů. Vedlejším efektem takového filtrování grafu je, že se zvýší složitost operace zjištění počtu vrcholů (resp. hran) v grafu z $O(1)$ na $O(n)$. To je způsobeno tím, že interně udržovaný údaj pořád odpovídá původní hodnotě. Vrcholy (resp. hrany) je proto potřeba projít a spočítat.

4.5 Bitmapový index

Optimalizace dotazovacích operací je založena na tzv. *bitmapovém indexu*. Z matematického pohledu je bitmapový index implementací *relace dosažitelnosti* $R \subseteq V^2$, která je definována vztahem:

$$aRb \iff \textit{existuje cesta z vrcholu } a \textit{ do vrcholu } b$$

Tato relace je v paměti uložena v podobě *bitových polí*. Při vytváření bitmapového indexu je ke každému vrcholu grafu alokováno bitové pole, jehož jednotlivé bity určují, které vrcholy jsou z daného vrcholu dosažitelné. Velikost každého jednoho bitového pole je tedy rovna celkovému počtu vrcholů grafu. Odtud plyne prostorová složitost bitmapového indexu: n^2 bitů, kde $n = |V|$ je počet vrcholů (funkcí).

Pro znázornění redukce složitosti dotazovacích operací zaveďme následující značení:

		$\mathbf{n} = \lvert V \rvert$	...	počet všech vrcholů (funkcí)
$\mathbf{m}$	$\leq$	n	...	počet vrcholů prohledávaných do hloubky

Porovnání složitosti jednotlivých dotazovacích operací je v tabulce 4.1. Dotaz na množinu následníků (tranzitivní uzávěr přímého následníka) lze získat v konstantním čase, neboť tato informace je v indexu přímo uložena. Jako výsledek dotazu tedy stačí vrátit odkaz na odpovídající bitové pole. Dotaz na množinu předchůdců vyžaduje inverzní index, který je definován jako relace R^{-1} (inverzní k relaci R). Stejně tak lze v konstantním čase získat informaci o existenci cesty – jako výsledek lze vrátit odpovídající bit bitového pole, které

dotazovací operace	prohledávání do hloubky	bitmapový index
V^-, V^+	$O(1)$	$O(1)$
tranzitivní uzávěr V^-, V^+	$O(m)$	$O(1)$
dotaz na existenci cesty	$O(m)$	$O(1)$
dotaz na existenci cyklu v grafu	$O(n^2)$	$O(n)$

Tabulka 4.1: Redukce složitosti dotazovacích operací pomocí bitmapového indexu

je svázané počátečním vrcholem cesty. Při hledání cest mezi dvěma vrcholy může přinést bitmapový index také výrazné snížení časové složitosti. Ze slepého prohledávání do hloubky se díky indexu stane informované prohledávání do hloubky. Navíc pokud cesta mezi vrcholy neexistuje, prohledávání lze zcela přeskočit.

Algoritmus 1 popisuje sestavení bitmapového indexu. Na začátku jsou množiny dosažitelných vrcholů každého vrcholu nastaveny na jejich přímé následníky. Potom jsou množiny následníků postupně šířeny přes hrany grafu, dokud se některá z množin mění. Jako optimalizaci lze zavést další bitové pole, které drží informaci o tom, které množiny byly v posledním kroku měněny. Vrcholy, jejichž množiny dosažitelných vrcholů se v posledním kroku neměnily lze v následujícím kroku přeskočit.

Algoritmus 1 Výpočet relace dosažitelnosti

Vstup: graf volání G

Výstup: množina dosažitelných vrcholů I_v pro všechny $v \in V_G$

Metoda:

```
1:  for all v ∈ V_G do
2:     for all s ∈ V_G^+(v) do                  // for all successors
3:        I_v(0) := I_v(0) ∪ {s}
4:     end for
5:  end for
6:  k := 0
7:  repeat
8:     k := k + 1
9:     for all v ∈ V_G do
10:       I_v(k) := I_v(k − 1)
11:       for all s ∈ V_G^+(v) do               // for all successors
12:          I_v(k) := I_v(k) ∪ I_s(k − 1)
13:       end for
14:    end for
15: until ∀v ∈ V_G : I_v(k) = I_v(k − 1)
16: for all v ∈ V_G do                          // treat I_v from last step as the result
17:    I_v := I_v(k)
18: end for
```

Jednotlivé množiny $I_v(k)$ představují množiny dosažitelných vrcholů v hloubce k. Vý-

sledkem algoritmu je množina I_v z posledního kroku. Počet kroků algoritmu (k) je shora omezen maximální hloubkou volání a konkrétní hodnota k se může lišit při sestavování dopředného a inverzního bitmapového indexu. Asymptotická časová složitost algoritmu pro sestavení bitmapového indexu je $O(n^2 \cdot d)$, kde $n = |V|$ je počet vrcholů (funkcí) a d maximální délka cesty v grafu. Implementace algoritmu pracuje s binárními operátory (OR, XOR) nad bitovými poli. To může (na některých architekturách) výrazně urychlit sestavení indexu.

Takto navržený algoritmus je efektivní v případě, že potřebujeme vypočítat bitmapové indexy pro všechny vrcholy v grafu. Během testování nástroje se však ukázalo, že při jednoduchých dotazech operace výpočtu všech indexů zdržuje. Komponenta pro výpočet bitmapových indexů proto byla doplněna o výpočet jednotlivých bitmapových indexů nezávisle na ostatních.

4.6 Hledání cest a tahů

Důležitou součástí dotazovacího nástroje je tzv. *prohledávač*. Jeho úkolem je najít všechny cesty nebo tahy v daném grafu. Výsledkem prohledávače je tedy množina. Pro některé typy grafu roste mohutnost výsledné množiny velice rychle vzhledem k počtu vrcholů v grafu. Počet cest nebo tahů v grafu lze spočítat předem pro acyklické grafy. Pro cyklické grafy lze alespoň stanovit horní omezení počtu tahů nebo cest pro daný graf. Vycházel jsem ze vztahu pro počet cest z [3] (v terminologii [4] orientovaných sledů). Označme n-tou mocninu matice sousednosti M_n:

$$M_n := (M_G^+)^n \tag{4.18}$$

Množinu všech sledů délky n z vrcholu v_i do vrcholu v_j označme $W_{v_i,v_j,n}$:

$$W_{v_i,v_j,n} := \{(e_0, e_1, ..., e_{n-1}) \in W \mid \quad Pv(e_0) = v_i \wedge Kv(e_n) = v_j\} \tag{4.19}$$

Potom platí:

$$\forall n \in \mathbb{N} \quad \forall v_i, v_j \in V \quad (M_n)_{ij} = |W_{v_i,v_j,n}| \tag{4.20}$$

Zjednodušeně řečeno, jednotlivé prvky matice $(M_n)_{ij}$ udávají počet sledů délky n mezi vrcholy v_i a v_j. Počet všech sledů délky n v grafu lze spočítat jako součet všech prvků matice:

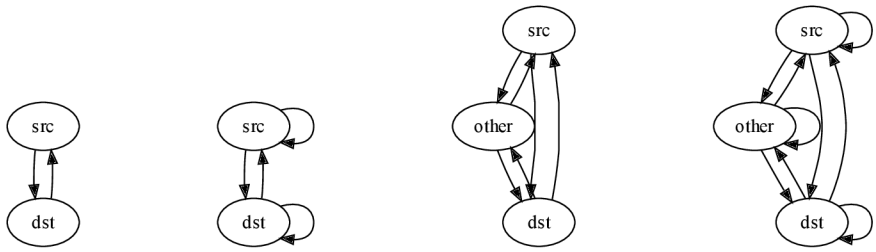
$$|W_n| = \sum_{\forall v_i, v_j \in V} (M_n)_{ij} \tag{4.21}$$

Pro acyklický graf je délka sledu shora omezena počtem hran v grafu a celkový počet sledů lze v tomto případě spočítat jako sumu přes všechny délky sledů:

$$|W| = \sum_{\forall n=1,...,|E|} |W_n| \tag{4.22}$$

V případě acyklického grafu je množina cest a tahů totožná s množinou sledů pro daný graf. Pro cyklický graf lze takto vypočtený údaj použít jako horní odhad pro počet cest nebo tahů v grafu (protože délky cest a tahů jsou vždy omezeny počtem hran v grafu).

