VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VIZUALIZACE ROZSÁHLÝCH GRAFOVÝCH DAT NA WEBU

LARGE GRAPH DATA VISUALISATION ON THE WEB

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ JARŮŠEK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Jarůšek Tomáš, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Vizualizace rozsáhlých grafových dat na webu**
Large Graph Data Visualisation on the Web
Kategorie: Web

Zadání:

1. Seznamte se s problematikou grafové reprezentace dat v databázích se zaměřením na datový model RDF.
2. Prostudujte možnosti vykreslování rozsáhlých grafových dat ve webovém prohlížeči. Zmapujte existující řešení a využitelné nástroje, např. prvek canvas.
3. Po konzultaci s vedoucím navrhněte nástroj pro vykreslení a interaktivní procházení grafových dat na webu.
4. Implementujte navržený nástroj v jazyce JavaScript.
5. Proveďte testování implementovaného řešení na rozsáhlých datech.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Lasila, I., Swick, R. R.: Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22 February 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222
- Žára, O.: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burget Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 21. října 2019

Abstrakt

Grafové databáze poskytují způsob uložení dat, který se zásadně liší od relačního modelu. Cílem této práce je poté vizualizovat tyto data a stanovit maximální objem, který jsou webové prohlížeče schopny najednou zpracovat. K tomuto účelu byla naimplementována interaktivní webová aplikace. Pro uložení dat je využit model RDF (Resource Description Framework). Ten reprezentuje data formou trojic se strukturou subjekt – predikát – objekt. Komunikace s touto databází, která běží na serveru je realizována pomocí REST API, samotný klient je poté implementován v jazyce JavaScript, kde vizualizaci zajišťuje HTML prvek canvas. Tu je možné provést pomocí třech speciálně navrhnutých metod: greedy, greedy-swap a force-directed. Výsledné hranice byly primárně zjištěny testováním časových náročností jednotlivých částí a silně závisejí na záměru uživatele. Limit byl stanoven na 150000 trojic v případě, kdy je nutné vykreslit maximální objem dat. Pokud je naopak cílem kvalita vizualizace a plynulost aplikace, tak se limit pohybuje v řádech tisíců.

Abstract

Graph databases provide a form of data storage that is fundamentally different from a relational model. The goal of this thesis is to visualize the data and determine the maximum volume that current web browsers are able to process at once. For this purpose, an interactive web application was implemented. Data are stored using the RDF (Resource Description Framework) model, which represents them as triples with a form of subject – predicate – object. Communication between this database, which runs on server and client is realized via REST API. The client itself is then implemented in JavaScript. Visualization is performed by using the HTML element canvas and can be done in different ways by applying three specially designed methods: greedy, greedy-swap and force-directed. The resulting boundaries were determined primarily by measuring time complexities of different parts and were heavily influenced by user's goals. If it is necessary to visualize as much data as possible, then 150000 triples were set to be the limiting volume. On the other hand, if the goal is maximum quality and application smoothness, then the limit doesn't exceed a few thousand.

Klíčová slova

grafová databáze, trojice, webový prohlížeč, vizualizace

Keywords

graph database, triple, web browser, visualization

Citace

JARŮŠEK, Tomáš. *Vizualizace rozsáhlých grafových dat na webu.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

Vizualizace rozsáhlých grafových dat na webu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Tomáš Jarůšek
31. května 2020

Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu mé diplomové práce Ing. Radku Burgetovi, Ph.D. za odbornou pomoc, cenné rady a vynaložené úsilí a čas, který mé práci poskytl.

Obsah

Kapitola 1

Úvod

Grafové databáze v poslední době nabývají na popularitě, a to hlavně díky schopnosti modelovat objekty z reálného světa v jejich přirozenější formě. Takové příležitosti se naskytují například na sociálních sítích, kde velice dobře reprezentují propojení jednotlivých osob mezi sebou. To vede k efektivnější práci s takovými daty. Navíc se jednodušeji vizualizují než jejich negrafové protějšky.

Jedním z nejrozšířenějších standardů v této oblasti je datový model RDF (Resource Description Framework) spravovaný konsorciem W3C. Ten poskytuje obecnou metodu pro popis dat definováním vztahů mezi datovými objekty a je založen na principu tvoření výroků o datech ve formě trojic, které mají formu subjekt – predikát – objekt. K vizualizaci takových dat poté může existovat několik důvodů, hlavním je ale snaha o lepší pochopení interních struktur dat. Platforem pro vizualizaci je také několik, tento projekt ale využívá webovou stránku, a to primárně kvůli tomu, že následuje dnešní trend, který se snaží přesunout veškerou funkcionalitu online. To usnadňuje práci uživateli, jsou s tím ale spojeny problémy. Reálné datasety mohou být velice rozsáhlé, prohlížeče však disponují limitovanými prostředky. Samotná vizualizace grafů také není triviálním úkonem, což povede k dalšímu navýšení požadavků na výkon. Cílem tedy bude implementovat webovou aplikaci, která bude vizualizovat grafové databáze několika metodami. Ty budou otestovány z hlediska kvality vizualizace a výkonu. Nakonec budou zhodnoceny jejich výsledky a bude stanoven počet trojic, který jsou webové prohlížeče schopny najednou vykreslit.

V teoretické části začínající kapitolou 2 budou rozebrány grafové databáze obecně – typy, struktura a vlastnosti. Kapitola 3 se poté podrobněji zaměří na jednotlivé aspekty modelu RDF. V navazující části 4 budou rozebrány použité technologie pro vývoj aplikace, na což naváže návrh popsaný v sekci 5. Ten se zabývá jak zvolenými algoritmy vizualizace, tak celkovou strukturou. Implementace je následně uvedena v kapitole 6 a specifikuje realizaci jednotlivých aspektů webové aplikace. Poslední sekce 7 poté popisuje provedené testování a poskytuje celkové shrnutí a vyhodnocení.

Kapitola 2

Grafové databáze

Grafová databáze je databáze, která pro dotazování a ukládání používá grafové struktury s uzly, hranami a dalšími vlastnostmi pro reprezentaci uložených dat. Základ tedy tvoří graf, jehož uzly zastupují datové položky. Hrany grafu poté reprezentují vztahy mezi nimi – data jsou propojena přímo mezi sebou a ve většině případů je možné je získat jednou operací. Grafové databáze navíc umožňují jednoduchou a intuitivní vizualizaci, což je užitečné pro analýzu i silně propojených dat.

Grafové databáze jsou součástí databázové třídy NoSQL, která se snaží vyřešit nebo omezit limitace existujících relačních databází. Vztahy mezi daty jsou v grafových databázích definovány explicitně na rozdíl od dalších návrhů typu NoSQL a relačních databází, u kterých jsou vztahy definovány implicitně. To má výhodu především v práci s komplikovanými hierarchickými strukturami, které se dají jen obtížně modelovat v ostatních typech databázových systémů. K tomu se k datům z těchto struktur dá jednoduše a rychle přistoupit. Grafové databáze jsou podobné k síťovým databázím. Podobají se tím, že reprezentují data pomocí obecných grafů, liší se však mírou abstrakce a obtížností průchodu grafem po hranách.

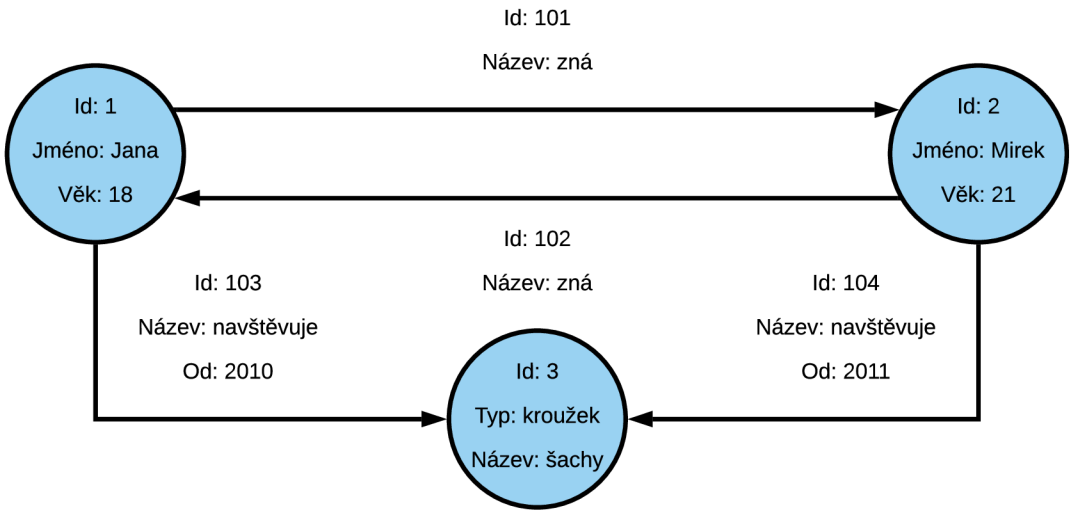
Princip uložení dat v grafových databázích může být realizován několika různými způsoby. Některé závisejí na již existujícím relačním databázovém systému, ty ukládají grafová data v tabulce (tabulka sama o sobě představuje logický objekt, a proto si tento přístup vynucuje další úroveň abstrakce mezi grafovou databází, systémem řídící databázi a fyzickými zařízeními, na kterých jsou samotná data uložena). Jiné používají pro ukládání dat dokumentové databáze nebo databáze typu key-value (v překladu klíč-hodnota), z čehož následně vyplývá, že implicitně spadají do modelu NoSQL. Většina takových grafových databází také přidává koncept značek a vlastností, což umožňuje snazší kategorizaci prvků, a tedy jejich mnohem rychlejší načítaní ve velkých skupinách.

Načítání dat z grafové databáze potřebuje dotazovací jazyk jiný než SQL, který byl navržen pro manipulaci s daty v relačních systémech, a tedy nemůže být ve své standardní formě jednoduše použit pro průchod grafem. Aktuálně zatím neexistuje univerzální standard pro grafové dotazovací jazyky, který by měl stejné postavení, jako má SQL pro relační databáze. Je tedy vytvořeno několik implementací, z nichž většina se vztahuje k jednomu konkrétnímu produktu. To ale neznamená, že snaha o standardizaci neexistuje. Standardizační snahy vedly k vytvoření víceúčelových dotazovacích jazyků, jako je Gremlin, SPARQL a Cypher. Kromě toho, že grafové databáze disponují dotazovacím rozhraním, umožňují některé implementace i přístup přes aplikační rozhraní (dále API).

Grafové databáze přitáhly velkou pozornost na přelomu tisíciletí díky úspěchu, který s nimi zaznamenaly velké počítačové firmy. Tyto implementace ale byly proprietární, což následně vedlo k rozmachu open-source implementací [6], [19], [21].

2.1 Struktura

Graf v grafové databázi je založen na konceptech grafové teorie. Je to množina objektů – uzlů a hran. Příklad takového grafu ukazuje následující obrázek 2.1.



Obrázek 2.1: Ukázka struktury jednoduchého grafu v grafové databázi.

Grafové databáze vyobrazují data ve skutečné podobě. To je realizováno převedením dat do uzlů a vztahů do hran. Uzly reprezentují entity jako například osoby, firmy, účty nebo jakoukoliv jinou věc, o které se má udržovat záznam. Obecně můžeme říci, že uzel je ekvivalentní záznamu nebo řadě v relační databázi nebo dokumentu v dokumentové databázi. Hrany naproti tomu reprezentují vztahy mezi nimi. Mohou být orientované nebo neorientované. V neorientovaném grafu jedna hrana reprezentuje jednu vlastnost pro oba propojené uzly, naproti tomu v orientovaném grafu mohou hrany s různou orientací reprezentovat naprosto rozdílné vlastnosti. Hrany jsou klíčovým konceptem v grafových databázích, protože do modelu přidávají abstrakci, která v relačních databázích nebo databázích typu NoSQL neexistuje.

Každý uzel a hrana poté mají svůj identifikátor, který je jednoznačně identifikuje. Dále mohou obsahovat vlastnosti, které je popisují. U uzlů to může například být jméno nebo věk, pokud uzel reprezentuje osobu. Vlastnosti hran poté konkrétně popisují vztah, který hrana reprezentuje. Dvě osoby mohou být ve vztahu s názvem „rodinný příslušník“ nebo osoby a škola mohou být ve vztahu pojmenovaném „student“.

V grafových databázích je propojení entit jasně viditelné, a tedy pozorování dat, která jsou takto reprezentována umožňuje lepší a efektivnější identifikaci struktur, které obsahují [19].

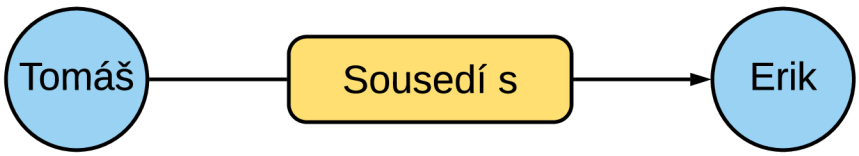
2.2 Vlastnosti

Grafové databáze jsou výkonným nástrojem pro dotazy, které implicitně vychází z datové struktury. Pokud je například nutné zjistit nejkratší vzdálenost mezi dvěma uzly v grafu, vypočítat jeho průměr nebo provést analýzu provázanosti několika uzlů, je možné tak učinit pomocí několika jednoduchých a přirozených dotazů, které přímo těží z grafové struktury.

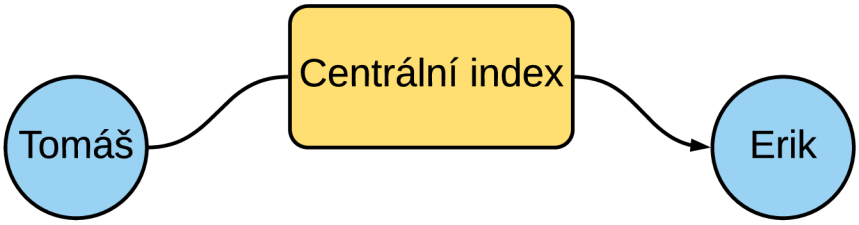
Grafy jsou k tomu vysoce flexibilní, což umožňuje uživateli vkládat nová data do existujícího grafu bez ztráty jakékoliv již existující aplikační funkcionality. Tato vlastnost se také pozitivně odráží v obtížnosti návrhu databázových systémů, protože není nutné znát přesnou strukturu předem. V relačních databázích je potřeba přesně určit počet tabulek, jejich obsah a vztahy, podle čeho se také bude silně odvíjet implementace dalších částí, které budou tuto databázi využívat. Naproti tomu v grafových databázích stačí navrhnout pouze hlavní části a další specifické vlastnosti mohou být přidány bez jakékoliv změny původní struktury.

Index-free adjacency

Index-free adjacency je koncept, který zajišťuje to, že všechny uzly přímo odkazují na všechny svoje sousedy (obrázek 2.2). Jedná se o jeden z pilířů grafových databází a předpokládá se, že všechny nativní implementace jej splňují. V relačních databázích se naproti tomu jakékoliv indexy na databázové prvky musejí získat od nějakého mezisystému, tedy prostředníka (obrázek 2.3).



Obrázek 2.2: Ukázka vztahu mezi dvěma osobami v grafové databázi. Uzly zde přímo obsahují odkazy na svoje sousedy. Převzato z [16].



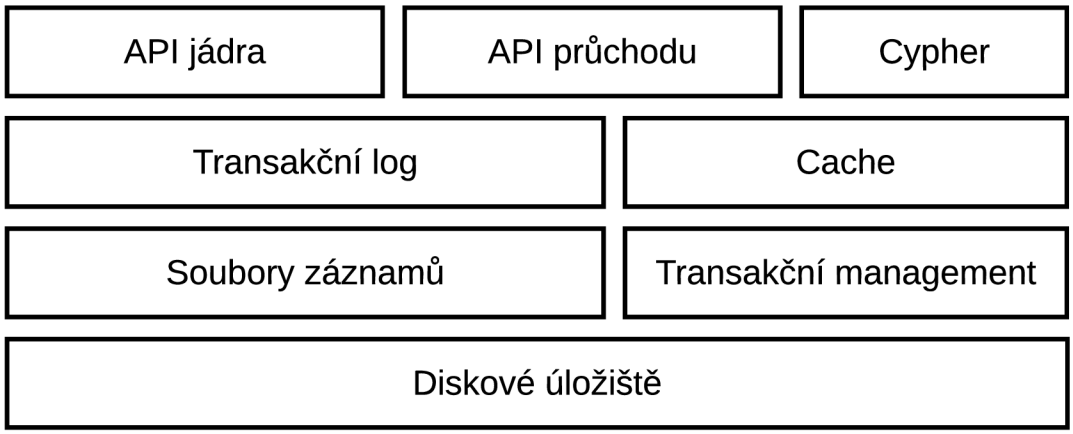
Obrázek 2.3: Ukázka vztahu mezi dvěma osobami v relační databázi. Pokud je nutné přistoupit k sousedovi uzlu, tak se nejdříve musí provést vyhledání pomocí mezisystému, ten poté vrací požadovaný index. Převzato z [16].

Tento koncept zapříčiňuje, že čas potřebný pro zpracování dotazu není proporcionální k objemu dat, ale pouze k té části grafu, která byla prohledána. To zabraňuje zhroucení rozsáhlých grafových databází. Je to také důvod, proč jsou grafové databáze tak dobré pro

procházení grafů, což je dále umocněno snahou tento koncept implementovat fyzicky – v tom případě jsou databázové indexy reprezentovány fyzickými adresami v paměti [10], [20], [16].

Uložení

Jak již bylo řečeno v úvodní kapitole, ukládací mechanismy se různí. Mohou být buď založeny na již existujících implementacích nebo vystavěny od nuly s maximální podporou grafové architektury. Nativní grafové databáze jsou navrženy tak, aby dosáhly co nejvyššího výkonu při obecném průchodu grafem, a proto bude dále v této kapitole analyzována pouze tato varianta. Následující obrázek 2.4 ukazuje strukturu systému Neo4j, který je zástupcem této kategorie.



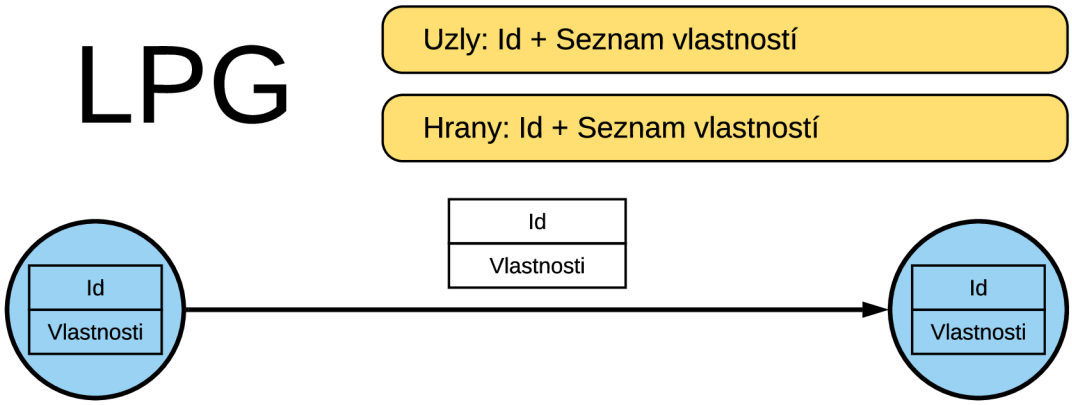
Obrázek 2.4: Interní struktura nativního grafového systému Neo4j. Převzato z [10].

Grafová data jsou uložena v logických částech, z nichž každá reprezentuje specifickou část grafu jako uzly, hrany a vlastnosti. Toto rozdělení je jeden z hlavních principů, který urychluje průchod. Každý uzel obsahuje pouze ukazatele na seznamy vlastností a vztahů, což ho dělá velice robustním. Grafový systém dále obsahuje dedikované API pro průchod grafem a cache (v překladu mezipaměť) pro rychlé znovupoužití již načtených dat. Jedna z částí je také vyhrazena pro obsluhu dotazování, v případě Neo4j se jedná o jádro jazyka Cypher. Zbylé části jsou podobné, jak v relačních systémech [10].

2.3 Grafové modely

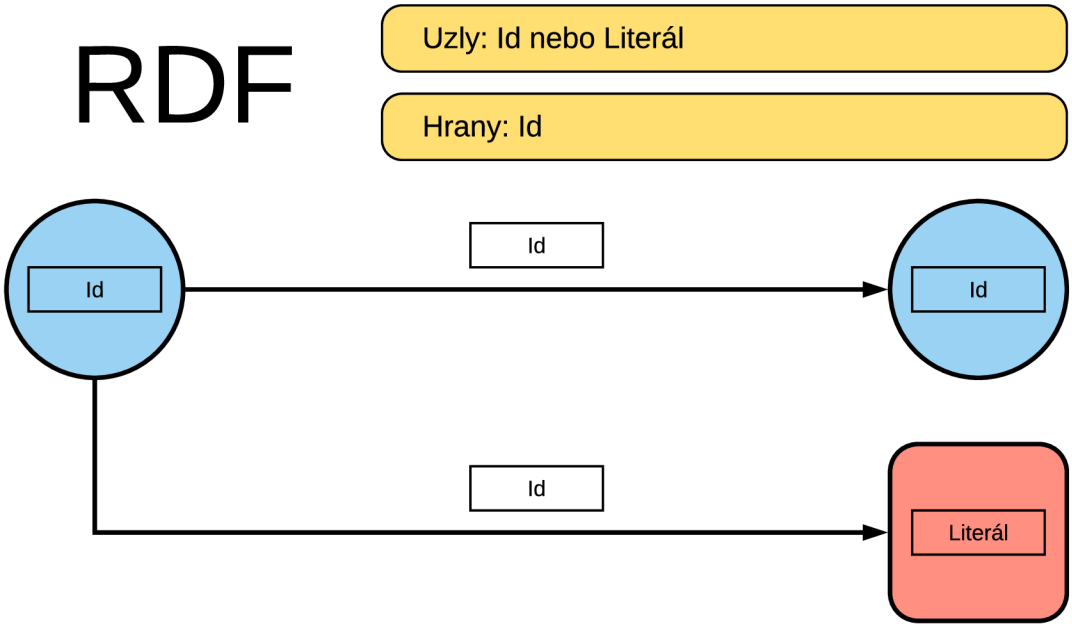
Existují dva hlavní grafové modely: labeled propery graph a resource description framework. Labeled property graph (dále LPG) se zaměřuje primárně na ukládání a dotazování, naproti tomu model typu resource description framework (dále RDF) se více orientuje na efektivní provádění změn v datech.

