VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

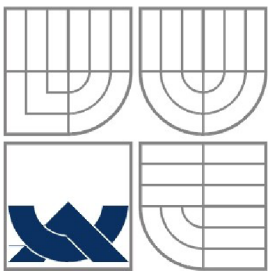
METODY PRO PRÁCI S GRAFY V DATABÁZI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

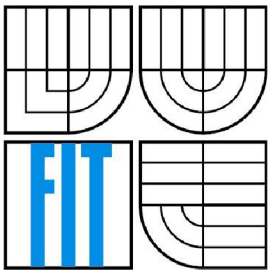
AUTOR PRÁCE
AUTHOR

BC. JOSEF HOVAD

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

METODY PRO PRÁCI S GRAFY V DATABÁZI

GRAPHS AND ITS METHODS IN DATABASES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. JOSEF HOVAD

VEDOUCÍ PRÁCE
SUPERVISOR

ING. OTA JIRÁK

BRNO 2011

Abstrakt

Práce seznamuje se základními pojmy teorie grafů a dále se způsoby reprezentace grafu jak v matematických úlohách, tak při programování. Dále představuje základní metody a problémy procházení grafů a teorie obecně. Jsou představeny možnosti správy grafových dat v různých typech databázových systémů, včetně systémů přímo vycházejících z teorie grafů. V praktické části práce je navržena efektivní metoda pro procházení grafy v databázi PostgreSQL. Tato metoda je otestována a demonstrována na prostřednictvím grafových algoritmů prohledávání, barvení a izomorfismu.

Abstract

The thesis introduces the basic concepts of graph theory and graph representation both in mathematics and programming. Furthermore, it presents basic methods and problems of graphs searching and theory in general. There are presented graph data management capabilities of different database systems including those directly based on the graph theory. In the practical part, there is designed an efficient method of graphs traversing in PostgreSQL database. The method was tested and demonstrated by the graph search algorithms, coloring and isomorphism.

Klíčová slova

Grafy, procházení grafů, reprezentace grafů, databáze, grafy v databázích, PostgreSQL, rozšíření PostgreSQL.

Keywords

Graph, graph theory, browsing graph, graph representation, database, graph management in database, PostgreSQL, PostgreSQL extension.

Citace

Josef Hovad: Metody pro práci s grafy v databázi, diplomová práce, Brno, FIT VUT v Brně, 2011

Metody pro práci s grafy v databázi

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Oty Jiráka.
Další informace mi poskytl především Ing. Petr Chmelař.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bc. Josef Hovad
15. 5. 2011

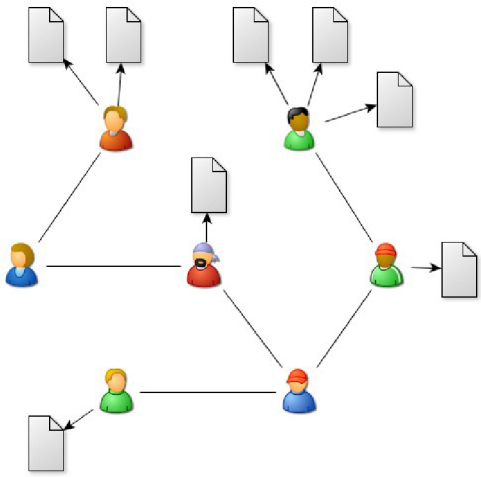
Poděkování

Děkuji vedoucímu práce Ing. Otu Jirákovi a Ing. Petru Chmelařovi za poskytnutí odborné pomoci.

Obsah

1 Úvod

Graf je matematická abstrakce, na kterou lze převést nespočet problémů reálného světa. Neformálně řečeno je grafem skupina libovolných objektů, mezi kterými existují vztahy. Objekty jsou v teorii grafů nazývány *vrcholy*, vztahy mezi nimi *hrany*. Jako graf si lze tedy představit například osoby reprezentované vrcholy, které mezi sebou sdílí vztahy přátelství reprezentované hranami (například na síti Facebook). Vrcholy grafu však nemusí reprezentovat objekty stejného typu. K vrcholům reprezentujícím osoby mohou být hranami připojeny další objekty jako jejich fotografie, videa, oblíbené stránky a podobně jako ukazuje Obrázek 1.1.



Obrázek 1.1: Příklad grafu sociální sítě

Grafem lze dále reprezentovat například města a cesty mezi nimi, vodovodní síť nebo odkazy na Wikipedii, na kterých provádím experimenty v této práci atd. V dnešní době je matematická teorie grafů značně rozpracována a řeší množství problémů, na které lze snadno abstrahovat problémy reálného světa. S grafy se často setká programátor a to především ve specifických formách stromových struktur. Přesto, že teorie grafů je matematicky důkladně zpracována, stále nabízí velký prostor pro další zajímavý výzkum.

1.1 Motivace

Samotná matematická teorie grafů je zavedený a relativně probádaný obor. Stejně tak je běžné využití teorie grafů v algoritmech. Přesto je v této problematice stále množství oblastí, které nabízí prostor pro výzkum a zlepšení. Příkladem může být problém zjišťování izomorfismu. V případě správy velmi rozsáhlých grafových struktur je vhodné použít adekvátní systém řízení báze dat, především v situacích, kdy již nejsme schopni strukturu grafu načíst do operační paměti. Pro tyto účely jsou dostupná komerční řešení, například od firmy Oracle, nebo specializované projekty vycházející

z teorie grafů, tzv. *graph databáze*. Zatím však neexistuje vhodné rozšíření práce s grafy pro velmi rozšířenou skupinu open source relačních databází, ve kterých spravuje data nespočet uživatelů různých zaměření. Vzhledem k aplikovatelnosti teorie grafů jsem přesvědčen, že vhodné rozšíření pro práci s grafy pro open source relační databáze by bylo velmi hodnotným přínosem.

Z těchto důvodů je prvním cílem této práce seznámit s teorií grafů a současnými možnostmi využití databázových systémů pro ukládání grafů. Dále si kladu za cíl navrhnout a realizovat vhodnou metodu správy grafu a jeho procházení v open source relačním databázovém systému PostgreSQL, protože tento systém disponuje vhodnými prostředky pro implementaci takové metody, respektive rozšíření. Na závěr práce chci ověřit a demonstrovat funkcionalitu navržených řešení a zhodnotit dosažené výsledky, především v kontextu dalšího vývoje řešené problematiky.

1.2 Struktura práce

Tato práce navazuje na stejnojmenný semestrální projekt, ve kterém jsem se zabýval především teoretickými základy teorie grafů. Semestrální projekt Metody pro práci s grafy v databázi seznamoval s teorií grafů a problémy, které lze s její pomocí řešit. Dále s možnými způsoby správy grafů v různých databázových systémech, včetně takových, které přímo vychází z teorie grafů. Na základě konzultací byla nastíněna metoda rozšíření práce s grafy pro databázi PostgreSQL, která je realizována v rámci této diplomové práce.

Při práci na semestrálním projektu jsem se seznámil s teorií grafů a to především v kontextu s informačními technologiemi. Prostudoval jsem možnosti správy grafových dat v databázových systémech a rozšířil znalosti o architektuře PostgreSQL databáze.

Ze semestrálního projektu přebírám především kapitolu 2, 3, 4 a 5, které jsou na několika místech rozšířeny o podstatné poznatky nabyté při realizaci navazující diplomové práce.

Tato diplomová práce je rozčleněna do osmi kapitol včetně úvodní, kterou právě čtete. V navazující druhé kapitole definuji východiska a cíle celé práce a dále se zaměřuji na popis nutných základů teorie grafů.

V kapitole 3 probírám možnosti jak graf reprezentovat při matematických výpočtech a dále vhodné reprezentace grafu v programech, kde některé z těchto možností využívám v navazujících praktických částech. Závěrem kapitoly třetí probírám dva základní způsoby prohledávání grafu. Jedná se o prohledávání grafu do šířky a do hloubky. Především se věnuji prohledávání grafu do šířky, které je následně implementováno pro navrženou metodu správy grafů v databázi.

V kapitole 4 se věnuji vybraným základním problémům řešeným v teorii grafů. Důkladně se zabývám především problémem barvení grafů a dále sofistikovanějším problémem hledání

izomorfismu grafů respektive grafu a podgrafu grafu. Těmito problémy se zabývám podrobněji, protože na nich budu dále demonstrovat navrženou metodu pro práci s grafy v databázi.

Kapitola pátá je věnována možnostem ukládání grafů v databázových systémech. Ty mohou být vhodným prostředkem pro uchovávání rozsáhlých souborů grafových dat. Zaměřuji se na popis možností ukládání grafů ve standardních, majoritně rozšířených relačních databázích a dále představuji i další typy databázových systémů, včetně specializovaných systémů vycházejících přímo z teorie grafů, tzv. grafové databáze.

Relační model databáze je suverénně nejrozšířenější a do dnešního dne neexistuje pro žádný open source relační databázový systém rozšíření pro efektivní správu grafových dat. Proto v této práci podobné rozšíření respektive metodu navrhuji. V kapitole 6. se zabývám návrhem rozšíření pro open source projekt PostgreSQL. Návrh vychází především z využití nízkoúrovňových identifikátorů záznamů, umožňujících maximální rychlost přístupu k datům a zachování spojové podstaty grafových dat.

V 7. kapitole demonstruji navrženou metodu efektivní struktury na několika algoritmech a porovnávám její časovou náročnost s řešením pomocí standardních SQL dotazů. Metoda je demonstrována na prohledávání grafu do šířky, na barvení grafů pomocí greedy coloringu a RLF coloringu a na VF2 algoritmu pro hledání izomorfismu.

Závěrečná osmá kapitola je věnována zhodnocení výsledků a přínosů práce a dále představuje možnosti dalších rozšíření a pokračování v probírané problematice.

2 Základní pojmy

S pomocí grafů lze popsat situace a řešit problémy, které můžeme modelovat jako konečné množství objektů (vrcholů) a vztahů mezi nimi (hran). V této kapitole se budu věnovat stručnému popisu základních pojmů terorie grafů. Vzhledem k množství dostupných materiálů, seznamujících se základy teorie grafů, se omezím pouze na stručné srhnutí informací nutných k pochopení problematiky řešené v této práci. V těch vycházím zejména z [BM82], [Hl07], [Ch06], [Ko04], [Li10], [St08].

2.1 Graf

Graf G je definován jako neprázdná množina vrcholů (V), z anglického *vertices,* a hran (E), z anglického *edges*.

$$G = (V, E) \tag{1}$$

Pokud bude množina hran E podmnožinou všech možných spojnic vrcholů V, tzn. neuvažujeme násobné hrany ani smyčky, pak se jedná o tzv. **obyčejný graf** (také „jednoduchý“ či „prostý“), kde:

$$E \subseteq \binom{V}{2} \tag{2}$$

Grafy, kde se vyskytují smyčky, tedy hrany spojující uzel sám se sebou, nazýváme obecné grafy a platí pro ně:

$$E \subseteq V \tag{3}$$

Grafy, kde jsou dva vrcholy spojeny více než jednou hranou (tzv. násobné hrany) jsou nazývány multigrafy, a platí pro ně:

$$E \to V \times V \tag{4}$$

Hrany grafů mohou být neorientované či orientované podle typu vztahu, který vyjadřují. Neorientované hrany vyjadřují oboustranné vztahy, orientované vyjadřují vztahy jenostranné. Podle typu hran lze graf označit jako neorientovaný (všechny hrany jsou neorientované), smíšený a orientovaný (všechny hrany jsou orientované), pro který platí:

$$E \subseteq V \times V \tag{5}$$

Pokud je e orientovaná hrana ve tvaru (x, y) kde x, y jsou vrcholy, je tento stav popisován tak, že „orientovaná hrana e vychází z x a končí v y“.

2.2 Další pojmy

V této části budou vysvětleny další pojmy, se kterými se lze v teorii grafů často setkat. Budou vysvětleny pojmy sled, tah, cesta, vzdálenost vrcholů, kružnice, úplný graf, souvislý graf, biparitní graf, izomorfismus, podgraf, stupeň vrcholu, skóre grafu, invarianty, ohodnocení hrany, metrika grafu, strom a kostra grafu.

Sled je posloupnost vrcholů a hran v orientovaném grafu $G = (V, E)$.

$$P = (v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n) \tag{6}$$

kde $v_1, v_2, \ldots, v_n$ jsou vrcholy množiny V a $e_1, e_2, \ldots, e_{n-1}$ jsou hrany množiny E, pro které platí $e_i = (v_i, v_{i+1})$.

Tah je sled o délce n mezi vrcholy v_1 až v_n, ve kterém se neopakuje žádná z hran.

Cesta je sled o délce n mezi vrcholy v_1 až v_n, ve kterém se neopakuje žádná hrana ani vrchol.

Vzdálenost vrcholů v_1 až v_n je počet hran nejkratší cesty, kterou můžeme mezi těmito vrcholy najít.

Kružnice (nebo také „cyklus“) je sled kde $v_1 = v_n$ a kde vrcholy v_1 až v_{n-1} jsou navzájem různé.

Úplný graf K_n, kde n je počet vrcholů, je graf ve kterém je každá dvojice vrcholů spojena hranou:

$$|E| = \binom{|V|}{2} \tag{7}$$

Souvislý graf je takový, že mezi jeho každými dvěma vrcholy můžeme najít cestu, která je spojuje. U orientovaných grafů rozlišujeme slabou a silnou souvislost. V silně souvislém orientovaném grafu existuje mezi každými dvěma vrcholy orientovaná i přesně opačná cesta. Graf je označen za slabě souvislý, pokud alespoň po převodu všech orientovaných hran grafu na hrany neorientované získáme souvislý graf. Maximální souvislý podgraf v nesouvislém grafu se označuje **komponenta**.

Biparitní graf je graf, jehož množinu vrcholů lze rozdělit na dvě čáti tak, že z vrcholů jedné části vedou hrany pouze do vrcholů druhé části.

Izomorfismus grafů $G = (V, E)$ a $G' = (V', E')$ existuje tehdy, pokud lze najít způsob jak vrcholy grafu G přejmenovat tak, aby odpovídaly vrcholům grafu G'. Takové přejmenování se nazývá bijektivní, neboli vzájemně jednoznačné zobrazení. Grafy G a G' jsou izomorfní pokud existuje bijektivní zobrazení:

$$f : V \to V' \text{ tak, že platí } : \{x, y\} \in E, \text{ právě když } \{f(x), f(y)\} \in E' \tag{8}$$

Invarianty jsou určité vlastnosti každého grafu, které lze vyjádřit číselnými hodnotami a jsou základem pro určení izomorfismu dvou grafů. Pokud chceme určit, zda jsou G = (V, E) a G' = (V', E') izomorfní, použijeme výsledek funkce gama nazvané invariant grafu (nebo „kódování“), která má alespoň jednu z vlastností:

a) pokud platí $\gamma(G) = \gamma(G')$, potom je G s G' izomorfní

b) pokud je G s G' izomorfní, pak platí $\gamma(G) = \gamma(G')$

Výsledkem γ funkce je invariant reprezentovaný číslem, maticí nebo polynomem.

Invarianty splňující vlastnost a) jsou pro graf G = (V, E):

- počet vrcholů (n = |V|),
- počet hran (m = |E|),
- nejvyšší a nejnižší stupeň ($\Delta(G)$ *a* $\delta(G)$),
- průměrný stupeň (deg(G), platí: $\Delta(G) \geq \deg(G) \geq \deg(G)$),
- skóre grafu.

Invariant splňující vlastnost b) je matice[1] sousednosti. Pokud jsou ale A(G) a A(G') matice sousednosti, pak jsou grafy G a G' izomorfní, pokud existuje permutační matice P pro kterou platí:

$$PA(G)P^{-1} = A(G'). \tag{9}$$

S touto podmínkou je splněna i vlastnost a).

Nalezením permutační matice tedy můžeme zjistit izomorfismus. Navíc lze pomocí matice sousedosti zjistit spektrum grafu, což je množina vlastních hodnot splňující podmínku a), například stejné minimální a charakteristické polynomy, determinant i stopu. Mezi další invarianty patří matice dosažitelnosti, matice vzdálenosti, uzlové chronické číslo, pokrytí uzlů, pokrytí vrcholů a další, jejichž vysvětlení lze najít v [Ch06] a [St08].

Podgraf H z grafu G získáme odebráním některých vrcholů a hran, platí tedy:

$$V(H) \subseteq V(G) \wedge E(H) \subseteq E(G) \tag{10}$$

Stupeň vrcholu $deg_G(v)$ v grafu G, který je obyčejným grafem, kde v je vrchol grafu G, označujeme počet hran grafu G obsahující vrchol v. U orientovaných grafů rozlišujeme vstupní $deg_+(v)$ a výstupní $deg_-(v)$ stupeň vrcholu, podle toho zda se jedná o hrany vycházející či končící v daném vrcholu. Pokud platí $deg_G(v) = 0$, je vrchol v izolovaný.

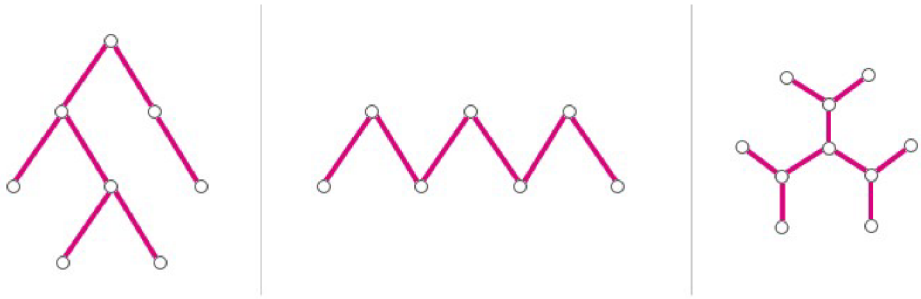
Skóre grafu je posloupnost stupňů grafu G: $deg_G(v_1)$, $deg_G(v_2)$, ... , $deg_G(v_n)$. Dvě skóre považujeme za stejná, pokud záměnou pořadí prvků jednoho z nich získáme jejich totožnost. Pokud grafy nemají stejné skóre, nejsou izomorfní.