Na obrázku 4.1 jsou znázorněny jednoduché grafy o dvou a třech vrcholech. V tabulce 4.2 jsou pak uvedeny odpovídající počty cest a tahů v grafu – jednak cesty/tahy, které vedou z vrcholu *src* do vrcholu *dst*, a jednak celkový počet cest a tahů v daném grafu. Nejvíce cest a tahů při stejném počtu vrcholů je v plně propojeném grafu (varianty (b) a (d)). Zejména počet tahů v grafu roste velice rychle vzhledem k počtu vrcholů. Pro plně propojený graf o 2 vrcholech, existuje v grafu 23 tahů, ale pro plně propojený graf o 3 vrcholech už existuje v grafu 1882 tahů. Pro plně propojený graf o 4 vrcholech už je prohledávač zcela nepoužitelný a jeho výpočet neskončí ani za několik hodin.



(a) (b) (c) (d)

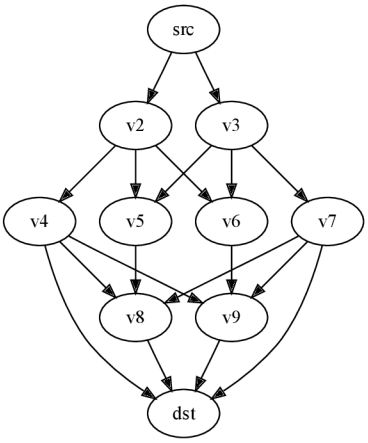
Obrázek 4.1: Příklady jednoduchých grafů pro demonstraci hledání cest a tahů

Naštěstí se běžné grafy volání (pro které je prohledávač určen) nepodobají plně propojeným grafům. Cykly v grafu (přímá/nepřímá rekurze) se vyskytují spíše ojediněle a celkově je počet hran také výrazně nižší v porovnání s plně propojeným grafem. Na obrázku 4.2 je ukázka typického grafu reálného programu, který vznikl jako výsledek ořezání (viz. dále). Takový graf bychom mohli například získat ořezáním grafu volání nějakého programu funkcemi `main` a `error`. Počty cest a tahů pro tento graf jsou rovněž uvedeny v tabulce 4.2. Jak bylo uvedeno výše, pro acyklický graf dostaneme stejný výsledek, hledáme-li cesty, jako hledáme-li tahy. Složitost prohledávání je tedy také v obou případech pro acyklický graf stejná.

G	$\|V(G)\|$	$\|E(G)\|$	cest $src \to dst$	všech cest	tahů $src \to dst$	všech tahů
obr. 4.1a	2	2	1	3	1	5
obr. 4.1b	2	4	1	3	4	23
obr. 4.1c	3	6	2	13	9	97
obr. 4.1d	3	9	2	13	153	1882
obr. 4.2	10	18	10	67	10	67

Tabulka 4.2: Závislost počtu cest a tahů na prohledávaném grafu.

Algoritmus pro hledání cest a tahů v grafu je navržen tak, aby pracoval s explicitním zásobníkem. U rekurzivně zapsaného algoritmu totiž hrozí vzhledem k objemu dat přetečení programového zásobníku. Alg. 2 popisuje jeho variantu pro hledání orientovaných tahů.



Obrázek 4.2: Abstraktní příklad – výsledek ořezání grafu volání vrcholy *src* a *dst*

Algoritmus pro hledání orientovaných cest se příliš neliší od popsaného algoritmu. Stačí upravit predikát v bodě 12 tak, aby se v cestě nemohly opakovat ani vrcholy. Prohledávač je proto implementován genericky jako šablona a jeho konkrétní typ je zvolen parametrem šablony.

Algoritmus 2 Hledání orientovaných tahů

Vstup: graf volání G, zdrojový vrchol v_{src}

Výstup: množina všech tahů X_v pro všechny $v \in V_G$, které začínají vrcholem v_{src}

Metoda:

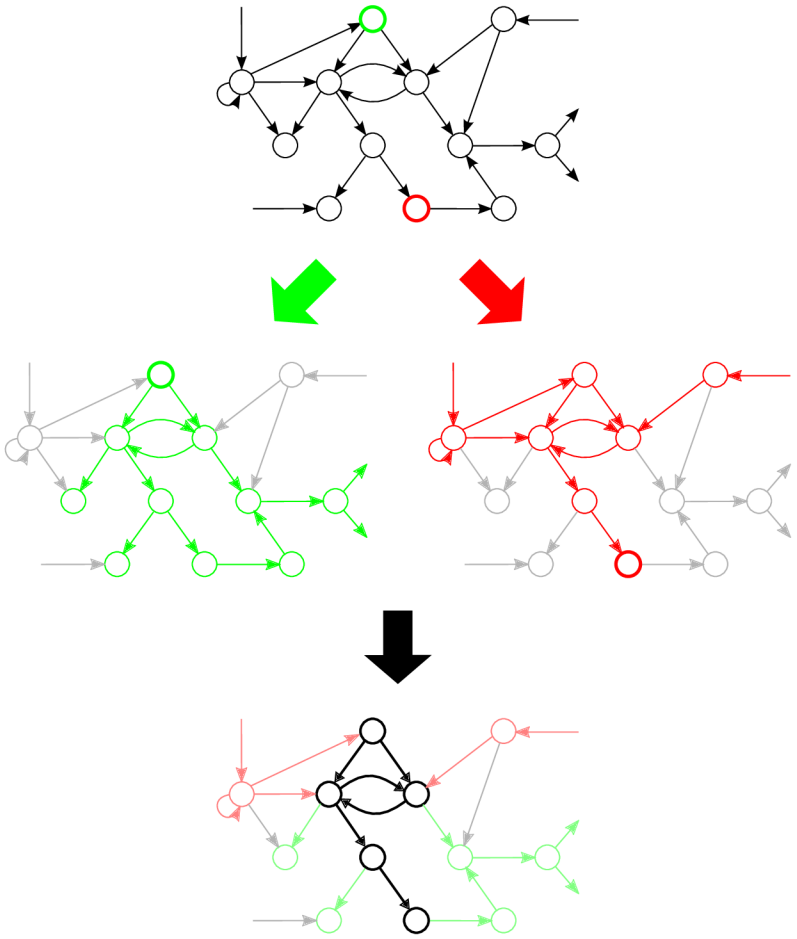
```
1:  for all v ∈ V_G do
2:    if v = v_src then
3:      X_v := {()}
4:    else
5:      X_v := {}
6:    end if
7:  end for

8:  S := {v_src}
9:  while S ≠ ∅ do
10:   v := pop(S)
11:   for all T ∈ X_v do                    // for all paths T leading to v
12:     for all {e ∈ E_G^+(v)|e is not in the path T} do
13:       T_new := T.e                      // append e to path T
14:       s := Kv(e)                        // destination of the edge
15:       if T_new ∉ X_s then
16:         X_s := X_s ∪ T_new
17:         push(S, s)
18:       end if
19:     end for
20:   end for
21: end while
```