V LPG mají všechny uzly a hrany unikátní identifikátor a množinu vlastností, které je popisují. Unikátnost je důležitá, aby šlo každý prvek jednoznačně identifikovat. Dále je se všemi prvky spojena interní struktura, ve které jsou obsaženy všechny jejich vlastnosti. Strukturu LPG demonstruje následující obrázek 2.5.



Obrázek 2.5: Struktura uzlů a hran v grafovém modelu LPG. Převzato z [8].

Model RDF je naproti tomu založen na trojicích, které mají strukturu subjekt – predikát – objekt. Každá z nich reprezentuje jeden vztah mezi dvěma entitami, které mohou být dvojího typu: objekt nebo literál. Objekty i hodnoty tedy budou v grafu reprezentovány uzlem. Z toho vyplývá, že na rozdíl od LPG, uzly ani hrany nebudou obsahovat interní strukturu, ale budou pouze definovány unikátním identifikátorem nebo u literálů přímo hodnotou. Strukturu RDF demonstruje následující obrázek 2.6.

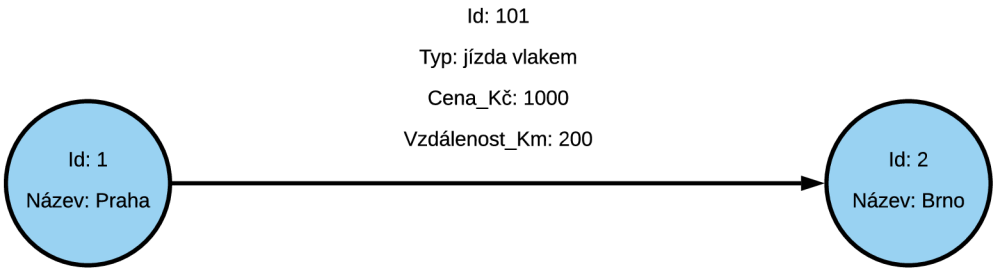


Obrázek 2.6: Struktura uzlů a hran v grafovém modelu RDF. Převzato z [8].

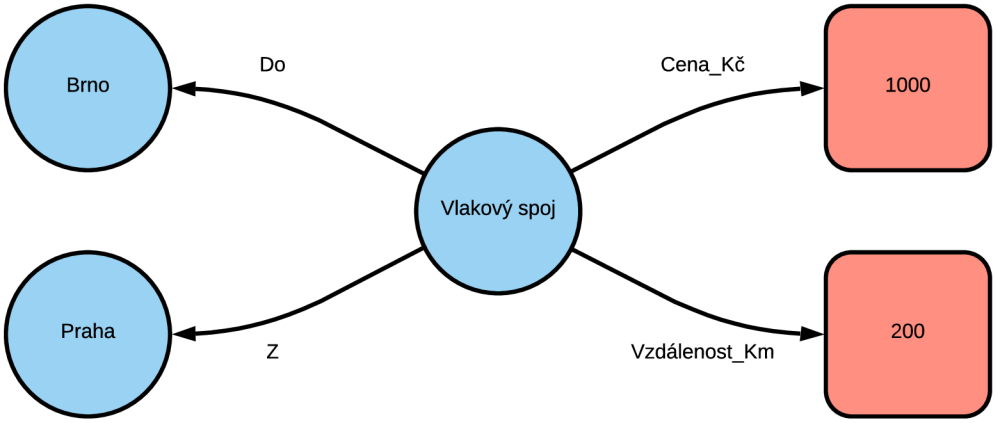
Protože model RDF neposkytuje žádné interní struktury pro vztahy, jsou vlastnosti obou modelů rozdílné v několika ohledech. První rozdíl je ten, že model RDF neumož-

ňuje do grafu vložit vztahy stejného typu. Pokud například existují entity s názvy „osoba“ a „museum“, je možné je v modelu LPG několikrát propojit stejně pojmenovaným vztahem „navštívil“ (identifikátory ale budou rozdílné). To potom umožňuje dotázat se na jeho kardinalitu. V modelu RDF mají vztahy pouze identifikátor, a proto se všechny další stejně pojmenované vztahy budou ignorovat a neposkytnou žádné další informace.

Další rozdíl spočívá v topologii výsledného grafu. Jako ukázku je možné zvolit reprezentaci jízdy vlakem z Prahy do Brna, která stojí 1000 Kč a má vzdálenost 200 km. Protože tento vztah obsahuje několik vlastností, povede to k velice rozdílným grafům, jak je vidět na obrázcích 2.7 a 2.8 [8].



Obrázek 2.7: Diagram reprezentující jízdu z Prahy do Brna v modelu LPG. Převzato z [8].



Obrázek 2.8: Diagram reprezentující jízdu z Prahy do Brna v modelu RDF. Převzato z [8].

2.4 Porovnání s relačními databázemi

Od osmdesátých let minulého století, kdy byly položeny základy relačního databázového modelu jsou relační databáze de facto standardem pro ukládání velkých objemů dat. Mají ale svoje nevýhody – vyžadují striktní schéma a nutnost normalizace dat vytváří omezení na typy vztahů, na které je možno se dotázat. Také zvyšující se objem dat, který je nutno zpracovávat způsobuje problémy v relačních databázích.

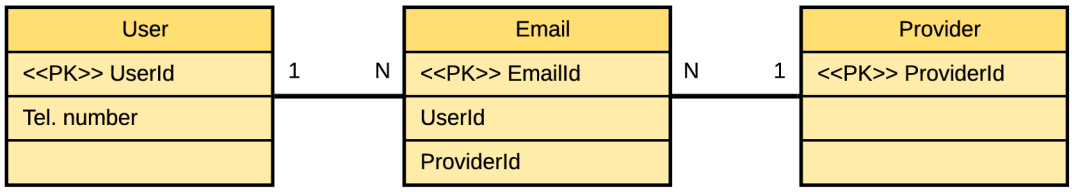
Hlavním důvodem pro normalizaci dat v relačních databázích je podpora vlastností ACID (atomicita, konzistence, izolovanost, trvalost) u transakcí. Normalizace vede k odstranění duplikátů z databáze a zachování datové konzistence. To ve výsledku vyústí v to, že data jsou rozdělena do velkého počtu tabulek. Tento přístup byl zaveden k dosažení rychlých přístupů po řádcích, problémy však nastanou při vytváření komplexních vztahů mezi uloženými daty. Takové vztahy sice mohou být analyzovány v relačním modelu, vede to však k vytvoření velice složitých dotazů, které ve výsledku povedou k provedení mnoha operací join přes různé atributy a tabulky. Další věc, která se musí zohlednit je omezení vycházející z cizích klíčů, což vede k dalšímu zpomalení komplexních dotazů.

Grafové databáze tedy v porovnání s relačními databázemi nabízejí rychlejší práci s daty, které se dají dobře reprezentovat asociativními datovými strukturami a obvykle také poskytují přirozenější mapování do struktur aplikací založených na objektově orientovaném modelu. Dále disponují lepší škálovatelností a ani u velkých datových sad nepotřebují pro dotazování operace join, které jsou, jak již bylo řečeno velmi drahé obzvláště u rozsáhlých tabulek. Grafové databáze také méně závisejí na předem daném schématu a dají se snadno modifikovat a rozšiřovat, což je dělá vhodnými pro aplikace, které pracují s často měnícími se daty nebo schémata. Naproti tomu, relační databáze obyčejně poskytují lepší výkon při opakování jedné operace nad rozsáhlými daty díky vhodnější formě jejich uložení.

Přestože grafové databáze zaznamenaly v poslední době nárůst popularity oproti relačním, neměl by grafový model sám o sobě sloužit jako důvod pro nahrazení nových nebo již existujících relačních databází. Grafové databáze by se měly brát v úvahu, pokud existují konkrétní důvody, které budou mít za následek snížení časových náročností nebo latence pro daný systém.

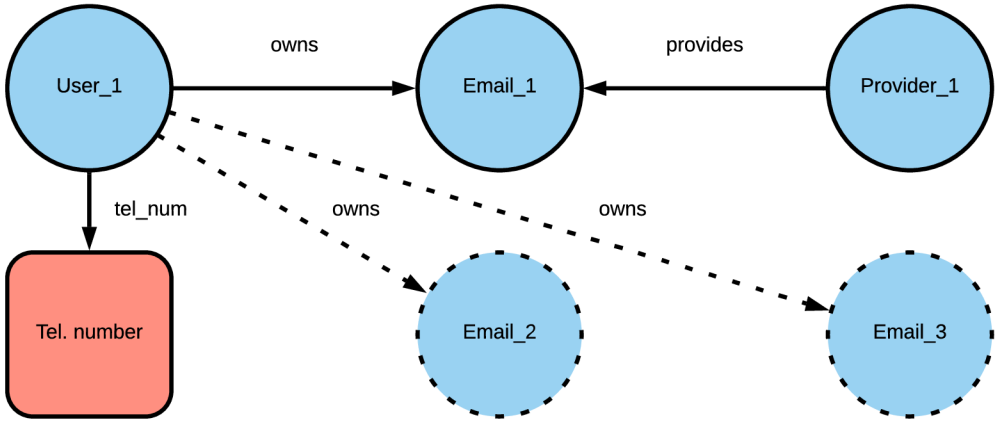
Příklady

Po teoretickém srovnání vlastností grafových a relačních databází se tato kapitola bude zabývat konkrétními instancemi pro lepší ilustraci. Nechť existuje následující relační (obrázek 2.9) a grafové (obrázek 2.10) databázové schéma reprezentující stejná data.



Obrázek 2.9: Relační schéma ukázkového příkladu.

Prvním požadavkem bude vyhledání všech uživatelů, jejichž telefonní číslo začíná předčíslím „420“. V relačních systémech by toto bylo realizováno nejprve prohledáním všech záznamů v tabulce *users*, která obsahuje telefonní čísla. U každého by se pak ověřilo, zda obsahuje požadované předčíslí. Tento krok je obzvláště v tabulkách s velkým počtem sloupců časově náročný, a proto relační databáze zavádí indexování. Data jsou uložena v menších podtabulkách, které obsahují pouze vybraná data a unikátní klíč. Pokud jsou telefonní čísla indexována, tak hledání proběhne pouze nad touto menší tabulkou, ze které se získají klíče vyhovujících záznamů. Poté proběhne jejich vyhledání v hlavní tabulce. Tabulky jsou obyčejně fyzicky uloženy v takovém formátu, aby indexování bylo co nejefektivnější. Na druhou



Obrázek 2.10: Grafové schéma ukázkového příkladu.

stranu grafové databáze nevyžadují další dělení dat na pozadí, protože telefonní čísla jsou již implicitně reprezentovány samostatnými entitami. Rychlosti vyhledání obou přístupů budou tedy srovnatelné.

Druhý požadavek vyžaduje vyhledání všech emailových schránek, které patří konkrétnímu uživateli. Jak již bylo řečeno v teoretické části, relační databáze sami o sobě neobsahují pevné vztahy mezi záznamy, místo toho jsou data provázána uložením primárního klíče záznamu z jedné tabulky v jiné tabulce. V aktuálním schématu je toto využito v tabulce *emails*, která reprezentuje seznam emailových schránek. Obsahuje sloupce *UserId* a *ProviderId*, které reprezentují jednotlivé uživatele a poskytovatele, a tím tedy realizují daná propojení. Aby bylo možné spojit uživatele s jeho emailovými schránkami, systém musí nejdříve načíst primární klíč vybraného uživatele, ten pak spojit s tabulkou *emails* pomocí sloupce *UserId* a vyextrahovat jejich informace. Tato operace join je časově náročná a při provádění komplikovaných dotazů se pravděpodobně bude muset provést vícekrát, což způsobí multiplikativní nárůst složitosti. Nakonec je nutné výsledky seřadit a přerovnat do formátu jaký požaduje dotaz. Grafové databáze naproti tomu explicitně ukládají vztahy mezi záznamy. Místo toho, aby uživatelé a emailové schránky byli spojeni přes klíč v tabulce *emails*, záznam o uživateli obsahuje ukazatel, který přímo ukazuje na záznam emailové schránky. Jakmile je tedy načten záznam o uživateli, je možné z něj přímo přejít na jakýkoliv jiný s ním spojený záznam, bez nutnosti vyhledávání. Tím odstraníme nutnost použití operací typu join.

Nakonec je možné spojit oba předchozí požadavky do jednoho. Mějme dotaz, který chce vybrat všechny emailové schránky, které patří uživatelům, jejichž telefonní číslo začíná předčíslím „420“. Kombinace obou předchozích postupů povede k takovémuto provedení: grafový databázový systém nejprve provede konvenční vyhledání všech vyhovujících uživatelů, poté ale získá všechny emailové schránky přes přímé vazby mezi nimi a tím pádem nemusí pracovat se všemi entitami *Email*. Zde je tedy přímo patrné urychlení, které grafové databáze nabízí.

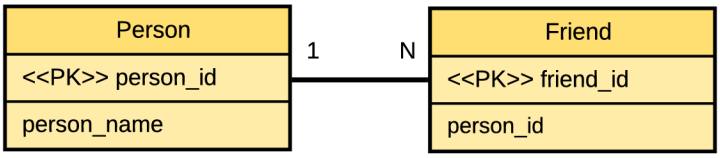
Znatelné navýšení výkonu však nastane ve chvíli, kdy vyhledání, které jsou nutné v dotazu provést zahrnuje entity, které jsou od sebe vzdáleny o dvě a více úrovní. V aktuálním schématu jsou emailové schránky nabízeny několika poskytovateli (tabulka *providers*). Pokud by teď bylo nutné najít všechny uživatele s emailovými schránkami od konkrétního

poskytovatele, zabralo by to relační databázi dvě operace join spolu s další režií na formátování výsledku. V grafové databázi by se vyhledali daní poskytovatelé, poté by ale pouze proběhl průchod pomocí propojení přes emailové schránky až k uživatelům. V tomto přístupu se vždy pracuje s daty, které jsou ve v vztahu s původní množinou poskytovatelů, tedy pouze se zlomkem celkového objemu dat. Pokud provedeme analýzu časové složitosti, tak zjistíme, že všechny kroky provedení tohoto dotazu spadají do logaritmické nebo konstantní časové složitosti. Tedy celková složitost takového dotazu je logaritmická vzhledem k objemu dat.

Když grafová databáze využívá model LPG, je možné s pomocí vlastností prvků dále zvýšit efektivitu dotazů za cenu vyšší složitosti. Pokud bude existovat škola, do které chodí osoby, tak jejich vlastnosti mohou být například „učitel“ nebo „žák“. To by poté umožnilo rychlejší vyhledání osob v konkrétní skupině. Vlastnosti je však možné přiřadit i vztahům. Vztah mezi učitelem a žákem může být pojmenován „učí“, což opět umožní rychlejší vyhledání všech žáků, které učitel vyučuje. Ekvivalentní SQL dotaz by k realizaci těchto úkonů potřeboval načíst data z tabulky propojující žáky a učitele. I když vlastnosti prvků zvyšují efektivitu dotazů, jejich hlavní výhoda je v přidání sémantiky pro konečné uživatele.

Relační databáze je tedy výhodnější zvolit pro data s plochou strukturou a s entitami položenými blízko u sebe, naproti tomu grafové databáze jsou vhodné pro hustě propojená data. To je v této době dělá populárními například pro ukládání uživatelů na sociálních sítích, kde velice efektivně modelují jejich vztahy mezi sebou.

Jako finální demonstraci provedeme srovnání jednoduchých dotazů v grafových a relačních databázích. Předpokládejme, že existuje následující schéma (obrázek 2.11) s tabulkami *people* a *friends*. Všechny dotazy poté vyhledávají Karlovi přátele. Dotazy v grafových dotazovacích jazycích pracují nad ekvivalentním grafovým schématem [21].



Obrázek 2.11: Relační schéma ukázkového příkladu pro demostraci dotazovacích jazyků.

Standardní relační dotaz v jazyce SQL:

```
SELECT p2.person_name
FROM people p1
JOIN friend ON (p1.person_id = friend.person_id)
JOIN people p2 ON (p2.person_id = friend.friend_id)
WHERE p1.person_name = 'Karel';
```

Grafový dotazovací jazyk Cypher:

```
MATCH (ee:person)-[:FRIEND-WITH]-(friend)
WHERE ee.name = "Karel"
RETURN ee, friend
```

SPARQL – rozšířený grafový dotazovací jazyk pro databáze využívající model Resource Description Framework:

```
PREFIX ex: <http://example.com/>
SELECT ?name
WHERE
{
    ?s ex:name "Karel";
    ex:knows ?o .
    ?o ex:name? name .
}
```

Kapitola 3

Resource Description Framework

Resource Description Framework (dále RDF, v překladu systém popisu zdrojů) je datový model standardizovaný konsorciem W3C. Poskytuje obecnou metodu pro popis dat definováním vztahů mezi datovými objekty. Byl navržen v roce 1999 a specifikace 1.0 byla publikována v roce 2004. Právě tento model bude využit pro reprezentaci grafových dat v tomto projektu.

Je založen na principu tvoření výroků o datech ve formě trojic, které mají formu subjekt – predikát – objekt. Subjekt označuje zdroj a objekt aspekty nebo vlastnosti zdroje. Predikát poté vyjadřuje vztah mezi subjektem a objektem.

Příklad trojice je možné demonstrovat na výroku „Honza měří 180 cm“. V RDF trojici subjekt reprezentuje „Honza“ a objekt „180 cm“. Predikát „měří“ poté definuje vztah mezi nimi. Zde je patrné, že se tento přístup výrazně liší od standardního modelu entita – atribut – hodnota. V něm by byl záznam uložen jako entita „Honza“, atribut „výška“ a hodnota „180 cm“.

Kolekce těchto trojic se poté obecně nazývá RDF store a reprezentuje značený orientovaný multigraf. To znamená, že databázový systém, který využívá RDF má všechny benefity a negativa, které plynou z teorie grafů a grafových databází. Ty již byly zmíněny v předchozí kapitole [15], [18].

3.1 Serializace

RDF poskytuje několik různých serializačních formátů, z nichž nejpoužívanější jsou:

- RDF/XML – formát založený na XML, první standardizovaný formát pro RDF
- JASON-LD – formát založený na JSON
- Turtle – kompaktní formát, snažící se o jednoduchost čtení pro uživatele
- N-triples – vychází z Turtle, je jednoduší, rychlejší generování a čtení

RDF/XML bývá někdy také označován zkratkou RDF, protože byl prvním definovaným standardem, je však nutné nezaměňovat RDF/XML serializační formát se samotným abstraktním RDF modelem [18].

3.2 Identifikace

Subjekt v RDF trojici je buď prázdný uzel nebo je použit uniform resource identifier (dále URI). Oba tyto stavy reprezentují zdroj. Zdroje reprezentované prázdnými uzly se nazývají anonymní a nedají se přímo identifikovat. Predikát je reprezentován také přes URI a identifikuje zdroj reprezentující vztah. Objekt naproti tomu může nabývat třech stavů. První je opět URI, druhý je prázdný uzel a třetí je řetězcový literál. Ten představuje jedinou možnost pro popis vlastností.

Ve webových aplikacích, kde je RDF populární například pro reprezentaci propojení osob na sociálních sítích, mohou RDF URI reprezentovat adresy, přes které se dá reálně přistoupit k datům na webu. Obecně ale platí, že RDF URI nejsou určeny pouze pro popis webových zdrojů, ale mohou nabývat jakýchkoliv hodnot a nemusí se přes ně dát nikam přistoupit. Proto se poskytovatelé i uživatelé musejí předem domluvit na jejich významu. Toto není přímo součástí RDF standardu, existuje však několik běžně používaných verzí. Jako příklad je možné uvést URI „http://example.com/bookstore#“, která může označovat uzel reprezentující knihkupectví. URI je poté možné zkracovat například tak, že „http://example.com/“ bude zapsáno jako „ex:“ [18].

3.3 Dotazovací Jazyky

Jediným de facto požívaným dotazovacím jazykem je SPARQL, který doporučuje samotné W3C. Syntaxe SPARQL vychází z SQL a je speciálně navržena pro práci s grafovými databázemi. Jediný další zástupce, který stojí za zmínění je Versa, což je kompaktní jazyk nezávislý na SQL a implementovaný v Pythonu.

SPARQL

SPARQL (čteno sparkl, zastupuje rekurzivní akronym SPARQL Protocol and RDF Query Language) je dotazovací jazyk určený pro manipulaci a získávání dat v RDF formátu. Standardizuje ho konsorcium W3C a existuje několik implementací v různých jazycích. Navíc existují nástroje, které dokáží přeložit SPARQL dotazy do jiných dotazovacích jazyků, jako je například SQL. Dotaz se skládá ze seznamů trojic (forma subjekt – predikát – objekt, která vychází z RDF). Každý prvek této trojice je poté možno nahradit neznámou (v originále wildcard). Jako výsledek jsou potom vraceny ty položky z databáze, které se shodují se vzorem definovaným v dotazu. K dispozici jsou také všechny běžně používané analytické a agregační funkce jako například SORT(), SUM() nebo MAX().