1 Více o reprezentaci grafů pomocí matic v kapitole 3.1

Ohodnocení hrany je vlastnost, kterou můžeme přiřadit hraně grafu. Může se jednat o číslo, nebo množinu hodnot různých typů. Pro jednoduchost definice připustíme kladná reálná čísla pro ohodnocení hran. Pak pro funkci ω, která hranám přiřadí ohodnocení, platí: $\omega : E(G) \rightarrow (0, +\infty)$

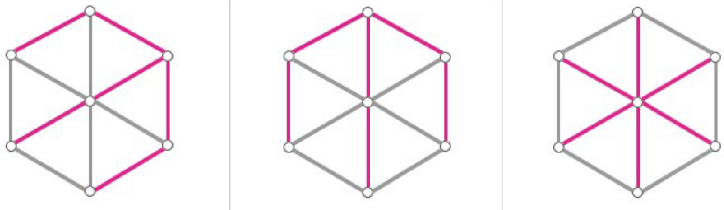
Metrika grafu je soubor vzdáleností mezi všemi dvojicemi vrcholů grafu. Jinak řečeno, metrikou grafu je matice *M*, ve které prvek *M[i,j]* udává vzdálenost mezi vrcholy *i* a *j*.

Strom je *souvislý* graf, který neobsahuje *kružnici*. Z toho plyne, že mezi každými dvěma uzly existuje cesta (graf je souvislý) a to právě jedna (neexistují kružnice). Pro strom dále platí eulerův vzorec pro stromy: $|V| = |E| + 1$. Stromy mají široké využití v informatice jako *datové struktury* pro ukládání dat. Vrchol grafu typu strom se typicky označuje jako **uzel**. Právě jeden uzel stromu je označen jako **kořen stromu**. Z něj vedou hrany k dalším vrcholům, tzv. **synům** (nebo také „dětem“). Kořen je pak pro své syny tzv. **otcem** (nebo také „rodičem“), stejně jako jsou otcem synové kořenu pro své syny. Pokud z uzlu nevedou hrany do dalších synů, pouze k otci, jedná se o tzv. **koncový uzel** (nebo také „list“). Reflexivně tranzitivními uzávěry relací otec a syn jsou **předek** a **potomek**. Základním typem stromové datové struktury je **binární vyhledávací strom**. Jde o strom, kde každý otec má maximálně dva syny a kde levý syn má menší hodnotu a pravý syn větší hodnotu. Tak lze vyhledávat podstatně rychleji, než při procházení všech prvků v seznamu. Pro reprezentaci stromu při programování se využívají dynamické datové struktury s ukazately, nikoli obvyklé matice popsané v kapitole 3.2. Příklady stromů uvádí Obrázek 2.1.



Obrázek 2.1: Příklady stromů, převzato z [Li10]

Kostra grafu *G = (V, E)* je libovolný podgraf *H*, který neobsahuje žádnou kružnici a zároveň spojuje všechny vrcholy grafu *G*. Platí, že kostrou *G* je strom *(V, E')* kde *E'* je podmnožina *E*. Příklady koster uvádí Obrázek 2.2.



Obrázek 2.2: Příklady koster grafu, převzato z [Li10]

Jádro grafu je podgraf orientovaného grafu splňující následující podmínky. Nechť $G = (V, E)$ je orientovaný graf, pak $W \subseteq V$ je jádrem grafu pokud $(u_0, u_1) \in E$ a $u_0 \in W$, pak $u_1 \notin W$. A dále platí, že jestliže $u_0 \notin W$, pak existuje $u_1 \in W$ tak, že $(u_0, u_1) \in E$.

3 Reprezentace a prohledávání grafů

Grafy lze z jejich podstaty obvykle snadno reprezentovat graficky. Typicky se reprezentují vrcholy grafu jako body nebo základní geometrické objekty, do kterých se zapíší základní informace o vrcholu, např. jeho název. Hrany mezi vrcholy jsou reprezentovány linkami nebo šipkami, které dané vrcholy spojují. Pro grafickou reprezentaci grafu není určen jednotný standard. Grafická reprezentace je vhodná především pro menší grafy, rozsáhlé grafy se stovkami vrcholů potlačí hlavní přednosti grafické reprezentace, kterými jsou přehlednost a čitelnost člověkem.

Pro potřeby matematické reprezentace či reprezentace v programech existují dva základní druhy reprezentace grafů a to spojová a maticová.

Popis pomocí matice má význam především v teoretické rovině, na rozdíl od spojové reprezentace, která je lépe využitelná v praxi. Maticová reprezentace grafu poskytuje vazbu mezi lineární algebrou a teorií grafů, čímž dostáváme prostředek pro algebraickou reprezentaci vlastností grafů. Spojová reprezentace je typická pro realizaci grafových algoritmů. Má menší paměťovou složitost a pro většinu algoritmů i lepší časovou složitost. V této kapitole vycházím především z [Ko04], [Li10], [St08].

3.1 Reprezentace grafu pro matematické výpočty

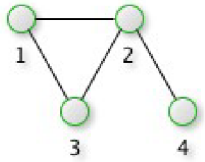
Reprezentovat graf graficky pomocí diagramu je nejsrozumitelnější pro jeho interpretaci člověkem. Často je však nutné zaznamenat graf pouze čísly, především pro exaktnost výpočtů. Pro tyto účely je možné použit matematickou definici množiny vrcholů a množiny hran. Tato varianta je vyhovující pokud se jedná o malý graf, nebo graf s nízkým počtem hran. Např.:

$$V=\{1,2,3\}, E=\{\{1,2\},\{1,3\}\}$$

Častěji se lze při řešení matematických úloh z grafy setkat se zápisem pomocí matic. Základním způsobem pro zapsání neorientovaného grafu je matice sousednosti. Vytvoříme ji tak, že označíme jednoznačnými identifikátory všechny vrcholy grafu a za rozměr matice zvolíme počet vrcholů grafu. Dále do matice zapisujeme na pozici [identifikátor vrcholu x, identifikátor vrcholu y] hodnotu 1, pokud mezi vrcholy x a y existuje hrana. Jinak zapíšeme 0. Pro $A(G)=[a_{ij}]$ tedy platí:

$$a_{ij} = \begin{matrix} 1, pokud\ \{v_i, v_j\} \in E, respektive\ (v_i, v_j) \in E \\ 0\ jinak \end{matrix} \tag{11}$$

V případě neorientovaného grafu je matice sousednosti symetrická. V případě grafu prostého je diagonála nulová, jinak ji mohou tvořit smyčky. Příklad matice sousednosti je na Obrázku 3.1.



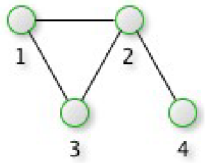
	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	0
4	0	1	0	0

Obrázek 3.1: Příklad zápisu grafu pomocí matice sousednosti

Pro řídké grafy, respektive grafy s nízkým počtem hran v poměru k počtu vrcholů, je vhodná matice incidence. Je to obdélníková matice grafu G = (V, E) o rozměru |V| x |E| a pro A(G) = [a_{ij}] platí:

$$a_{ij} = \begin{matrix} 1, pokud \{v_i, v_j\} \in E \\ 0\ jinak \end{matrix} \tag{12}$$

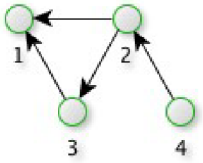
Matice znázorňuje zda daný vrchol inciduje s danou hranou. S maticí incidence nelze zaznamenat smyčky nebo násobné hrany. Příklad matice incidence je na Obrázku 3.2.



	1.2	1.3	2.3	2.4
1	1	1	0	0
2	1	0	1	1
3	0	1	1	0
4	0	0	0	1

Obrázek 3.2: Příklad zápisu grafu pomocí matice sousednosti

Výše uvedenými způsoby je možné reprezentovat neorientované grafy. Pro reprezentaci orientovaného grafu lze využít lehce modifikované předchozí techniky. V definici množiny hran pro reprezentaci orientovaného grafu striktně dodržujeme umístění vrcholu. Vrchol, ze kterého hrana vychází, je definován vždy na prvním místě a vrchol, ve kterém hrana končí, na druhém místě. V matici sousednosti označíme jeden rozměr pro vrcholy, z nichž zaznamenané hrany vychází, a v druhém rozměru jsou vrcholy, v nichž zaznamenané hrany končí. Tím pádem výsledná matice není symetrická, protože hrany zaznamenáváme (číslo 1) pouze v jejich směru. Příklad tohoto řečení ilustruje Obrázek 3.3.



odkud/kam	1	2	3	4
1	0	0	0	0
2	1	0	1	0
3	1	0	0	0
4	0	1	0	0

Obrázek 3.3: Reprezentace orientovaného grafu pomocí matice sousednosti

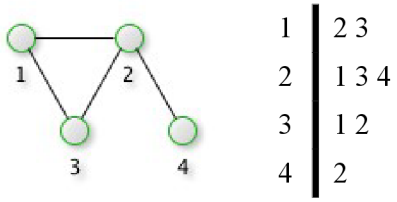
Podobně lze realizovat reprezentaci multigrafu či orientovaného multigrafu, kde se na místa hran zaznamenávají celá čísla určující počet hran, který dané vrcholy spojují.

3.2 Reprezentace grafu při implementaci programů

V programování neexistuje standardizovaný způsob reprezentace grafu, protože ten se obvykle liší podle vlastností grafu a řešených úloh. Na základě těchto vlastností pak je reprezentace zvolena. Případem, který je třeba zmínit, jsou také dynamické stromové struktury, což jsou grafy, kde se obvykle využívá sofistikovaných postupů práce s ukazately a pamětí. Typicky využívanými způsoby reprezentace grafu při implementaci programu jsou následující.

Matice sousednosti je v programu dvourozměrné pole o velikosti |V| × |V|. Tato reprezentace má prostorovou složitost $O(n^2)$ a konstantní časovou složitost pro vyhledávání.

Seznam sousednosti je nejčastěji implementován jako dynamický seznam objektů. Každý objekt reprezentuje vrchol a obsahuje dynamický seznam svých sousedních vrcholů. Má prostorovou složitost $O(|E|)$ pro graf $G = (V, E)$. Časová složitost vyhledávání je maximálně $O(n)$. Příklad seznamu sousednosti je na Obrázku 3.4.



Obrázek 3.4: Ukázka seznamu sousednosti pro neorientovaný graf

Matice sousednosti je vhodná pro reprezentaci grafů s velkým množstvím hran. Jinak lze očekávat množství redundantních dat a bývá výhodnější použít seznam sousednosti, pro jeho nižší paměťovou náročnost a často i nižší výpočetní složitost.

V kapitole 5 se budu zabývat reprezentací grafů v systémech řízení báze dat, respektive v databázích.

3.3 Způsoby prohledávání grafu

Prohledávání grafu patří mezi elementární úlohy a je často využíváno v algoritmech pro řešení dalších úloh v teorii grafů. Základní metodou prohledávání je tzv. prohledávání do šířky. Vycházím zejména z [Ko04].

3.3.1 Prohledávání do šířky

Prohledávání v grafu G = (V, E) probíhá tak, že se systematicky procházejí všechny hrany z vyznačeného vrcholu *s*, s cílem najít každý vrchol, který je z *s* dosažitelný. Zároveň se zjistí vzdálenost ke každému z *s* dosažitelnému vrcholu a získá se tak **strom hledání do šířky, tzv. BF-strom** obsahující všechny z *s* dosažitelné vrcholy. Pro všechny uzly v tomto stromu je pak cesta ke kořenu *s* nejkratší cestou k tomuto vrcholu v grafu G. Jedná se tedy i o strom nejkratších cest z *s* do dosažitelných vrcholů. Hranice vrcholů, které již byly navštíveny, se rovnoměrně posouvá na celé své *šířce*. Tím pádem jsou navštíveny vrcholy ležící ve vzdálenosti *k* od vrchlu s dříve, než první uzel ve vzdálenosti *k + 1*.

V průběhu prohledávání je každý vrchol označen nejprve jako FRESH, potom jako OPEN a nakonec jako CLOSED. Při startu jsou všechny vrcholy FRESH. Právě navštívený vrchol je označen jak OPEN. Jako CLOSED je označen uzel, který už nepatří do hranice mezi prohledanými a neprohledanými, tedy ten, který už zaručeně nemá žádného nového souseda.

Z myšlenky prohledávání grafu do šířky vychází další podstatné algoritmy jako například Dijkstrův algoritmus pro nalezení nejkratší cesty mezi dvěma vrcholy nebo Jarníkův-Primův pro nalezení nejmenší kostry grafu.

Algoritmus prohledávání do hloubky je implementován v experimentální části této práce, proto uvedu i jeho pseudokód na základě kterého jsem algoritmus implementoval. Čerpám z [Ko04]:

```
BFS(G, s)
for každý uzel u ∈ U −{s}
  do stav[u] := FRESH
    d[u] := ∞
    p[u] := NIL
stav[s] := OPEN
d[s] := 0
p[s] := NIL
INIT QUEUE; ENQUEUE(s)
while not EMPTY QUEUE do
  u := QUEUE FIRST
  for každé v ∈ Adj[u] do
    if stav[v] = FRESH then
      stav[v] := OPEN
      d[v] := d[u] + 1
      p[v] := u
      ENQUEUE(v)
  DEQUEUE
  stav[u] := CLOSED
```

Algoritmus tedy probíhá tak, že všechny uzly mimo *s* označí jako nové, zatím nedosažitelné z *s* a bez předchůdce. Samotný uzel *s* otevře s nulovou vzdáleností od *s*, rovněž bez předchůdce a uloží jej do fronty. Dokud fronta není prázdná, vybírá z ní první otevřený uzel *u* a jeho sousedy, kteří jsou noví, otevírá je, určuje jim vzdálenost i předchůdce a ukládá do fronty. Poté odebere daný uzel *u* z fronty, zavře jej a opakuje předchozí s dalším otevřeným uzlem ve frontě.

Časová složitost prohledávání grafu do šířky je $O(|V|+|E|)$.

3.3.2 Prohledávání do hloubky

Jedná se o analogii k prohledávání do šířky, celkového průchodu grafem je také dosaženo obdobnou metodikou. Při prohledávání je nutné zajistit, abychom neprohledávali stejný vrchol vícekrát, proto jsou navštívené vrcholy označovány identifikátorem předchozího prohledaného vrcholu. Prohledávání pak probíhá tak, že algoritmus přechází z aktuálně prohledávaného vrcholu do dalšího, který ještě není označen jako prohledaný. Pokud jsou v okolí aktuálního uzlu všechny uzly označené jako prohledané, vrací se algoritmus do předchozího uzlu, ze kterého na daný vrchol původně vstupoval a snaží se najít a navštívit zatím neprohledané vrcholy v jeho okolí. Tento proces končí, pokud se algoritmus vrátí do vrcholu, ve kterém bylo prohledávání zahájeno a pokud jsou všechny okolní vrcholy kolem výchozího označeny jako prohledané. Při prohledávání do hloubky vzniká tzv. DF-les vycházející z okolních vrcholů výchozího vrcholu a každá jeho komponenta je tzv. DF-strom. V závislosti na implementaci je možné prohledávané vrcholy označovat vhodným způsobem pro pozdější zpracování jinými algoritmy. Princip prohledávání do hloubky je využitý také ve VF2 algoritmu implementovaném v rámci této práce. Proto uvádím i pseudokód prohledávání do hloubky, který čerpám z [Ko04]:

```
DFS(G)
  for každý uzel u ∈ U
    do stav[u] := FRESH
      p[u] := NIL
  i:=0
  for každý uzel u ∈ U
    do if stav[u] = FRESH
      then DFS-Projdi(u)

DFS-Projdi(u)
  stav[u] := OPEN
  i := i+1; d[u] := i
  for každý uzel v ∈ Adj[u]
    do if stav[v] = FRESH
      then p[v] := u
        DFS-Projdi(v)
  stav[u]:= CLOSED
  i := i + 1; f[u] := i
```

Algoritmus nejprve nastaví všechny uzly na nové a bez předchůdce. Nastaví čítač značek na nulu a pak zkouší všechny uzly a pro nové prochází jejich potomky do hloubky. Prohlásí uzel *u* za otevřený a přidělí mu první značku. Projde hranu *(u, v)* a pro nový uzel *v* stanoví za jeho předchůdce *u* a projde potomky uzlu *v*. Pokud je procházení uzlu *u* dokončeno, dá mu koncovou značku.

4 Problémy v teorii grafů

V této kapitole budou rozebrány vybrané nejznámější problémy v teorii grafů. Zabývám se především hledáním cest v grafech, barvením grafů, toky v sítích a na závěr uvádím příklad sofistikovanějšího problému, kterým je hledání izomorfismu v grafu. V této kapitole čerpám a vycházím z [BM82], [Hl07], [Ch06], [Ko04], [Li10], [St08]. Některé z algoritmů byly implementovány v experimentální části této práce a to barvení grafu a hledání izomorfismu v grafu.

4.1 Hledání cest v grafech

V této kapitole rozeberu nejznámější problémy týkající se hledání cest v grafu podle stanovených podmínek. Tato problematika má nespočet aplikací v praxi, například v systémech vyhledávajících spoje ve veřejné dopravě, ve spedicích nákladní dopravy nebo pro hledání nejkratších cest v mapě, například v GPS navigacích v automobilech.