4.7 Ořezání grafu

V předchozí kapitole bylo uvedeno, jak vysoká je složitost prohledávače cest a tahů. Prohledávač je zcela jistě nejsložitější (z pohledu časových a prostorových nároků) komponenta celého dotazovacího nástroje. Je tedy snaha učinit prohledávání co možná nejvíce efektivní. Není dobré spouštět prohledávač nad celým grafem, ale nad jeho nejmenším možným podgrafem. Samozřejmostí je, že ve výsledku musíme dostat stejné množiny cest a tahů jako při zpracování celého grafu. Takovým podgrafem je právě *podgraf indukovaný množinou sledů*, který byl definován v kapitole 4.2. Ten lze efektivně získat s lineární časovou složitostí pomocí grafového filtru a bitmapového indexu, jak ukazuje obr. 4.3. V návaznosti na předchozí definice označme množinu vrcholů, které patří do nějaké sledu na cestě z v_{src} do v_{dst} symbolem A a relaci dosažitelnosti symbolem R. Potom lze množinu A získat následujícím vztahem:

$$A = \{x \in V(G) | v_{src} R x\} \cap \{x \in V(G) | x R v_{dst}\} \tag{4.23}$$



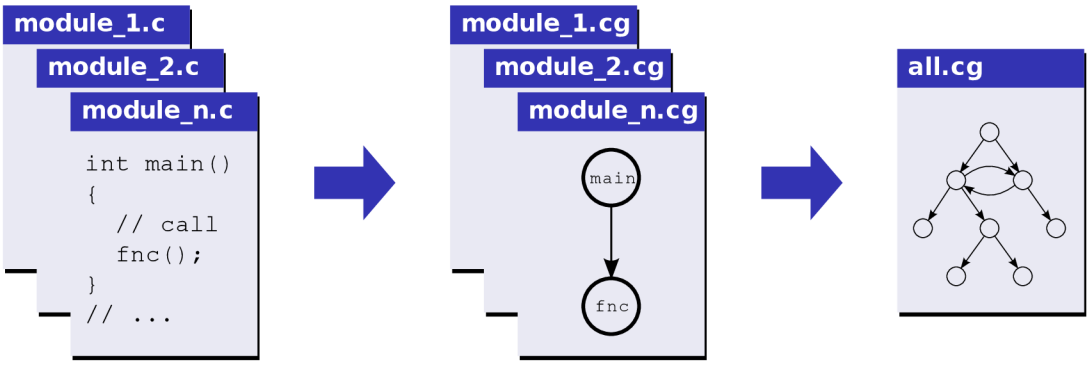
Obrázek 4.3: Operace ořezání grafu

Izomorfním zobrazením do algebry bitových polí se z operace průniku dvou množin stane operace logického součinu dvou bitových polí. Vytvoření filtrovaného grafu je operace s konstantní časovou složitostí. Celková časová složitost je tedy lineární. Operace ořezání grafu se používá jako předzpracování při hledání tahů/cest, ale také je přístupná uživateli. Tato operace samozřejmě není omezena na jediný zdrojový a cílový vrchol – snadno ji lze rozšířit na množiny zdrojových a cílových vrcholů.

4.8 Sestavovací program grafu volání

Jak bylo uvedeno v kapitole 3, grafy volání jsou získávány při překladu jednotlivých modulů. Překladem tedy získáme tolik oddělených grafů volání, kolik modulů překládáme. Dotazovací nástroj však na svém vstupu očekává jediný graf volání, je tedy nutné spojit tyto grafy volání do jednoho, který je bude popisovat jako celek. A to je právě úloha sestavovacího programu. Při sestavování binárních modulů a vytváření knihovny nebo spustitelného programu se sestavují také grafy volání, jak ilustruje obr. 4.4.

Ať už jsou grafy volání získávány jakkoliv, vždy je potřeba nějak vymezit rozsah plat-



Obrázek 4.4: Sestavování grafu volání

nosti jednotlivých symbolů[1]. Některé symboly jsou platné na úrovni modulu, v terminologii jazyka C se takovým symbolům říká statické. Naproti tomu existují globální (v terminologii jazyka C externí) symboly, jejichž platnost přesahuje hranice modulu. Takový symbol může být definován pouze v jednom modulu a pouze jednou. Ostatní moduly, které s globálním symbolem pracují, pak obvykle obsahují jeho deklaraci. Ve výsledku tedy získáme binární soubor a graf volání, který popisuje jeho obsah. Činnost sestavovacího programu grafu volání je přitom podobná činnosti binárního sestavovacího programu. Také se pracuje se symboly a také je potřeba řešit rozsah platnosti symbolů.

Spolu se zásuvným modulem pro generování grafů volání jsem dostal i plně funkční sestavovací program grafů volání. Tento sestavovací program pracuje s pevně definovaným formátem grafu volání a jeho reprezentací v paměti. Já jsem se rozhodl udělat o něco obecnější sestavovací program, který pracuje s abstrakcí grafu. Navržený sestavovací program je tedy nezávislý na vstupním/výstupním formátu a částečně i na reprezentaci grafu.

Sestavovací program je k dispozici jako šablona založená na BGL a tuto šablonu potom používá spustitelný sestavovací program. Vstupní grafy volání mohou obsahovat velké množství deklarací dovezených z hlavičkových souborů. Při dotazování jsou však užitečné deklarace pouze těch funkcí, jež jsou někde v kódu volány. Ostatní deklarace o analyzovaném programu nic neříkají a není je proto potřeba zahrnovat do výsledného grafu volání. Sestavovací program tyto zbytečné deklarace filtruje pomocí grafového filtru na samotném začátku zpracování.

Spolu s každým symbolem je ukládána informace o jeho původním výskytu – v současném formátu vstupních dat se jedná o název souboru a číslo řádku. V případě definice nás pouze zajímá, kde byl symbol definován, a odkud byl potom odkazován. Informace o deklaracích u definovaných symbolů nejsou užitečné a v průběhu sestavování se zahazují. Naopak některé symboly uvnitř analyzované části programu definovány nejsou a potom nás bude zajímat jejich deklarace. Většinou se jedná o symboly definované v systémových knihovnách, místem deklarace jsou potom systémové hlavičkové soubory.

[1] V současné verzi dotazovacího nástroje se pracuje pouze s názvy funkcí/metod. Do budoucna se však počítá s rozšířením pojmu symbol také pro názvy proměnných.

4.9 Objektově orientovaný návrh knihovny

Objektový model dotazovacího nástroje (a sestavovacího programu) by se dal rozdělit na několik částí podle toho, na jaké úrovni abstrakce tyto části pracují. Čím vyšší je úroveň abstrakce, tím více jsou komponenty znovupoužitelné. Snažil jsem se proto co možná nejvíce aplikační logiky přesunout do úrovně obecně využitelných grafových algoritmů. Následuje stručné uvedení jednotlivých úrovní abstrakce (vrstev):

Abstrakce grafu *(vrstva 1)*
: S abstrakcí obecného grafu pracují šablony `BitmapIndexer` (obr. 4.5) a `PathFinder` (obr. 4.6), které zapouzdřují implementaci algoritmů 1 a 2. Při návrhu těchto šablon jsem se inspiroval knihovnou BGL, na které jsou šablony postavené. Pro pohyb v grafu jsou používány abstraktní iterátory a abstraktní deskriptory vrcholů a hran. Parametrem šablony je typ grafu, který definuje způsob reprezentace grafu (maticová reprezentace, seznam sousednosti, ...), popisky vrcholů (např. název symbolu a místo jeho definice) a popisky hran (např. místo volání). Konkrétní typy iterátorů a deskriptorů jsou z typu grafu extrahovány pomocí tzv. *rysů (traits)* [2].

Abstrakce grafu volání *(vrstva 2)*
: Graf volání už je definovaný jako konkrétní typ BGL grafu (v současné verzi obousměrný seznam sousednosti) s pevně danými popisky vrcholů. Tam, kde to je možné, počítá implementace pouze s existencí některých složek popisků vrcholů a nevyžaduje konkrétní typ grafu. Příkladem mohou být šablony `SymbolMap`, `Linker` a `DropUnusedDeclarations`.