SPARQL definuje čtyři typy dotazů:

- SELECT – vrací nezměněná data z databáze, tedy formou tabulky
- CONSTRUCT – vrací data přetransformovaná do formátu RDF
- ASK – dotaz vracející hodnotu „true“ nebo „false“
- DESCRIBE – popisuje výsledek dotazu konstrukcí relevantního RDF grafu

Přesný popis formátu je možné demonstrovat na jednoduchém dotazu typu SELECT, který má za úkol vrátit všechny hlavní města, která leží v Africe. Výsledek bude obsahovat názvy těchto měst a států, ve kterých leží.

```
PREFIX ex: <http://example.com/exampleOntology\#>
SELECT ?capital, ?country
WHERE
{
    ?x ex:cityName ?capital;
    ex:isCapitalOf ?y.
    ?y ex:countryName ?country;
    ex:isInContinentex:Africa.
}
```

Příkaz začíná definicí prefixu, tím se zkrátí URI a zpřehlední se tak zbytek dotazu. Poté následuje klíčové slovo SELECT a seznam neznámých. Ty se označují symbolem „?“. Tento příklad obsahuje neznámé „?capital“ a „?country“ – to budou výsledky, které dotaz vrátí ve formě tabulky se dvěma sloupci. Nakonec dotaz obsahuje klíčové slovo WHERE. V jeho těle budou vloženy všechny vzory, které musí výsledky splňovat.

Pokud trojice končí středníkem, tak další trojice, které po ní následují nemusí obsahovat kompletní definici a prázdná místa budou doplněna z předchozí trojice. To znamená, že v tomto příkladu má druhá trojice „ex:isCapitalOf ?y“ stejný význam jako „?x ex:isCapitalOf ?y“. Celý dotaz se poté dá formulovat čtyřmi trojicemi. Neznámá „?x“ reprezentuje entitu město a „?y“ entitu stát. První trojice vyjadřuje to, že u měst je nutné vrátit pouze vlastnost název a druhá poté specifikuje, že entita město musí být hlavním městem libovolného státu. Třetí opět vyjadřuje nutnost vrácení pouze vlastnosti název, tentokrát ale státu. Poslední pak zužuje výběr pouze na ty státy, které leží v Africe. Systém poté projde databází a vrátí ty záznamy, které všechny vzory splňují.

SPARQL navíc obsahuje rozsáhlý soubor dalších specifikací, které definují další funkcionality mimo dotazování. Aktuální specifikace jazyka SPARQL poskytují například příkazy UPDATE, INSERT a DELETE, pomocí kterých se dají vkládat nové trojice nebo modifikovat již existující [2], [14].

3.4 Příklad

Funkcionalitu RDF je možné ilustrovat na jednoduchém příkladě, který popisuje kontakt na osobu jménem Eric Miller [15]. Struktura tohoto kontaktu je vyobrazena na obrázku 3.1 a skládá se z těchto částí:

- URI „http://www.w3.org/People/EM/contact#me“ identifikuje entitu kontakt.
- URI „http://www.w3.org/2000/10/swap/pim/contact#Person“ identifikuje entitu osoba.
- URI „mailto:em@w3.org“ identifikuje entitu emailová schránka.
- Zbylé URI identifikují vlastnosti entity kontakt.
- „Eric Miller“ a „Dr.“ jsou konkrétní hodnoty (literály).

Příklad uložení kontaktu ve formátu RDF/XML:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#"
xmlns:eric="http://www.w3.org/People/EM/contact#"
```

```
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me">
        <contact:fullName>Eric Miller</contact:fullName>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me">
        <contact:mailbox rdf:resource="mailto:em@w3.org"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me">
        <contact:personalTitle>Dr.</contact:personalTitle>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me">
        <rdf:type rdf:resource=
        "http://www.w3.org/2000/10/swap/pim/contact#Person"/>
    </rdf:Description>
</rdf:RDF>
```

Obrázek 3.1: Graf reprezentující kontakt na osobu Eric Miller. Převzato z [15].

Kapitola 4

Technologie klienta a serveru

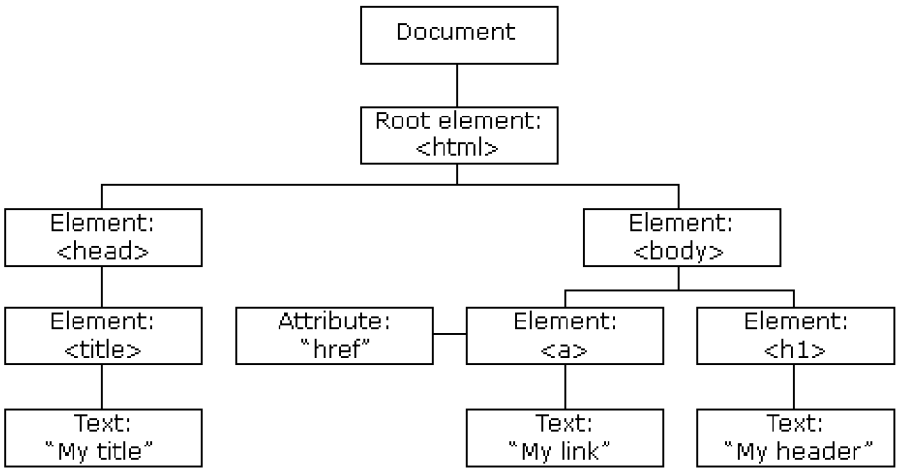
Protože je v tomto projektu funkcionalita klienta poskytována webovým prohlížečem, jsou technologie, které budeme využívat již z velké části dané. Struktura webové stránky je definována v jazyce HTML, CSS poté definuje její vzhled. Z jazyka HTML je pro nás klíčový prvek canvas, který nám umožní vykreslovat grafiku přímo do webové stránky. Toto celé je poté dynamicky řízeno skripty v jazyce JavaScript. Funkcionalitu grafového databázového úložiště poté zajišťuje framework rdf4j vytvořený v jazyce Java, který je určený pro manipulaci s RDF daty a běží v aplikaci Apache Tomcat. Komunikace mezi klientem a serverem je realizována pomocí rdf4j API, což je speciální varianta webového REST API, kterou poskytuje právě framework rdf4j.

4.1 HTML

HTML (Hypertext Markup Language) je v informatice název značkovacího jazyka používaného pro tvorbu webových stránek, které jsou propojeny hypertextovými odkazy. Jedná se o interpretační jazyk, to znamená, že zdrojový kód se nepřekládá do spustitelného tvaru, ale je prováděn interpretem (v tomto případě webovým prohlížečem) na klientovi. Jazyk HTML je tvořen prvky (v originále element), což jsou objekty, které říkají něco o svém obsahu. Jméno prvku ohraničené ostrými závorkami „<“ a „>“ označujeme jako tag (v překladu značka). Může být párový nebo nepárový. Prvky se mohou do sebe vnořovat, nikdy se ale nesmí křížit. Každý prvek poté může obsahovat určité atributy (parametry, které určují vlastnosti prvku). Atribut se zapisuje do otevíracího tagu vedle jména ve formátu „jméno_atributu="hodnota"“ [9].

HTML DOM

DOM (Document Object Model) je standard pro přístup k dokumentům. Poskytuje neutrální rozhraní, které umožňuje programům a skriptům dynamicky číst a měnit obsah, strukturu nebo styl dokumentu. Tento standard se lze využít pro jakékoliv typy dokumentů, primárně se však používá v jazycích XML a HTML. HTML DOM tedy poskytuje standardní objektový model a programové rozhraní pro HTML. Definuje HTML prvky jako objekty, ty pak disponují vlastnostmi, funkcemi pro přístup a událostmi (v originále event). Přitom zůstane zachována hierarchie, kterou má HTML dokument – objekty tedy budou tvořit strom. Příklad takového stromu se nachází na obrázku 4.1. Pomocí tohoto rozhraní je pak možné dynamicky měnit a číst obsah již načtené webové stránky [5].



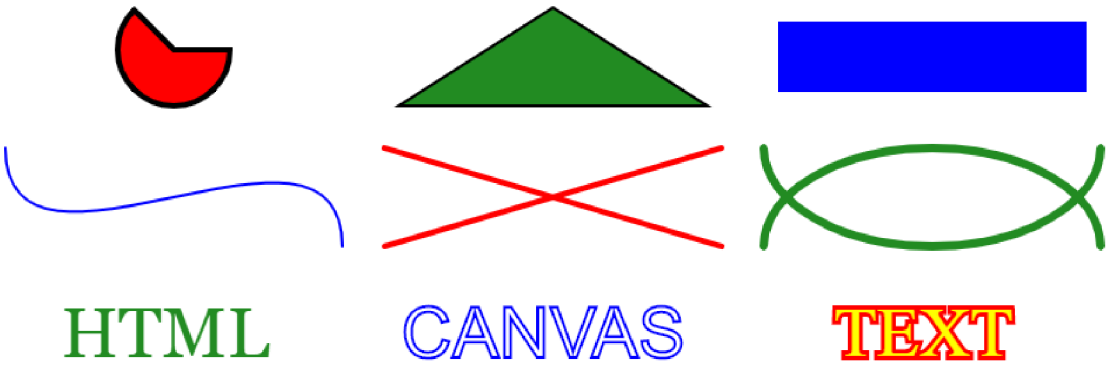
Obrázek 4.1: Demonstrační struktura jednoduché webové stránky v HTML DOM. Převzato z [5].

Prvek canvas

Jedná se o párový tag, který je speciálně určen pro vykreslování grafiky ve webových stránkách. Ovládá se pomocí skriptovacího jazyka JavaScript a umožňuje dynamické změny i po načtení stránky. Pokud takové změny provádíme v konstantních intervalech – běžně šedesátkrát za sekundu, je možné vytvářet animace, čehož bude využito při vizualizaci grafové databáze. Jeho funkcionalita je dostupná ve většině běžně používaných prohlížečích. Samotný tag může mít například takovýto tvar:

```
<canvas id="myCanvas" width="200" height="100"\></canvas>
```

Ten vytvoří objekt canvas uvnitř webové stránky, jehož identifikátor je „myCanvas“. Bude 200 px široký a 100 px vysoký. K tomuto objektu pak přistoupíme pomocí DOM struktury ze skriptu.



Obrázek 4.2: Ukázka základní funcionality prvku canvas.

Objekt canvas navíc obsahuje desítky funkcí, které pokrývají většinu základní funkcionality pro práci s 2D objekty. To zahrnuje vykreslování základních tvarů jako je bod, čára,

kruh, mnohoúhelníky a další. U těch je poté možné měnit barvy výplní, ohraničení a jiné vlastnosti. Existuje zde i podpora pro vykreslování textu. Funkcionalita je demonstrována na obrázku 4.2 [4], [3].

4.2 Rdf4j

Eclipse rdf4j[1] je framework vytvořený v jazyce Java určený pro manipulaci s RDF daty. To zahrnuje vytváření, zpracování a následné uložení RDF dat do databáze. Na ty je poté možné se dotazovat. Rdf4j plně podporuje dotazovací jazyk SPARQL spolu s jeho rozšířeními na vytváření, úpravu a mazání záznamů. Poté poskytuje jednoduché API založené na REST architektuře, pomocí kterého je možné s databázovým systémem komunikovat z jakéhokoliv typu aplikace. Samozřejmostí je také podpora všech hlavních používaných souborových formátů (např. RDF/XML, Turtle, N-Triples), což je důležité pro import a export dat. Framework může běžet na různých serverech s podporou servletových kontejnerů, doporučený je však Apache Tomcat. K dispozici jsou poté dvě nezávislé implementace uložení dat [13]:

- Memory store – Data jsou při práci uložena primárně v paměti RAM, synchronizace s diskem probíhá později. Tento přístup poskytuje vysokou rychlost operací, je však omezen kapacitou paměti RAM.
- Native store – Veškerá práce s daty probíhá přímo přes vstupně-výstupní rozhraní disku. Tento přístup poskytuje lepší rozšiřitelnost, konzistenci a spolehlivost, operace jsou však prováděny pomaleji.

Server a Workbench

Rdf4j Server a rdf4j Workbench jsou aplikace zajišťující správu databáze. Obě disponují grafickým uživatelským rozhraním a dá se k nim v základu přistoupit přes adresy „http://localhost:8080/rdf4j-server“ a „http://localhost:8080/rdf4j-workbench“.

Rdf4j Server poskytuje HTTP přístup k databázi a dělá ji tedy viditelnou jako SPARQL uzel. K této aplikaci primárně přistupují externí klienti, a proto obsahuje jen minimum funkcionality pro uživatele (informace o serveru a server log).

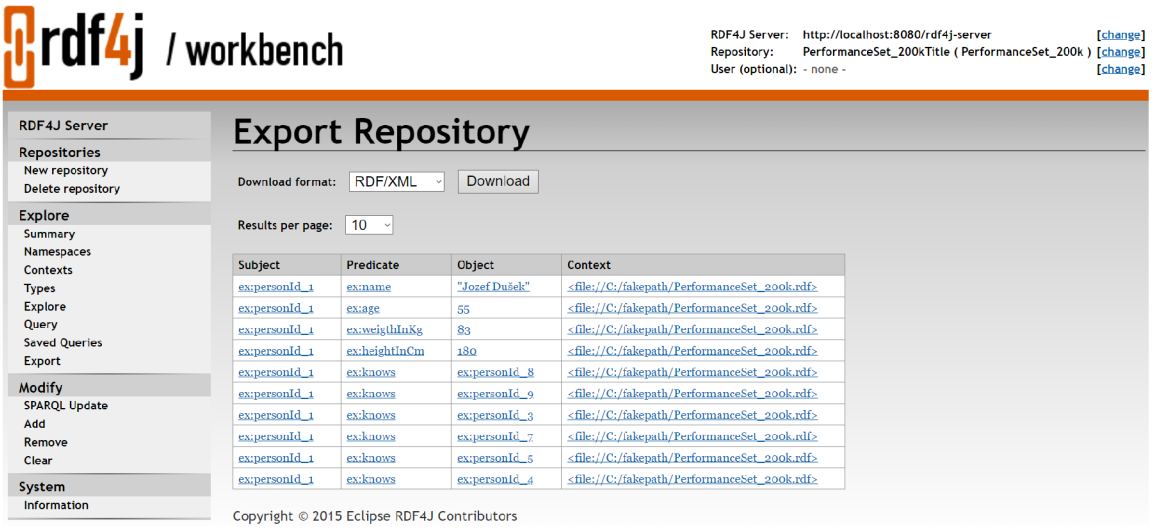
Rdf4j Workbench poté zajišťuje všechnu ostatní uživatelskou funkcionalitu. Rozhraní této aplikace je vyobrazeno na obrázku 4.3.

Základem je vytváření, mazání a exportování databází. Po vytvoření je databáze prázdná, je do ní však možné manuálně načíst data z externího souboru. Jak bylo řečeno v úvodu, většina používaných formátů je podporována. Stejným způsobem se poté dají data exportovat. Dále je možné prohlížet obsah databáze, a to buď vypsáním seznamu všech trojic, nebo přes dotaz. Protože rdf4j podporuje rozšíření jazyka SPARQL pro úpravu a mazaní trojic, je tato funkcionalita také zahrnuta v této aplikaci [12].

Apache Tomcat

Apache Tomcat[2] je webový server a servlet kontejner, ve kterém bude v tomto projektu běžet framework rdf4j. Je vyvíjený jako open source projekt a implementuje javové servlety, JSP (Java Server Pages) a EJB (Enterprise JavaBeans). Jako celek tedy poskytuje čistě

[1] https://rdf4j.org
[2] http://tomcat.apache.org



Obrázek 4.3: Uživatelské rozhraní aplikace rdf4j Workbench určené ke správě databáze.

javové prostředí HTTP webového serveru, ve kterém potom mohou běžet javové aplikace. Tomcat samotný je implementovaný v jazyce Java a je možné ho ovládat přes konzoli nebo grafické uživatelské rozhraní.

Je srovnatelný s komerčními produkty od IBM, Oracle a dalších. Ty poskytují zpravidla větší robustnost a lépe propracovanou bezpečnostní stránku, Tomcat oproti tomu vyniká jednoduchostí, transparentností a menší náročností na výpočetní výkon [1].

Skládá se z několika hlavních částí:

- Catalina – servletový kontejner podle specifikace Sun Microsystems
- Coyote – konektor, který podporuje HTTP protokol – důsledkem je, že se Catalina jeví jako standardní webový server, který pracuje se soubory jako HTTP dokumenty
- Jasper – správce JSP, kompiluje JSP soubory do javového kódu jako servlety, ty pak zpracuje Catalina

4.3 REST API

REST (Representational State Transfer) je architektura, která umožňuje přistupovat k datům na určitém místě pomocí standardních metod HTTP.

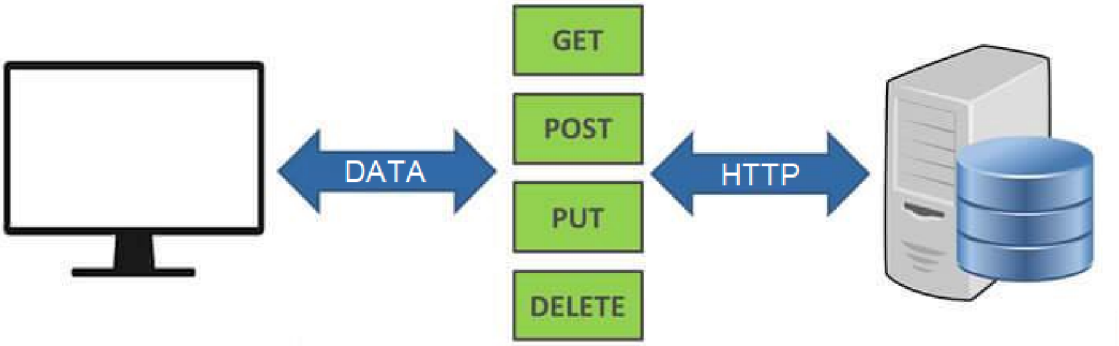
Komunikace je typu klient – server, REST však zajišťuje, že jejich implementace jsou na sobě naprosto nezávislé. Je tedy možné měnit kód klienta nebo serveru bez toho, aniž by bylo ovlivněno fungování druhé strany. Důležité je jen to, že obě strany ví, jaký je formát zasílaných zpráv, což také podporuje modularitu.

Další důležitou vlastností je bezstavovost. To znamená, že server ani klient nepotřebují vědět žádné informace o stavu druhého uzlu. Mohou tedy přijímat a zpracovávat zprávy nezávisle bez ohledu na předchozí komunikaci. To je vynuceno využíváním zdrojů namísto obecných požadavků ke zpracování. Zdroje jsou definovány jednoznačným URI identifikátorem a jedná se o jakékoliv objekty, které je potřeba uložit nebo poslat dalším službám.

Komunikace v architektuře REST probíhá na základě zasílaní požadavků (typ request), ty načítají nebo modifikují data a server na ně odpovídá (typ response). To demonstruje

obrázek 4.4. Požadavek se skládá z hlavičky, cestě ke zdroji, volitelných parametrů a specifikace typu požadavku. Ty existují čtyři a mají specifická využití:

- GET – přístup k datům
- POST – vytvoření nových dat
- DELETE – mazání existujících dat
- PUT – úprava existujících dat



Obrázek 4.4: Schéma komunikace v REST API. Převzato z [17].

V hlavičce se poté specifikuje formát dat, ve kterém bude probíhat jejich výměna. Základní je text/plain, existuje však spousta dalších jako například image/png, video/mp4 nebo application/xml [7].

Rdf4j API

Jak již bylo zmíněno v úvodu, komunikaci bude zajišťovat protokol rdf4j API, který vychází z REST architektury a dodržuje standard „SPARQL 1.1 Protocol W3C Recommendation“, což zaručuje, že rdf4j server funguje jako plnohodnotný SPARQL uzel (v originále SPARQL endpoint). Poskytuje tedy standardizovaný mechanismus pro vzdálenou práci s grafovými databázemi.

Disponuje několika formáty pro přenos trojic, výchozí je „text/plain“. Nejpoužívanější jsou vypsány níže:

```
RDF/XML: application/rdf+xml
N-Triples: text/plain
Turtle:  text/turtle
```

Protože výsledky dotazů nemusí být nutně trojice, tak používají jiné formáty:

```
SPARQL Query Results XML Format: application/sparql-results+xml
SPARQL Query Results JSON Format: application/sparql-results+json
```

Nejpoužívanějším typem požadavku je GET, který se používá pro získání dat z databáze. Do toho spadají požadavky na seznam všech jmenných prostorů (v originále namespace) nebo trojic. Také se pomocí něj provádí specifické dotazy, ty se ale nejdříve musí zakódovat do syntaxe URL. V následujících příkladech má databáze název „mem-rdf“.

Požadavek na seznam jmenných prostorů:

```
GET /rdf4j-server/repositories/mem-rdf/namespaces
Host: localhost
Accept: application/rdf+xml
```

Demonstrační dotaz:

```
SELECT ?book ?who
WHERE
{
    ?book <http://example.org/creator> ?who
}
```

Požadavek se zakódovaným dotazem:

```
GET /rdf4j-server/repositories/mem-rdf?query=SELECT%20%3Fbook%20%3Fwho%20%0
AWHERE%20%7B%3Fbook%20%3Chttp%3A%2F%2Fexample.org%2Fcreator%3E%20%3Fwho%7D
Host: localhost
Accept: application/sparql-results+xml
```

Trojice lze také ukládat, měnit a mazat. K tomu se používají typy POST, PUT a DELETE. Takové požadavky jsou tvořeny obdobně jak u typu GET. Vložení trojic do databáze:

```
PUT /rdf4j-server/repositories/mem-rdf/statements
Host: localhost
Content-Type: application/rdf+xml

[DATA IN RDF/XML FORMAT]
```

Smazání všech trojic v databázi:

```
DELETE /rdf4j-server/repositories/mem-rdf/statements
Host: localhost
```

V případě, že vše proběhlo v pořádku, tak server zareaguje odpovědí „200 OK“ a přímým zasláním požadovaných dat v daném formátu, pokud byly nějaké vyžadovány. V opačném případě vrátí chybovou zprávu [11].

Kapitola 5

Návrh

Po teoretické analýze použitých technologií se tato kapitola zaměří na aplikaci samotnou. Hlavním cílem této práce je vizualizace grafových dat v prohlížeči s následným zjištěním limitu trojic, které prohlížeče zvládnou při aktuálním výkonu najednou vykreslit. Tyto cíle tvořily hlavní faktor při celkovém návrhu. Interaktivní vizualizace bude realizována pomocí několika rozdílných metod, které poskytují různé výkony a vzhled. Po testování a vyhodnocení by tedy mělo být možné identifikovat tu nejvhodnější pro tyto účely. Samotná data budou uložena na serveru, ke kterým se bude vzdáleně přistupovat. Veškerá práce s nimi však bude probíhat lokálně. Protože se práce zabývá nalezením výkonnostního limitu, měla by aplikace jako celek pracovat co nejefektivněji. To znamená, že by měly být využity optimální datové struktury a algoritmy. Také grafické uživatelské rozhraní (dále GUI) by mělo být minimalistické jen s nejdůležitějšími funkcemi, aby zbytečně nespotřebovávalo limitované zdroje.

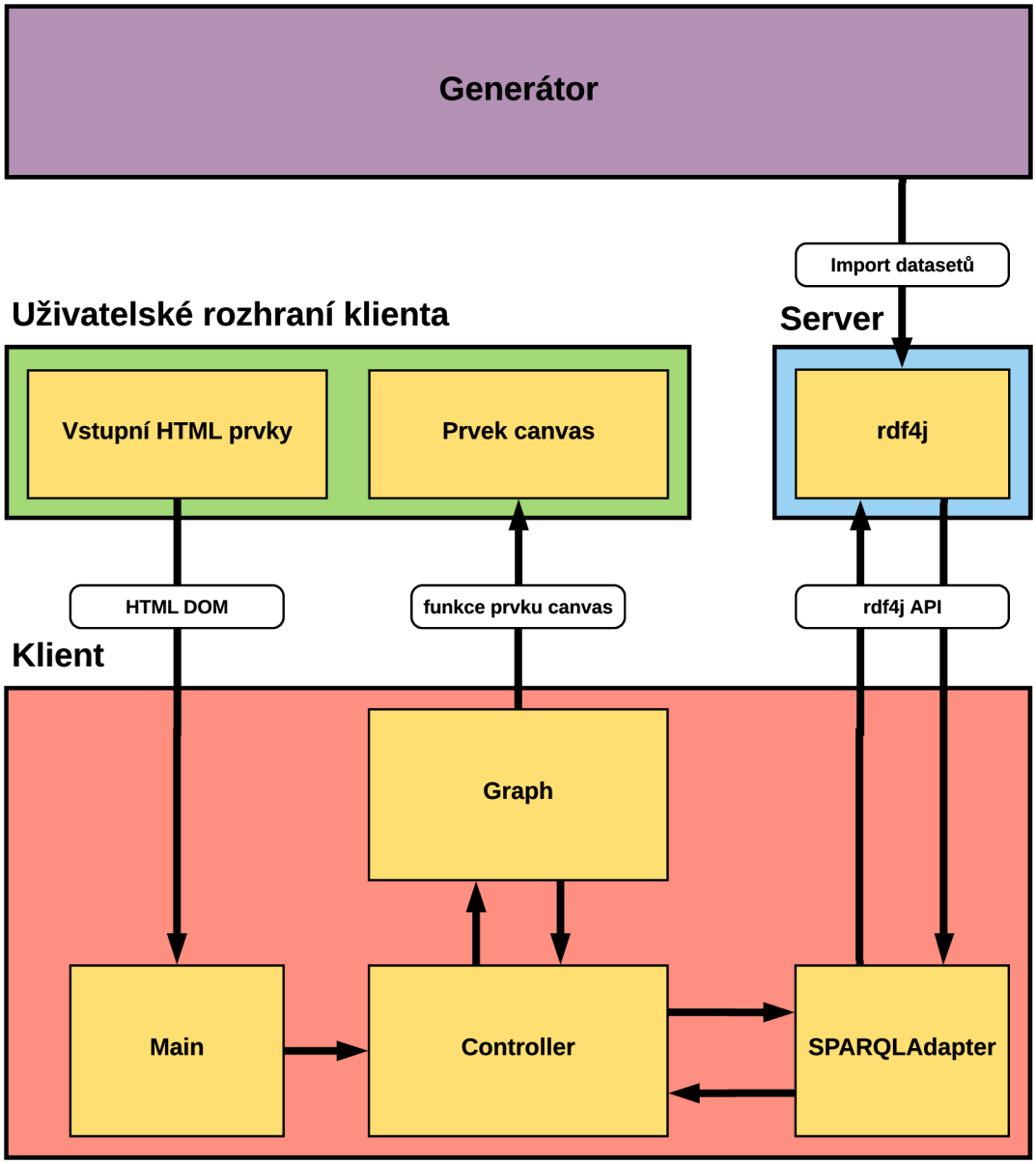
Poslední část, která se musela přesně definovat v této fázi byl generátor RDF dat. Ten sice přímo nesouvisí s hlavní aplikací, od počátku projektu se ale zvažovali jeho výhody a nevýhody. Nakonec bylo rozhodnuto v jeho prospěch a jednalo se o první část, která byla přesně navržena a implementována.

Celková struktura aplikace se poté skládá z několika samostatných částí, které mezi sebou komunikují. Ta je vyobrazena na obrázku 5.1.

5.1 Generátor

Generátor byl od počátku navržen s jasnými cíli za účelem pohodlného vývoje hlavní aplikace a jednoduchého vygenerování potřebných datasetů pro testování. Ty byly následující:

- Výsledkem generátoru bude dataset typu RDF.
- Vygenerovaný dataset bude uložený do jednoho souboru ve formátu RDF/XML, aby ho bylo možné nahrát do databázového systému.
- Dataset musí obsahovat alespoň čtyři rozdílné entity, mezi kterými budou tvořeny propojení.
- Propojení by neměly být naprosto náhodné, generování by tedy mělo proběhnout na základě abstraktní struktury.
- Vlastnosti entit by neměly být náhodné řetězce.



Obrázek 5.1: Schéma aplikace.

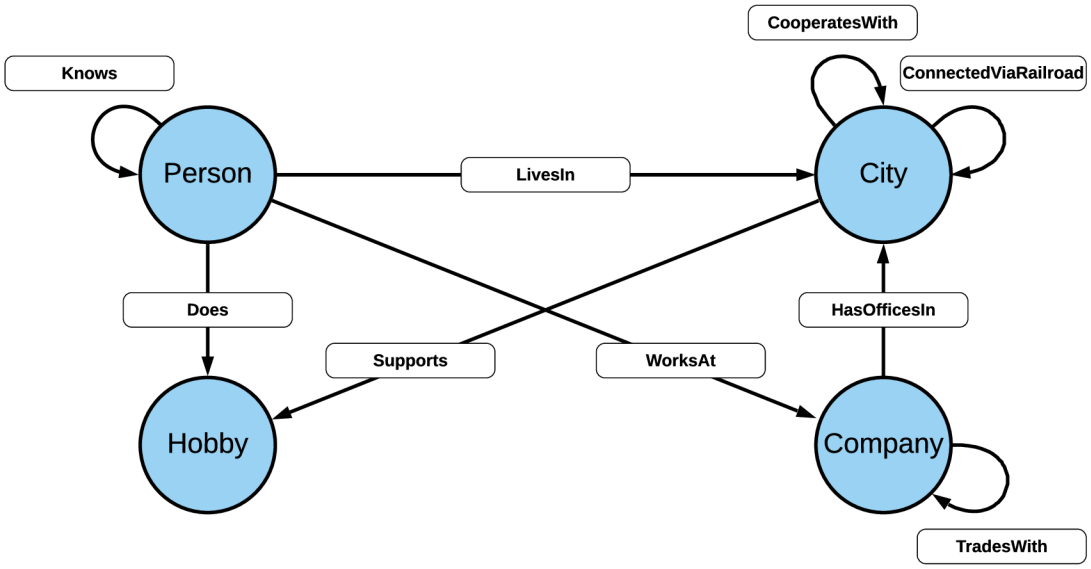
- V datasetu se mohou objevit takové situace, kdy dva uzly mají více než dvě propojení mezi sebou.
- Generování vytváří datasety náhodným způsobem dle zadaných parametrů a rozmezí.
- Co nejvíce parametrů musí být možné upravit.

Prvním krokem tedy bylo navrhnutí jednotlivých entit. Aby byl výsledný dataset co nejpřirozenější a splňoval dané podmínky, byly vybrány následující: osoba, společnost, město a koníček. Ty jsou běžně používané a poskytují několik způsobů, jak navrhnout vazby

mezi nimi. Osoby se potom velice hodí pro vytvoření zadané abstraktní struktury. Všechny by poté měly obsahovat nějaké interní vlastnosti, aby ale datasety nebyly jen shlukem náhodných znaků, tak by ke generaci měla být využita data z reálného světa.

Dalším krokem byl návrh propojení osob, které budou tvořit jádro datasetu. Byl zvolen takový postup, který nepropojuje osoby naprosto náhodně, ale snaží se jejich rozložení vytvořit tak, aby výsledek simuloval reálnou strukturu známostí. Základní princip je založen na faktu, že u každého člověka je větší pravděpodobnost, že bude znát konkrétní osobu z blízkého okolí než úplně náhodnou z celého světa. Lidé budou například mít více známostí z města ve kterém žijí než z města na druhé části republiky. Výsledná stromová struktura může mít několik úrovní a bude připomínat kolekci hustě propojených shluků, které budou spojeny s dalšími shluky pomocí relativně malého počtu propojení.

Poslední krok spočíval v zakomponování ostatních entit do datasetu. Ty budou připojeny bez struktury, a to náhodným výběrem daného počtu ze seznamu možností. Celková struktura vztahů je vyobrazena na diagramu 5.2.



Obrázek 5.2: Schéma vztahů, které může generátor vytvořit mezi jednotlivými entitami.

Za povšimnutí stojí vztahy *CooperatesWith* a *ConnectedViaRailroad*, které byly navrženy pro specifickou funkci. Protože oba spojují entity město, znamená to, že mezi dvěma městy mohou být až čtyři různé vztahy, což bylo vyžadováno jedním z originálních kritérií.

Aby bylo generování co nejrychlejší, implementační jazyk by měl být kompilovaný. Zvolen tedy byl C++.

5.2 Server

Dalším krokem bylo zvolení vhodného úložiště pro grafová data. Protože výkonnostní limity budou testovány v klientské části aplikace, výkon samotného uložiště nehrál při výběru velkou roli. Navíc se předpokládá, že systém úložiště zvládne pracovat s více trojicemi než webový prohlížeč, který je k tomu ještě vykresluje. Celkově se ve fázi návrhu předpokládalo použití datasetů s počtem trojic v rozmezí statisíců až jednotek miliónů. Nejdůležitějším

faktorem při výběru uložiště byla tedy funkcionalita a dostupnost. Prvním požadavkem bylo, aby systém jako celek mohl běžet na lokálním serveru za účelem jednoduchého vývoje klienta a pohodlné manipulaci s uloženými daty. Dalším benefitem je také jednodušší instalace a nastavení. Pro přenos dat v tomto projektu byl zvolen v podstatě nejpoužívanější formát RDF/XML. V něm jsou ukládány datasety produkované dříve navrhnutým generátorem a podporuje ho také většina volně dostupných datových repositářů. Proto musí být schopen importovat data v tomto formátu i databázový systém. Dalším důležitým požadavkem byla podpora REST API pro vzdálenou komunikaci, jelikož se jedná o jediný způsob, jakým mezi sebou mohou server a klient komunikovat. Posledním požadavkem poté bylo, aby systém byl volně dostupný. Po zhodnocení několika kandidátů byl nakonec vybrán framework rdf4j. Ten splňuje veškeré požadavky: je volně dostupný, umí pracovat v podstatě se všemi typy souborů, může běžet lokálně v aplikaci Apache Tomcat a disponuje rdf4j REST API. K tomu navíc obsahuje uživatelsky přívětivé GUI pro manuální manipulaci a nastavení.

5.3 Klient

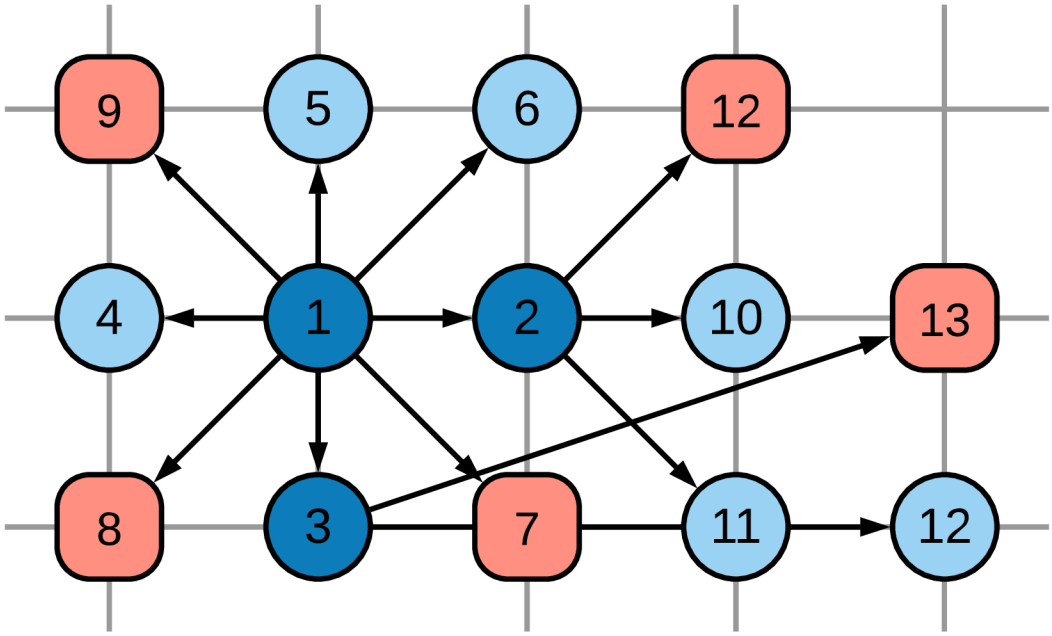
Poté následoval návrh klienta. Ten měl od počátku několik základních požadavků. Dle zadání má klient běžet ve webovém prohlížeči, a tedy bude mít formu webové stránky. Z toho vyplývá, že nástroje, které budou pro tvorbu klienta využity jsou de facto předem dány prostředím. Webové stránky vycházejí z jasně dané struktury, kterou poté prohlížeče umí zpracovat. To znamená, že struktura stránky bude definována v jazyce HTML a styl v jazyce CSS. Všechno ostatní (řízení, zpracování vstupů, komunikace se serverem, vykreslování) bude implementováno v jazyce JavaScript. Klient musí být schopen přijmout a zpracovat data ze serveru, které poté budou vizualizovány několika různými způsoby. To je realizováno za pomoci speciálního HTML prvku canvas, který umožňuje dynamické vykreslování do stránky. Posledním důležitým kritériem bylo pak to, že klient má být interaktivní uživatelská aplikace. Musí tedy disponovat uživatelským rozhraním. To bude vytvořeno pomocí standardních webových nástrojů a ovládat se bude klávesnicí a myší.

Vizualizační algoritmy

Vizualizace grafů není triviálním úkonem, a to hlavně díky tomu, že grafy mohou mít velice různé struktury. Není tedy jednoduché najít univerzální metody, které by vykreslily všechny grafy perfektně. Grafy mohou být řídké, husté, spojité, nespojité a k tomu různé části grafu mohou tvořit hierarchie. Další vlastnosti jako je orientovanost nebo přítomnost několikanásobných propojení také hrají roli, neovlivňují ale celkovou strukturu. Pro umístění uzlů je poté k dispozici celá plocha a není zřejmé v jakých relativních pozicích by se měly jednotlivé uzly nacházet. Dále se například musí určit, zda se uzly a propojení mohou překrývat.

Do výsledné aplikace byly nakonec vybrány tři metody pro rozložení uzlů lišící se principem, výkonem a estetikou. Formát výsledku je však stejný – každému uzlu je přiřazena souřadnice v prostoru. Tyto metody využívají obecné principy, ale jako celek byly speciálně navrženy pro tento projekt. Jedná se o metody nazvané greedy, greedy-swap a force-directed. Několik vlastností však mají společných. Protože grafové databáze povoluji existenci nespojitých datasetů, je důležité, aby tuto vlastnost podporovaly i tyto algoritmy. Také by se měly snažit vyhnout překryvu uzlů, protože to způsobuje nepřehlednost.

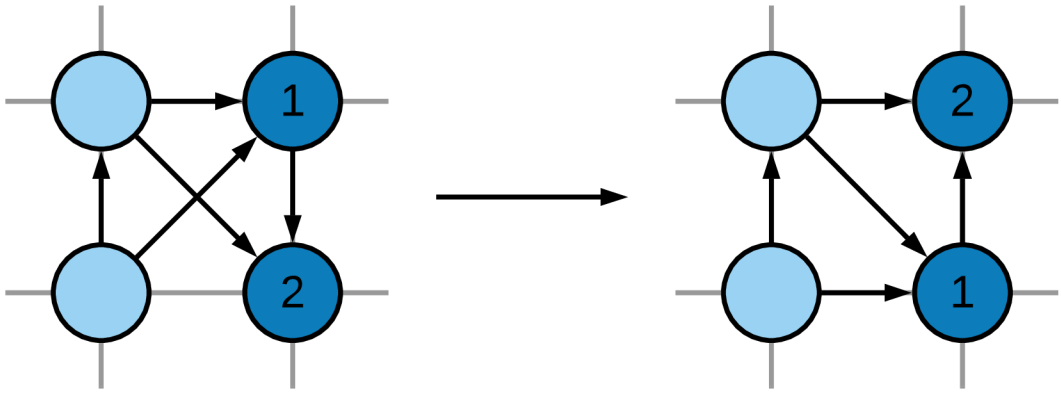
Greedy metoda je z nich nejjednodušší. Její základní koncept je prostý – uzly se vkládají do mříže tak, aby byly co nejblíže jejich sousedům. Tato varianta pracuje se čtvercovou mříží. Do ní se doprostřed nejdříve vloží náhodný uzel. Poté se postupně zpracovávají jeho sousedé, kteří se umisťují na nejbližší možná volná místa. Takhle se pokračuje, dokud všechny uzly nejsou umístěny v mříži. Tento postup je demostrován na diagramu 5.3 a přináší několik výhod a úskalí. Použitím mříže se v podstatě obejde problém rozložení uzlů v prostoru, který spotřebovává nejvíce výkonu. Počet možných pozic je limitovaný a při vkládání se vůbec nemusí brát v úvahu již umístěné uzly, metoda je tedy velice rychlá. Výsledný vzhled však není optimální. První problém je ten, že metoda velmi špatně vyobrazuje hierarchie. Sousedé jsou bráni náhodně a zpracovávají se po jednom. Struktura v okolí uzlu tedy ovlivňuje výsledek jen minimálně. Další problém poté vytváří mříž samotná. Uzly se nachází na přímkách a pokud budou propojeny, tak to bude vést k velice nepřehledným překryvům propojovacích čar. Celkově se tedy jedná o velice jednoduchou metodu, která však očekávaně neposkytuje optimální výsledky.



Obrázek 5.3: Diagram fungování greedy metody. Světle modré uzly jsou objektové, světle červené jsou literální. Čísla označují pořadí vložení do mříže. Tmavě modře jsou poté vyznačeny ty objektové uzly, které už byly zpracovány (tedy jejich sousedé byly vloženi do mříže).

Druhou použitou metodou je greedy-swap. Vychází z předchozí greedy metody, snaží se však řešit její problémy dalším zpracováním výsledného rozložení. Inspirace pro tuto metodu vychází z algoritmu 2-opt swap pro řešení problému obchodního cestujícího. Hlavní aspekt, který se poté snaží vylepšit je vykreslení podstruktur a hierarchií. I když se standardní greedy metoda snaží dát propojené uzly co nejblíže k sobě, tak fakt, že se s již vloženými uzly nemanipuluje způsobuje to, že se ke konci skutečná struktura grafu rozpadá. Greedy-swap tedy vychází z jednoduchého pozorování. Když jsou hustě propojené uzly blíže u sebe,

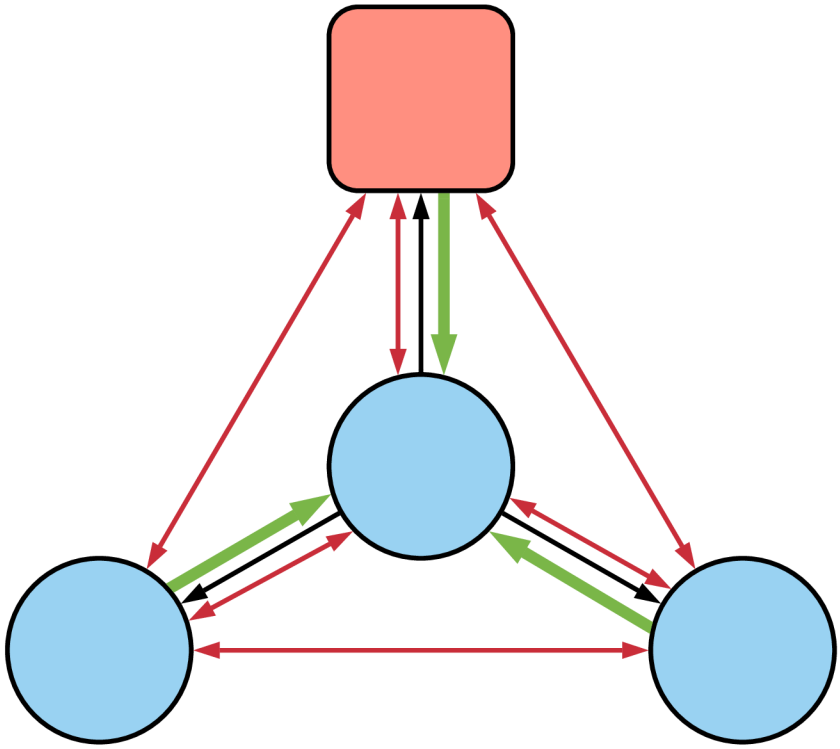
tak se sníží celková vzdálenost propojení v grafu. Princip metody je tedy následující – graf se vykreslí greedy metodou a v tomto výsledku se začnou prohazovat uzly. Uzel se prohodí právě tehdy, když se tím sníží celkový součet délek všech hran v grafu (obrázek 5.4). To pokračuje, dokud v grafu neexistuje žádný další prohoz, který by vedl ke zlepšení (je nalezeno lokální minimum součtu délek). Tento postup je časově náročnější než u základní metody, vizualizace by však měla poskytnout mnohem lepší výsledky. Literální uzly by vždy měly zůstat v blízkosti rodičovského uzlu a uzly v hustě propojených shlucích by měly být co nejblíže u sebe. Problémy spojené s vykreslením do mříže však zůstávají.



Obrázek 5.4: Ukázka prohozu v greedy-swap metodě. Vlevo je ukázána původní struktura, napravo se nachází stav po prohozu. Je vidět, že došlo ke zmenšení součtu délek – jedna úhlopříčná hrana se změnila na vodorovnou.

Poslední metoda využitá v tomto projektu je force-directed graph drawing (česky vykreslení grafu založené na silách). Ta pracuje na úplně jiném principu než předchozí dvě metody a poskytuje naprosto jiný styl výsledku. Obecný princip této metody je založen na fyzikální simulaci. Všechny uzly se navzájem odpuzují silou, která je závislá na vzdálenosti mezi nimi. Toto odpuzování poté vyvažuje přitažlivá síla (také závislá na vzdálenosti), kterou mezi sebou mají uzly, které jsou propojeny. Důležité pak je, že funkce, které určují velikost odpudivé i přitažlivé síly mají jiné, ale specificky zvolené průběhy. Uzly se díky tomu ustalují v předem dané vzdálenosti a nepřekrývají se. To je demonstrováno na obrázku 5.5. Průběh algoritmu má poté dvě hlavní fáze. Nejprve se uzly rozprostřou do prostoru. Poté následuje samotná fyzikální simulace. Pro každý uzel se vypočte součet všech jednotlivých vektorů sil. Pak se uzly posunou ve směru těchto vektorů. Toto se opakuje tolikrát, kolikrát je zadáno. Protože se jedná o simulaci, čím více iterací se provede, tím více se pozice uzlů budou blížit ideálnímu ustálenému stavu. Tato metoda je výkonově nejnáročnější, poskytuje ale nejpřirozenější vizualizaci. Při dostačujícím počtu iterací by měla velice efektivně vyobrazovat struktury a hierarchie v grafu. Protože tato metoda může umístit uzly do jakékoliv pozice v prostoru, problém s překryvy, který měly obě předchozí metody zde nenastává.

Nakonec bylo důležité určit samotný vzhled, pomocí kterého budou vykresleny výstupy všech algoritmů. Ten by měl být co nejjednodušší. Uzly budou reprezentovány kruhy, literální a objektové jsou rozlišeny barvami. Propojení jsou reprezentována přímkou. Orientovanost bude určena šipkou a vícenásobná propojení bude reprezentovat vícenásobná linie. Popisy uzlu jsou obsaženy uvnitř kruhu, popisy propojení jsou pak umístěny v jejich středu.



Obrázek 5.5: Demostrace všech sil, se kterými pracuje metoda force-directed. Všechny uzly se navzájem odpuzují (červené šipky), pokud jsou však uzly propojeny, tak se navíc přitahují, což vyváží odpudivou sílu (zelené šipky).

V případě, že je propojení několikanásobné, tak budou popisy rozprostřeny po celé délce, aby se nepřekrývaly. Popisy vždy zachovávají vodorovnou orientaci. Veškeré vykreslení bude realizováno pomocí funkcí prvku canvas.

Uživatelské rozhraní

Jak již bylo řečeno v úvodu kapitoly, GUI by mělo být minimalistické, aby se uspořil výkon a zachovala jednoduchost ovládání. Primárním ovladačem byla zvolena myš, klávesnice bude využita jen pro zadávání textu. Hlavními ovládacími prvky tedy budou tlačítka, výběrová menu, posuvníky, zaškrtávací boxy a prostor pro text.

Nejdříve bylo nutné definovat funkcionalitu, která nesouvisí s vizualizací. Ta ve výsledku zahrnuje zadání cesty k serveru a názvu repositáře, výběr vizualizační metody, zadání počtu iterací pro force-directed algoritmus a nastavení limitu pro maximální počet trojic načtených ze serveru, aby se předešlo zahlcení.

Návrh vizualizační části poté vypadal následovně: Prostor pro vykreslení trojic by měl být co největší s menu umístěném na boku. Aplikace by tedy měla využít celý prostor okna. Funkcionalita by poté měla opět splňovat několik požadavků. Práci by mělo jít začít dvěma způsoby, a to buď vykreslením konkrétního zvoleného uzlu nebo celé databáze do vyhrazeného prostoru daným algoritmem. S tímto výsledkem by mělo jít dále manipulovat. Pro účel prohlížení by měl jít graf přibližovat, oddalovat a posouvat (posunout by se měly

dát i uzly samostatně). Aby se nemusel pokaždé zadávat identifikátor uzlu pro změnu načtené části grafu, mělo by být možné uzly označit (a zvýraznit). Označené uzly by pak mělo být možné expandovat (přidat do grafu jejich sousedy) nebo zachovat (v grafu zůstanou pouze označené uzly). Nakonec by také mělo být možné graf překreslit bez opětovného načtení dat.

Struktura

Posledním úkonem byl poté návrh samotné struktury klienta. Ta má nelehký úkol – musí vyhovět všem předchozím návrhovým požadavkům, a přitom zajišťovat jednoduchý vývoj, rozšiřitelnost a dobrý výkon. Aplikace jako taková byla tedy rozdělena do několika samostatných a robustních částí, z nichž každá má na starosti jiný aspekt aplikace. Ty spolu pak komunikují.

Jádro klienta budou tvořit části *main* a *controller*. Main bude zajišťovat inicializaci aplikace po startu spolu se správou vstupů a výstupů. Také v něm bude běžet programová smyčka aplikace, která zajišťuje její dynamičnost. Hlavní řídící centrum však bude tvořit *controller*. Ten bude obsahovat veškerou logiku aplikace, což znamená, že bude zpracovávat požadavky uživatele a poté podle nich a aktuálního stavu bude volat další části, které poté dané úkony provedou a jejich výsledky zobrazí na výstup.

První takovou částí je *SPARQLAdapter*. Ta bude zajišťovat veškerou funkcionalitu spojenou se serverem. Bude tvořit SPARQL dotazy podle uživatelských vstupů a posílat je na server. Také bude odpovědi přijímat. Ty zpracuje a pošle zpět žadateli. Druhou a pro náš projekt nejdůležitější částí poté bude *graph*. Ta bude zajišťovat veškeré úkony spojené s vizualizací trojic. Nejdříve vytvoří abstraktní grafovou strukturu z přijatých trojic, umístí je do prostoru dle zadaného algoritmu a celý výsledek vykreslí do plátna ve webové stránce (do toho také spadá zajištění interaktivity v již vykresleném grafu). Právě tato část musí pracovat co nejrychleji, a proto by zde měla být největší snaha o efektivní implementaci.

Kapitola 6

Implementace

Po specifikaci návrhu bude nyní popsána implementace všech částí aplikace a generátoru. Ta by se měla snažit co nejpřesněji splnit dané cíle. Celý vývoj probíhal v operačním systému Windows 10. Server tvoří databázový framework rdf4j verze 3.0.1, který běží v aplikaci Apache Tomcat 9.0. Vývoj klienta probíhal v internetovém prohlížeči Opera 68.0. Veškerý HTML, CSS a JavaScript kód byl poté editován v programu Notepad++ verze 7.8.5. Výsledkem je webová stránka, která je kompatibilní s většinou běžných prohlížečů. Generátor byl implementován v jazyce C++ 17. K programování a kompilaci bylo využito vývojové prostředí Visual Studio 2019. Výsledkem kompilace je jeden .exe soubor.

6.1 Generátor

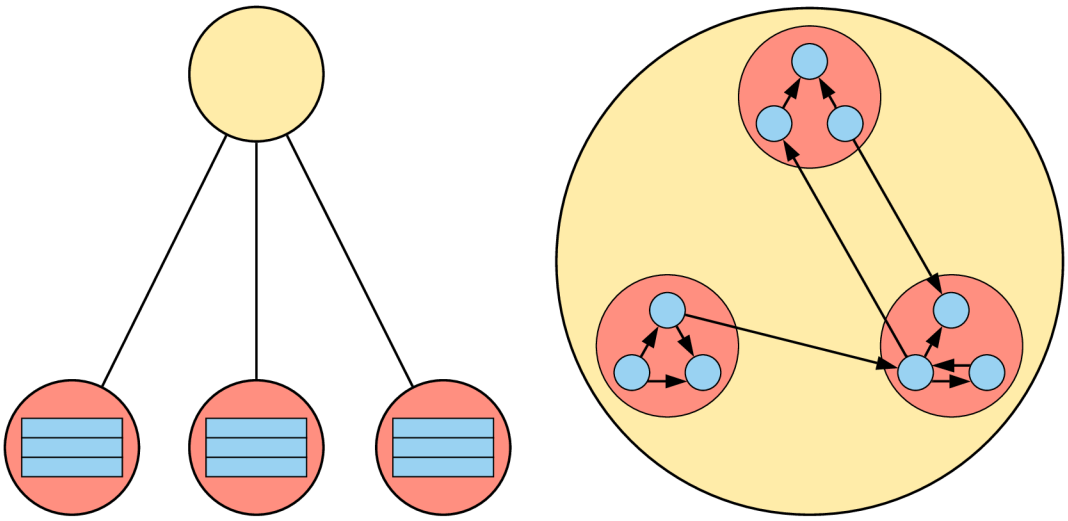
Generování začíná nastavením parametrů do globálních proměnných. Těch existuje několik a je pomocí nich možné měnit většinu aspektů generování. Pokud je to možné, zadávají se ve tvaru rozmezí – dá se tedy například specifikovat, že osoba bude mít 2 až 5 přátel.

Nejprve jsou zvoleny počty měst, firem a koníčků, které bude možné využít. Ty jsou ale omezeny počtem položek v externích souborech, počet osob poté vyplývá z ostatních parametrů. U stromu, který reprezentuje shlukovou hierarchii je možné měnit počet synů a jeho hloubku. U propojení osob se dá nastavit koeficient rozložení, který určuje počet propojení mezi různými úrovněmi shluků. Parametry všech ostatních propojení jsou řízeny přímým zadáním požadovaného počtu.

Pokračuje se načtením a zpracováním všech dat z externích souborů (formát .txt). V těch jsou obsaženy seznamy jmen a příjmení, výčet koníčků a informace o městech a firmách. Jména osob vychází ze seznamu reálných jmen, výčet koníčků poté obsahuje několik náhodných, ale smysluplných názvů. Údaje o společnostech byly převzaty ze seznamu sta největších firem v česku a města přebírají informace ze seznamu všech českých měst. Další číselné vlastnosti byly poté dogenerovány náhodně.

Poté proběhne rekurzivní vystavění stromu. Postupuje se od kořene, který reprezentuje shluk obsahující všechny osoby. Ten se poté dále dělí na podshluky, dokud se nedosáhne požadované hloubky. Jakmile je dosažena, vloží se do listových shluků požadovaný počet osob. To je ukázáno na obrázku 6.1.

Tvorba propojení mezi osobami probíhá jako první a pro každou osobu zvlášť. Počet propojení je pro každého předem určený, nejdříve je ale nutné rozdělit je do skupin, které určí jejich množinu cílů. Výpočet poměru těchto skupin je založen na geometrické posloupnosti a řídí se jedním parametrem. Pokud má například hodnotu 0.5 a generuje se strom



Obrázek 6.1: Demonstrace vztahu mezi stromem a výslednou strukturou databáze. Hierarchie má v tomto příkladu dvě úrovně a počet entit (shluků nebo přímo osob) ve shluku je nastaven na tři. Nalevo je strom použitý ke generování, napravo je výsledná struktura. Stejné barvy reprezentují stejné entity – žluté a červené jsou shluky, modré jsou osoby. Propojení mezi osobami není ve stromu vyobrazeno.

se třemi úrovněmi, tak bude výsledné rozložení v poměru 1:1/2:1/4. Při hodnotě 0.1 by poměr byl 1:1/10:1/100. To tedy znamená, že nejvíce propojení bude vytvořeno s osobami ve stejném shluku, menší část připadne na propojení s osobami vzdálenými o jednu úroveň a zbytek připadne na propojení s osobami vzdálenými o dvě úrovně.

Po vytvoření struktury osob následuje připojení ostatních entit. Jednotlivá propojení jsou generována samostatně a ovlivňuje je jeden parametr, který nastavuje, kolik takových propojení bude entita obsahovat. Pokud bude definováno, že osoba provozuje 3 až 5 koníčků, tak se nejdříve zvolí náhodný počet z rozmezí a poté se vyberou náhodné cíle z množiny všech koníčků. Nakonec je také důležité zmínit, že entity obsahují interní vlastnosti, které jsou také reprezentovány propojeními jako je například: jméno, věk, název nebo počet obyvatel. Ty jsou předem přesně definovány a nejdou měnit.

Tím je dokončeno generování a následuje uložení do souboru. Protože je výsledek aktuálně reprezentován třídami a ukazateli, je nutné ho převést do struktury typu RDF. Každá entita se převádí samostatně a při generování jim bylo přiřazeno unikátní číslo, které se použije při vytvoření identifikátoru. Ten může být například pro osobu „personId_1“ a slouží jako název uzlu, který poté jednoznačně určuje danou osobu. Interní vlastnosti, jako je výška nebo věk se převedou do jedné trojice, ve které bude objekt literální uzel – tedy konkrétní hodnota. Za takového zástupce je možné uvést trojici „personId_1“ – „heightInCm“ – „180“. Všechny propojení, které spojují dvě entity se také převedou do jedné trojice, objekt i subjekt však bude zastoupen objektovým uzlem, který zastupuje entitu. To ukazuje například trojice „personId_1“ – „knows“ – „personId_15“.

Je důležité poznamenat, že generátor negarantuje vytvoření spojitého datasetu. Každá entita má svoje vlastnosti (z toho vyplývá, že vždy bude využita v alespoň nějaké trojici), takže pokud je parametrem specifikováno, že má být vytvořeno 60 měst, tak je jisté, že se

všech 60 zástupců ve výsledném datasetu objeví. Kdyby ale dataset obsahoval například pouze 10 osob, které by bydlely v jednom městě, tak by nebyly všechny využity. Reálně se však předpokládá, že počet osob několikanásobně převýší počet ostatních entit a v tom případě dataset spojitý bude. Z hlediska specifikace modelu RDF se ale tak jako tak jedná o validní možnost, a tedy není nijak řešena v tomto generátoru.

Nakonec jsou jednotlivé trojice převedeny do syntaxe RDF/XML a uloženy do jediného souboru. Ten je poté možné přímo načíst do databázového serveru.

6.2 Klientská aplikace

Implementace klienta dodržuje strukturu definovanou v návrhu. V této kapitole budou poté podrobně analyzovány její jednotlivé části (*main*, *controller*, *graph* a *SPARQLAdapter*), které jsou v kódu reprezentovány sadami funkcí a třídami. Hlavní zaměření bude na jejich účel spolu s použitými technologiemi, datovými strukturami a algoritmy.

Main

Jedná se o část, která tvoří základ klienta. Spouští se po začátku aplikace, obsahuje globální parametry a zajišťuje několik důležitých funkcí. První z nich je inicializace. Získá se grafický kontext prvku canvas a do HTML struktury se vloží ovládací prvky (tlačítka, posuvníky atd., <canvas> je definován přímo v HTML kódu). Také se aktivuje myš nastavením patřičných event funkcí. K tomu je využito HTML DOM. Dále jsou při inicializaci vytvořeny instance všech definovaných tříd. Poté následuje start programové smyčky (v originále frameloop). Její pseudokód je následující:

```
function Frameloop()
{
    CalculateDeltaTime();
    SychnronizeHTMLElements();
    controller.ManageApplication();
}
```

Je vypočítán čas, který uběhl od provedení předchozí smyčky, pak se provede synchronizace HTML prvků. To zahrnuje dvě věci: načtení aktuálních hodnot vstupů do globálních proměnných, díky kterým může jakákoliv třída přistoupit k jejich aktuálnímu stavu a následnou synchronizaci více prvků dohromady, kde je to vyžadováno (například když hodnota může být měněna posuvníkem i zadáním hodnoty textově – oba prvky ale stále musí obsahovat stejné hodnoty, nehledě na to, který byl změněn). Tento krok spravuje všechny vstupy kromě myši (k té je také možno přistoupit přes globální proměnné, ty se ale mění pomocí asynchronních volání) a tlačítek (jejich stisk přímo zavolá asynchronní funkci). Poté je řízení předáno třídě *controller*. Samotný frameloop je řízen prohlížečem, který se jej snaží udržet na 60 cyklech za sekundu (dále fps – frames per second). Při nedostatku výkonu se aplikace zpomalí.

Controller

Tato třída obsahuje veškerou řídící logiku aplikace, která je závislá na aktuálním stavu a zadaných vstupech. Skládá se ze synchronní a asynchronní části. Princip řízení je poté následující: zpracují se vstupy a podle aktuálního stavu se zavolají metody z jiných tříd, které provedou požadované úkony. Struktura synchronní části je následující:

```
ManageApplication()
{
    graph.ClearCanvas();
    graph.UpdateAllVariablesOnFrameStart();
    ProcessMouseInputsAndApplyThem();
    graph.DrawGraph();
}
```

Ta zajišťuje vykreslování již přijatých a zpracovaných dat pomocí třídy *graph* do plátna. To je nejprve vymazáno, poté následuje reset interních proměnných. Pak se přejde ke zpracování vstupů z myši (vizualizace v plátně se ovládá pouze myší), podle kterých se zavolají konkrétní metody například pro posun, přiblížení nebo oddálení. Jakmile jsou všechny změny aplikovány, nový stav grafu je vykreslen.

Asynchronně se poté provádí komunikace s databází a všechny akce, které se aktivují stiskem tlačítka. Jejich průběh je obdobný a je demonstrován na následujícím příkladu, který načítá všechny data z repositáře.

```
LoadWholeDatabase_REQUEST()
{
    graph.ClearGraph();
    SPARQLAdapter.SetServer(inputServerAddress);
    SPARQLAdapter.SetRepository(inputRepositoryName);
    SPARQLAdapter.ConstructAndExecuteQuery_LoadAllNodes();
}

LoadWholeDatabase_ACCEPT(data)
{
    graph.AddTriplesToGraph(data);
    graph.DistributeNodesInPlane_METHOD();
}
```

Jak již bylo řečeno, všechny asynchronní vstupy zachytává *main*, ten je však okamžitě předá zavoláním metody třídě controller. Načítání dat začíná vymazáním aktuálních, poté se nastaví adresa serveru a název repositáře na hodnoty vložené uživatelem. Pak následuje samotný požadavek na server, který zajišťuje třída *SPARQLAdapter*. Ten je také asynchronní. Po dokončení je tedy zpětně zavolána další metoda, která data přijme, vloží do grafu a provede jejich vizualizaci dle zadaného algoritmu.

SPARQLAdapter

Třída *SPARQLAdapter* zajišťuje veškerou komunikaci s databázovým serverem. K tomu také náleží příprava před odesláním požadavku a zpracování přijatých dat. Ukázkový pseudokód uvedený níže navazuje na příklad z předchozí kapitoly o třídě *controller*.