4.1.1 Hledání nejkratší cesty

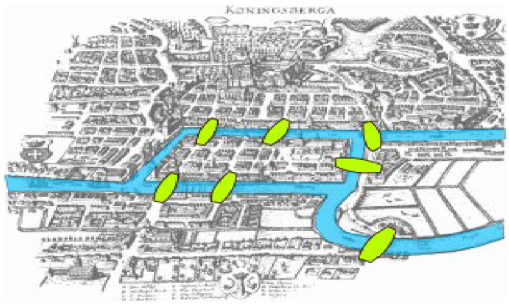
Jedná se o NP-úplný problém, při kterém se hledá cesta mezi dvěma vrcholy, kde součet ohodnocení hran je nejnižší možný. **Floyd-Warshallův algoritmus** je nejjednodušším algoritmem pro nalezení nejkratší cesty. Spočívá v porovnávání všech možných cest mezi dvěma vrcholy a výběru nejkratší z nich. Algoritmus postupně vylepšuje odhad na nejkratší cestu, až je jisté, že je optimální. Velkým problémem tohoto algoritmu je jeho výpočetní složitost. Z tohoto důvodu výhodnější je například Dijkstrův algorimus, jehož výpočet je rychlejší, ovšem je náročnější na implementaci. Dalšími algoritmy řešícími tento problém jsou A* vyhledávací algoritmus, Bellman-Fordův algoritmus a Johnsonův algoritmus.

4.1.2 Minimální kostra

Neformálně řečeno, úloha nalezení minimální kostry popisuje jak „co nejlevněji“ spojit všechny vrcholy grafu, neboli jak spojit všechny vrcholy s použitím hran minimálního ohodnocení. Hledání minimální kostry je možné pouze u grafů s ohodnocenými hranami. Obvyklým a relativně snadným algoritmem pro nalezení minimální kostry grafu je Kruskalův algoritmus. Provádí spojování vrcholů hranami s nejmenším ohodnocením dokud nejsou spojeny všechny vrcholy. Dalšími algoritmy pro řešení tohoto problému jsou Borůvkův algoritmus a Jarníkův algoritmus. Obdobně může být někdy užitečné **nalezení maximální kostry grafu**.

4.1.3 Sedm mostů Königsbergu

Úloh „cestovních problémů“ je mnoho. Pro zajímavost uvádím i „historickou úlohu“ teorie grafů. Jejím zadáním je, zda je možné přejít všechny mosty tehdejšího města Königsberg tak, že se přes každý půjde právě jednou. Viz obrázek 4.1. Euler dokázal, že to není možné.



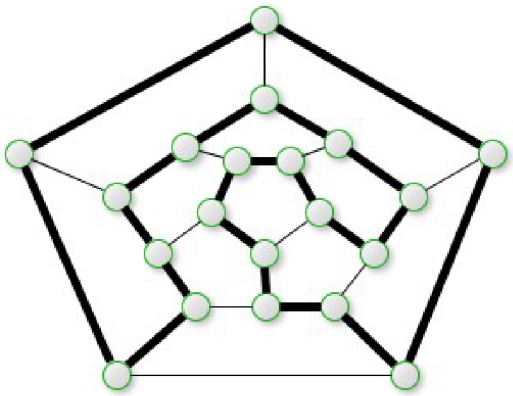
Obrázek 4.1: Sedm mostů Königsbergu, převzato z [Obr1]

Zavedl tak pojem eulerovský tah, což je tah, který obsahuje každou hranu grafu právě jednou. Pokud v grafu existuje uzavřený eulerovský tah, je označován jako eulerovský graf. Tzn. neformálně řešeno ho lze graficky realizovat „jedním tahem“.

Je-li $G = (V, E)$ neorientovaný graf a $P=(v_0, e_1, v_1, \ldots, e_n, v_m)$ posloupnost, pro kterou platí, že $|E|=n$ a $\forall i, j=0,\ldots,n, i \neq j : e_i \neq e_j$, nazveme tuto posloupnost eulerovským tahem. Pokud navíc platí, že $v_0 = v_m$, jedná se o uzavřený eulerovský tah.

4.1.4 Hamiltonovská kružnice

Hledání hamiltonovské kružnice je NP-úplný problém. Jedná se o cestu, která právě jednou projde všechny vrcholy grafu a při tom existuje hrana mezi prvním a posledním vrcholem cesty. Pokud mluvíme o hamiltonovském grafu, znamená to, že v něm lze tuto kružnici najít. Příklad hamiltonovské kružnice je na Obráku 4.2.



Obrázek 4.2: Hamiltonovská kružnice

Přesto, že je úloha obdobná s eulerovskými grafy, je rozhodnutí o hamiltonovském grafu zatím nevyřešeným problémem. Není totiž známá žádná jednoduchá, nutná a postačující podmínka k tomu, abychom graf mohli označit jako hamiltonovský. Stejně tak zatím neexistuje algoritmus pro nalezení hamiltonovské kružnice v grafu. Známo je několik podmínek postačujících k tomu, aby byl graf hamiltonovský. Pro graf s $u \geq 3$ uzly jsou to následující:

- Diracova podmínka: Každý uzel má stupeň alespoň $\frac{1}{2} u$.
- Oreho podmínka: Každá dvojice uzlů nespojených hranou má součet stupňů alespoň u.
- Pósova podmínka: Pro každé přirozené číslo $k < \frac{1}{2} u$ je počet uzlů, jejichž stupeň nepřevyšuje k, menší než k.

Problematika hamiltonovské kružnice a grafu je tedy příkladem oblasti, která stále nabízí otevřený prostor pro další výzkum.

4.1.5 Problém obchodního cestujícího

Úloha je zobecněním hledání Hamiltonovské kružnice a jedná se o NP úplný problém. Má časté využití v praxi a pro její řešení se využívají heuristické nebo genetické algoritmy poskytující přibližné výsledky. Jedná se o řešení úlohy nad prostým neorientovaným grafem $G = (V, E)$, s ohodnocením hran $w: H \rightarrow R^+$, kdy je třeba nalézt hamiltonovskou kružnici s minimální w-délkou. Název úlohy je odvozen od typického transportního problému, kde je určitý počet míst potřeba projet co nejkratší okružní jízdou a každé z míst navštívit právě jednou. Heuristické algoritmy vužívané pro řešení tohoto problému obvykle neposkytují žádné záruky toho jak moc se liší od optimální cesty. Dovedou to však polynomální algoritmy jako třeba Christofidův algoritmus, který se zárukou najde cestu, která je nejhůře o polovinu delší.

4.2 Barvení grafu

Původním zadáním problému tohoto typu je obarvit libovolnou mapu států čtyřmi barvami tak, aby žádné dva sousední státy nebyly obarveny stejnou barvou. Předložil ho roku 1852 Francis Guthrie a vyřešil ho až v roce 1977 Kennethem Appelem a Wolfgangem Hakenem s pomocí tehdy dostupné výpočetní techniky. Při řešení problému bylo do teorie grafů zavedeno množství podstatných konceptů.

Pro řešení zmíněné úlohy **čtyř barev** se používá princip, kdy každý stát bude reprezentovat jeden vrchol grafu a státy spolu sousedící budou spojeny hranou. Jednotlivým vrcholům místo názvu státu nebo barvy přiřazujeme čísla reprezentující barvu. Tak lze problém formulovat matematicky:

Nechť $G = (V,E)$ je graf, k přirozené číslo. Zobrazení $b: V \rightarrow \{1,2,...,k\}$ nazveme obarvením grafu G pomocí k barev, pokud pro každou hranu $\{x,y\} \subseteq E$ platí $b(x) \neq b(y)$.

Důkaz, že lze libovolný graf obarvit čtyřmi barvami je matematicky náročný, byl proveden pomocí počítače a jeho popis lze nalézt v [RS97].

Určité jednoduché grafy je možné obarvit i menším počtem barev než čtyřmi. Počtem nutných barev pro obarvení daného grafu je tzv. **chromatické číslo**.

Mezi známé úlohy problému barvení grafu patří již výše zmíněný **problém čtyř barev**, dále **Erdős-Faber-Lovászova hypotéza**, která je vysvětlena například v [CL88] a zatím nebyla dokázána. **Úplné obarvení** je úloha, kde musí mít jinou barvu všechny sousedící vrcholy, všechny hrany, které se dotýkají v daném vrcholu a také vrchol oproti všem hranám, které z něj vychází nebo v něm končí. Počet potřebných barev pro úplné obarvení je **úplným chromatickým číslem**. Tato úloha zatím nebyla vyřešena.

Mezi nejznámější algoritmy pro barvení grafu patří greedy coloring, DSATUR, RLF, IBSC, BSC.

V experimentální části této práce implementuji dva algoritmy pro obarvení vrcholů grafu. Barvy jsou v algoritmech abstrahovány na číselné hodnoty. Nejtriviálnějším a nejrychlejším algoritmem pro barvení grafu je tzv. greedy coloring, jehož princip je následující. Algoritmus prochází sekvenčně všechny vrcholy grafu a zjišťuje obarvení sousedních vrcholů. Danému vrcholu potom přiřazuje barvu $k+1$, kde k je nejvyšší obarvení některého ze sousedů právě barveného vrcholu. Přesto, že je algoritmus velmi rychlý, nedosahuje v porovnávání s ostatními dobrých výsledků. S velmi nízkou ztrátou rychlosti barvení lze algoritmus modifikovat a jeho výsledky alespoň částečně zlepšit. A to sice například tak, že barvené uzly seřadíme sestupně podle počtu jejich sousedů. Jsou pak tedy obarveny nejprve uzly s větším počtem sousedů.

S kvalitou obarvení (nižší počet použitých barev) roste u algoritmů pro barvení grafu složitost a časová náročnost. Vzhledem k tomu, že testy v experimentální části této práce probíhají nad grafem wikipedie, který je velmi rozsáhlý, zvolil jsem jako druhý algoritmus pro úlohu barvení kompromis a to sice RLF algoritmus, který dosahuje uspokojivých výsledků v přijatelném čase. Při výběru tohoto algoritmu jsem vycházel z [CA]. Při implementaci tohoto RLF algoritmu jsem vycházel především z [RLF]. Uvádím zde pseudokód tohoto algoritmu:

```
    colornumber = 0; //number of used colors
    while (number_of_uncolored_vertex > 0) do
      determine a vertex x of maximal degree in G;
      colornumber = colornumber + 1;
      F(x) = colornumber;
      NN = set of non-neighbors of x;
      while (cardinality_of_NN > 0) do
        //Find y in NN to be contracted into x
        maxcn = 1; // becomes the maximal number of common
neighbors
        ydegree = 1; // becomes degree(y)
        for every vertex z in NN do
          cn=number of common neighbors of z and x;
          if (cn > maxcn or (cn == maxcn and degree(z) < ydegree))
then
            y = z;
            ydegree = degree(y);
            maxcn = cn;
          end if;
        end for;
        if (maxcn == 0) then //in this case G is unconnected
          y=vertex of maximal degree in NN;
        end if;
        F(y) = colornumber;
        contract y into x;
        update the set NN of non-neighbors of x;
      end while;
      G = G - x; //remove x from G
    end while;
```

Barvení je realizováno ve smyčce, která probíhá dokud nejsou obarveny všechny vrcholy grafu. Barvení, respektive průchod hlavní smyčkou, začíná výběrem neobarveného vrcholu *u* s nejvyšším počtem sousedních vrcholů. Tento vrchol je obarven a následně je zjištěna množina uzlů, které nesousedí s vrcholem *u* ani s jiným vrcholem stejné barvy jako *u*. Tato množina je nazvána NN. Z této množiny je vybírán vrchol *s*, který má s aktuálním vrcholem *u* nejvíce společných sousedících vrcholů, případně vrchol s největším počtem sousedních vrcholů (pokud nejsou nalezeni společní sousedé *u* a *s*). Vrchol *s* je obarven stejnou barvou jako *u* a výše uvedený proces je opakován, dokud je do množiny NN zjištěn alespoň jeden vrchol. Pokud je po zjištění množiny NN tato prázdná, dochází k opakování celého hlavního cyklu, tzn. výběr nového uzlu *u* a probíhá barvení novou barvou. Tento algoritmus je implementován v experimentální a demonstrační části této práce.

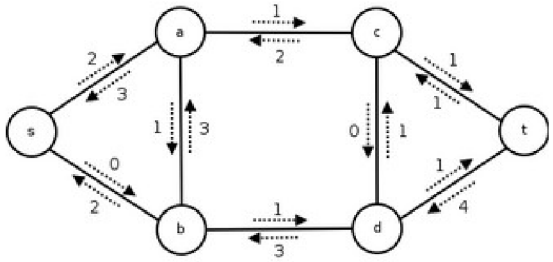
4.3 Toky v sítích

Toky v sítích patří mezi nejvíce aplikované části teorie grafů. Nalézají široké uplatnění v informatice, dopravě a dalších oborech. Modelem bývá orientovaný graf reprezentující síť (informační, dopravní, vodovodní, plynovou apod.), přes kterou je přepravováno médium. Omezení přepravní kapacity sítě reprezentuje ohodnocení hran modelového grafu a směr toku orientace hrany. Tok v každé hraně je uvažován beze ztrát. Pokud se jedná o zápornou velikost toku, jedná se o tok proti původnímu směru.

Typickými úlohami toku v sítích je tedy například zjišťování nejlevnějšího toku, propustnosti, minimálního toku a podobně. Tyto úlohy lze řešit v jakékoli síti, kterou lze abstrahovat na orientovaný graf s ohodnocenými hranami. Pro studium toků bylo významné vydání [FF62], na základě kterého pak bylo formulováno a řešeno mnoho úloh tohoto typu.

V síti můžeme rozeznat **tok od zdroje ke spotřebiči** a **cirkulaci**, který znázorňuje obrázek 4.3. Pokud se jedná o cirkulaci, je splněn Kirchhoffův zákon pro všechny vrcholy. U toku od zdroje ke spotřebiči platí Kirchhoffův zákon pro všechny vrcholy kromě zdroje a spotřebiče.

Síť je definována jako čtveřice $S = (G, q, s, t)$, kde $G = (V, E)$ je obyčejný orientovaný graf, $s \in V$ je zdroj sítě, $t \neq s$ je spotřebič sítě S a $q : E \rightarrow R^+$ je nezáporné hodnocení hran, tzv. kapacita sítě S. Pro každou hranu je definována kapacita hrany $q(e)$.



Obrázek 4.3: Příklad sítě a toků v síti, převzato z [Obr2]

Více informací o tocích v síti je možné najít v [Ko04].

4.4 Izomorfismus

Pojem izomorfismus grafů byl zmíněn již v kapitole 2.2. Při hledání izomorfismu jde o to zjistit, zda jsou dva grafy $G = (V, E)$ a $G' = (V', E')$ izomorfní, respektive pokud se vyjádřím velmi neformálně, zda lze u jednoho grafu přejmenovat a přemístit vrcholy tak, aby odpovídal druhému. Jedná se o problém třídy NP. Zatím nebylo dokázáno, že je i NP-úplný. Jeho řešení klade vysoké paměťové a časové nároky. Nejtriviálnějším způsobem zjištění izomorfismu je permutace všech vrcholů V a V', zde však složitost řešení exponenciálně roste s přibývajícími vrcholy. Sofistikované algoritmy pro zjištění izomorfismu většinou používají a kombinují následující metody:

- Invarianty grafu, pomocí kterých lze snadno a rychle rozhodnout, že dva grafy izomorfní nejsou.
- Rozkladem a vytvořením matice vzdáleností lze graf rozložit na spojité podgrafy a omezit počet permutací.
- Dále se využívá nalezení cyklů nebo klik grafu, algoritmů umělé inteligence apod.

Izomorfismus je možné zjistit permutací vrcholu grafu, respektive nalezením permutační matice, což je však vhodné pouze pro malé grafy. Stejně vysokou složitost má i hledání izomorfismu pomocí kanonické formy. Mezi nejznámější použitelné algoritmy pro zjištění izomorfismu patří Ullmannův algoritmus. Pro použití na rozsáhlých grafech je pak vhodný především VF algoritmus nebo jeho zlepšená verze VF2, který vychází z Ullmannova. Podrobně se zjišťováním izomorfismu zabývá [Ch06] a konkrétně těmito algoritmy [St08]. V experimentální části této práce jsem v rámci testů svých řešení implementoval algoritmus VF2, který je vhodný pro hledání izomorfismu grafů i pro hledání izomorfismu grafu a podgrafu grafu. A to především pro velmi rozsáhlé grafy, kterým graf wikipedie je (tisíce vrcholů a hran). Při zkoumání a popisu algoritmu jsem vycházel především z [VF2]. Porovnávání dvou grafů $G_1 = (V_1, E_1)$ a $G_2 = (V_2, E_2)$ v algoritmu VF2 se skládá ze zobrazení M, které přiřazuje vrcholy grafu G_1 k vrcholům grafu G_2 a opačně podle daných omezení. Uvádím relativně abstraktní pseudokód a následně podrobnější popis, který čerpám z [VF2]:

```
PROCEDURE Match(s)
  INPUT: přechodný stav s; počáteční stav s0 kde M(s0)=Ø
  VÝSTUP: mapování mezi dvěma grafy
  IF M (s) pokrývá všechny vrcholy grafu G2 THEN
    OUTPUT M(s)
  ELSE
    Vypočítej množinu možných párů P(s) pro zahrnutí do M(s)
    FOREACH p ∈ P(s)
      IF pravidla proveditelná pro zahrnutí p fo M(s) THEN
        Zpracuj stav s' získaný přidáním p do M (s)
        CALL Match(s')
      END IF
    END FOREACH
    Obnovení předchozích hodnot polí při návratu z rekurze
  END IF
END PROCEDURE Match
```

Zobrazení M je vyjádřeno jako množina párů (n, m), kde $n \in G1$ a $m \in G2$, reprezentující zobrazení vrcholu n na vrchol m. Zobrazení $M \subset V1 \times V2$ je nazváno izomorfismem podgrafu pokud M je izomorfismem mezi G_2 a podgrafem grafu G_1.

Proces hledání funkce zobrazení může být popsán stavovým prostorem. Každý stav s ze zobrazení může být přiřazen k částečnému řešení zobrazení $M(s)$ což je podmnožina zobrazení M. $M(s)$ jednoznačně identifikuje podgrafy grafů G_1 a G_2, neboli $G_1(s)$ a $G_2(s)$ získané výběrem vrcholů z G_1 a G_2, které jsou obsaženy v $M(s)$ a spojeny hranami.