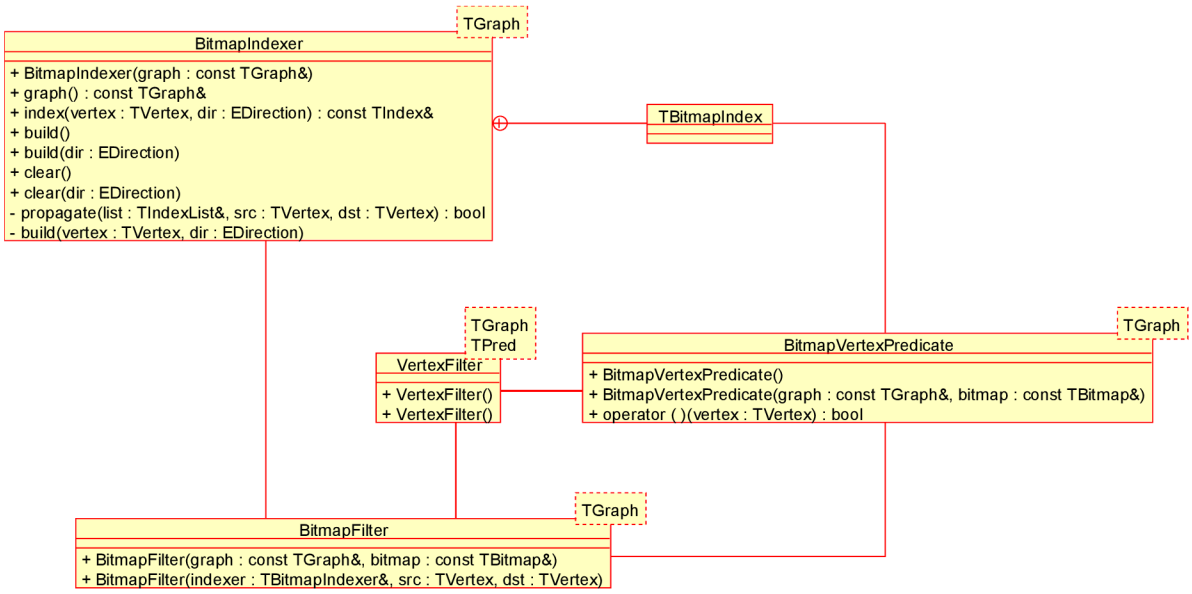
Konkrétní reprezentace grafu volání *(vrstva 3)*
: Na nejnižší úrovni abstrakce pracují komponenty pro vstup/výstup grafu volání, které tvoří bránu mezi interní reprezentací grafu a okolním světem. V současné verzi je podporován vstupní formát používaný nástrojem cgt (uvedeným v kapitole 3) a výstupní formáty cgt a Graphviz.

Dále bude uveden stručný popis stěžejních tříd/šablon:

`BitmapIndexer`
: Diagram tříd šablony `BitmapIndexer` je na obr. 4.5. Při vytváření objektu je konstruktoru předána reference na indexovaný graf. Tento graf musí být platný a neměnný po celou dobu existence objektu. Pouhé vytvoření objektu nezpůsobí žádnou objemnou alokaci paměti, ani náročný výpočet. Jednotlivé (dopředné nebo inverzní) bitmapové indexy jsou zpřístupněny metodou `index()`. Pokud bitmapový index není k dispozici, je vypočítán automaticky a uložen ve vyrovnávací paměti bitmapových indexů pro případný další dotaz. Metodou `build()` lze vypočítat dosud nevypočtené indexy najednou pomocí alg. 1. Metoda `clear()` naopak všechny již vypočtené indexy smaže a tím uvolní paměť. Metody `build()` a `clear()` mají čistě optimalizační charakter a nemohou nijak ovlivnit výsledek vrácený metodou `index()` – tento fakt je také ověřován v odpovídajícím unit testu (viz. dále).

`BitmapFilter`

Šablona `BitmapFilter` (rovněž zachycena na obr. 4.5) implementuje grafový filtr, který používá jako predikát bitové pole (*bitmapu*). Obecnější varianta konstruktoru očekává jako parametr bitové pole o velikosti počtu vrcholů originálního grafu. Vedle toho je k dispozici konstruktor, který realizuje operaci ořezání grafu zdrojovým a cílovým vrcholem, jak byla definována v kapitole 4.7.



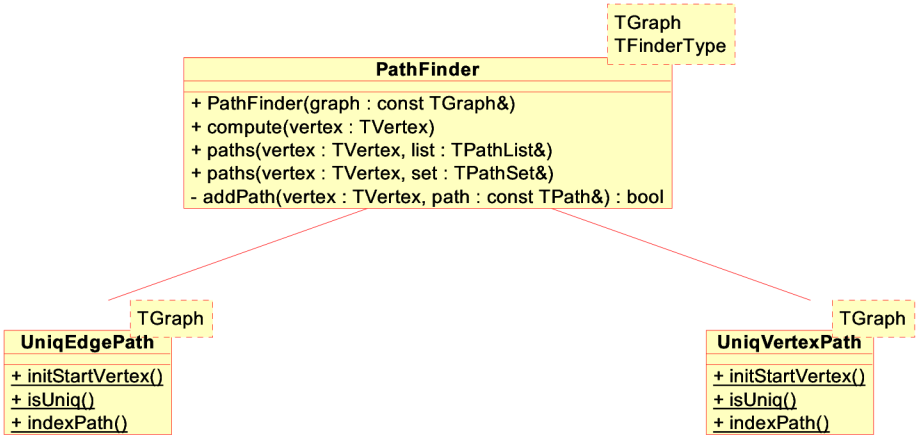
Obrázek 4.5: Diagram tříd – šablona `BitmapIndexer`

`PathFinder`

Šablona `PathFinder` (obr. 4.6) funguje jako univerzální prohledávač orientovaných cest a tahů. Konkrétní typ prohledávače je určen druhým parametrem šablony – `UniqVertexPath` pro hledání orientovaných cest a `UniqEdgePath` pro hledání orientovaných tahů. Výpočet cest a tahů je spuštěn metodou `compute()` se zdrojovým vrcholem jako parametrem. Výsledkem algoritmu jsou cesty/tahy do všech vrcholů grafu ze zadaného zdrojového vrcholu. Vypočtené množiny cest pro jednotlivé cílové vrcholy jsou zpřístupněny metodou `paths()`. Jak bylo uvedeno v kapitole 4.7, vhodným vstupem prohledávače je ořezaný graf, tj. instance třídy `BitmapFilter`.

`Linker`

Šablona `Linker` obaluje typ grafu (daného parametrem `TGraph`), která udržuje informace nezbytné pro sestavování grafů volání. Jednotlivé grafy se sestavují pomocí metody `link()`, která je opět šablona. Je tedy možné sestavovat grafy různého typu, což je nezbytná podmínka pro použití grafových filtrů. Filtrovaný graf je totiž (kvůli předefinování iterátorů) jiného typu než původní graf. Spustitelný sestavovací program používá jako vstup grafový filtr s predikátem `DropUnusedDeclarations`.



Obrázek 4.6: Diagram tříd – šablona `PathFinder`

`CgtReader`, `CgtWriter`

Tyto třídy implementují vstup/výstup grafu volání ve formátu cgt, který je naopak zase zcela oddělen od abstrakce grafu, aby bylo možné tyto třídy znovu použít v programech, které s abstrakcí grafu vůbec nepracují. Most [6] pro načítání grafu tvoří šablona `CgtGraphBuilder`, která dostane typ grafu jako parametr šablony. Zároveň ale implementuje rozhraní `ICgtReaderListener`, které definuje sadu funkcí zpětně volaných parserem formátu cgt. Při zápisu formátu cgt je společným jmenovatelem šablona funkce `write()` definovaná ve vrstvě 2, která jako parametr dostane objekt grafu (typu `TGraph`) a objekt zapisovače (typu `TWriter`).