```
ConstructAndExecuteQuery_LoadAllNodes()
{
    this.ConstructQuery();
    this.RequestSPARQLQueryResult()
    {
        this.ConstructRequest();
        this.SendRequest();
```

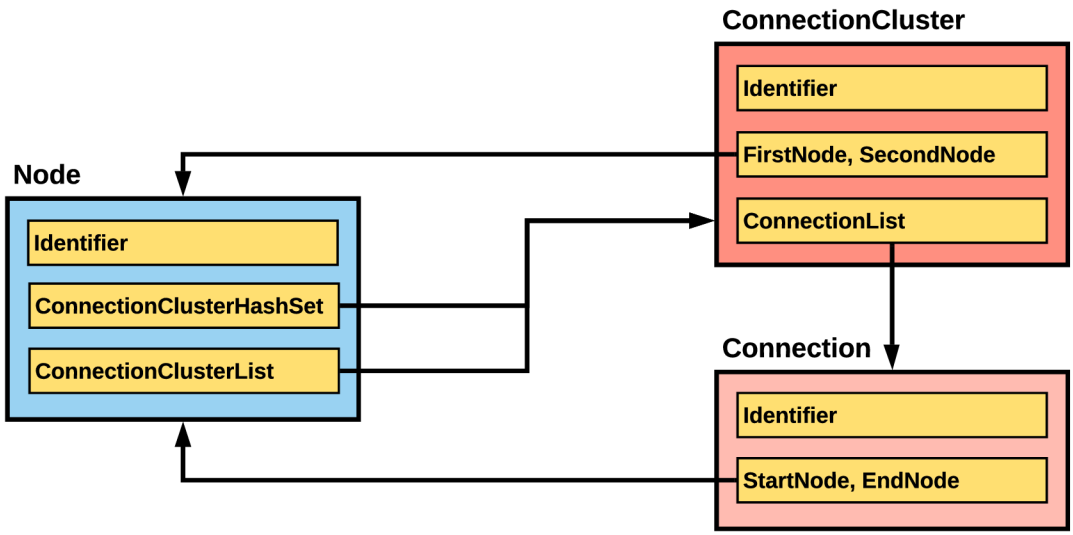
```
    }
}