Na základě této definice pak přechod ze stavu s do stavu s' reprezentuje přírustek k části grafu asociovaného s s ve stavovém prostoru pro zobrazení vrcholů (m,n).

Mezi všemi stavy stavového prostoru vede pouze malé množství k nalezení izomorfismu. Existují však situace, které předem vylučují jeho nalezení. Proto VF2 zavádí množinu podmínek, které zajišťují, že jsou generovány pouze stavy, ve kterých je možné izomorfismus najít. Pomocí nich se počet generovaných stavů značně omezuje. Jsou označovány jako „omezující podmínky“

a definovány jednotnou formou funkcí *F(s, n, m)*, která vrací pravdivostní hodnotu *true* v případě, že přidání zobrazení *(n, m)* ke stavu *s* splňuje všechny předpoklady zaručující možnost nalezení izomorfismu. Konečný stav je potom izomorfismem mezi G_1 a G_2 nebo izomorfismem podgrafu mezi podgrafem z G_1 a grafem G_2. Omezující podmínky lze dále rozdělit na syntaktická omezení a sémantická omezení. Syntaktická omezení jsou závislá na struktuře vstupního grafu, sémantická na jeho atributech.

V počátečním stavu s_0 zobrazení neobsahuje žádný prvek, tzn. $M(s_0) = \emptyset$. Pro každý následující stav algoritmus určí množinu *P(s)* s páry vrcholů, které je možné přiřadit k současnému stavu *s*. Pro každý pár $p = (n, m)$, který patří do *P(s)* jsou vyhodnoceny omezující podmínky a pokud jsou splněny, tzn. *F(s, n, m)* je *true*, pak je vyhodnocen následující stav $s' = s \cup p$ a celý proces je rekurzivně aplikován na stav s'.

Poznamenejme, že algoritmus prohledává stavový prostor grafu na základě prohledávání do hloubky, které popisuji v kapitole 3.3.2. Zjednodušeně řečeno, stavy mohou být dosaženy různými cestami a proto se ošetřuje možnost opětovného generování stejných stavů.

Výpočet kandidátů *P(s)*, které je možné zahrnout do současného stavu *s*, nejprve uvažuje vrcholy přímo spojené s $G_1(s)$ a $G_2(s)$. Nechť pak $T_1^{out}(s)$ a $T_2^{out}(s)$ jsou množiny uzlů, které ještě nejsou v parciálním zobrazení, jedná se o koncové vrcholy hran incidujících s $G_1(s)$ a $G_2(s)$. Analogicky pak $T_1^{in}(s)$ a $T_2^{in}(s)$. Množina *P(s)* je tvořena všemi páry *(n, m)*, kde *n* náleží $T_1^{out}(s)$ a *m* náleží $T_2^{out}(s)$, pakliže není některá z těchto množin prázdná. Stejně tak množina *P(s)* obsahuje dvojice náležící $T_1^{in}(s)$ a $T_2^{in}(s)$.

Při svém chodu algoritmus využívá datových struktur typu jednorozměrných polí o velikosti odpovídající počtu vrcholů G_1 (3 pole) respektive G_2 (3 pole). Ve dvou polích se uchovává aktuální zobrazení vrcholů, zbývající tvoří množinu možných párů *P(s)* a udržují aktuální hodnotu hloubky v prohledávání stavového prostoru náležící stavu, ve kterém vstoupil vrchol do dané množiny. Tato pole jsou sdílena všemi stavy rekurze a tím je dosaženo toho, že paměťová je dána počtem vrcholů v grafech G_1 a G_2. Prohledávání do hloubky pak zajišťuje, že v paměti může být maximálně *N* stavů v jednom okamžiku, kde *N* je počet vrcholů. Paměťová složitost je tedy *O(N)*.

5 Grafy v databázích

Databáze, neboli také datové základny či systémy řízení báze dat, jsou uspořádané soubory dat uložené na paměťových mediích se softwarovými prostředky pro operace s daty.

Od počátku vývoje databází, kdy se zprvu řešil jen problém specifikace struktury dat v souboru na paměťovém médiu, vzniklo mnoho databázových modelů založených na matematických strukturách a principech a s různou úrovní abstrakce.

Mezi základní databázové modely, patří následující:

- Plochý (Flat) model – jedná se o dvoudimensionální pole, respektive tabulku, kde je každé části dat přiřazen unikátní klíč. Položky v jednom sloupci mají stejný datový typ a položky v jednom řádku jsou ve vzájemném vztahu.
- Hierarchický model – data jsou organizována do stromové struktury. Každé části dat je přiřazen identifikátor předchůdce (nadřazeného prvku) a položky v téže úrovni jsou seřazeny.
- Síťový model – data jsou organizována do spojové struktury. Každá položka dat obsahuje záznamy obsahující data a množinu identifikátorů položek, které jsou s aktuální položkou ve vzájemném vztahu. Jedná se tedy o graf. Z tohoto modelu databáze také vychází tzv. grafové databáze, které rozebírám v kapitole 5.3.
- Relační model – data jsou ogranizována do relací, obvykle označovaných jako tabulky, viz dále.
- Dimensionální model – je specializovanou formou relačního modelu a určený pro datové sklady a OLAP dotazování.

Další případné modely (objektový, hvězdice apod.) pouze kombinují nebo modifikují některé z předcházejících modelů.

Nejznámější a nejrozšířenější je stále relační model databáze. Dále existuje mnoho modelů, které jsou dnes považovány za slepé cesty vývoje, nebo jsou velmi specializovány pro konkrétní účely (datové sklady apod.). Především díky rozvoji internetu a potřebě ukládat velké objemy dat s předem známými nároky na styl vyhledávání v nich, vznikají databázové systémy, které jsou dnes označovány NoSQL neboli Not Only SQL. V kontextu grafů je podstatný grafový model databáze, který přímo vychází z teorie grafů. Grafové databáze jsou součástí trendu NoSQL databází. Jsou vhodné pro práci s daty, kde vztahy mezi jednotlivými záznamy hrají nejpodstatnější roli.

V této kapitole čerpám především z [AN10] a [AG05].

5.1 Relační model databáze

Relační model databáze je nejstarším obecně známým, začal vznikat již v šedesátých letech dvacátého století. Od té doby do dneška jsou systémy tohoto typu suverénně majoritním prostředkem pro správu dat. Nebudu se zde tedy zabývat popisem relačních databází a vysvětlováním základních pojmů, protože předpokládám jejich implicitní znalost. Použitou terminologii a další informace o relačních databázích lze najít např. v [CB89].

Aby byly operace v relačních databázích dostatečně rychlé, používají se databázové indexy. Pomocí indexů, které jsou typicky spravovány jako binární stromy, lze najít požadované záznamy v nesrovnatelně kratší době (pro n záznamů v $O(log_2 n)$), než kdybychom procházeli databázi lineárně záznam po záznamu (pro n záznamů až v $O(n)$). Díky indexům lze v relačních databázích spravovat téměř neomezené objemy běžných[2] dat a provádět vyhledávání a spojování tabulek v *uspokojivě* krátkých časech.

Grafová data lze do relačních databází uložit mnoha způsoby, nicméně při vzrůstajícím objemu dat může docházet k jejich neefektivnímu procházení, protože v relační databázi nejsou implicitně reprezentována ve své přirozené spojové podobě. Typickým příkladem způsobu uložení grafových dat mohou být *osoby, mezi kterými jsou vztahy přátelství*. Modelem je tedy tabulka osob *persons*, kde každá osoba má ID a jméno, z pohledu grafu každá z osob reprezentuje jeden vrchol. Mezi osobami jsou vztahy přátelství, které reprezentuje tabulka *friends* obsahující sloupce ID osoby *person_id* a ID přítele *friend_id*. Z pohledu grafu se jedná o hrany. Více na toto téma lze najít v [GvS]. Pokud naším zájmem budou data grafové struktury a pokud dosáhnou vysokého objemu, nastává situace, kdy je při dotazech nutné spojování tabulek, i přes vhodné indexování, výrazně zpomalující operace.

Při vysokém objemu grafových dat a potřebě charakteristických sofistikovaných dotazů (např.: zjištění nejkratší cesty mezi dvěma městy, izomorfismu a podobně) je tedy výše popsaná normalizace grafových dat do tabulek *naivní*. Proto komerční databáze jako například Oracle nabízejí vhodná rozšíření pro práci s grafovými daty, jako například Network Data Model (NDM) jako součást Oracle Spatial. Tzn. zavedení přímé podpory na implementační úrovni pro uvažovanou specializaci dat.

Cílem této a následující diplomové práce je navrhnout a realizovat základ obdobného rozšíření pro PostgreSQL databázi, viz. Kapitola 6.

2 Účetní systém, inventář, rezervační systém a podobně.

5.2 NoSQL

Další databázové modely, které jsou dnes často využívány, jsou označovány jako NoSQL databáze. To je současné označení pro databáze, které se liší od standardního relačního modelu. Nemusí obsahovat tabulky nebo operace spojování tabulek při dotazech a mohou například efektivněji řešit problematiku horizontálního rozšiřování databáze. Typicky využívají jiný způsob dotazování dat než jazyk SQL. Vědecké články tyto typy databází obvykle označují jako strukturovaná úložiště. V trendu NoSQL databází lze vnímat následující kategorie.

Key/value úložiště jsou projekty vycházející z databáze Dynamo od firmy Amazon ukládající všechna data do globálního úložiště se strukturou obdobnou hašovací tabulce, respektive plochému modelu databáze. Více v [GD07]. Návrh je plně podřízen rychlému vyhledávání v extrémně velkých objemech dat. Dalšími projekty tohoto typu jsou Dynomite, Veldemort, Tokyo.

BigTable je skupina projektů vycházejících ze stejnojmenné databáze vytvořené Googlem. Data se ukládají do jedné velké tabulky a jednotlivým typům záznamů jsou relevantní určité skupiny sloupců, respektive plochému modelu databáze. Návrh je plně podřízen rychlému vyhledávání v extrémně velkých objemech dat. Známými projekty tohoto typu jsou Hbase, Hypertable, Cassandra.

Key/value a Bigtable jsou východiskem prakticky pro všechny aktuální projekty v oblasti NoSQL databází a z jejich myšlenek se vždy nějakým způsobem vychází. Jedná se prakticky vždy o plochý databázový model. Key/value a Bigtable, nebo kombinace těchto přístupů, používají téměř všechny majoritní webové služby jako Twitter, Google (zahrnující všechny jeho služby), Facebook a podobně. Ačkoli se NoSQL databáze liší v návrhu a struktuře, spojuje je snaha řešit obdobnou skupinu problémů.

Dokumentové databáze jsou řešení skladující libovolně strukturované dokumenty, nad kterými se tvoří *views*, obvykle pro každý opakující se dotaz. *Views* zajišťují rychlý přístup k dokumentům bez ohledu na rozsah databáze, libovolné *ad-hoc* dotazy jsou však problematické. Vhodné využití naleznou v úzkém napojení na koncept objektově orientovaného programování, kde jednotlivé záznamy mohou snadno reprezentovat samotné objekty. Nejznámějšími zástupci jsou Apache CouchDB a MongoDB.

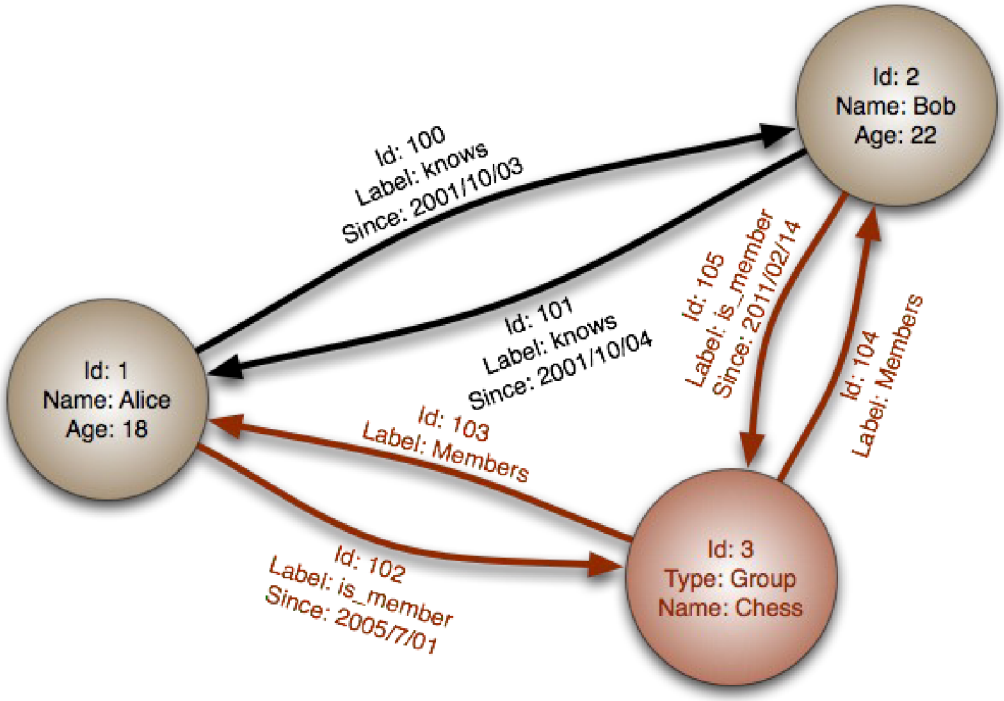
Grafové databáze jsou kategorií, kterou diskutuji v následující kapitole.

Dále existují další především experimentální projekty, které se pokoušejí využívat nové přístupy pro ukládání dat a opět více či méně vycházejí z předchozích kateogorií. Zmíním např. Gladius DB, ScarletDME, db4o apod.

5.3 Grafový model databáze

Grafové databáze jsou vhodné pro práci s daty, kde **vztahy mezi jednotlivými záznamy hrají nejpodstatnější roli**. Tyto databáze vycházejí ze síťového modelu.

Na rozdíl od relačního databázového modelu nejsou data rozdělována do tabulek, ale tvoří rozsáhlou spojitou strukturu, respektive graf. Jednotlivé vrcholy jsou spojeny přímými referencemi reprezentujícími hrany. To také způsobuje náročné rozdělování dat, například při potřebě rozdělení na více počítačů, při horizontálním rozšiřování databáze. (Ovšem tento problém je třeba řešit i u relačního modelu databáze modelujícího graf *naivním* způsobem.) I přes nevýhodu náročného rozdělení dat na více počítačů, je zde zřejmá hlavní výhoda tohoto řešení a to **konstantní čas přístupu k sousedním uzlům** v grafu a to bez ohledu na rozsah grafu.



Obrázek 5.1: Struktura dat v grafové databázi, převzato z [GDW]

Vrcholy grafu obvykle nesou určité charakteristiky, např. ID a jméno osoby kterou reprezentují, stejně tak hrany mohou mít charakteristiky jako propustnost, délku a podobně. To je znázorněno na Obrázku 5.1. Charakteristiky vrcholů a hran jsou obvykle uloženy v hašovací tabulce u každého vrcholu, respektive hrany. Tyto charakteristiky, podle kterých vyhledáváme vrchol či hranu našeho zájmu, jsou vyhledány stejným způsobem jako v relačních databázích s použitím databázového indexu. Index může být realizován externím systémem, vhodným například pro fulltextové vyhledávání, nebo vestavěným indexovacím stromem. Graf sám o sobě může být také chápán jako

index. Jakmile je pomocí indexovaného vyhledávání nalezen požadovaný vrchol, probíhá zjišťování dalších sousedních vrcholů v konstantním čase.

Právě přechody do sousedních uzlů v konstantním čase jsou hlavním benefitem grafového modelu databáze a dramaticky urychlují řešení většiny problémů teorie grafů.

V [AG05] je prezentováno několik řešení grafového modelu databáze. Představiteli grafových databází jsou následující projekty, více viz [GO]:

- Neo4j je plně transakční databáze implementovaná v jazyce Java. Ukládá strukturovaná data přímo v grafové struktuře na disku. Díky implementaci v Javě je snadno začlenitelná. Více informací lze najít na [NEO].
- DEX je výkonná knihovna pro strávu rozsáhlých grafů nebo sítí, více viz. [DEX].
- HyperGraph je projekt pro obecné potřeby, disponuje rozšiřitelností, přenositelností, je distribuovaný, začlenitelný do jiných projektů a jedná se o open source. Více na [HG].
- InfoGrid je webová grafová databáze s mnoha doplňky pro tvorbu webových aplikací s API architektury REST. Více informací lze najít na [IG]
- VertexDB je vysoce výkonný databázový server. Je postavený na HTTP protokolu a data jsou klientovi zasílána ve formátu JSON. Další informace jsou k dipozici na [VDB]
- AllegroGraph je moderní, vysoce výkonná databáze založená na RDF Frameworku. Více viz. [AG].

Mezi další známé grafové databáze patří InfiniteGraph, FlockDB, Sones nebo OrientDB.

6 Návrh a řešení metody práce s grafy v databázi

Na první pohled by se z výše uvedeného mohlo zdát, že řešení práce s grafy v databázi je nasnadě a spočívá ve výběru vhodného projektu z grafových databází. Vzhledem k tomu, že jsou ale relační databáze od svého vzniku do dnešní doby suverénně dominantním modelem databáze (a to i přesto, že v historii již bylo navrženo mnoho alternativních konceptů), považuji za vhodné zaměřit se právě na tento typ databází. To zvláště proto, že pro žádnou relační open-source databázi zatím neexistuje vhodné rozšíření pro práci s grafy. Dále pak proto, že řešení by mohlo pomoci projektům, které z části ukládají data vhodná pro použití relační databáze a z části grafová data, která by však bylo neefektivní spravovat ve zcela oddělené grafové databázi.

Z open-source relačních databází volím vytvoření rozšíření pro práci s grafy pro databázi PostgreSQL, protože se nyní jedná o technologicky nejpokročilejší projekt této kategorie disponující vhodnými prostředky pro implementaci uvažovaného rozšíření.