Kapitola 5

Implementace dotazovacího nástroje

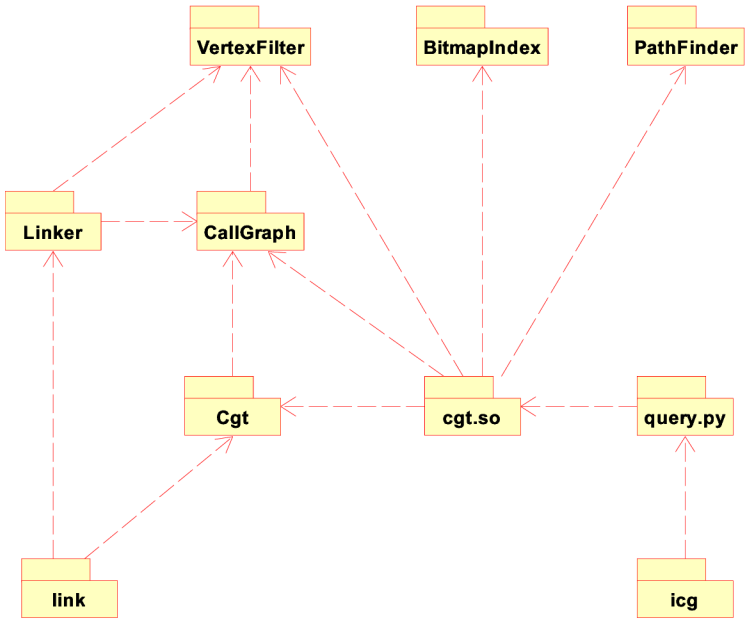
Jak bylo uvedeno v kapitole 4, návrh je založen na C++ knihovně BGL. Pro implementaci grafových algoritmů jsem proto zvolil také C++. Uživatelské rozhraní pro zadávání dotazů nad grafem je tvořeno modulem napsaným v Pythonu, který jsem dostal z větší části již hotový. Pro svoji práce jsem tedy použil hlavně jazyk C++. Propojení C++ knihovny s uživatelským rozhraním v Pythonu je realizováno pomocí knihovny Boost Python (viz. dále).

Grafové algoritmy byly implementovány genericky pomocí šablon, jsou proto k dispozici pouze ve formě hlavičkových souborů (podobně jako šablony knihovny BGL). Výjimku tvoří modul pro vstup/výstup formátu cgt a modul implementující napojení na Python, které se kompilují odděleně. Cílem návrhu šablon vrstvy 1 byla jejich vysoká znovupoužitelnost. Pro zvýšení kvality kódu těchto šablon (a snadnější údržbu) byly k jednotlivým šablonám napsány unit testy. Tyto unit testy se rovněž kompilují jako běžné C++ moduly.

Na obr. 5.1 jsou znázorněny závislosti mezi jednotlivými moduly. Tyto moduly jsou seřazeny shora dolů podle vrstvy, na které pracují (viz. kapitola 4.9). Úplně nahoře jsou moduly implementující obecné grafové algoritmy. Nejníže jsou moduly, které tvoří uživatelské rozhraní – jejich názvy odpovídají spustitelným souborům, které jsou dostupné uživateli.

5.1 Uživatelské rozhraní

Uživatelské rozhraní současné verze dotazovacího nástroje je založeno na jazyce Python. To znamená, že zadávané dotazy jsou platné řetězce jazyka Python. Pomocí různých konstrukcí (lambda výrazy, přetěžování operátorů, ...) je vytvořen nový jazyk pro zadávání dotazů nad grafem. Díky tomu je možné například používat již hotové funkce v jazyce Python jako predikát v zadávaném dotazu. Nebo je naopak možné provést dotaz nad grafem uvnitř nějakého složitějšího programu. Jádrem uživatelského rozhraní je modul `query.py`, který jsem dostal již hotový a plně funkční. Mým úkolem bylo vytvořit knihovnu založenou na BGL a na ni tento modul napojit. Součástí této práce je záplata pro modul `query.py`,



Obrázek 5.1: Diagram závislosti modulů

která nahrazuje implementace dotazovacích operací v jazyce Python jejich ekvivalenty založenými na BGL.

Propojení grafových algoritmů s uživatelským rozhraním zajišťuje sdílený objekt `cgt.so`, který je založený na knihovně *Boost Python* [7]. Tato knihovna umožňuje hladkou spolupráci objektů jazyka Python s objekty jazyka C++. Migrace objektů mezi Python a C++ přitom nepředstavuje pro programátora prakticky žádnou práci navíc. Je potřeba pouze nadefinovat, které třídy budou exportovány a pod jakými názvy. Následuje jednoduchý příklad exportované třídy:

```
class vset_subgraph {
    public:
        vset_subgraph(vset &);
        std::string dump_for_graphviz();

    private:
        typedef BitmapFilter<TGraph> TFilter;
        TFilter filteredGraph_;
};

BOOST_PYTHON_MODULE(cgt) {
    class_<vset_subgraph>
        ("vset_subgraph", init<vset &>())
        .def("dump_for_graphviz", &vset_subgraph::dump_for_graphviz);
}
```

Modul `query.py` nemá smysl spouštět samostatně. Tento modul je navržený pro import do jiných Python modulů. Proto byl vytvořen samostatně spustitelný program `icg` pro zadávání dotazů nad grafem, který obaluje modul `query.py`. Jako parametr očekává jméno

jediného souboru obsahující graf volání, který je při startu načten. Program `icg` může potom pracovat ve dvou režimech – interaktivním a dávkovém. Odpovídající režim je zvolen automaticky podle toho, jestli je standardní vstup přesměrován, nebo připojen k terminálu. Interaktivní režim je zajišťován knihovnou *readline*, která mimo jiné udržuje perzistentní historii dotazů a umožňuje v ní vyhledávat dříve zadané dotazy. V dávkovém režimu jsou načítány příkazy ze standardního vstupu a výsledky posílány na standardní výstup.

5.2 Sestavovací program

Sestavovací program se jmenuje `link`. Jako parametry příkazové řádky očekává názvy souborů, které obsahují jednotlivé grafy volání. Sestavený graf volání je potom vypsán na standardní výstup. V současné verzi je podporován pouze formát cgt, ale do budoucna je počítáno s rozšířením pro další vstupní/výstupní grafové formáty (jako např. GraphViz). Implementace sestavovacího programu je velice jednoduchá. Na začátku je vytvořena instance třídy `Linker`. Potom jsou postupně načteny grafy volání pomocí `CgtReader`/`CgtGraphBuilder` a sestaveny. Výsledný graf je nakonec vypsán pomocí generické funkce `write` a zapisovače `CgtWriter`.

5.3 Externí názvy symbolů C++

Překladač jazyka C++ používá pro jména symbolů tzv. *name mangling* [13] – dekorace názvu symbolu jeho typem, případně jmenným prostorem apod. Díky tomu je např. přetěžování funkcí zcela transparentní z pohledu sestavovacího programu, který místo stejného názvu funkce s jinými parametry jednoduše vidí jiný symbol. Formát cgt s výhodou pracuje se symboly ve stejném tvaru jako odpovídající binární moduly – v případě jazyka C++ tedy s jejich externími názvy. Na této úrovni pracuje také sestavovací program grafů volání.

Z pohledu uživatele však nejsou externí názvy dobře čitelné. Třída `CgtReader` proto volitelně provádí tzv. *demangling* symbolů. Externí názvy pro jazyk C++ jsou popsány ve specifikaci binárního rozhraní C++ [1] a funkce pro jejich vytváření a dekódování jsou např. součástí balíku *binutils*. Převod externího názvu do lidmi čitelné podoby je zajišťován funkcí `cplus_demangle` z archivu `libiberty.a`. Bohužel spolu s balíkem binutils není distribuován hlavičkový soubor, který obsahuje deklaraci funkce `cplus_demangle` (a souvisejících maker). Bylo tedy potřeba odpovídající deklarace zkopírovat na úrovni zdrojového kódu. Nevýhodou takového přístupu je, že není možné počítat s kompatibilitou při použití jiné verze binutils.