Onload()
{
        controller.LoadWholeDatabase_ACCEPT(
        this.ParseQueryXMLTriplesInto2DTable(response));
}
```

Proces zaslaní požadavku na server začíná vytvořením SPARQL dotazu. Třída obsahuje několik šablon, do kterých jsou doplněny konkrétní identifikátory dle potřeby. Jakmile je vytvořen, je nutné ho zakódovat do URL formátu. Poté se může přejít k vytvoření samotného požadavku. Ten respektuje strukturu rdf4j API, která byla definována v 4.3. Ta je vyplněna aktuálními hodnotami a požadavek je odeslán. Následně jsou data asynchronně přijata ve specifikovaném formátu. Tento řetězec je poté rozdělen do struktur, které umí přijmou třída *graph*. To je realizováno pomocí regulárních výrazů, které jsou běžně dostupné v jazyce JavaScript. Zpracovaná data jsou poté vrácena třídě *controller*.

Graph

Jako poslední bude analyzována třída *graph*, která jako celek zajišťuje veškerou funkcionalitu související s vizualizací. Je ze všech nejrozsáhlejší a bude tedy rozebrána po částech. Ty se zaměří na vkládání dat do grafu, strukturu uložení grafu, vizualizační algoritmy, manipulaci s vykreslenými daty a samotné vykreslení do webové stránky.

Analýza tedy začne rozebráním struktury, pomocí které je graf uložen. Její implementaci byla věnována maximální pozornost, aby co nejlépe splnila návrhové cíle. Důraz byl kladen především na to, aby přístup k datům a vkládání dat bylo co nejrychlejší. V průběhu vývoje byla několikrát upravována. Finální verze je vyobrazena na obrázku 6.2.



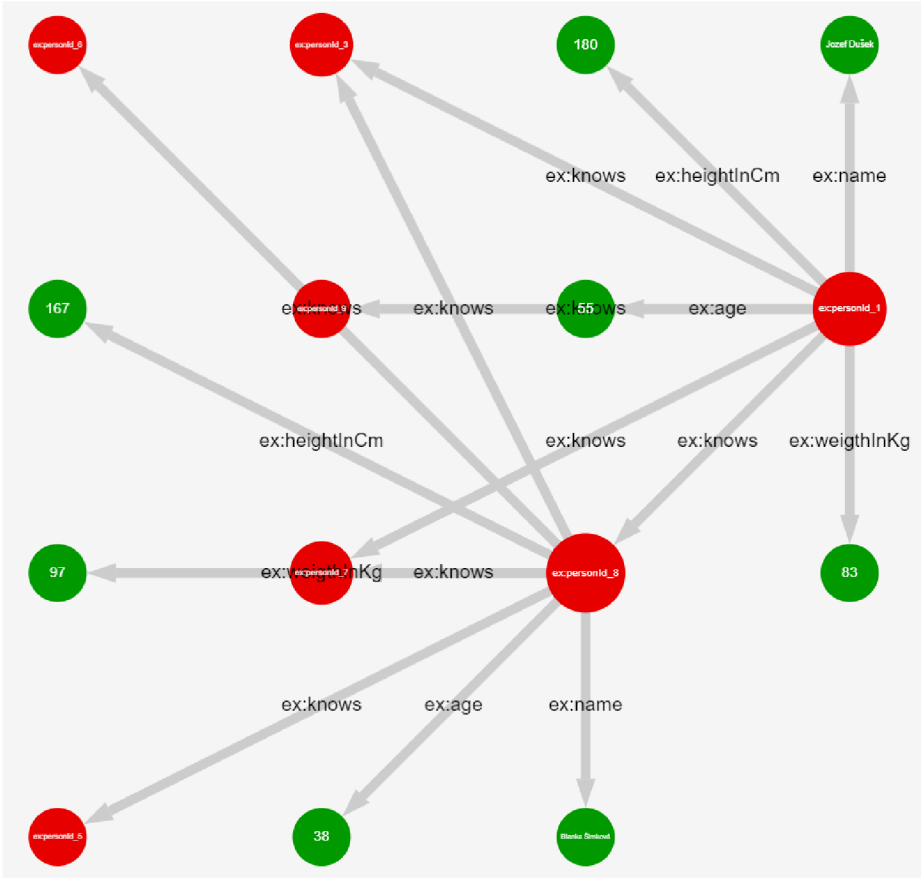
Obrázek 6.2: Struktura uložení grafu na klientovi.

Tvoří ji tedy tři třídy: *Node* (z ní dále dědí třídy *ObjectNode* a *LiteralNode*), *Connection* a *ConnectionCluster*. Všechny disponují jednoznačným identifikátorem, podle kterého se dají dohledat. Pokud již entita má identifikátor v databázi, tak je převzat, jinak je vytvořen uměle. Node zastupuje uzel, *ConnectionCluster* obaluje všechny propojení mezi dvěma uzly a *Connection* poté reprezentuje jedno konkrétní. Tyto třídy jsou propojeny odkazy. *Node* obsahuje seznam všech entit *ConnectionCluster*, v těch je následně uchován seznam jednotlivých propojení. Entity *Connection* i *ConnectionCluster* poté zpětně ukazují na uzly (orientovaná je ale jen entita *Connection*). Aby bylo procházení co nejefektivnější, jsou zde několikrát použity hashovací tabulky. Využívá se faktu, že mají konstantní složitost vyhledávání v závislosti na počtu uložených prvků. Je pomocí nich například implementovaný seznam všech uzlů v databázi nebo seznam všech *ConnectionCluster* entit v uzlu. To zajišťuje najití jakékoliv entity v grafu v konstantním čase. Pokud by tedy bylo nutné přistoupit k seznamu všech propojení mezi uzly s identifikátory „u1“ a „u2“, vyhledal by se nejdříve první uzel v hashovací tabulce obsahující všechny uzly, v něm by se opět pomocí hashovací tabulky našla konkrétní *ConnectionCluster* entita. Celkově by tedy byla operace provedena v konstantním čase. Existují však situace, kdy je potřeba rychle projít všechny entity, proto většina případů obsahuje zároveň i normální seznam obsahující odkazy na stejná data. To vede k navýšení potřebné paměti, není ale naráženo na žádné limity a stále je zachována lineární prostorová složitost vhledem k součtu uzlů a hran.

Dalším důležitým aspektem struktury, který musel být efektivně navržen je vkládání dat, protože stejně jako procházení bude prováděno za běhu aplikace. Navíc počty vkládaných trojic budou vysoké. Jak již bylo demonstrováno před chvílí, vyhledávaní entit v grafu má konstantní složitost. K tomu bude navíc využita další vlastnost hashovacích tabulek – konstantní časová složitost vložení prvku. Tyto vlastnosti jsou poté spojeny dohromady. Nejdříve se při vkládání musí zjistit, zda prvek v databázi už existuje. K tomu poslouží jednoduché vyhledání. Když je prvek nalezen, tak se nebude dělat nic nebo se rozšíří. Pokud nebude nalezen, tak se vytvoří. To kopíruje standard RDF. Celkově je tedy dosaženo konstantní složitosti vložení trojice, což je optimální výsledek. Znamená to, že když bude do grafu nutné vložit několik nových trojic, čas potřebný k provedení nebude záviset na objemu dat, který už je v databázi, ale pouze na počtu nových položek. Celkově tedy vložení n trojic bude disponovat časovou složitostí $O(n)$.

Jakmile jsou trojice uloženy v grafu, je možné přejít k samotné vizualizaci. Začíná se umístěním uzlů do prostoru. To má tři kroky: předzpracování, provedení daného algoritmu a finalizaci. Předzpracování mají algoritmy unikátní, finalizace je však společná pro všechny.

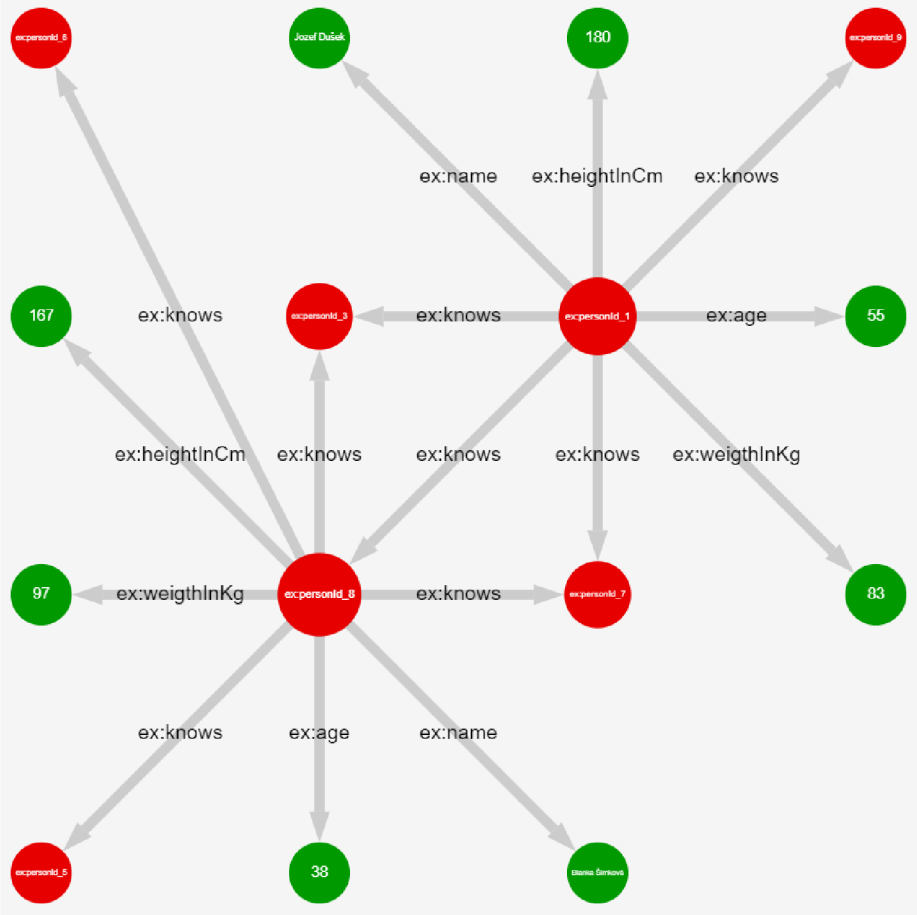
Greedy metoda začíná přípravou čtvercové mříže. Ta má nejmenší možný rozměr, který pojme všechny uzly. Tedy pokud by bylo nutné vykreslit 78 uzlů, zvolený rozměr by byl 9x9 a zbyly by 3 volné pozice. Poté se do mříže začnou vkládat uzly. K uchování množiny uzlů, které už jsou vloženy je využita hashovací tabulka (tedy dotaz na to, zda je uzel umístěn v mříži má konstantní časovou složitost). Pro uložení množiny uzlů, které čekají na zpracování je využita fronta. První uzel je zvolen náhodně a je vložen do středu mříže. Poté se projdou všichni jeho sousedé a najde se pro ně v mříži takové volné místo, které je nejblíže aktuálnímu uzlu. Tam se umístí. Sousedé jsou přitom také vloženi do fronty, ze které se v další iteraci vyjme nový uzel ke zpracování (jedná se o průchod do šířky). Iterace také obsahuje větev, která řeší situaci, kdy je graf nespojitý. Toto pokračuje, dokud nejsou všechny uzly umístěny. Celková časová složitost této metody vzhledem k počtu prvků n je $O(n^2)$, protože pro každý prvek je nutné prohledat všechny zbylé volné pozice, jejichž počet je úměrný n. Výsledek této metody je vyobrazen na obrázku 6.3.



Obrázek 6.3: Ukázka výsledného rozložení uzlů metodou greedy.

Metoda Greedy-swap poté navazuje na předchozí výsledek a funguje po krocích. Algoritmus zkouší prohodit pozici každého uzlu se všemi ostatními, tedy časová složitost kroku vzhledem k počtu uzlů n je $O(n^2)$. Pokud by prohození vedlo ke snížení součtu všech délek propojení v grafu, tak se uskuteční, jinak se s uzly nic neděje. Pokud bylo v jednom kroku provedeno alespoň jedno prohození, tak se provede krok další. To pokračuje, dokud neexistuje žádné prohození, které by vedlo ke zlepšení metriky. Tento přístup by mohl v nejhorším případě vést až k exponenciální časové složitosti, kdyby byly zlepšení velice pozvolné, v praxi však konverguje mnohem rychleji. Konečnost je ale zaručena, protože součet vzdáleností nelze donekonečna snižovat – existuje absolutní minimum. Reálné vyhodnocení časové náročnosti se provede až při testování. Výsledek této metody je vyobrazen na obrázku 6.4.

Nakonec zbývá metoda force-directed. Jak bylo řečeno v návrhu, její koncept je velice jednoduchý, implementace však přinesla několik problémů. Některé byly vyřešeny, jiné jsou pouze minimalizovány. Protože vychází z fyzikální simulace založené na silách, tak je nutné, aby při startu nebyly uzly příliš blízkou sebe, to by totiž vedlo k obrovským odpudivým silám, které by vyústily v přetečení proměnných. V předzpracování jsou tedy uzly rozmístěny daleko od sebe. V této situaci by měly převážit přitažlivé síly – uzly by se tedy měly přitáhnout do finální pozice, ne odpudit. Průběh algoritmu poté opět probíhá po krocích, jejichž počet si může nastavit uživatel. Čím více kroků bude provedeno, tím více se bude výsledek blížit optimálnímu ustálenému stavu – to je demonstrováno na obrázku 6.5.



Obrázek 6.4: Ukázka výsledného rozložení uzlů metodou greedy-swap.

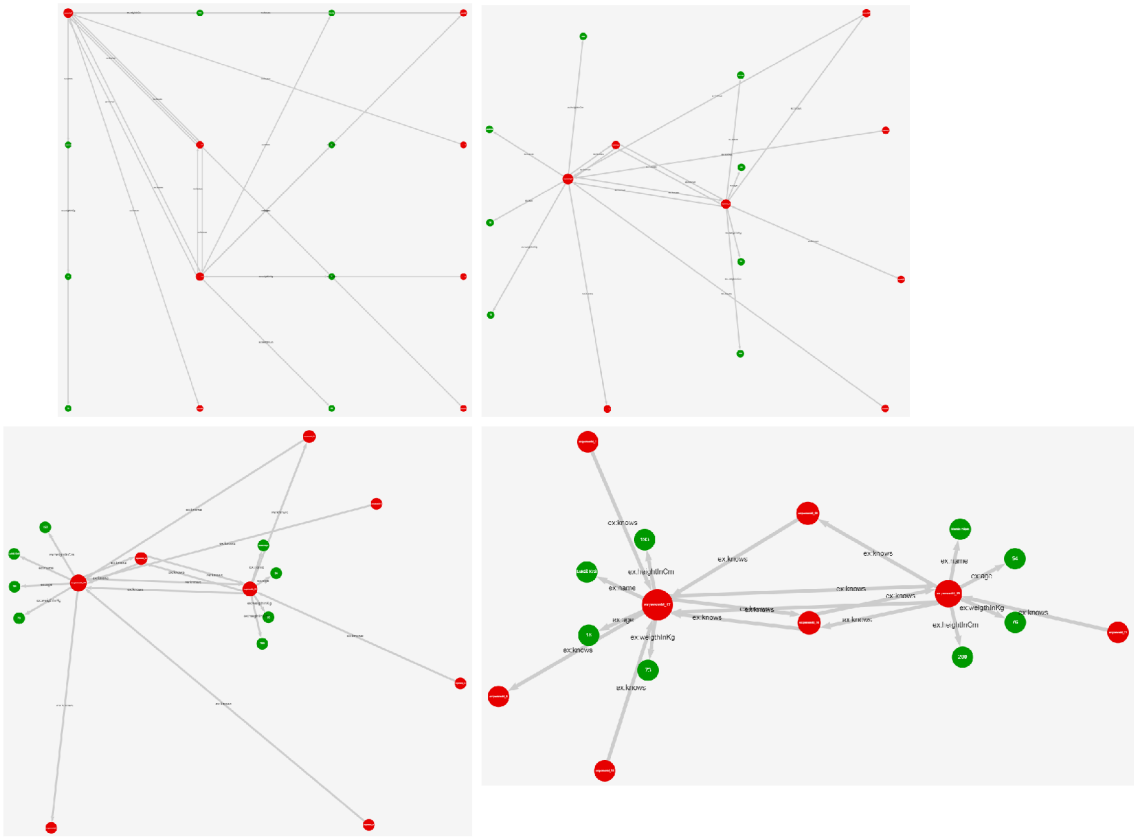
Krok poté probíhá následovně. Pro každý uzel se vypočítá vzdálenost d s každým dalším uzlem. Ta bude určovat velikosti přitažlivých a odpudivých sil. Ty jsou reprezentovány vektorem – směr definuje přitažlivost nebo odpudivost, délka poté velikost. Všechny uzly, ať propojené nebo ne se navzájem odpuzují. Vektor síly směřuje přímo opačně od druhého uzlu a velikost je určena vzorcem 6.1.

$$\frac{kp_1p_2}{d} \tag{6.1}$$

kde:

- k je konstanta
- p_1 je počet propojení prvního uzlu
- p_2 je počet propojení druhého uzlu
- d je vzdálenost uzlů

Protože jsou ve vzorci zohledněny i počty propojení, tak obsáhlejší uzly budou dále od sebe, což povede k logickému oddělení a omezení zahlcení. Síla je poté nepřímo úměrná vzdálenosti. To je nutné kvůli tomu, aby se grafy s velkým počtem uzlů příliš nerozprostřeli.



Obrázek 6.5: Ukázka výsledků metody force-directed při různém počtu iterací. Ty jsou následující: vlevo nahoře – 0, vpravo nahoře – 5, vlevo dole – 50, vpravo dole – 500.

Přitahovány jsou poté všechny propojené uzly. Vektory směřují navzájem na sebe a velikost je definována vzorcem 6.2.

$$kdP \tag{6.2}$$

kde:

- k je konstanta
- d je vzdálenost uzlů
- P je počet propojení mezi uzly

Jak je vidět, tak na rozdíl od odpudivé síly je závislost přitažlivé síly lineárně závislá na vzdálenosti. To povede k tomu, že bude existovat místo kde budou obě v rovnováze (to dá se řídit konstantami). V té se budou uzly snažit ustálit. Přitažlivá síla literálních uzlů je poté dále vynásobena pěti, aby se zabránilo jejich přílišnému vzdálení od rodičovských uzlů.

Jakmile jsou vypočítány všechny vektory reprezentující odpudivé a přitažlivé síly, tak následuje jejich sečtení do jednoho výsledného vektoru pro každý uzel. Nakonec jsou uzly posunuty ve směru a podle síly daného vektoru. Celková časová složitost kroku vzhledem k počtu uzlů n je $O(n^2)$. Je však nutné mít na paměti, že se krok provede několikrát. Výsledek této metody je vyobrazen na obrázku 6.6.

Obrázek 6.6: Ukázka výsledného rozložení uzlů metodou force-directed.

Jak již bylo zmíněno, s touto metodou bylo při implementaci několik problémů. Hlavním problémem bylo a stále v určité míře je to, že algoritmus není prostorově stabilní vůči počtu uzlů. Tím se myslí to, že relativní vzdálenosti uzlů se mění a projevuje se to tak, že v rozsáhlých grafech vznikají mezi uzly větší mezery. Tento problém byl omezen zvolením vhodných konstant u výpočtu sil, nikoliv však eliminován. Další problém byl s překryvem literálních uzlů obzvláště v případě, kdy jich bylo hodně u sebe od jednoho rodiče. To bylo vyřešeno zvýšením odpudivé síly mezi literálními uzly na trojnásobek. Požadavek na vizualizaci nespojitých datasetů se také jevil jako problémový, nakonec však nebyl nijak řešen, protože i když části grafu nejsou spojeny, tak odpudivé síly klesají rapidně se vzdáleností – části se tedy od sebe nevzdálí daleko. Poslední problém představovaly různé krajní případy. Za příklad takové situace je možné uvést případ, kdy databáze obsahuje uzly se stovkami propojení. Poté se může stát, že součet sil bude příliš velký, což povede k oscilaci uzlu nebo přetečení hodnot proměnných. Toto právě z části řeší velký iniciální rozestup uzlů, simulace však i ve finální verzi muže skončit neúspěchem. Chyba je však odchycena. Metoda má tedy stále svoje úskalí, vizualizace naprosté většiny případů však funguje v pořádku.