6.1 Prostředky pro implementaci rozšíření PostgreSQL

PostgreSQL je rozšiřitelné, protože jeho chod je řízen informacemi uloženými v interních tabulkách. Většina standardních relačních databázových systémů ukládá informace o databázích, tabulkách, sloupcích apod. v systémových tabulkách, které jsou dostupné uživateli stejně jako jiné tabulky. PostgreSQL se liší v tom, že v těchto tabulkách ukládá podstatně větší množství informací včetně informací o datových typech, funkcích, přístupových metodách a podobně. Protože jsou tyto tabulky editovatelné uživatelem, lze touto cestou PostgreSQL rozšiřovat. Ostatní relační databázové systémy jsou v tomto směru obvykle rozšiřitelné změnou zdrojových kódů nebo zavedením rozšíření dodávaných výrobcem. PostgreSQL je implementován v jazyce C se standardem ANSI C a v tomto jazyce (a dalších jazycích) nabízí možnost implementovat další rozšíření[3]. Přesto, že je díky tomu dosaženo maximální rychlosti, vývoj v jazyce C není vždy příliš komfortní. To je však úspěšně vyváženo množstvím maker a knihovních funkcí, které jsou k dispozici. Je tak možné definovat **vlastní funkce, operátory, datové typy i vlastní indexy**. Formou těchto rozšíření jsou realizovány projekty jako PostGIS (podpora geografických objektů) nebo Orafce (rozšíření o část sofistikované

3 Za rozšíření lze samozřejmě chápat i vlastní funkce definované v PL/pgSQL či dalších jazycích.

funkcionality, kterou disponuje databáze Oracle). V této kapitole velmi stručně shrnu hlavní aspekty implementace C rozšíření pro PostgreSQL, vycházím především z [PD] a [PCZ].

Pro všechny datové typy dostupné v SQL se implementaci C rozšíření používá generický datový typ `Datum`, který obsahuje ukazatel nebo přímo hodnotu, pokud se tato hodnota do datového typu `Datum` vejde (4 nebo 8 bajtů podle toho zda se pracuje na 32 nebo 64 bitové platformě). Programátor musí vědět s jakým datovým typem v rámci `Datum` pracuje a není zde v tomto směru žádná kontrola ze strany překladače. To umožňuje také použití polymorfních datových typů v parametrech a návratových hodnotách funkcí, se kterými je však spojena větší režije.

S datovými typy souvisí také předávání parametrů funkcím. Parametry lze předávat přímo, nicméně je doporučeno předávat a vracet parametry pomocí maker, tzv. V1 konvencí. Definice funkce je pak tedy vždy ve tvaru `Datum nazev_funkce(PG_FUNCTION_ARGS)`. Samotné argumenty pak lze ve funkci získat pomocí maker `PG_GETARG_typ(int)`, kde `typ` je typem parametru a jako argument předáme makru pořadí parametru, který získáváme. Voláním `PG_GETARG_TEXT_P(0);` je tedy získán parametr na první pozici aktuální funkce v typu korespondujícím s SQL datovým typem TEXT. Pro možnost využívat tuto užitečnou a ulehčující konvenci je nutné nejprve zavolat tzv. signaturu funkce ve tvaru `PG_FUNCTION_INFO_V1(nazev_funkce);`

Pro zamezení možnosti použít rozšíření zkompilované pro jinou platformu (32 nebo 64bit), má v sobě každé rozšíření zakomponovanou informaci o tom, v rámci jaké verze PostgreSQL bylo zkompilováno. Při inicializaci rozšíření se pak modul zavede, nebo je vrácena chyba. Jedná se o tzv. signaturu modulu a slouží pro ni makro `PG_MODULE_MAGIC`.

Pro převod mezi hodnotami typu `Datum` a datovými typy jazyka C slouží opět makra, která zakrývají relativně náročné procesy pro deTOAST[4] a dekompozici typů s variabilní délkou.

Pro práci s řetězci slouží knihovna stringinfo, která usnadňuje jinak relativně nepohodlný proces spojování a formátování řetězců v jazyce C.

Typu char* jazyka C odpovídá typ `CString`, který se používá pro vstupně/výstupní operace. Každý datový typ v PostgreSQL má registrovanou jednu funkci s názvem `in` a jednu funkci s názvem `out`. Ve funkci `in` se parsuje vstup, ve funkci `out` formátuje výstup. CString se mimo `in` a `out` funkcí nedoporučuje používat z důvodu, že nepodporuje TOAST a jeho délka je omezena na 8KB.

Dalším základním aspektem vývoje rozšíření pro PostgreSQL je správa paměti. Pro alokování paměti je v PostgreSQL opět vytvořen komplexnější unifikovaný systém, snižující fragmentaci paměťového prostoru a usnadňující práci programátora. Při implementaci rozšíření je paměť

4 TOAST je pojem označující v PostgreSQL systém ukládání větších množství dat. Velikost datové schránky v PostgreSQL je 8KB a při přesažení této velikosti je využit TOAST. Jedná se o tabulku, která je vytvořena pro každou tabulku, která obsahuje záznamy, jejichž položky přesahují 8KB. Data v záznamu původní tabulky pak nahradí adresa dat uložených v TOAST tabulce.

doporučeno alokovat vždy pomocí funkce `palloc` a uvolňovat pomocí funkce `pfree`. Paměť alokovaná pomocí funkce `palloc` je automaticky uvolněna vždy po dokončení aktuální transakce, aby se zabránilo únikům paměti.

Před použitím rozšíření v rámci PostgreSQL serveru je nutné je zkompilovat tak, aby byl vytvořen soubor, který server může dynamicky načíst. Jedná se o tzv. *sdílenou knihovnu*. Vytváření sdílených knihoven je analogické s vytvářením spustitelných souborů. Navíc veškerý kód musí být vytvořen tak, aby byl nezávislý na umístění v rámci diskového prostoru. Samotnou kompilaci sdílené knihovny je možné provést překladačem gcc nebo cc a liší se podle operačního systému. Pro představu uvádím příklad triviálního rozšíření využívajícího výše uvedené postupy:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);  /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}
```