5.4 Překlad a sestavení programu

Pro úspěšný překlad a sestavení ze zdrojových kódů a provozování dotazovacího nástroje jsou potřeba následující prerekvizity:

- `CMake 2.4+` (build systém)
- `Boost 1.35+` (balík knihoven)
- `python 2.5` (interpret jazyka Python)
- `readline` (knihovna pro interaktivní vstup)
- `binutils` (balík nástrojů pro sestavování programů)
- `GraphViz` (nástroj pro vizualizaci grafů)

Automatický build systém `CMake` zajistí nalezení požadovaných knihoven. Pokud je statická knihovna `libiberty.a` umístěna v jiném adresáři než ostatní systémové knihovny, je potřeba její adresu nastavit ručně, případně vytvořit symbolický odkaz v některém adresáři, kde jsou knihovny hledány. Pro sestavení projektu stačí zadat příkaz `make` v adresáři projektu. Spolu se zdrojovými soubory je distribuována také sada unit testů pro šablony pracující na vrstvě 1. Příkazem `make check` lze tyto unit testy spustit. V současné verzi dotazovacího nástroje není explicitně kontrolována náročnost dotazu, ani předpokládaná doba jeho zpracování. Vzhledem k paměťové složitosti prohledávače cest a tahů je proto dobré omezit dostupnou virtuální paměť pomocí příkazu `ulimit` tak, aby nedocházelo k nadměrnému využívání odkládacího souboru.

Kapitola 6

Výsledky testování

Vytvořený nástroj byl testován na různě velkých grafech volání vygenerovaných ze skutečných programů. V této kapitole budou uvedeny výsledky testování dotazovacího nástroje a sestavovacího programu na několika příkladech. Na přiloženém CD je možné najít další grafy volání vygenerované ze zdrojových kódů různých programů. Pro generování grafů byly použity jak programy psané v jazyce C, tak programy psané v jazyce C++.

6.1 Sestavování grafu

Implementované řešení bylo mimo jiné testováno na zdrojových kódech linuxového jádra (verze 2.6.19.7). Linuxové jádro jsem nejprve přeložil pomocí upraveného překladače gcc a zásuvného modulu pro generování grafů volání. Tím jsem získal ke každému překládanému modulu jeho odpovídající graf volání v textovém souboru (ve formátu cgt). Grafy volání jednotlivých modulů jsem potom sestavoval do jediného. To sice není za všech okolností zcela správně[1], ale pro účely testování sestavovacího programu (a následně dotazovacího nástroje) je tento přístup dostatečně úspěšný. Sestavovací program sám o sobě nedisponuje rekurzivním průchodem adresáře. K tomu lze snadno použít nástroje z balíku *findutils*:

```
$ find ./linux-2.6.19.7 -name \*.cg | xargs link > kernel.cg
```

V tabulce 6.1 jsou shrnuty statistiky sestavování grafu volání linuxového jádra, který obsahuje více než 40000 vrcholů. V prostředním sloupci tabulky jsou pro srovnání uvedeny odpovídající hodnoty původní implementace sestavovacího programu. Nuly v pravém sloupci jsou způsobeny tím, že v současné verzi sestavovacího programu nejsou nijak zohledněny deklarace ani definice proměnných. Rozšíření grafu pro znázornění vztahů mezi proměnnými je jeden z úkolů, který by bylo dobré v příští verzi nástroje nějak vyřešit. Vzhledem k současnému formátu generovaných grafů volání má smysl zpracovávat pouze proměnné platné na globální úrovni. Odlišnosti mezi hodnotami v prostředním a pravém sloupci tabulky jsou zapříčiněny drobnými rozdíly ve zpracování *inline* funkcí a podobných pokročilých technik, které jsou ve zdrojových kódech jádra používány.

Další možné vylepšení se týká výkonu sestavovacího programu. Nový sestavovací program je při sestavování všech modulů jádra asi 4× pomalejší než jeho původní implementace. Při profilování sestavovacího programu jsem zjistil, že úzkým hrdlem je naivní

[1] Kromě samotného kódu jádra jsou v adresáři také pomocné moduly, které mají na starosti jeho konfiguraci, překlad a sestavení.

	Původní implementace	Současná implementace
Doba sestavování grafů volání	13.80 s	41.79 s
Doba sestavování již sestaveného grafu	1.42 s	1.62 s
Počet odkazovaných souborů	5 897	2 534
Počet deklarovaných proměnných	32 813	0
Počet definovaných proměnných	85 638	0
Počet deklarovaných funkcí	1 841	43
Počet definovaných globálních funkcí	12 316	14 525
Počet definovaných statických funkcí	55 035	27 169

Tabulka 6.1: Porovnání výsledků původní a současné implementace sestavovacího programu

implementace parseru formátu cgt, která vstup načítá po řádcích a následně každý řádek testuje několika regulárními výrazy. Nabízí se řešení ve formě jednoprůchodového čtení vstupu pomocí Flex/Bison. Vzhledem k dobrým výsledkům původního sestavovacího programu však bude nejlepší, když se z něj vytáhne ručně psaný parser formátu cgt a napojí se na šablonu `CgtGraphBuilder`. To se mi zatím nepodařilo, neboť původní implementace parseru není vzhledem k vysoké optimalizaci dostatečně zapouzdřená. Z pohledu uživatele však sestavování není časově kritická operace (ve srovnání s dobou překladu jádra), byly proto upřednostněny úkoly s vyšší prioritou. V případě dotazování nad grafem pomocí `icg` se výkon parseru projevuje jenom při zpracování dotazů samostatně. Při proudovém nebo interaktivním zpracování lze s výhodou využít toho, že vstup je načten (a případně indexován) pouze jednou.

6.2 Dotazování nad grafem

Práce s dotazovacím nástrojem bude nejprve předvedena na jednoduchém školním projektu. Na rozdíl od abstraktních příkladů z kapitoly 4.6 bude tentokrát na vstupu graf volání vygenerovaný z existujícího programu. Pro ukázku byl zvolen jednoduchý překladač neexistujícího jazyka. Úkolem je zjistit, proč program vypisuje nesmyslné chybové hlášení ve funkci `ParseError`, přičemž k programu není k dispozici žádná dokumentace. Předpokládejme, že máme již vygenerovaný a sestavený graf volání programu v souboru `proj.cg`. Prvním krokem je spuštění `icg` v interaktivním režimu. Objeví se příkazová řádka dotazovacího nástroje:

```
$ icg proj.cg
proj.cg>
```

Nyní je možné vypsat všechny symboly pomocí příkazu `cg`, případně nechat vykreslit graf volání pomocí příkazu `plot(cg)`. Graf volání vykreslený jako celek je však dosti nepřehledný. Úkolem je zjistit, za jakých okolností může program skončit ve funkci `ParseError`. Nabízí se tedy ořezání grafu funkcemi `main` a `ParseError`:

```
proj.cg> plot(cg.prune('main', 'ParseError'))
```