```
CREATE FUNCTION copytext(text) RETURNS text
AS 'DIRECTORY/funcs', 'copytext'
LANGUAGE C STRICT;
```

PostgreSQL je dále možné rozšiřovat definicí uložených procedur v jazyce PL/pgSQL. Tento jazyk ideově vychází z PL/SQL firmy Oracle a využívám ho pro velkou část rozšíření implementovaného v rámci této práce. Pro obdobné účely jako PL/pgSQL lze využít i jazyky PL/Tcl, PL/Perl a PL/Python.

6.2 Identifikátory záznamů

PostgreSQL disponuje několika úrovněmi identifikátorů záznamů uložených v tabulkách. V této kapitole jednotlivé identifikátory představím a to především v kontextu rychlosti přístupu k samotným záznamům jejich prostřednictvím. Vycházím zde především z [PD].

Z pohledu běžného uživatele jsou záznamy typicky označeny volitelným unikátním identifikátorem, tzv. primárním klíčem. Nad primárním klíčem je vytvořen index a získání záznamu je provedeno v čase *O(log(n))*. To je standardní a pro naprostou většinu případů užití vhodný způsob identifikace.

Dále je k záznamu možné přistoupit prostřednictvím identifikátoru objektu *oid* (object ID). Oid je jedinečným identifikátorem interních prostředků databáze a dělí se na další typy podle toho, zda identifikuje relaci (regclass), jméno funkce (pregproc), jméno funkce s argumenty (regprocedure), operátor (regoper), operátor s argumenty (regoperator) nebo datový typ (regtype). Na základě nastavení konfigurační proměnné default_with_oids je pak možné klauzulí WITH OIDS respektive WITHOUT OIDS nastavit zavedení oid pro každý záznam v tabulce. Přesto, že se jedná o interní identifikátor určený pouze pro pokročilé využití, není s jeho použitím dosaženo rychlejšího přístupu k záznamu. Pro identifikaci záznamů se tedy doporučuje použití uživatelem definovaného primárního klíče.

Na nejnižší úrovni databázového systému PostgreSQL je záznam identifikován tzv. *Tuple identifier* neboli, *tid* určujícím fyzické umístění záznamu. Jedná se o datový typ systémového sloupce *ctid*[5] každé tabulky databáze. Tento identifikátor využívají indexy pro finální přístup k samotnému záznamu a jedná se tedy o nejrychlejší možnost jak k záznamu v tabulce přistoupit, protože neprobíhá průchod indexem (typicky s časovou složitostí *O(lon(n))*), ale přímý přístup k fyzickému úložišti a to v čase konstantním. Vztahuje se vždy na konkrétní tabulku. Komplikací užití tohoto identifikátoru je jeho častá změna v čase a to vždy při aktualizaci nebo přesunu záznamu a také při operaci VACUUM FUUL. Proto jeho využití není pro veškeré běžné případy doporučeno, což ovšem není případ této práce. Práci s *tid* respektive systémovým sloupcem *ctid* obsahující tento identifikátor se věnuji v následující kapitole 6.3.

Pro úplnost uvádím i další existující systémové identifikátory, kterými jsou *tableoid* identifikující z které tabulky záznam pochází, *xmin* identifikuje transakci vkládající záznam, *cmin* identifikuje příkaz vkládání, *xmax* identifikuje transakci smazání záznamu, *cmax* identifikuje příkaz mazání záznamu.

5 Mezi systémové sloupce v tabulkách patří i další, typicky pomocné identifikátory pro běh databáze, více viz. System Columns v dokumentaci PostgreSQL.

6.3 Identifikátor TID

Jak bylo popsáno v předchozí kapitole, identifikátor *tid* určuje fyzické umístění záznamu, který lze jeho prostřednictvím získat v konstantním čase. *Tid* identifikátor se skládá ze dvou částí, první hodnota označuje datový blok a druhá offset. Záznam je možné vyžádat SQL dotazem s určením hodnoty sloupce *ctid*, např: `SELECT * FROM vertices WHERE ctid = '(0,1)'::tid;` nebo rychleji funkcí v jazyce C. Funkce pro přímý přístup k datům jsou definovány ve zdrojovém souboru src/backend/access/heap/heapam.c .

Pro přímý přístup k datům je nutné nejprve otevřít relaci pomocí funkce `heap_openrv` a poté je možné použít funkci `heap_fetch`. Funkci `heap_fetch` předáme identifikátor otevřené relace, aktuální snapshot, pointer na strukturu `HeapTupleData` kde v prvku `t_self` struktury `ItemPointerData` předáme data *tid* identifikátoru, dále buffer pro získaná data. Na závěr určíme nutné detaily a to sice způsob nakládání s bufferem v případě, že funkce selže, a dále zda volání funkce zohlednit pro statistické potřeby.

Funkce `heap_fetch` vloží ve struktuře `HeapTupleData` do prvku `t_data` struktury `HeapTupleHeader` samotná data. Tyto je pro vrácení uživateli nutné zkopírovat funkcí `heap_copytuple`, která pro data alokuje nový blok paměti. Funkce `heap_copytuple` nenastaví všechny prvky nové struktury typu `HeapTupleData`, proto je nutné zbývající ošetřit makry `HeapTupleHeaderSetDatumLength`, `HeapTupleHeaderSetTypeId` a `HeapTupleHeaderSetTypMod`.

Poté je možné uvolnit buffer předaný funkci `heap_fetch`, uzavřít otevřenou relaci a převést data do universálního typu `Datum` pro vrácení uživateli.

Identifikátor *tid* je využíván ve všech indexech PostgreSQL databáze a funkce `heap_fetch` je využívána pro přístup k záznamům i při jejich dotazování standardním způsobem. Jedná se tedy o nejrychlejší možný způsob jak záznam v databázovém systému PostgreSQL získat.

6.4 Metoda efektivní grafové struktury

Jak již bylo popsáno v kapitole 5.1, správa grafu v tabulkách relační databáze a vytvoření vazeb pomocí standardních primárních a cizích klíčů není vhodnou reprezentací grafu, protože nerespektuje spojovou podstatu grafových dat. Z toho důvodu jsem navrhl strukturu, která bude ze spojové podstaty vycházet a kde budou přechody v grafu (získání okolních vrcholů) probíhat v konstantním čase.

V navržené struktuře probíhají přechody mezi záznamy nikoli pomocí primárních a cizích klíčů, ale pomocí *tid* identifikátorů a pomocí funkcí implementovaných v jazyce C. Tím je zaručeno,

že přechod proběhne v konstatním čase a v maximální rychlosti, kterou je databázový systém a úložiště (typicky pevný disk) schopné dosáhnout. V tomto konceptu je třeba řešit problém změny identifikátoru *tid* a to nejčastěji při operaci UPDATE nad záznamem, která je relativně vysoce frekventovaná[6].

V návrhu struktury vycházím především z potřeb algoritmů, které implementuji v rámci této práce a přihlížím také k potřebám ostatních běžných grafových algoritmů. V rámci práce budu implementovat prohledávání grafu do šířky, zjištění chromatického čísla grafu pomocí greedy coloringu, obarvení grafu pomocí RLF coloringu a zjištění izomorfismu podgrafu pomocí sofistikovanějšího algoritmu VF2. V této škále algoritmů, od triviálních (greedy coloring) po relativně sofistikované (VF2), se setkávám primárně s potřebou zjištění okolních uzlů od daného vrcholu. To také v návrhu zohledním a dále návrh přizpůsobím tak, aby bylo možné procházet i na základě „The Graph Traversal Pattern" jak je popsáno v [AN10].

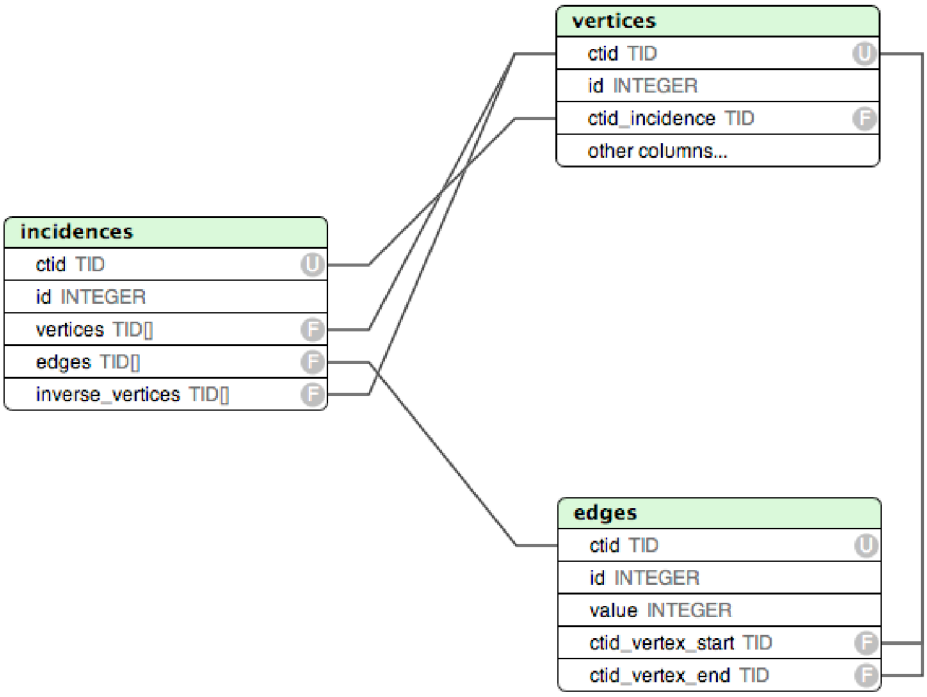
Strukturu grafu jsem rozdělil do tří tabulek. Tabulka obsahující vrcholy má název *vertices* a obsahuje ID jednotlivých vrcholů a hodnoty náležící k danému vrcholu, které jsou bezprostředně nutné pro algoritmy uvažovaných operací nad grafem a odkaz do tabulky incidencí *ctid_incidence* typu *tid*. Tabulka obsahující hrany má název *edges* a obsahuje ID jednotlivých hran a ohodnocení hrany a *tid* identifikátory počátečního a koncového uzlu, které hrana spojuje. (Přestože testovacím grafem je graf stránek wikipedie, kde není obsaženo ohodnocení hran, byla tato možnost do struktury zařazena.) Pro uložení dalších informací náležících k vrcholům a hranám grafu, které nejsou nezbytně nutné pro běh uvažovaných operací respektive algoritmů, slouží další libovolné tabulky, mezi kterými jsou na tabulky efektivní struktury (*vertices* a *edges*) vytvořeny reference standardními klíči prostřednictvím sloupce ID v tabulkách *vertices* a *edges*.

Mezi tabulkami *vertices* a *edges* je vložena tabulka incidencí vrcholů a hran s názvem *incidences*. Každý vrchol tabulky *vetices* obsahuje referenci *ctid_incidence* což je hodnota *ctid* záznamu v tabulce *incidences* patřící k danému vrcholu. Samotný záznam incidencí pak obsahuje následující hodnoty, respektive sloupce.

Sloupec *vertices* obsahuje pole identifikátorů *tid*, které náleží záznamům respektive vrcholům v tabulce *vertices,* do kterých vedou hrany z vrcholu aktuální incidence. Sloupec *edges* obsahuje pole identifikátorů *tid*, které náleží záznamům respektive hranám v tabulce *edges* a reprezentuje samotné výstupní hrany z vrcholu aktuální incidence. Stejná pozice prvků v polích *vertices* a *edges* definuje pár výstupní hrany a cílového vrcholu. V neposlední řadě je pro spoustu algoritmů a operací užitečný sloupec *inverse_vertices*, který obsahuje pole indentifikátorů *tid*, které náleží záznamům respektive vrcholům v tabulce *vertices*, ze kterých vedou hrany do vrcholu aktuální incidence.

6 V porovnání s operací VACUUM FULL, při které také dochází ke změně *tid* záznamů, ale která probíhá obvykle méně často a proto její ošetření neklade tak vysoké nároky.

Celou efektivní strukturu schematicky ilustruje Obrázek 6.1. (Vzhledem k použití polí a k tomu, že pro vazby nejsou použity primární a cizí klíče, nejedná se o ER diagram.)



Obrázek 6.1: Metoda efektivní struktury

Celá struktura tedy defakto tvoří databázový index pro procházení grafovými daty.

Pro funkčnost celé struktury je nutné reflektovat změny *ctid* jednotlivých záznamů při jejich aktualizaci (operaci UPDATE). Aby toho bylo možné dosáhnout je nutné, aby se alespoň v jednom místě efektivní struktury identifikátor *tid* při aktualizaci neměnil, protože jinak se struktura dostává do rekurzivního a nikdy nekončícího promítání změn jednotlivých *tid*. Toho je dosaženo pomocí BEFORE UPDATE triggeru implementovaného v jazyce C. V jazyce C je implementován jako funkce inplace_update_trigger, která s použitím knihovní funkce PostgreSQL `heap_inplace_update` zajistí, že *tid* není při aktualizaci záznamu (operaci UPDATE) změněno. Při implementaci tohoto řešení jsem vycházel především z [IU]. Toto opatření s sebou nese omezení, která jsou rozebrána v kapitole 6.6 a se kterými návrh počítá a jsou vhodně ošetřena.

6.5 Základní operace nad efektivní strukturou

Pro základní správu dat v efektivní struktuře jsem implementoval funkce abstrahující vkládání vrcholů a hran do struktury a stejně tak jejich smazání. Dále jsou implementovány demonstrační funkce pro dávkovou inicializaci grafové struktury, v případě této diplomové práce se jedná o inicializaci efektivní datové struktury pro graf stránek na wikipedii. Vzhledem k tomu jsem

strukturu navrhl primárně pro práci s orientovaným grafem, nicméně lze jí vhodně využít i pro práci s grafem neorientovaným[7].

Vložení vrcholu do struktury je realizováno funkcí `insert_vertex` s parametrem ID vrcholu. Funkce vrací hodnotu typu boolean. Hodnota *true* je vrácena pokud vložení vrcholu proběhne bez poblémů, *false* pokud dojde k chybě. Typicky pokud již v efektivní struktuře existuje vrchol s daným ID. V rámci vložení vrcholu je vytvořen také záznam v tabulce *incidences*, navázaný na právě vkládaný vrchol a nezbytně nutný pro funkčnost struktury.

Vložení hrany mezi dva vrcholy je realizováno funkcí `insert_edge` s parametry ID hrany, hodnota hrany (typu integer), ID výchozího vrcholu a ID konečného vrcholu. Funkce opět vrací hodnotu typu boolen. Je vráceno *true* pokud vložení hrany proběhne bez problému, *false* pokud dojde k chybě. Typicky pokud již existuje hrana s daným ID nebo pokud neexistuje jeden z vrcholů či relevantních záznamů v tabulce *incidences*. Funkce vloží záznam do tabulky *edges* a také do záznamu incidencí. K výchozímu vrcholu do polí *vertices* a *edges* a ke konečnému vrcholu do pole *inverse_vertices*.

Smazání vrcholu je realizováno funkcí `delete_vertex` s parametrem ID vrcholu a opět vrací hodnotu typu boolean podle toho zda operace proběhla úspěšně stejně jako předchozí funkce. Zároveň jsou smazány všechny hrany, které z tohoto vrcholu vycházejí nebo které do tohoto vrcholu vedou a také adekvátně upraveny záznamy incidencí u všech sousedních vrcholů, včetně vrcholů sousedících přes hrany opačného směru.

Smazání hrany je realizováno funkcí `delete_edge` s parametrem ID hrany a vrací hodnotu typu boolen podle toho zda operace proběhla úspěšně stejně jako předchozí funkce. Zároveň jsou smazány všechny záznamy v polích *vertices*, *edges* a *inverse_vertices*, které jsou ke hraně navázány.

Pro inicializaci efektivní struktury jsem implementoval demonstrační funkce `init_vertices` a `init_edges`. Funkce `init_vertices` prochází jednotlivé vrcholy uložené v tabulce mimo efektivní strukturu a vkládá záznam o nich do efektivní struktury. V případě této práce se jedná o tabulku page v databázi wikipedie. Funkce `init_edges` pracuje obdobně a vkládá všechny hrany do efektivní struktury, v případě této práce z tabulky pagelinks z databáze wikipedie. Pro zakládání efektivní struktury nad jinou existující databází (sociální síť komunitního webu a podobně) je nutné tyto funkce vhodně modifikovat pro zvolený zdroj dat.

6.6 Procházení efektivní struktury

Efektivní procházení grafem uloženým v databázi je jedním z hlavních cílů této práce, proto v návrhu řešení a rozšíření pro PostgreSQL kladu na tuto operaci hlavní důraz a snažím se ji vytvořit

7 Neorientovaný graf lze do struktury vložit tak, že mezi každými dvěma vrcholy, mezi kterými vede neorientovaná hrana, vložíme dvě orientované a vzájemně protisměrné hrany.

maximálně efektivní. Pro základní procházení grafem vycházím z [AN10] a uvažuji dvě základní přechodové operace a to sice přechod od vrcholu k hranám, které s ním incidují a dále přechod z hrany k vrcholu se kterým inciduje. Na základě architektury efektivní struktury grafu je možné snadno získat přímo okolní vrcholy (po směru výchozích hran i proti směru vstupních hran), což je pro většinu operací a algoritmů nejpodstatnější.

Pro průchod grafem jsou implementovány funkce `get_vertex` s parametrem *tid* hledaného vrcholu a vracející kompletní záznam daného vrcholu z tabulky *vertices* a funkce `get_incidences` s parametrem *tid* záznamu incidencí a vracející kompletní záznam všech incidencí a hran příslušných k určitému vrcholu. Pomocí tohoto záznamu tedy zíkáváme přechod do okolí vrcholu a tímto způsobem (voláním funkcí `get_vertex` a `get_incidences`) realizujeme průchod strukturou grafu.

Obě funkce jsou implementované v jazyce C a využívají pro přístup k datům funkci `heap_fetch` tak, jak je popsáno v kapitole 6.3. Průchod grafem je tedy realizován v konstantních časech s maximální rychlostí, kterou jsme schopni s PostgreSQL a použitým hardware dosáhnout.

Efektivnost struktury při procházení grafem demonstruje algoritmus pro prohledávání grafu do šířky BFSearch implementovaný v experimentální a demonstrační části této práce.

6.7 Omezení návrhu a jejich řešení

Omezení návrhu vycházejí především z použití BEFORE UPDATE triggeru, který zamezuje změně identifikátoru *tid* při aktualizaci záznamu v tabulce *vertices*. Tento trigger používá funkci `heap_inplace_update` a proto musí být reflektovány následující situace.

Nový záznam, při aktualizaci v tabulce *vertices*, musí být stejné velikosti jako původní záznam. Není přípustná ani změna z hodnoty NULL na „NOT NULL“ hodnotu či naopak. Proto v tabulce *vertices* figurují pouze sloupce s pevnou velikostí datového typu. V rámci všech implementovaných algoritmů jsou ve sloupcích tabulky *vertices* potřebné pouze hodnoty typu integer. Při vkládání nových záznamů je do těchto sloupců vkládána nula, abychom předešli nerealizovatelné nutnosti změny z hodnoty NULL na numerickou hodnotu. V případě, že je v rámci algoritmu nutné k vrcholům uložit hodnoty datového typu s proměnnou velikostí, je pro tento účel možné vytvořit například TEMP tabulku.

Nad sloupci v tabulce *vertices*, ve kterých dochází ke změnám hodnoty, nelze vytvořit databázový index. Může díky tomu docházet k nesprávnému řazení výsledků a dalšímu nevhodnému chování. Stejně tak není možné realizovat cizí klíč nad sloupcem, který je aktualizován.

Dále při této operaci dochází k porušení atomičnosti a izolovanosti transakce. To je vhodně ošetřeno tak, že ve funkci, která mění záznam v tabulce *vertices*, je nejprve zavolán příkaz SELECT

FOR UPDATE a tím je zajištěno, že k záznamu nebude mít přístup jiná transakce až do skončení současné.

Posledním potenciálním problémem navržené metody je údržbová operace VACUUM FULL, která mění *tid* ve sloupci *ctid* u všech záznamů v databázi. Pokud je v tabulce smazán záznam, vznikne v souboru s daty volné místo. Funkce VACUUM FULL na tato volná místa přesunuje existující záznamy a optimalizuje tak rozdělení obsazeného a volného prostoru v souboru s daty. Poté je možné celý soubor zkrátit a operační systém může vzniklý diskový prostor využít pro jiné účely. Celkově je tato funkce vhodná pouze pro případy, kdy byla většina databázových dat smazána. V opačném případě může mít vedlejší efekty ve formě zvětšení objemu databázových indexů a jejich zpomalení. V navržené metodě práce s grafy pak pravděpodobně dojde k porušení struktury grafu. Mimo triviální řešení, kterým je tuto operaci nad databází s grafem nepoužívat, se nabízí další možná cesta. Tou je vytvoření vlastního indexu, jehož implementace by po volání VACUUM FULL zajistila přepočítání referencí pro všechny prvky v efektivní struktuře grafu. V rámci této práce jsem zmíněný index neimplementoval a situaci řeším nepoužíváním operace VACUUM FULL, což je pro mé experimentální účely plně dostačující.

7 Experimenty a demonstrace metody

V této části práce rozeberu podstatné aspekty implementace vytvořených rozšíření a především experimenty s jeho použitelností, což bude demonstrováno na několika grafových algoritmech. Budu porovnávat především čas průběhu při použití algoritmu implementovaného pomocí standardního postupu spojování (JOIN) tabulek vrcholů a hran vůči použití algoritmu, který v implementaci využívá navrženou metodu efektivní struktury pro práci s grafy v PostgreSQL. Teoretickým předpokladem většiny experimentů je, že se potvrdí očekávání vyšší efektivity procházení grafem při použití efektivní struktury popsané v kapitole 6.

V rámci vývoje algoritmů, pomocí kterých chci ověřit implementovanou metodu procházení grafovými daty v databázi, jsem se rozhodl pro využití jazyka PL/pgSQL. Je více než pravděpodobné, že implementace v jazyce C by byla rychlejší, nicméně v rámci experimentů mi jde především o výše zmíněnou komparaci a proto shledávám PL/pgSQL a jeho vyšší úroveň abstrakce jako vhodnější prostředek.

Uvedené testy probíhaly na počítači Apple MacBook2,1 s procesorem 2.16GHz Intel Core 2 Duo, operační pamětí 2GB 667MHz DDR2 SDRAM a pevným diskem FUJITSU MHW2120BH s kapacitou 120.03GB a rychlostí 5400 RPM a 8MB Cache. Operačním systémem na testovací platformě byl Mac OS X 10.6.7 s jádrem verze Darwin 10.7.0. Všechny testy proběhly na databázi PostgreSQL verze 8.4.4 kompilovanou GCC i686-apple-darwin10-gcc-4.2.1 (GCC) 4.2.1 (Apple Inc. build 5659), 64-bit. Dále se tento počítač označuji zkráceně „MacBook“.

Dále pak na vysoce výkonném serveru Minerva3 dostupném v rámci CVT FIT VUT v Brně. Na tomto serveru však nebylo zaručeno, že můj experiment je jediným spuštěným procesem. Dále tento počítač označuji „Minerva3“.

Pro testy byl použitý graf stránek limburské a české wikipedie z 18. dubna 2011, toho data dostupný z [AW]. Česká wikipedie má značný rozsah a to 500352 vrcholů spojených 3465044 hranami. Limburská wikipedie byla náhodně vybraná, protože rozsah její databáze splňoval požadavek na menší objem dat pro rychlejší průběh testování při vývoji. Konkrétně obsahuje 19683 vrcholů a 218522 hran. Dále je využita pro komparaci s během algoritmů na obsáhlejší české wikipedii.

7.1 BF-search – prohledávání do šířky

Prvním experimentálním algoritmem, který jsem v rámci práce implementoval, bylo zcela základní prohledávání grafu do šířky. To především proto, že cílem práce je zaměřit se na procházení grafy. Vycházel jsem z pseudokódu algoritmu uvedeného v kapitole 3.3.2 a dále jsem tento algoritmus

rozšířil o možnost zvolit hloubku, do které je graf prohledáván. Vzhledem k rozsahu grafu Wikipedie, na kterém byl algoritmus následně testován, bylo toto omezení vhodné. Bez omezení hloubky prohledávání, neboli při prohledávání kompletního grafu wikipedie, se algoritmus snadno dostane k velmi vysoké časové náročnosti.

Algoritmus jsem implementoval v jazyce PL/pgSQL a to v několika verzích. V první verzi jsem využil pro implementaci fronty vrcholů pole jazyka PL/pgSQL a pro procházení navrženou metodu efektivní struktury. První verze implementace algoritmu je ve zdrojovém kódu přiloženém k práci reprezentována funkcí `bf_search_1` a přijímá dva parametry. Prvním parametrem je ID vrcholu grafu, od kterého je zahájeno prohledávání, druhým parametrem je maximální hloubka prohledávání. Funkce `bf_search_1` vrací jako výsledek jednorozměrné pole datového typu `bfs_q_item`. Datový typ `bfs_q_item` reprezentuje prohledané vrcholy a to konkrétně položkami *ctid* vrcholu (*ctid_vertex*), *ctid* incidencí náležících k vrcholu (*ctid_incidence*), vzdálenost od výchozího vrcholu (*d*) a *tid* nadřazeného prvku (*p*). Díky prvku *p* je tedy ve vráceném poli definována stromová struktura, tzv. BF-Strom, který se prohledáváním do hloubky získává.

V druhé verzi implementace algoritmu jsem pro průchod grafem využil standardních SQL dotazů s klauzulí JOIN. Pro implementaci fronty využívané v algoritmu je opět využito jednorozměrné pole. Ve zdrojových kódech je tato verze algoritmu označena jako `bf_search_1_sql` a přijímá stejné parametry jako první `bf_search_1`. Návratovou hodnotou je jednorozměrné pole datového typu `bfs_q_item_sql`. Jedná se o obdobu typu `bfs_q_item`, kde jsou však identifikátorem vrcholu standardní ID. Tzn. `bfs_q_item_sql` obsahuje ID vrcholu (*id_vertex*), vzdálenost od výchozího vrcholu (*d*) a ID předchůdce (*p*).

Protože se mi prohledávání do hloubky pomocí implementovaných funkcí `bf_search_1` a `bf_search_1_sql` zdálo subjektivně pomalé, hledal jsem ještě možnost, jak algoritmus optimalizovat. Zjistil jsem, že v jazyku PL/pgSQL není zcela optimalizována práce s poli a zkusil jsem frontu v algoritmu implementovat místo pole dočasnou tabulkou, tzv. TEMP tabulkou. Jak bude dále demonstrováno, využití TEMP tabulky chod algoritmu značně zrychlí.

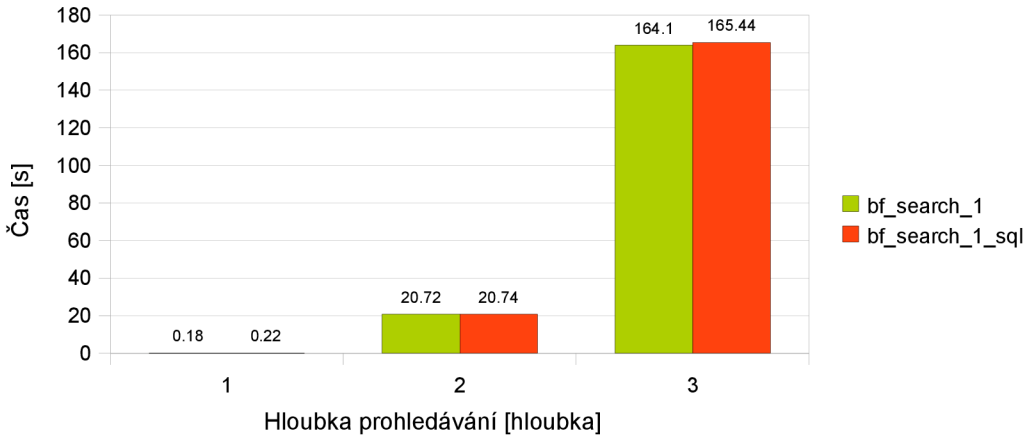
Třetí verze implementace algoritmu využívá pro fronty TEMP tabulku a dále navrženou metodu efektivní struktury a jejího procházení. Ve zdrojovém kódu je reprezentována funkcí `bf_search_2` a přijímá stejné parametry jako předchozí verze. Funkce `bf_search_2` ukládá výsledky prohledávání do TEMP tabulky s názvem result. Záznamy v tabulce korespondují s datovým typem `bfs_q_item`, tabulka tedy obsahuje sloupce *ctid_vertex*, *ctid_incidence*, *d* a *p* stejně jako datový typ `bf_q_item`.

Čtvrtá verze implementace algoritmu využívá pro frontu také TEMP tabulku a graf prochází pomocí standardních SQL dotazů. Je ve zdrojových kódech reprezentována funkcí `bf_search_2_sql` a přijímá stejné parametry jako předchozí funkce. Funkce

`bf_search_2_sql` ukládá výsledky prohledávání do TEMP tabulky s názvem result. Záznamy v tabulce korespondují s datovým typem `bfs_q_item_sql`, tabulka tedy obsahuje sloupce *id_vertex*, *d* a *p* stejně jako datový typ `bf_q_item_sql`.

7.1.1 Komparační test s BF-search

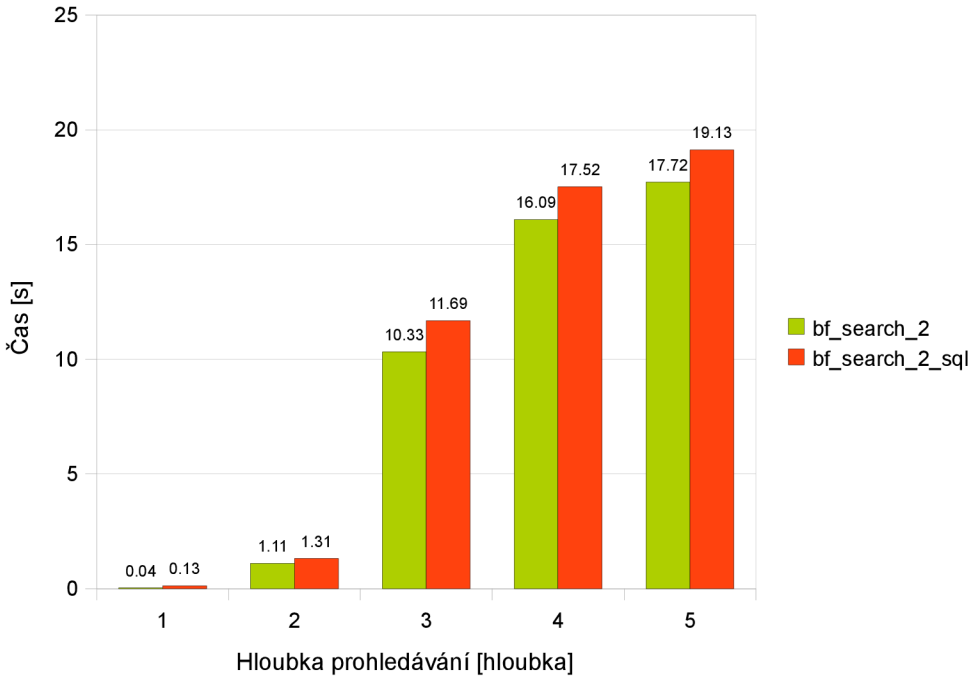
V tomto testu se chci zaměřit především na porovnání rychlosti při použití navržené metody efektivní struktury oproti použití standardních SQL dotazů. Teoretickým předpokladem je, že navržená metoda bude vykazovat lepší výsledky. Zároveň porovnám rychlost při použití polí a dočasných tabulek pro implementaci front, zásobníků, seznamů, stromů a podobných struktur v PL/pgSQL. V rámci testu je provedeno vyhledávání do šířky nad náhodně zvolenými vrcholy limburské wikipedie na počítači „MacBook“.



Obrázek 7.1: Porovnání prohledávání metodou efektivní struktury a standardními SQL dotazy s implementací fronty pomocí pole.

V testech bylo při prohledávání do hloubky prohledáno 119 uzlů v první úrovni, 1933 uzlů v druhé úrovni, 6642 ve třetí úrovni, 8633 uzlů ve čtvrté a 9292 v páté úrovni. Z uvedených grafů na Obrázku 7.1 a 7.2 vyplývá, že časová náročnost je nižší při použití navržené efektivní grafové struktury. Dále ve všech experimentech vykazuje nižší časovou náročnost implementace fronty pomocí TEMP tabulky.

Osobně jsem očekával markantnější časový rozdíl při použití navržené metody efektivní struktury. Nicméně limburská wikipedie, kterou jsem musel z časových důvodů použít, není na tolik obsáhlá aby přechody mezi uzly s časovou složitostí *O(log(n))* byly na tolik fatálně zpomalující a většitnu času při průběhu algorimu zpotřebuje jiná režije než samotné získávání záznamu.



Obrázek 7.2: Porovnání prohledávání metodou efektivní struktury a standardními SQL dotazy s implementací fronty pomocí TEMP tabulky.

Vzhledem k výsledkům experimentu, kdy jsou v PL/pgSQL operace s TEMP tabulkami podstatně rychlejší než operace s poli (což bylo ověřeno i v dalších snahách používat při implementaci pole), v následujících experimentech používám pro implementaci dynamických datových struktur TEMP tabulky.

7.2 Chromatické číslo grafu / greedy coloring

Dále se zaměřuji na algoritmy pro barvení grafu. Zcela základním a triviálním způsobem barvení je tzv. greedy coloring, který popisuji v kapitole 4.2. Pro tento algoritmus jsem implementoval základní optimalizaci largest first coloring. Tato optimalizace spočívá v tom, že barvení postupuje od vrcholů s nejvyšším stupněm, neboli s nejvyšším počtem sousedních vrcholů. Algoritmus je tedy implementován tak, že probíhá ve dvou fázích. V první fázi ke každému vrcholu zjistí jeho stupeň a uloží výsledek do databáze. V druhé fázi probíhá samotné obarvení grafu. Vzhledem k nekvalitnímu obarvení shledávám vhodné využití tohoto algoritmu například pro zjišťování chromatického čísla grafu pro potřeby komparace s jinými grafy.

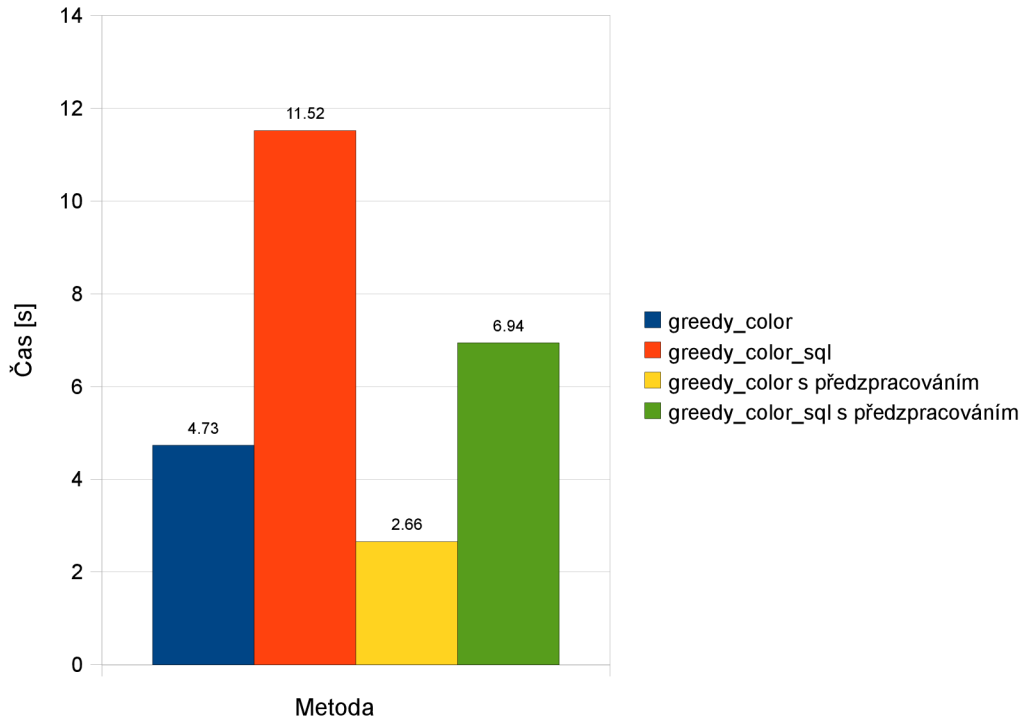
V této demonstraci navržené metody efektivní struktury jsem se zaměřil opět na komparaci s procházením grafu pomocí standardních SQL dotazů. Jedná se o mírně odlišnou úlohu oproti prohledávání grafu do hloubky, neboť zde nejsou tak vysoké paměťové nároky, protože neobsahuje

žádnou frontu ani jiné dočasné úložiště. Navíc v první fázi algoritmu nedochází k procházení grafu, ale pouze k zjišťování počtu okolních uzlů, respektive stupňů uzlů. Test provedu i bez úvodní části, s předzpracovaným grafem, kde jsou stupně vrcholů zjištěny. I v této úloze navržená metoda vykazuje podstatně lepší výsledky než standardní SQL dotaz. Před průběhem algoritmu jsou pro korektní průběh vždy vynulovány hodnoty určující barvu ve sloupci tabulky *vertices*.

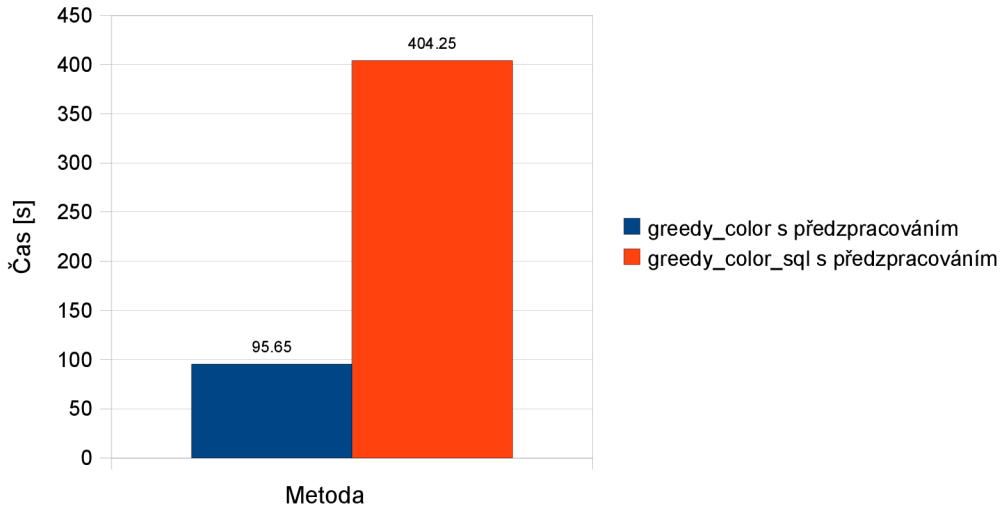
První verze implementace grafu využívá prostředky navržené metody efektivní struktury. V přiložených zdrojových kódech je reprezentována funkcí `greedy_color`. Druhá verze implementace využívající standardní SQL dotazy je prezentována funkcí `greedy_color_sql`.

7.2.1 Komparační test zjištění chromatického čísla

Teoretickým předpokladem tohoto experimentu je rychlejší průběh verze implementace s použitím efektivní grafové struktury. Pro ověření předpokladu jsem spustil experiment nad grafem limburské a české wikipedie.



Obrázek 7.3: Porovnání navržené metody efektivní struktury oproti SQL dotazování při zjištění chromatického čísla grafu na limburské wikipedii a počítači „MacBook“.



Obrázek 7.4: Porovnání navržené metody efektivní struktury oproti SQL dotazování při zjištění chromatického čísla grafu na české wikipedii a počítači „Minerva3“.

Z grafů na Obrázku 7.3 a 7.4 znázorňujících rychlost při použití jednotlivých verzí implementace algoritmu zjištění chromatického čísla vyplývá, že byly splněny teoretické předpoklady experimentu a navržená metoda efektivní grafové struktury vykazuje v této demonstraci násobně lepší výsledky. Pro úplnost uvádím, že graf limburské wikipedie byl obarven 319ti barvami. Chromatické číslo grafu pro komparaci s dalšími grafy by tedy bylo 319. Graf české wikipedie byl obarven 25040 barvami.

7.3 RLF coloring

Dále jsem implementoval dokonalejší algoritmus barvení grafu RLF coloring, využívající rekurzivního průchodu grafem. Algoritmus vykazuje podstatně kvalitnější obarvení než greedy coloring. Jeho časová složitost je v nejhorším případě $O(|V|^3)$. Algoritmus je popsán v kapitole 4.2. Algoritmus je implementován pomocí dvou funkcí. První z nich je funkce `rlf_color` a jedná se o hlavní průchod smyčkou barvení. Funkce je ve zdrojových kódech přiložených k této práci. Druhá funkce `rlf_color_update_nn` slouží pro výpočet množiny nesousedních vrcholů nazývané NN, viz. popis v kapitole 4.2. Množina NN je v algoritmu implementována pomocí TEMP tabulky, protože je efektivnější než implementace pomocí pole, jak bylo zjištěno v předchozích experimentech. Algoritmus označuje barvami přímo záznamy v tabulce *vertices*. Tomuto přístupu jsem dal přednost, protože barvené grafy jsou velmi rozsáhlé a realizaci dočasné struktury pro výsledek v rozsahu několika desítek tisíc záznamů neshledávám jako efektivní.

V pseudokódu tohoto grafu je uvedno přepočítání množiny NN po obarvení každého vrcholu. Protože výpočet množiny NN je značně náročný, optimalizoval jsem tento proces a množinu NN

zjišťuji pouze když dochází k zahájení hlavního cyklu a obarvení prvního uzlu novou barvou. Dále z množiny NN pouze odstraňuji uzly, které již po obarvení dalšího vrcholu grafu nesmí obsahovat. Tímto jsem dosáhl podstatně nižší časové náročnosti algoritmu.

Další optimalizací je zjištění počtu společných sousedních vrcholů pro první vrchol obarvený učitou barvou a pro další barvené vrcholy v množině NN. Toto zjištění provádím v rámci výpočtu množiny NN a tím je opět redukována časová náročnost barvení.

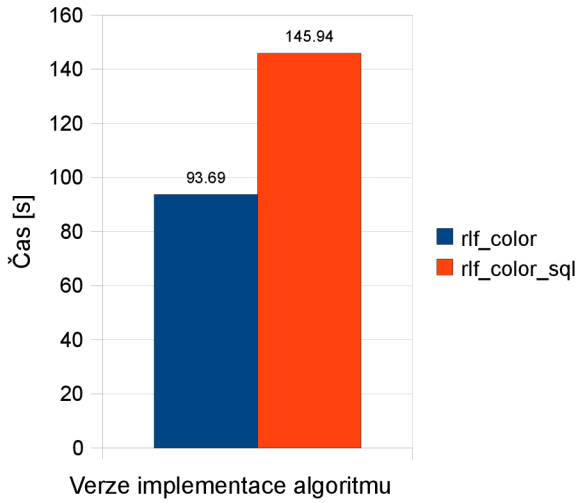
V algoritmu využívám i operace s jednorozměrnými poli, včetně mnou implementované operace `array_intersect` pro zjištění průniku polí. To z důvodu, že záznamy incidencí se nacházejí v polích a jejich konverze do TEMP tabulek by byla minimálně stejně neefektivní jako vykonání těchto několika nutných operací přímo s poli.

Pro komparaci jsem vytvořil opět identickou verzi algoritmu využívající standardních SQL dotazů, který je reprezentován funkcemi `rlf_color_sql` a `rlf_color_update_nn_sql` a pomocnými funkcemi `rlf_color_update_nn` a `rlf_color_update_nn_sql`.

7.3.1 Komparační test verzí implementace RLF coloring

Teoretickým předpokladem tohoto experimentu je, že algoritmus pro barvení, který využívá navržené metody efektivní struktury, bude mít podstatně nižší časovou náročnost. Test proběhne nad grafem limburské wikipedie. V testu jsem se zaměřil na rychlost průběhu algoritmu a proto jsem ho spustil na předzracovaném grafu se zjištěnými stupni uzlů v grafu.

Výsledek testu prezentuje graf na Obrázku 7.5. Je zřejmé, že průběh algoritmu je při implementaci pomocí navržené efektivní struktury výrazně méně časově náročný. Teorietický předpoklad tohoto testu byl tedy splněn. RLF obarvil graf limburské wikipedie 75 barvami.



Obrázek 7.5: Porovnání implementace algoritmu pomocí navržené metody s SQL dotazováním na grafu limburské wikipedie a počítači „MacBook“.

7.3.2 Komparace kvality obarvení greedy vs. RLF coloring