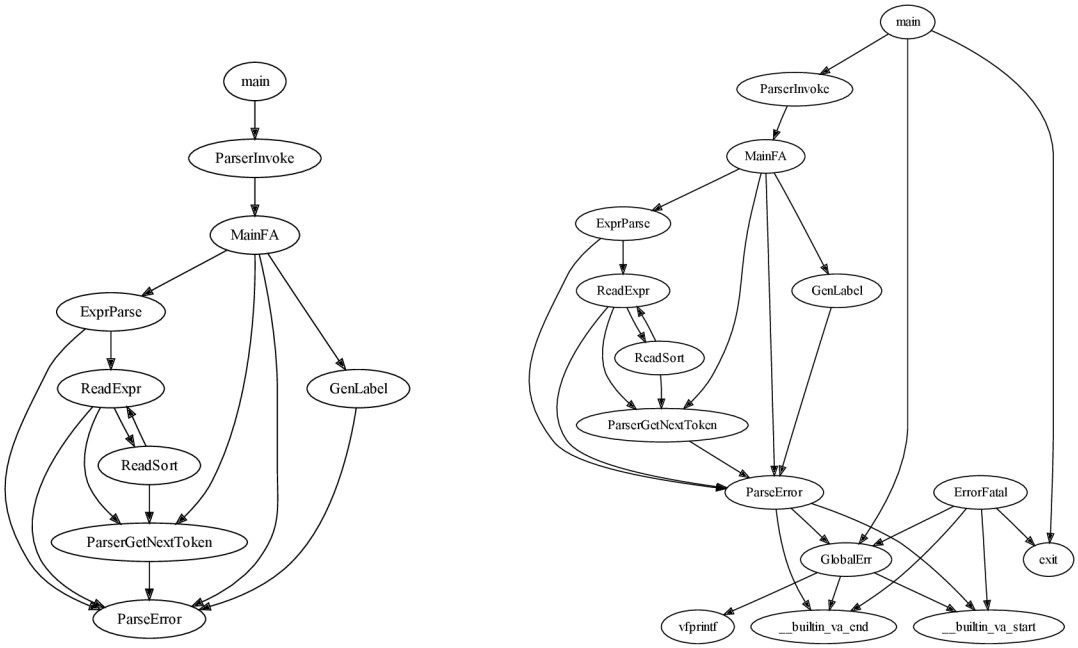
Výsledek dotazu je zachycen na obr. 6.1a. Grafem jsou popsány všechny možné cesty/tahy mezi funkcemi `main` a `ParseError`. Na první pohled je vidět (potenciálně nebezpečné)

nepřímé rekurzivní volání funkcí `ReadExpr` a `ReadSort`. Jako příklad složitějšího dotazu uvedu sjednocení původního podgrafu (množinou vrcholů, na které je indukován) s množinou přímých/nepřímých předchůdců všech funkcí, které začínají řetězcem `Error`:

```
proj.cg> plot(cg.prune('main', 'ParseError') + cg['Error.*'].trcallees())
```

Výsledek dotazu je zachycen na obr. 6.1b. Pomocí prohledávače je možné vypsat všechny cesty mezi dvěma vrcholy (případně množinami vrcholů):

```
proj.cg> paths('main', 'ParseError')
(main, ParserInvoke, MainFA, ParseError)
(main, ParserInvoke, MainFA, GenLabel, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ParserGetNextToken, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ReadSort, ParserGetNextToken, ParseError)
(main, ParserInvoke, MainFA, ParserGetNextToken, ParseError)
--- total solutions found: 7
```



(a) (b)

Obrázek 6.1: Výsledky dotazů nad grafem volání (jednoduchý školní projekt)

Vypsání cest v textové podobě má smysl spíše při proudovém zpracování, kdy je na standardní výstup `icg` napojen rourou nějaký filtr. S výsledkem prohledávače lze ale také pracovat jako s datovým typem množina cest, provádět operace sjednocení, průniku apod. Stejným způsobem lze položit dotaz na množinu tahů:

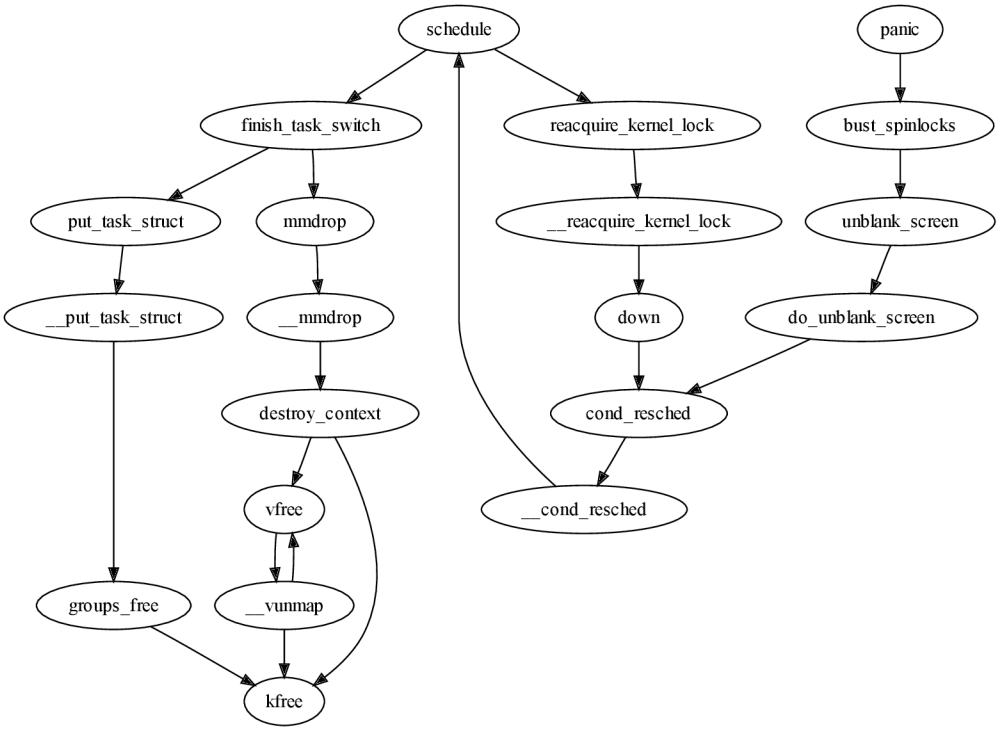
```
proj.cg> trails('main', 'ParseError')
(main, ParserInvoke, MainFA, ParseError)
```

```
(main, ParserInvoke, MainFA, GenLabel, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ParserGetNextToken, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ReadSort, ParserGetNextToken, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ReadSort, ReadExpr, ParseError)
(main, ParserInvoke, MainFA, ExprParse, ReadExpr, ReadSort, ReadExpr, ParserGetNextToken, ParseError)
(main, ParserInvoke, MainFA, ParserGetNextToken, ParseError)
--- total solutions found: 9
```

Na příkladu je vidět, že díky rekurzi (podmínka nutná, nikoliv postačující) je v grafu více tahů než cest a že všechny cesty jsou zároveň obsaženy v množině tahů. Jako poslední příklad kreslení grafu uvedu dotaz nad grafem volání linuxového jádra:

```
kernel.cg> plot(cg.prune('panic','kfree'), 'graph', 'pdf', True)
```

Volitelné parametry příkazu `plot` udávají název grafu (`graph`), výstupní formát grafu (`pdf`) a poslední parametr říká, že mají být z grafu odstraněny násobné hrany při kreslení. Výsledek dotazuje je na obr. 6.2. V grafu se vyskytují dva orientované cykly a tedy podezřelá místa, kde by mohlo teoreticky dojít k přetečení zásobníku – to může mít v případě jádra tragické důsledky. Z příkladu je patrné, že cyklus v grafu nestačí k tomu, aby prohledávač vrátil různé výsledky při hledání cest a hledání tahů – obě množiny jsou v tomto případě totožné.