Teoretickým předpokladem je kvalitnější obarvení grafu při použití RLF coloringu oproti greedy coloringu a tím do značné míry ověření správnosti implementace algoritmů[8]. Tento předpoklad byl v předchozích testech splněn. RLF coloring spotřebuje k obarvení grafu použité wikipedie 75 barev oproti greedy coloringu, který spotřebuje 319 barev. Oproti tomu je greedy coloring několikanásobně rychlejší. Porovnáváme verzi s předzpracovaným grafem, greedy coloring tedy obarvil graf za 2,66 sekundy, zatímco RLF coloring barvil graf 93.69 sekundy. Teoretické předpoklady jsou potvrzeny.

7.4 VF2

Při implementaci VF2 algoritmu jsem přesně vycházel z postupu uvedeného v [VF2] a dále ze zdrojových kódů přiložených k [St08]. Zjišťování izomorfismu patří mezi komplexní a netriviální úlohy. Cílem jeho implementace je ověřit především robustnost navržené metody v náročných výpočtech a použitelnost při komplikovaných nasazeních v praxi.

Algoritmus je implementován opět v jazyce PL/pgSQL. Interně používá několik dynamických struktur, které jsou implementovány TEMP tabulkami (nahrazujícími pole). Viditelnost TEMP tabulek v rámci celé databázové session daného uživatele je při implementaci VF2 žádaná a využívaná. Algoritmus je implementován hlavní funkcí `vf2_izomorf` obsahující zavedení dočasných datových struktur, dále funkcí rekurze prohledávání nazvanou `vf2_backtrack` a dále čtyřmi pomocnými funkcemi.

Předpokládám, že „menší“ graf G_1, pro který hledáme izomorfní podgrafy ve „větším“ grafu G_2, předám funkci jako parametr. „Větší“ graf G_2, ve kterém algoritmus vyhledává izomorfní podgrafy je uložen v navržené efektivní struktuře v PostgreSQL. Vstupní graf G_1 je předán jako jednorozměrné pole s prvky datového typu simple_graph. Tento datový typ obsahuje dvě jednorozměrná pole typu integer nazvaná *vertices_in* a *vertices_out*, do kterých se zaznamenávají indexy vrcholů spojených vstupními a výstupními hranami s aktuálním vrcholem. V poli obsahujícím graf G_1 tedy záleží na umístění jednotlivých vrcholů, protože je prostředkem pro jejich vzájemnou referenci.

Funkce `vf2_izomorf` tedy zavádí potřebné datové struktury. Těmi jsou TEMP tabulky *vf2_tmp1* a *vf2_tmp2*, které svou velikostí korespondují s počtem vrcholů v grafech G_1 a G_2. Tyto tabulky reprezentují aktuální zobrazení vrcholů. Na pozici, reprezentované sloupcem i, kde $i = n$ v tabulce *vf2_tmp1* je ve sloupci *rs1* uložena pozice vrcholu, který je v páru s vrcholem pozice $i = n$, pokud je n v *M1(s)*, jinak nula. To samé platí pro *vf2_tmp2*. Tabulka *vf2_tmp1* dále obsahuje sloupce *in1* a *out1*, tabulka *vf2_tmp2* dále sloupce *in2* a *out2*. Ve sloupci *in1* záznamu pozice $i = n$ je nenulová

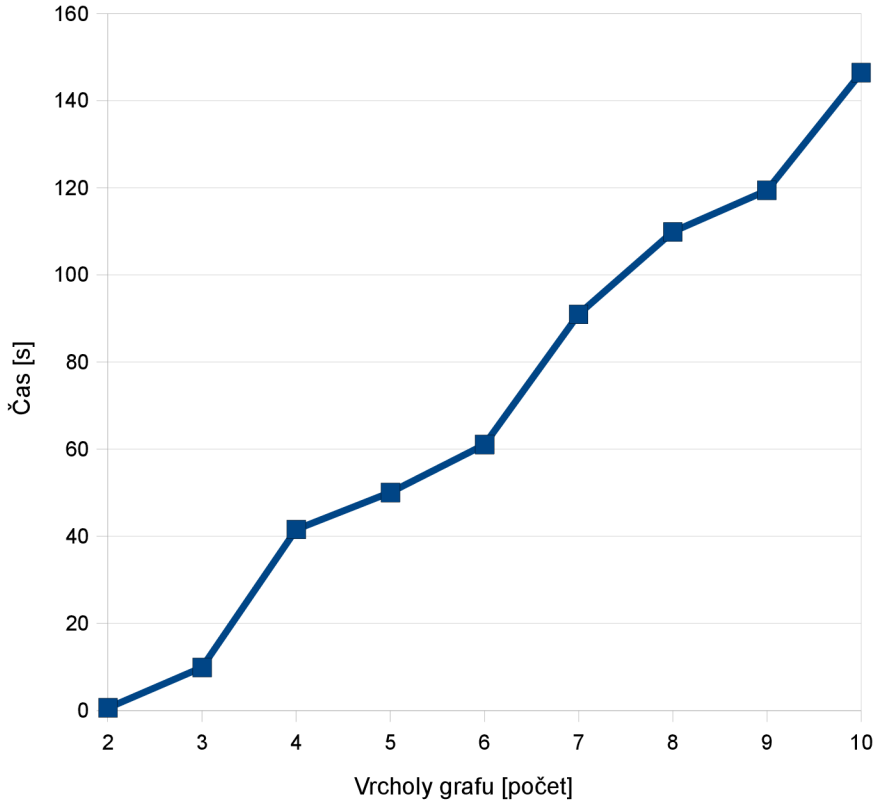
8 Správnost algoritmů byla ověřena i dalšími způsoby.

hodnota pokud záznam pozice $i = n$ je obsažen v $M1(s)$ nebo v $T_1^{in}(s)$. Toto platí analogicky pro sloupce *out1*, *in2* a *out2*. Dále je vytvořena TEMP tabulka *vf2_global*, která obsahuje globální proměnné pro celý algoritmus, konkrétně počet nalezených izomorfismů (pokud není hledání omezeno na první nalezený izomorfismus). V databázi je vytvořen datový typ `g_data` skládající se z položek typu integer obsahující počítadla, která se předávají v rekurznivních voláních.

Funkce `vf2_backtrack` je výchozí funkcí rekurzivního prohledávání grafu obdobně jako probíhá metoda prohledávání do hloubky, jak bylo již zmíněno v kapitole 4.4. Funkce nalezne možný pár a následně jsou ověřeny omezující podmínky pomocí funkce `vf2_is_possible_pair`. Pokud jsou splněny, je pár možné zkusit zahrnut do aktuálního stavu s pomocí funkce `vf2_new_pair` a dojde k zanoření do rekurze. Pokud nejsou, pokračuje určením dalšího možného páru. O obnovení předchozích hodnot polí při návratu z hlubších úrovní rekurze se stará funkce `vf2_back` s pomocí proměnné typu `g_data`.

7.4.1 Demonstrace metody efektivní struktury při hledání izomorfismu

V této demonstraci se zaměřím na nalezení izomorfního podgrafu s různým počtem uzlů. Teoretickým předpokladem experimentu je, že mé řešení bude časově méně náročné.



Obrázek 7.5: Test zjišťování izomorfismu

Z důvodu extrémní časové náročnosti této úlohy jsem zjišťoval pouze první výskyt izomorfismu. Dle hodnot v grafu je vidět, že mé řešení dosahuje velice uspokojivých výsledků. Izomorfismus byl zjišťován pro podgraf s různým počtem vrcholů (2 až 10), počet hran n byl pak v grafu $G = (V, E)$ vždy $n = |V| - 1$.

7.5 Vyhodnocení experimentů

V experimentální části jsem se zaměřil na ověření funkcionality navržené metody efektivní struktury pro správu grafů v databázovém systému PostgreSQL. Experimetny jsem provedl nad různými algoritmy a porovnával jsem především rozdíl časové náročnosti při použití navržené metody oproti použití standardních SQL dotazů. Dále jsem touto formou demonstroval funkcionalitu celého řešení.

Ve všech případech vykázala navržená metoda efektivní struktury více či méně lepší výsledky. Osobně jsem očekával podstatně nižší časovou náročnost efektivní metody v algoritmu prohledávání do šířky. Malé rozdíly obou metod v tomto algoritmu připisuji rozsahu wikipedie na které byly testy z časových důvodů prováděny, kde nejvíce času při průchodu algoritmem zabere jiná režije než samotné dotazy. Se zvyšujícím se rozsahem struktury grafu by se měl rozdíl časových náročností obou testovaných metod zvyšovat ve prospěch mnou navržené metody.

Při testech nad algoritmy pro barvení grafu byly rozdíly ve prospěch navržené metody již celkem markantní. To především díky tomu, že navržená metoda disponuje vhodným zjištěním okolí vrcholu i přes inverzní hrany incidující s vrcholem. Není tedy nutné zjišťování vstupních a výstupních hran incidujících s vrcholem v rámci dvou samostatných dotazů.

Při testech algoritmu VF2 pro zjišťování izomorfismu jsem se zaměřil především na demonstraci robustnosti celého řešení i na sofistikovanějším algoritmu jakým VF2 je. Veškeré testy zjištění izomorfismu proběhly úspěšně v přijatelně krátkých časech. Spouštěl jsem také test pro nalezení všech izomorfismů podgrafu o velikosti 7mi vrcholů, ale ten jsem z časových důvodů ukončil v momentě, kdy algoritmus nalezl již přes 50tisíc izomorfismů.

Všechna měření jsem prováděl několikrát a v grafech byly uváděny průměrné hodnoty výsledků.

8 Závěr

V práci jsem se zabýval teorií grafů a jejími aspekty v kontextu informačních technologií a možností správy grafů v databázových systémech. Po obecném uvedení do problematiky v rámci první kapitoly, východisek a cílů celé práce jsem popsal nutné základy teorie grafů.

Dále jsem se v práci zaměřil na možnosti jak graf reprezentovat při matematických výpočtech a na vhodné reprezentace grafů v programech, kde jsem některé z těchto možností využil v navazujících praktických částech. V této části práce jsem probral také dva základní způsoby prohledávání grafů a to prohledávání grafů do šířky a do hloubky, především jsem se věnoval prohledávání grafů do šířky, které bylo následně implementováno pro navrženou metodu správy grafů v databázi.

Ve 4. kapitole práce jsem se věnoval vybraným základním problémům řešeným v teorii grafů. Důkladně především problémem barvení grafů a dále sofistikovanějším problémem hledání izomorfismu grafů respektive grafu a podgrafu grafu. Těmito problémy jsem se zabýval podrobněji, protože jsem na nich později demonstroval navrženou metodu pro práci s grafy v databázi. V problematice hledání izomorfismu jsem do značné míry navázal na práci Ing. Petra Chmelaře, Zjišťování izomorfizmu mezi grafy [Ch06].

V následující části jsem představil možnosti ukládání grafů v databázových systémech. Zaměřil jsem se na popis možností ukládání grafů ve standardních, majoritně rozšířených relačních databázích a dále představil i další typy databázových systémů, včetně specializovaných systémů vycházejících přímo z teorie grafů, tzv. grafové databáze.

Protože relační model databáze je suverénně nejrozšířenější a do dnešního dne neexistuje pro žádný open source relační databázový systém rozšíření pro efektivní správu grafových dat, v této práci jsem podobné rozšíření respektive metodu navrhl. V kapitole 6. jsem se zabýval návrhem rozšíření pro open source databázový systém PostgreSQL. V návrhu jsem vycházel především z využití nízkoúrovňových identifikátorů záznamů, umožňujících maximální rychlost přístupu k datům a zachování spojové podstaty grafových dat.

V závěru práce jsem úspěšně demonstroval navrženou metodu efektivní struktury na několika algoritmech a porovnal její časovou náročnost s řešením pomocí standardních SQL dotazů. Metoda byla demonstrována na prohledávání grafů do šířky, na barvení grafů pomocí greedy coloringu a RLF coloringu a na VF2 algoritmu pro hledání izomorfismu.

Přínos práce shledávám především v navržené a realizované metodě správy grafů v PostgreSQL databázi, protože demonstruje, že na jeho principech je možné realizovat efektivní grafovou strukturu i ve standardní relační databázi, která disponuje vhodnými prostředky pro

implementaci rozšíření. Při vyhledávání informací jsem se s obdobným přístupem nesetkal (stejně ani s jakýmkoli projektem, řešícím efektivně správu grafů v některém open source relačním databázovém systému) a na poli open source databází se momentálně jeví jako unikátní. To může být obzvláště přínosné vzhledem k množství dat a projektů, které open source databázové systémy a především PostgreSQL využívají.

Je nutné říci, že je navržená a realizovaná metoda představena defakto ve své počáteční a experimentální formě a nabízí mnoho prostoru pro vylepšení pro případné produkční nasazení. Pro produnkční nasazení by bylo nutné především zdokonalit zapouzdření celého rozšíření, implementovat podstatně širší soubor základních grafových operací, zavést možnost vkládat do struktury libovolný počet grafů z různých skupin tabulek dané databáze, dořešit vhodné ošetření operace VACUUM FULL a podobně. Téměř nutností pro produkční nasazní by bylo také vhodné zasazení do systému cachování dotazů a výsledků v databázovém systému, kde v tomto ohledu je SQL dotazování na vysoké úrovni. Dále by bylo vhodné testovat navrženou metodu nad širokým spektrem grafů.

Závěrem konstatuji, že problematika grafů v databázích je velice zajímavé téma a domnívám se, že by si mezi vývojáři databázových systémů zasloužila více pozornosti. Toho by se ovšem s rozvojem internetu a důrazem na grafové struktury dat, například v podobě sociálních sítí typu Facebook, mohla brzo dočkat.

Literatura

[AN10] RODRIGUEZ, Marko; NEUBAUER, Peter. The Graph Traversal Pattern. In Graph Data Management: Techniques and Applications [online]. [s.l.] : [s.n.], 2011 [cit. 2011-05-24]. Dostupné z WWW: <http://arxiv.org/abs/1004.1001>.

[AG] AllegroGraph [online]. 2011 [cit. 2011-05-24]. Dostupné z WWW: <http://www.franz.com/agraph/allegrograph/>.

[AG05] ANGLES, Renzo; GUTIERREZ, Claudio. Survey of Graph Database Models [online]. Chile : [s.n.], 2005 [cit. 2011-05-24]. Dostupné z WWW: <http://www.dcc.uchile.cl/~rangles/research/publications/tr-dcc-2005-10.pdf>.

[AW] Wikimedia Downloads [online]. 2011 [cit. 2011-05-24]. Index of /cswiki/latest/. Dostupné z WWW: <http://dumps.wikimedia.org/cswiki/latest/>.

[BM82] BONDY, J. A.; MURTY, U. S. R. Graph Theory With Applications.. 5. Ontario, Canada : Elsevier Science Publishing Co., Inc., 1982. 264 s. Dostupné z WWW: <http://book.huihoo.com/pdf/graph-theory-With-applications/>. ISBN 0-444-19451-7.

[CA] KLOTZ, Walter. Graph Coloring Algorithms∗ [online]. [s.l.] : [s.n.], 2010 [cit. 2011-05-24]. Dostupné z WWW: <http://ldc.usb.ve/~meza/ci-5651/e-a2010/articulos/Sequential%20and%20Backtracking%20algorithms%20for%20Graph%20coloring%202002.pdf>.

[CB89] FLEMING, Candace C.; VON HALLE, Barbara. Handbook of Relational Database Design. Ontario, Canada : Addison-Wesley Professional, 1989. 624 s. ISBN 978-0201114348.

[CL88] CHANG, W. I.; LAWLER, E. L. Edge coloring of hypergraphs and a conjecture of ErdÖs, Faber, Lovász. Berlin : Springer Berlin / Heidelberg, 2010. 295 s. ISBN 0209-9683.

[DEX] Sparsity Technologies [online]. 2011 [cit. 2011-05-24]. DEX. Dostupné z WWW: <http://www.sparsity-technologies.com/dex>.

[FF62] FORD, L. R.; FULKERSON, D. R. Flows in Networks. Princeton (N.J.) : Princeton University Press, 1962. 208 s.

[GD07] Amazon.com. Dynamo: Amazon's Highly Available Key-value Store [online]. Stevenson, : [s.n.], 2004 [cit. 2011-05-24]. Dostupné z WWW: <http://bnrg.eecs.berkeley.edu/~randy/Courses/CS294.F07/Dynamo.pdf>.

[GDW] Wikipedia [online]. 2011 [cit. 2011-05-24]. Graph database. Dostupné z WWW: <http://en.wikipedia.org/wiki/Graph_database>.

[GO] MyNoSQL [online]. 2010 [cit. 2011-05-24]. Quick Review of Existing Graph Databases. Dostupné z WWW: <http://nosql.mypopescu.com/post/498705278/quick-review-of-existing-graph-databases>.

[GvS] ALBERTON, Lorenzo. TechPortal [online]. 2010 [cit. 2011-05-24]. Graphs in the database: SQL meets social networks. Dostupné z WWW: <http://nosql.mypopescu.com/post/498705278/quick-review-of-existing-graph-databases>.

[HG] The HyperGraph [online]. 2003 [cit. 2011-05-24]. Dostupné z WWW: <http://hypergraph.sourceforge.net/>.

[Hl07] HLINĚNÝ, Petr. Teorie Grafů (FI: MA010) [online]. Brno : [s.n.], 2008 [cit. 2011-05-24]. Dostupné z WWW: <http://www.fi.muni.cz/~hlineny/Vyuka/GT/Grafy-text07.pdf>.

[Ch06] CHMELAŘ, Petr. Zjišťování izomorfizmu mezi grafy [online]. Brno, 2006. 18 s. Oborová práce. Vysoké učení technické Brno, Fakulta informačních technologií. Dostupné z WWW: <http://www.fit.vutbr.cz/~chmelarp/public/06grafizo.pdf>.

[IG] InfoGrid [online]. 2009 [cit. 2011-05-24]. Dostupné z WWW: <http://infogrid.org/>.

[Ko04] KOLÁŘ, Josef. Teoretická informatika. Praha : ČVUT, 2004. 200s. ISBN 80–900853–8–5.

[Li10] JIROVSKY, Lukás. Teorie grafů ve výuce na střední škole [online]. Praha, 2010. 82 s. Diplomová práce. Univerzita Karlova v Praze, Matematicko-fyzikální fakulta. Dostupné z WWW: <http://teorie-grafu.elfineer.cz/dipl_teorie_grafu.pdf>.

[NEO] Neo4j [online]. 2007 [cit. 2011-05-24]. Dostupné z WWW: <http://neo4j.org/>.

[Obr1] Wikipedia [online]. 2011 [cit. 2011-05-24]. Seven Bridges of Königsberg. Dostupné z WWW: <http://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg>.

[Obr2] Wikipedia [online]. 2011 [cit. 2011-05-24]. Flow network. Dostupné z WWW: <http://en.wikipedia.org/wiki/Flow_network>.

[PCZ] STĚHULE, Pavel. PostgreSQL [online]. 2011 [cit. 2011-05-24]. C a PostgreSQL - interní mechanismy. Dostupné z WWW: <http://www.postgres.cz/index.php/C_a_PostgreSQL_-_intern%C3%AD_mechanismy>.

[PD] PostgreSQL 8.4.8 Documentation [online]. 2009 [cit. 2011-05-24]. Dostupné z WWW: <http://www.postgresql.org/docs/8.4/static/>.

[IU] DEOLASEE, Pavan. Postgresql-in blogspot [online]. 2008 [cit. 2011-05-24]. In place UPDATE. Dostupné z WWW: <http://postgresql-in.blogspot.com/2008/04/postgresql-in-place-update.html>.

[RLF] NGUYEN, Duc. CodeProject [online]. 2010 [cit. 2011-05-24]. Graph coloring using Recursive-Large-First (RLF) algorithm. Dostupné z WWW: <http://www.codeproject.com/KB/recipes/graph_coloring_using_RLF.aspx>.

[RS97] ROBERTSON, Neil, et al. The Four-Colour Theorem. In Journal of combinatorial theory. [s.l.] : Academic Press, 1997. s. 44.

[St08] STEJSKAL, Roman. Zjišťování izomorfizmu grafů v databázi. Brno, 2008. 47 s. Diplomová práce. Vysoké učení technické Brno, Fakulta informačních technologií.

[VDB] VertexDB [online]. 2010 [cit. 2011-05-24]. Dostupné z WWW: <http://www.dekorte.com/projects/opensource/vertexdb/>.

[VF2] CORDELLA, Luigi, et al. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE [online]. 2004, 10, [cit. 2011-05-24]. Dostupný z WWW: <https://gforge.inria.fr/docman/view.php/1841/6109/Cordella_2004.pdf>.