Obrázek 6.2: Výsledek dotazu nad grafem volání linuxového jádra

6.3 Naměřené doby zpracování dotazovacích operací

V předchozí kapitole bylo uvedeno, jaké výsledky je možné pomocí nástroje získat a jak. Tato kapitola se více zaměřuje na časovou náročnost jednotlivých operací. Jak bylo zmíněno v kapitole 4.6, prohledávač cest a tahů je časově nejnáročnější část dotazovacího nástroje. V návrhu plánovače byly nastíněny jeho teoretické limity (v podobě plně propojených grafů). Ty však nevypovídají o jeho náročnosti při analýze reálných grafů volání. Pro účely měření časové náročnosti bylo potřeba vybrat vhodně velký graf volání. Ukázalo se, že některé dotazy nad grafem volání linuxového jádra jsou příliš složité. Zvolil jsem proto dostatečně jednoduchý graf volání, ve kterém je možné prohledat všechny cesty/tahy v grafu v přijatelném čase. Použitý graf je vygenerovaný z části zdrojových kódů balíku *elfutils* a obsahuje 1701 vrcholů (funkcí).

V tabulce 6.2 jsou uvedeny doby provádění jednotlivých dotazů. Pro srovnání jsou v jednom sloupci uvedeny odpovídající údaje původní implementace prohledávače cest. Běžně používané dotazy představuje horní polovina tabulky, kde nástroj opravdu vrací výsledky v čase přijatelném pro interaktivní dotazování (do časů je navíc zahrnuta i doba načtení grafu volání, která se při interaktivním režimu nijak neprojevuje). Množina cest a tahů, které začínají ve funkci `main` se mohutností už příliš neliší od množiny všech cest a tahů v grafu. Dotaz proto trvá o něco déle, než by si uživatel přál, ale zlepšení oproti

	Původní čas	Současný čas	Zrychlení	Výsledků	Výsledků/1s
`paths ('main','error')`	36.45 s	0.10 s	365×	273	2 730
`trails('main','error')`		0.10 s		309	3 090
`paths ('.*' ,'error')`	129.90 s	0.24 s	541×	1 217	5 071
`trails('.*' ,'error')`		0.37 s		1 534	4 146
`paths ('main','.*' )`	5 h 40 min	49.36 s	413×	112 748	2 284
`trails('main','.*' )`		58.70 s		129 000	2 198
`paths ('.*', '.*' )`	20 h 17 min	124.40 s	587×	424 548	3 413
`trails('.*', '.*' )`		162.98 s		503 600	3 090

Tabulka 6.2: Škálovatelnost prohledávače na reálném grafu volání (1701 vrcholů)

původní implementaci je přibližně stejné (viz. sloupec zrychlení). Graf volání opět obsahuje cykly, prohledávač tedy vrací různé výsledky při hledání cest a při hledání tahů. Doba prohledávání je pro obě varianty také různá.

Sestavení bitmapového indexu je úkol s výrazně nižší složitostí (viz. kapitola 4.5). Díky tomu bylo možné operaci sestavení indexu testovat na výrazně větších grafech. Zvolil jsem proto graf volání linuxového jádra. Doby sestavení bitmapového indexu jsou uvedeny v tabulce 6.3. Z výsledků je patrné, že doby sestavování dopředného a zpětného indexu se pro konkrétní grafy mohou lišit. Každý již vypočtený index je uložen ve vyrovnávací paměti, proto jsou v druhém řádku nulové časy odpovídající konstantní časové složitosti. Operace smazání všech indexů má lineární časovou složitost, tomu odpovídají příznivé časy v posledním řádku tabulky.

	Dopředný index	Zpětný index
První vyžádání všech indexů	2.06 s	2.79 s
Druhé vyžádání všech indexů	0.00 s	0.00 s
Smazání všech indexů	0.02 s	0.02 s

Tabulka 6.3: Čas potřebný k sestavení bitmapového indexu (41694 vrcholů)

Kapitola 7

Závěr

Jedním z analyzovaných nástrojů na zpracování grafu byl již hotový prototyp dotazovacího nástroje a sestavovacího programu. Z tohoto prototypu navržený a implementovaný nástroj z větší části vychází. Jako hlavní pilíř návrhu byla použita generická knihovna pro práci s grafy (BGL). Díky této knihovně v kombinaci s kompilovaným jazykem C++ bylo možné dosáhnout vysokého výpočetního výkonu, který je potřeba zejména při hledání cest a tahů v grafu. Oproti původní implementaci prohledávače je nově vytvořený prohledávač asi 500× rychlejší. Vhodným nástrojem pro vizualizaci grafu se ukázal GraphViz, který byl na dotazovací nástroj hladce napojen. Výsledky dotazovacích operací je tak možné jednoduše vizualizovat, což byl jeden z cílů při vývoji nástroje. Činnost dotazovacího nástroje byla předvedena na několika příkladech. Ukázalo se, že kromě rychlé analýzy jednoduchých programů si nástroj poradí také se složitějšími problémy, jako je například analýza grafu volání linuxového jádra.

Současná verze nástroje je plně funkční a může být používána pro analýzu grafů volání rozsáhlých programů. Tím ale vývoj nástroje nekončí. Celou řadu jeho vlastností je možné ještě vylepšit. Prvním krokem pravděpodobně bude napojení původního parseru formátu cgt na nově vytvořenou abstrakci grafu. Také by bylo dobré zvážit podporu dalších vstupních/výstupních formátů pro reprezentaci grafů volání. Současný objektový model je na tato rozšíření již připraven. Další prostor pro vylepšení je ve zpracování funkcí volaných přes ukazatel. Současné řešení ukazatele na funkce nijak nezpracovává. To je z praktického pohledu hlavní nevýhoda při analýze linuxového jádra. Rozhraní většiny subsystémů jádra je totiž založeno na strukturách, které obsahují pouze ukazatele na funkce. Zpracování těchto ukazatelů způsobem, který je pro uživatele nějak užitečný, je však netriviální problém. Mimo jiné to vyžaduje rozšíření zásuvného modulu pro překladač, se kterým jsem zatím přišel do styku jen v roli uživatele.

Lze předpokládat, že vývoj nástroje neskončí odevzdáním této diplomové práce. Nástroj je nyní zveřejněný na www[1] spolu se svými zdrojovými kódy, které jsou chráněny otevřenou licencí[2]. Díky tomu se do jeho vývoje mohou zapojit další vývojáři, které tato myšlenka zaujala a mají nápady, jak nástroj vylepšit.

[1] https://fedorahosted.org/cgt/

[2] GPLv3 – k dispozici na http://www.gnu.org/licenses/gpl-3.0.txt

Literatura

[1] Itanium C++ ABI. http://www.codesourcery.com/public/cxx-abi/abi.html, 2009.

[2] Alexandrescu, A.: *Moderní programování v C++*. Computer Press, 2004, ISBN 80-251-0370-6.

[3] Choe, G. H.: *Computational ergodic theory*. Springer, 2005, ISBN 3540231218.

[4] Demel, J.: *Grafy a jejich aplikace*. ACADEMIA, 2002, ISBN 80-200-0990-6.

[5] Free Software Foundation, I.: RTL Representation. http://gcc.gnu.org/onlinedocs/gccint/RTL.html, 2008.

[6] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1997, ISBN 0-201-63361-2.

[7] de Guzman, J.; Abrahams, D.: The Boost Python Library. http://www.boost.org/doc/libs/1_35_0/libs/python/doc/tutorial/doc/html/index.html, 2005.

[8] Krupková, V.: Diskrétní matematika. 2004.

[9] Nešetřil, J.: *Teorie grafů*. SNTL, 1979.

[10] *Sheng*, L.; *Özsoyoğlu*, Z. M.; *Özsoyoğlu*, G.: A Graph Query Language and Its Query Processing. http://art.cwru.edu/TOpapers/ICDE99.pdf, 2008.

[11] Siek, J.; Lee, L.-Q.: The Boost Graph Library. http://www.boost.org/doc/libs/1_35_0/libs/graph/doc/index.html, 2001.

[12] Wikipedia: Call graph. http://en.wikipedia.org/wiki/Call_graph, 2008.

[13] Wikipedia: Name mangling. http://en.wikipedia.org/wiki/Name_mangling, 2008.