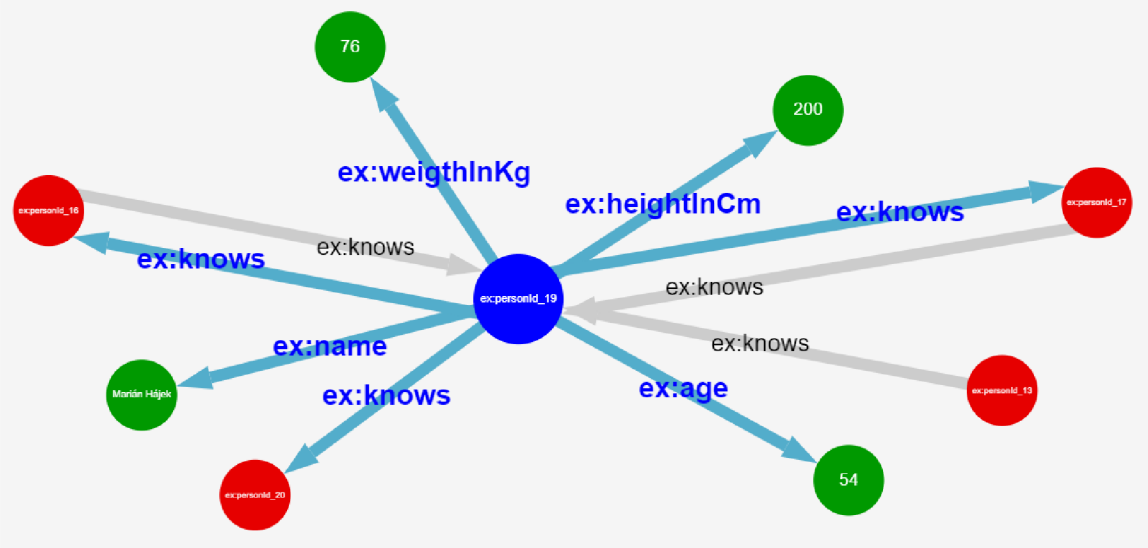
Po provedení algoritmů následuje finalizace. Jejím hlavním cílem je vypočtení parametrů pro vykreslení. Nejprve se vypočítají koeficienty pro velikosti kruhů a šířek propojení. Protože se absolutní pozice uzlů v grafu nemění po dokončení algoritmu, tak je také nutné vypočíst transformační matice pro převod grafu z reálných souřadnic do plátnových. Nakonec se uzly vloží do struktury quadtree.

Nyní je možné přejít k samotnému vykreslení do dvourozměrného plátna. To je realizováno pomocí standardních funkcí HTML prvku canvas. Ty mají pro tento projekt dvě velice užitečné vlastnosti: jsou samovolně akcelerovány na grafické kartě a umějí pracovat s transformacemi. Kombinace těchto vlastností je tedy využita ke zvýšení výkonu vykreslování tak, že všechny možné transformace grafu, které je nutné provádět v této aplikaci nejsou prováděny změnou souřadnic uzlů, ale vypočtením transformačních matic. Ty jsou poté před vykreslením vynásobeny mezi sebou a výsledná matice je předána prvku canvas, který pomocí ní transformuje graf do výsledné pozice za použití výkonu grafické karty. Nakonec je důležité zmínit, že funkcionalita prvku canvas zahrnuje princip frustum culling (objekty, které nejsou vidět nejsou vykresleny). Celý proces tedy začíná vynásobením matic. Poté se vykreslí tvary a linie následované popisy. To vykonávají samotné třídy uzlů a propojení, tento krok je tedy uskutečněn proiterováním všech těchto objektů v databázi a zavoláním patřičných metod. Tím je završena statická vizualizace.

Nakonec bude analyzována poslední část implementace – zajištění interaktivity. Uživatel může vizualizaci přibližovat, oddalovat a posouvat pomocí myši. Implementace převzala principy z webové stránky Google Maps poskytující mapy – posun kolečka o jednu pozici povede k přiblížení nebo oddálení o 25% a absolutní pozice na kterou myš ukazuje se při

tom nemění. Posunout celý graf je možné držením levého tlačítka myši. To vše je realizováno vypočtením dalších transformačních matic, které určují velikost a pozici pohledového okna. Uzly je potom také možné posouvat jednotlivě držením pravého tlačítka myši. Aby však aplikace nebyla zpomalena při posuvu uzlů v rozsáhlých grafech, tak je využita struktura quadtree, které byla vystavěna po dokončení zvoleného algoritmu. To má vzhledem k počtu uzlů v grafu n složitost $O(n \log n)$, jedná se však o jednorázový proces, který bude zabírat jen zlomek času v porovnání s výpočty samotných algoritmů. Klíčový fakt je potom ten, že vyhledání prvku v této struktuře má časovou složitost $O(\log n)$. Tedy nalezení prvku, který uživatel posunul bude také spadat do této třídy složitosti. Samotné posunutí je poté realizováno změnou absolutních souřadnic uzlu. Dalším způsobem, jak ovlivnit vizualizaci je možnost vypnout zobrazení popisů a orientaci propojení (například graf na obrázku 7.3 má vypnuté popisy i orientaci). To navýší výkon a zpřehlední graf. Nakonec se uzly dají zvýraznit (také se zde využívá quadtree pro nalezení uzlu, ukázka se nachází na obrázku 6.7). To může být použito ke zvýšení přehlednosti, primárním účelem je však jejich označení pro další manipulaci. Tou je buď expanze nebo zachování. Oba tyto úkony vyústí k vytvoření dotazu na server.

Tímto je dokončena celková analýza této aplikace z hlediska implementace. Finální vzhled je vyobrazen na obrázku 6.8.



Obrázek 6.7: Demonstrace zvýraznění uzlu v načteném grafu.

Obrázek 6.8: Finální vzhled výsledné aplikace.

Kapitola 7

Testování a výsledky

Finální fáze tohoto projektu se skládala ze sběru dat, testování a vyhodnocení. Použité datasety byly nejen vygenerovány, ale i staženy z externích zdrojů. Hlavním cílem testování poté bylo zjistit časové složitosti všech aspektů aplikace při velkém počtu vizualizovaných trojic. Konečné vyhodnocení poté bralo v úvahu jak objektivní výsledky, tak výsledný vzhled vizualizací.

Veškeré testování a generování pobíhalo na stolním počítači s operačním systémem Windows 10 64-bit, procesorem Intel i7-2600K a grafickou kartou NVIDIA GeForce GTX 1060.

7.1 Data

Většina datasetů, které byly použity pro testování pocházely z implementovaného generátoru. Hlavním důvodem pro toto rozhodnutí byla možnost zvolení přesných parametrů. Díky tomu bylo možné vygenerovat testovací sady o přesně daných počtech trojic a obsahující specifické struktury. Účelem externích datasetů bylo poté ověřit to, že klient respektuje standard a umí zpracovat data i z jiných zdrojů.

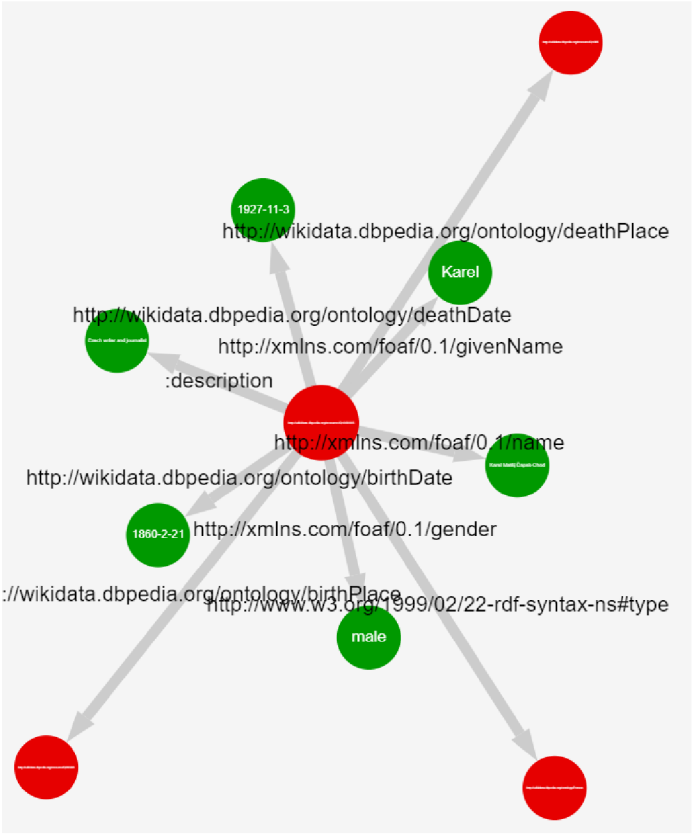
DBPedia

Zdrojem externích dat byla DBPedia[1]. Je to projekt, který se snaží vyextrahovat strukturovaná data z projektů Wikimedia (to zahrnuje online encyklopedii Wikipedia). Ty jsou poté přetransformovány do grafové struktury a jsou volně dostupné ke stažení. Pro potřeby testování stačilo použít pouze jediný dataset nazvaný *Dbpedia*. Ten vychází z datasetu *Person Data*, který obsahuje informace o osobách vyextrahovaných z anglické a německé Wikipedie (část jeho obsahu vyobrazuje obrázek 7.1). Před tím, něž ho ale bylo možné použít, tak musel být překonvertován do formátu .rdf, protože DBpedia ukládá data ve formátu .tll (Turtle). K tomu byl použit volně dostupný konvertor RDF2RDF[2]. Při tom byl ořezán jeho obsah na 99995 trojic, výsledná velikost je poté 10369 kB.

Generované datasety

Datasetů bylo vygenerováno několik a vždy se specifickým účelem. První z nich – *AllFeatures* představuje obyčejného zástupce. Využívá všechny možnosti generátoru a je určen

[1] https://wiki.dbpedia.org
[2] http://www.l3s.de/~minack/rdf2rdf/

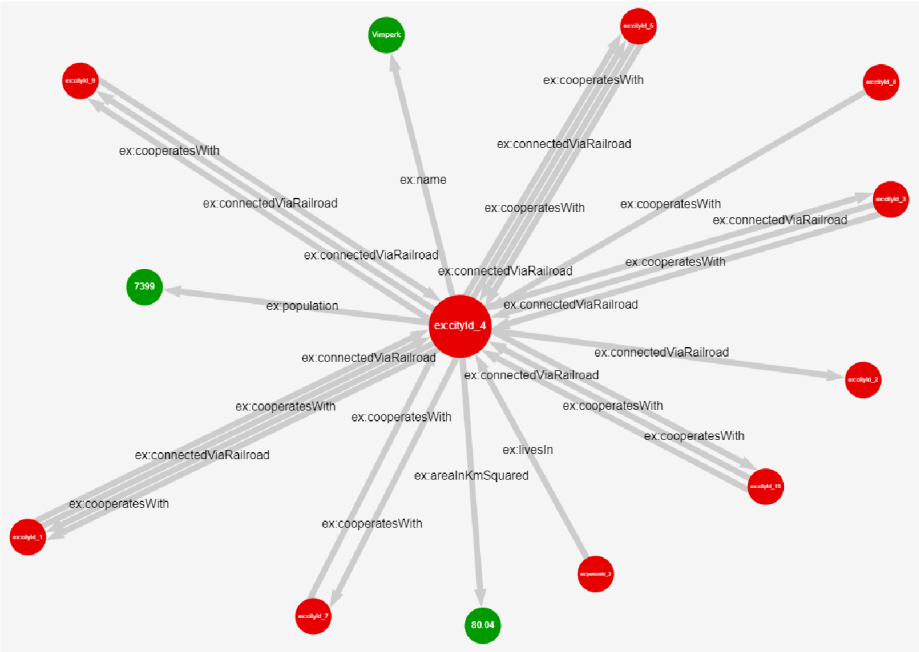


Obrázek 7.1: Ukázka vizualizace externího datasetu *Dbpedia*. Od generovaných datasetů se liší pouze absencí prexifů – to prodlužuje názvy a zvyšuje nepřehlednost.

pro demonstraci základních prvků aplikace. Druhý je *MultipleConnections* – ten slouží pro demonstraci vykreslení několikanásobných propojení (obrázek 7.2). Následuje sada tří datasetů *StructureDepth*. Ty obsahují pouze osoby a jejich účelem je otestovat schopnosti jednotlivých algoritmů vyobrazit logické struktury v datech. Každý disponuje jiným počtem hierarchických úrovní. Poslední je poté *PerformanceSet_200k* – ten bude použit k otestování výkonu všech částí aplikace. K tomuto účelu není nutné generovat datasetů více, protože počet načtených trojic je možné omezit v aplikaci. Přesnější specifikace všech zástupců jsou obsaženy v následující tabulce 7.1, která obsahuje také časy potřebné k vygenerování. Ty se pohybovaly nejvýše v řádech jednotek sekund (časy byly měřeny pomocí knihovny *chrono*), použití kompilovaného jazyka se tedy osvědčilo.

Tabulka 7.1: Specifikace vlastností vygenerovaných datasetů.

Název	Počet osob ve shluku	Hloubka hierarchie	Počet trojic	Velikost [kB]	Čas generování [ms]
AllFeatures	5	3	1553	120,01	2,01
MultipleConnections	5	1	152	13,93	0,32
StructureDepth2	5	2	175	14,64	0,338
StructureDepth3	5	3	1125	90,00	2,50
StructureDepth4	5	4	6875	538,26	21,36
PerformanceSet_200k	10	4	204711	15323,13	2260,95



Obrázek 7.2: Ukázka vizualizace datasetu *MultipleConnections*, který předvádí schopnost aplikace vykreslit vícečetná propojení.

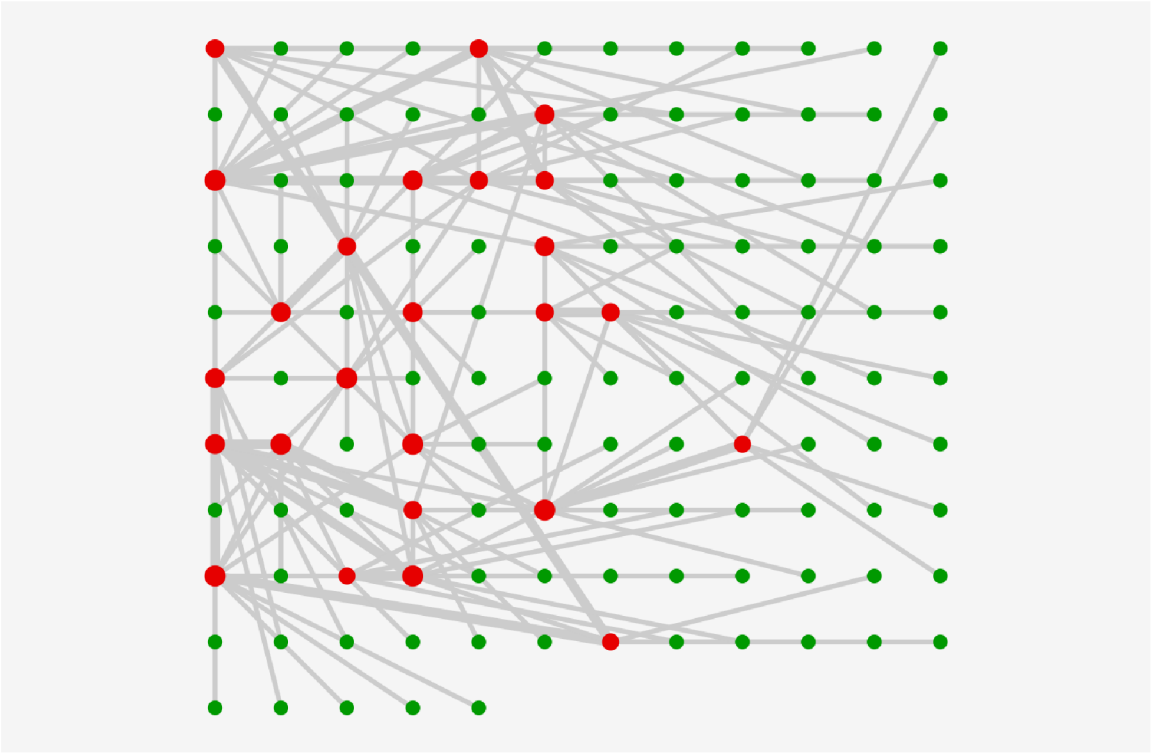
7.2 Testování kvality vizualizace

Kvalitní grafová vizualizace by měla splňovat dvě hlavní vlastnosti. První z nich je vyobrazení struktur obsažených ve grafu. To znamená, že mělo by být jasně patrné v jakém vztahu jsou jednotlivé entity podle jejich relativních pozic. Druhou je poté přehlednost – prvky by se neměly překrývat a měly by se dát dobře rozlišit. Obě tyto vlastnosti budou hlavním bodem testovaní a vyhodnocení. Počet iterací force-directed metody byl přitom nastaven na 2000.

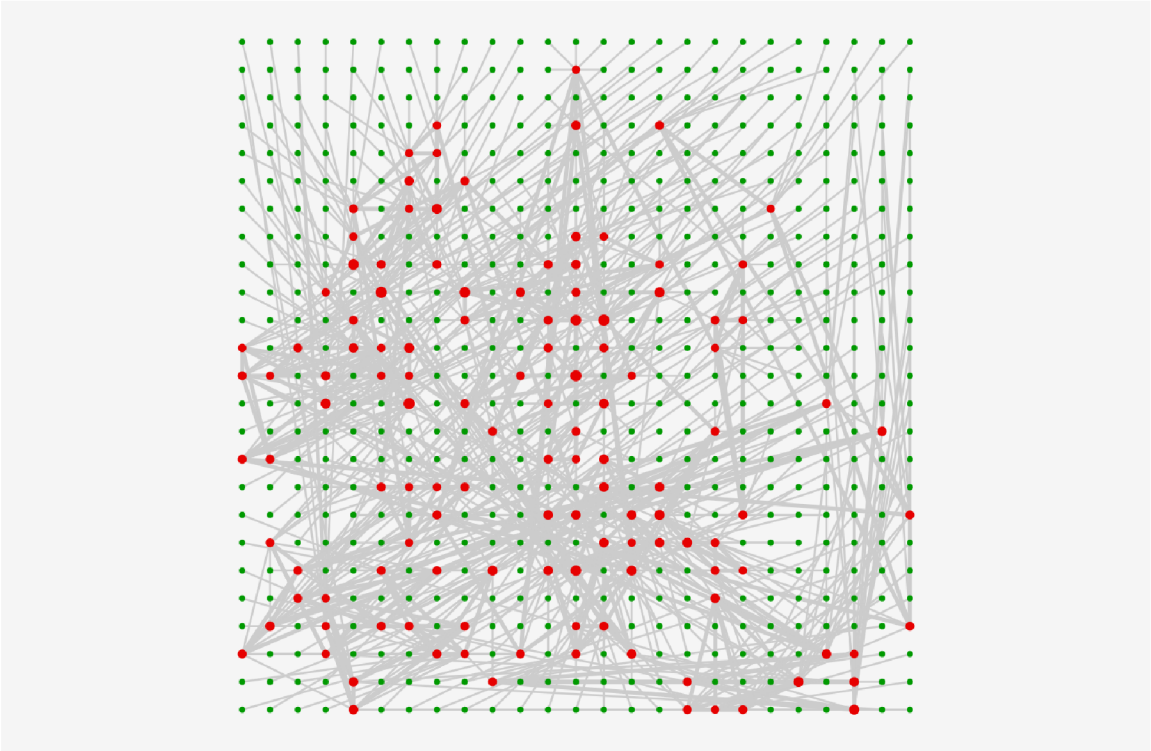
První na řadu tedy přijde analýza vyobrazení struktur jednotlivými algoritmy. Jak již bylo řečeno, pro tento účel byly nachystány tři datasety: *StructureDepth2*, *StructureDepth3* a *StructureDepth4*. Shluky v těchto datasetech vždy obsahují přesně pět osob, hloubka hierarchie je však rozdílná. *StructureDepth2* tedy tvoří pět shluků o pěti osobách, *StructureDepth3* pět shluků o pěti shlucích o pěti osobách atd. Hodnoceno poté bude to, jak zřetelná bude tato hierarchie ve výsledku. Analýza bude provedena postupně počínaje algoritmem greedy.

Jak je vidět na obrázcích 7.3, 7.4 a 7.5, tak algoritmus greedy neposkytuje dle očekávání dobré výsledky v tomto ohledu. Je sice zřetelné, že propojené uzly se shlukují k sobě, není ale patrná hierarchie, kterou datasety obsahují nehledě na počet úrovní nebo uzlů.

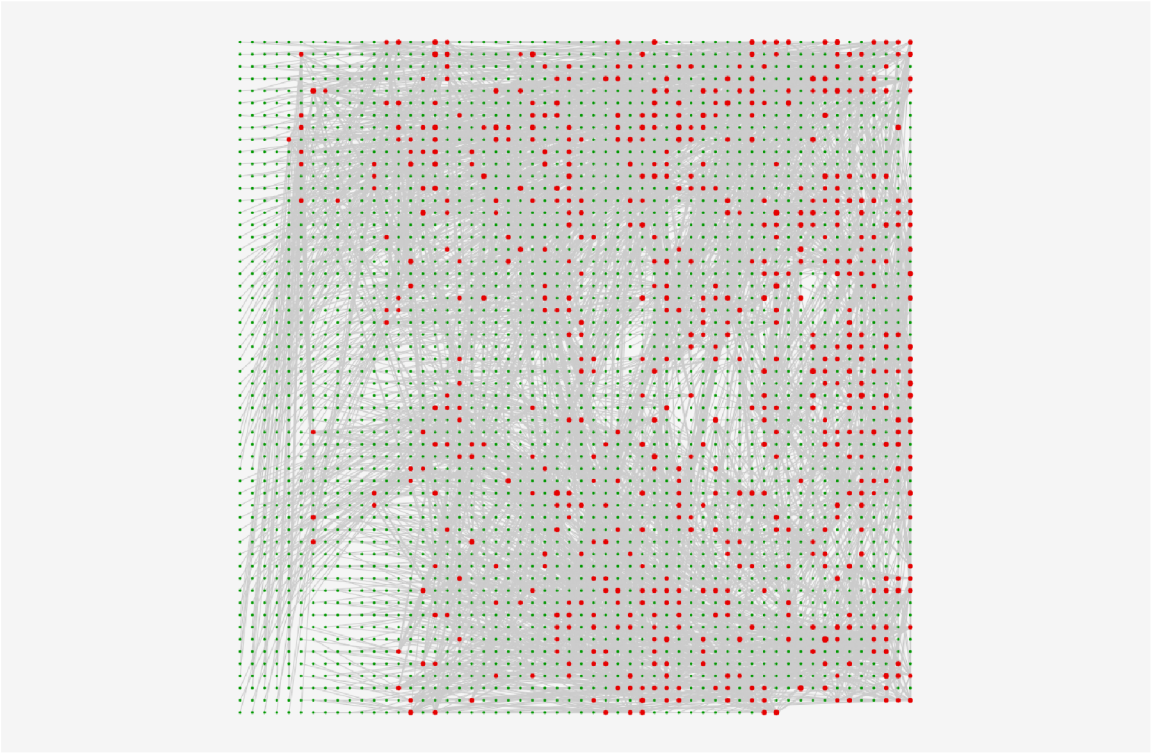
Další testovanou metodou byla greedy-swap. Ta se snaží vylepšit předchozí výsledky metody greedy dalším zpracováním, což se jí úspěšně daří, jak je vidět na obrázcích 7.6, 7.7 a 7.8. V prvním případě je vidět lepší oddělení shluků, ty jsou ale stále vidět jen čtyři. Další problém nastává při navýšení počtu hierarchií. V druhém a třetím případě je sice vidět logické oddělení, ale jen jedné úrovně. Obecně se však jedná o znatelné zlepšení předchozí metody.



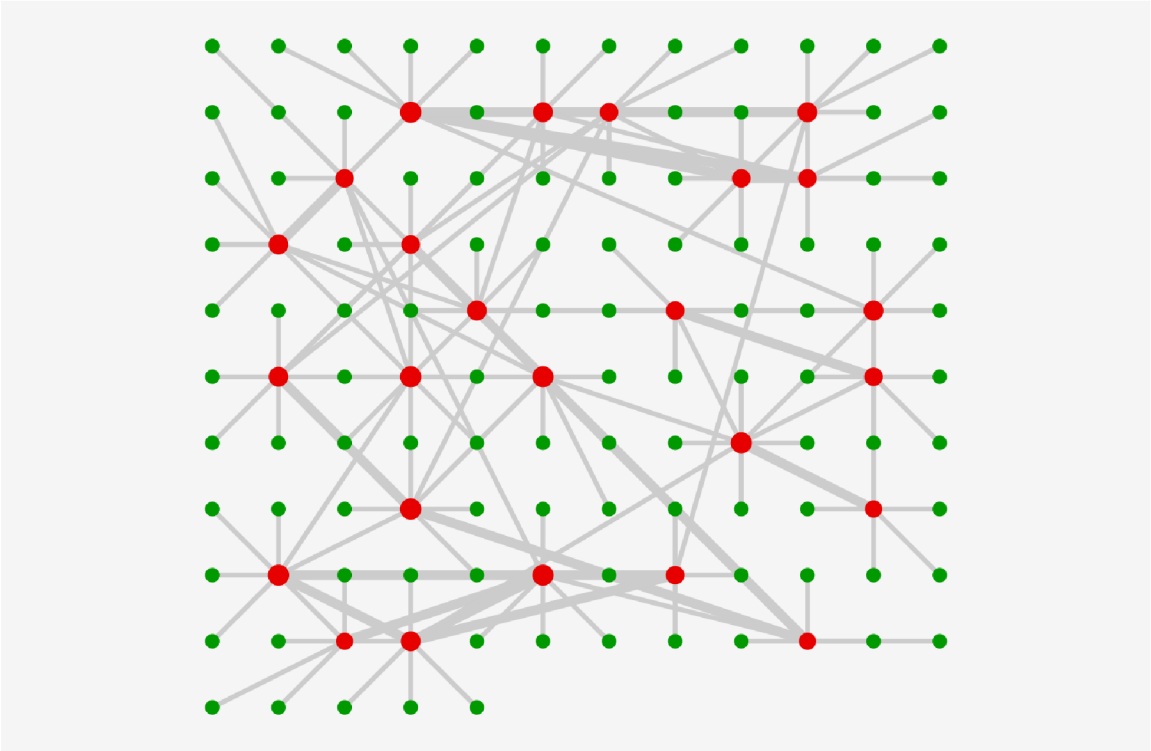
Obrázek 7.3: Vizualizce datasetu *StructureDepth2* algoritmem greedy.



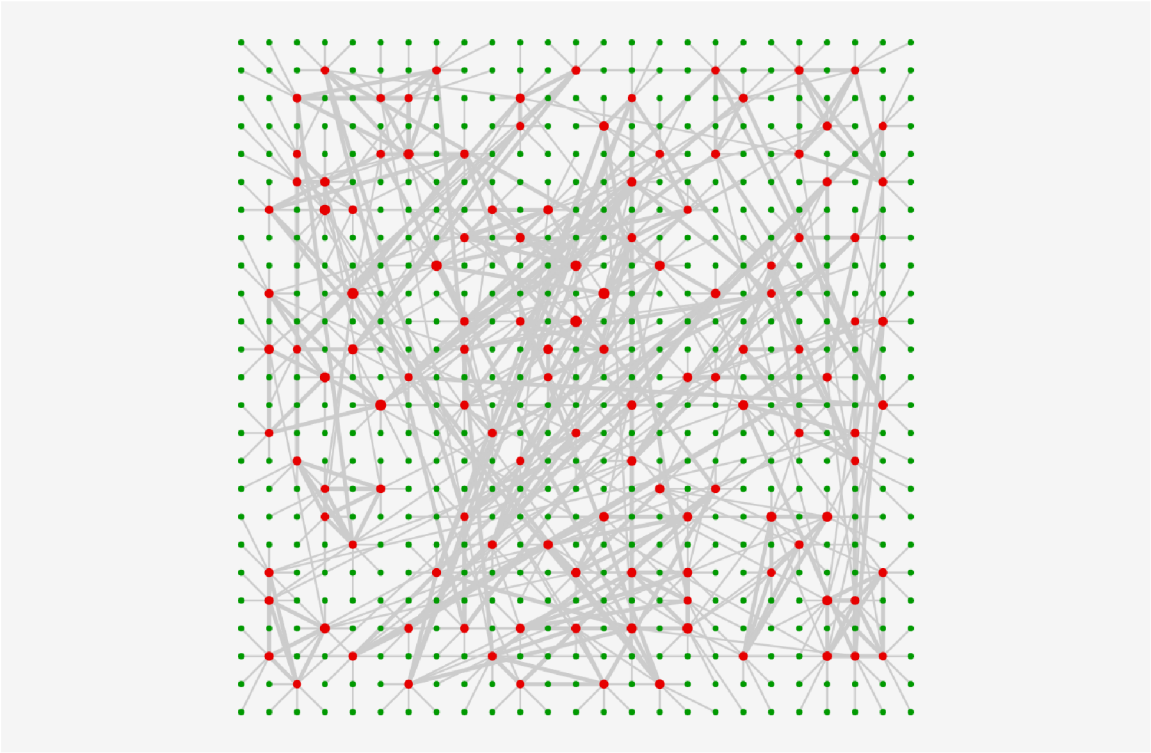
Obrázek 7.4: Vizualizce datasetu *StructureDepth3* algoritmem greedy.



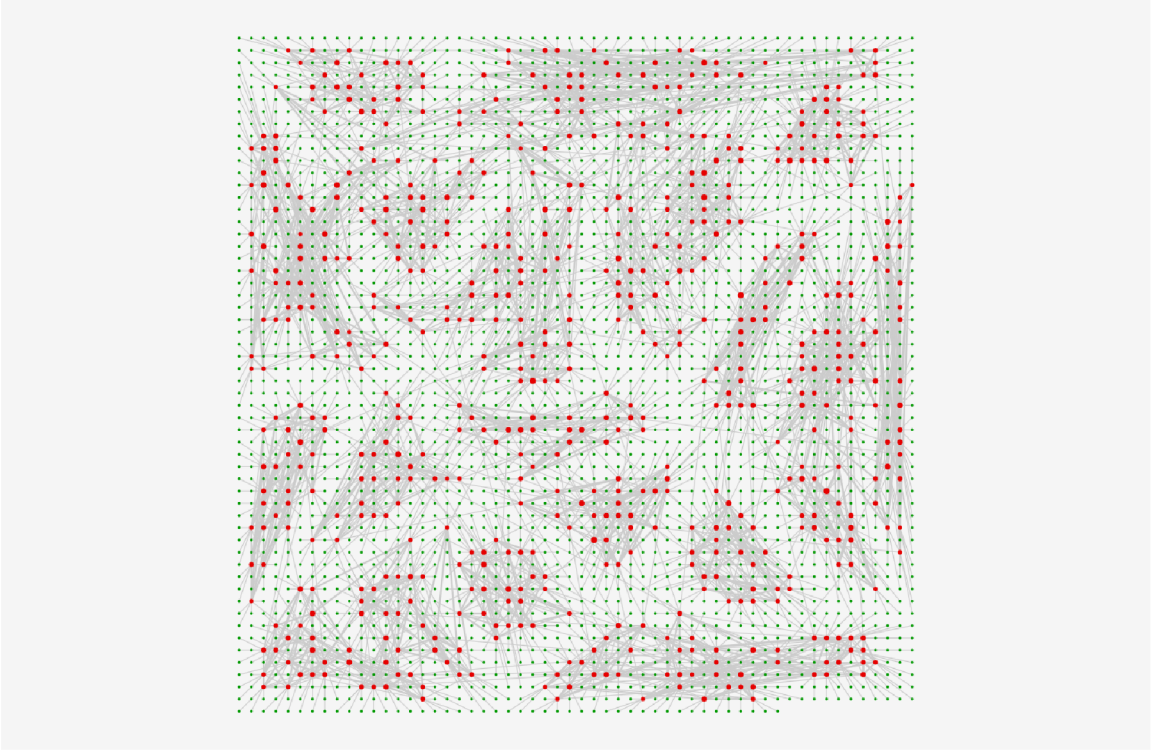
Obrázek 7.5: Vizualizce datasetu *StructureDepth4* algoritmem greedy.



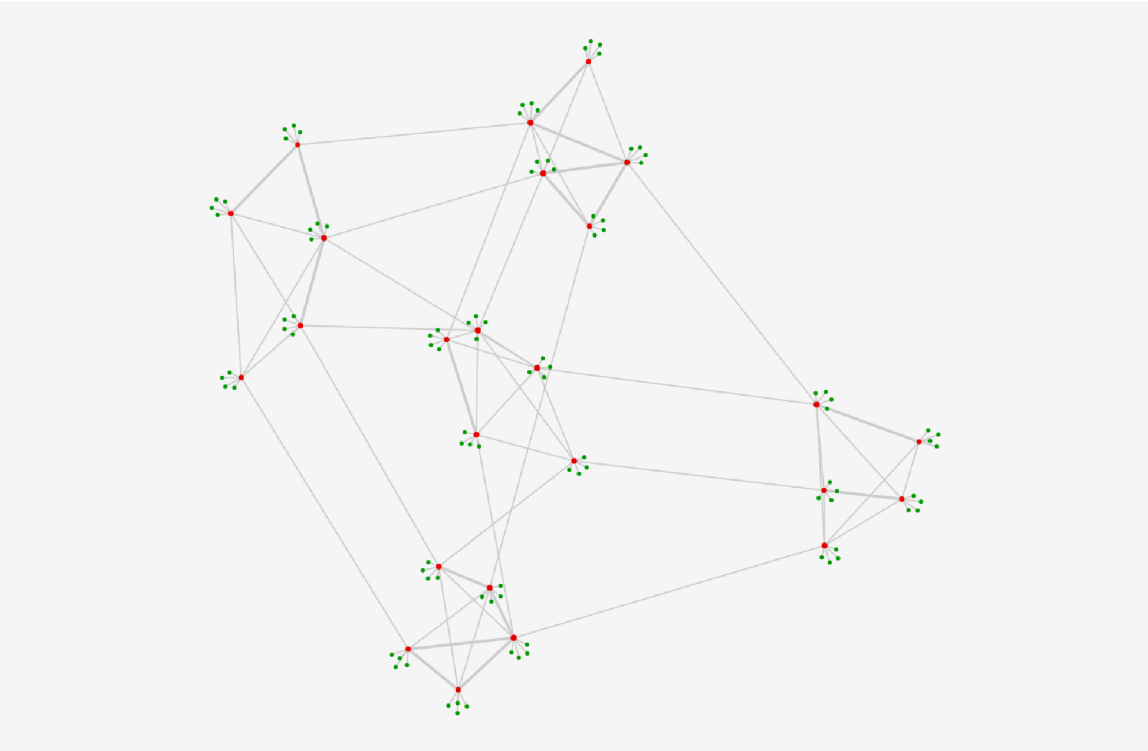
Obrázek 7.6: Vizualizce datasetu *StructureDepth2* algoritmem greedy-swap.



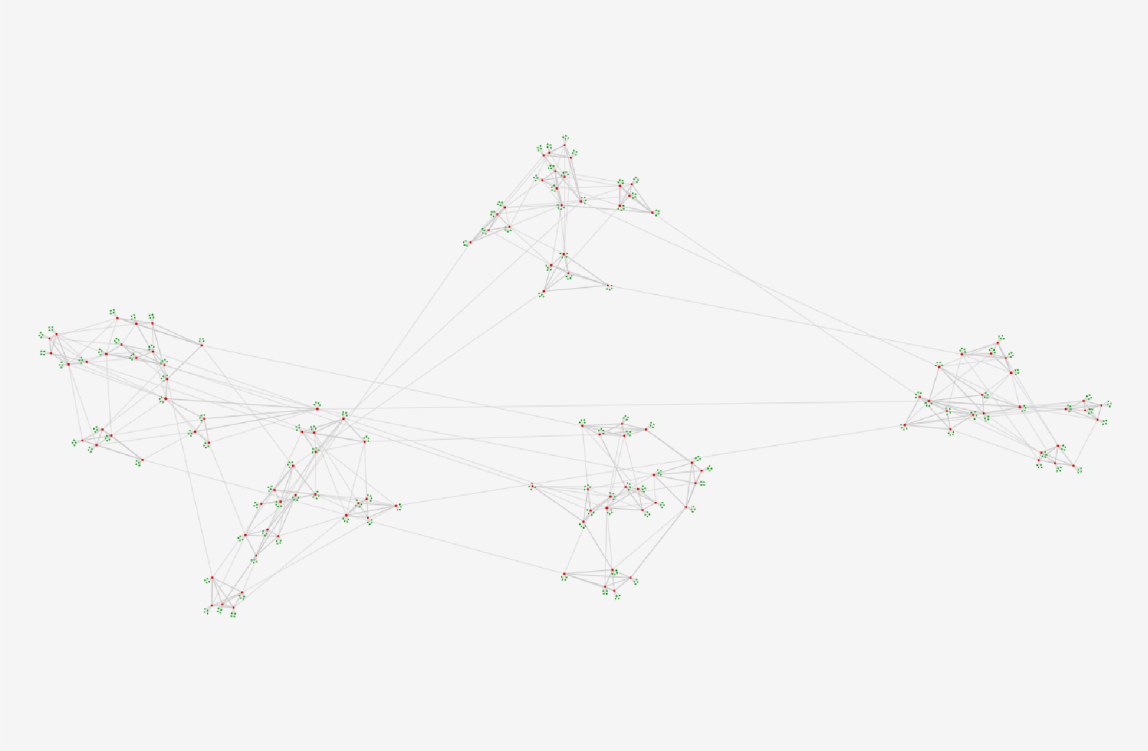
Obrázek 7.7: Vizualizce datasetu *StructureDepth3* algoritmem greedy-swap.



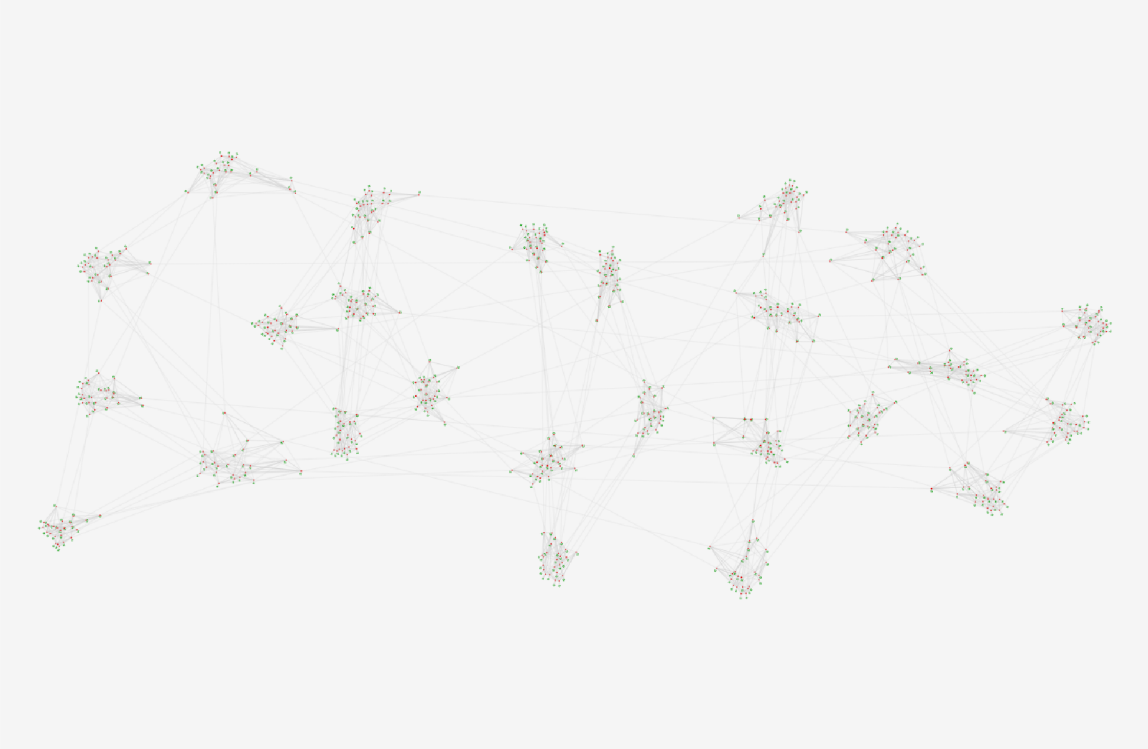
Obrázek 7.8: Vizualizce datasetu *StructureDepth4* algoritmem greedy-swap.



Obrázek 7.9: Vizualizce datasetu *StructureDepth2* algoritmem force-directed (2000 iterací).



Obrázek 7.10: Vizualizce datasetu *StructureDepth3* algoritmem force-directed (2000 iterací).



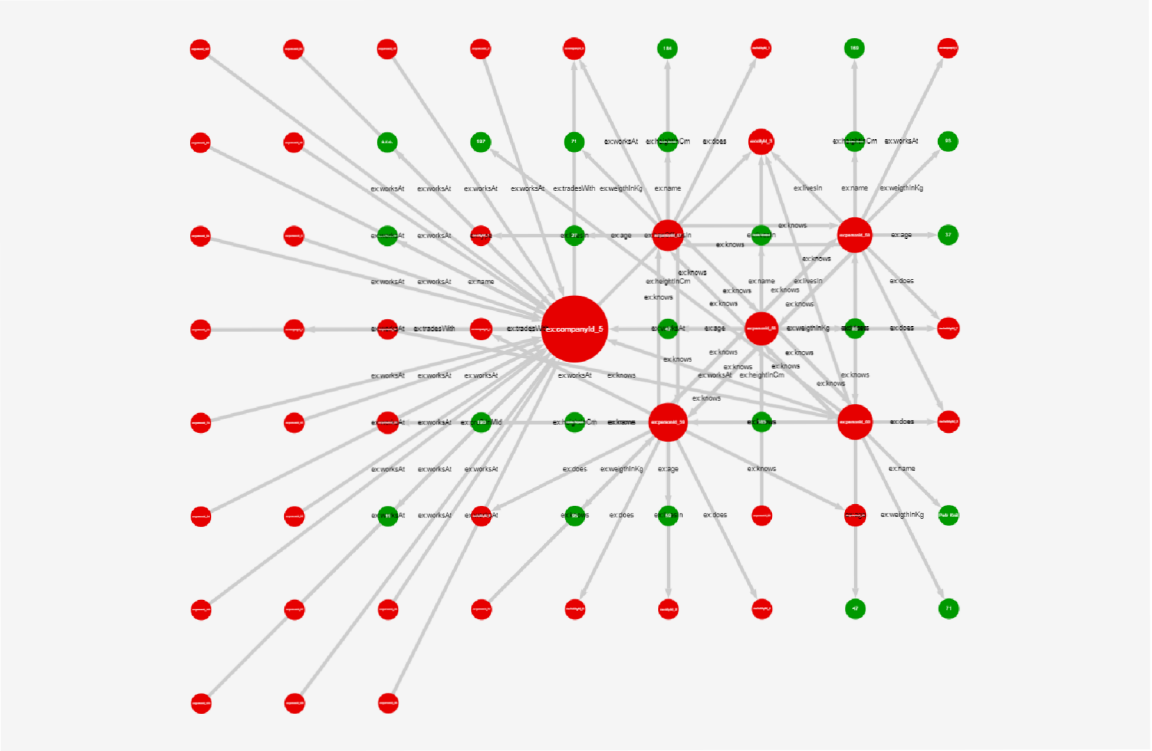
Obrázek 7.11: Vizualizce datasetu *StructureDepth4* algoritmem force-directed (2000 iterací).

Nakonec tedy zbývá metoda force-directed. Z jejích výsledků (obrázky 7.9, 7.10 a 7.11) je na první pohled patrné, že vyobrazuje struktury lépe než obě předchozí metody. První případ je perfektní, pět shluků je jasně viditelných. V druhém případě jdou poté vidět obě úrovně hierarchie současně. Při přidání třetí se však opět začínají ztrácet a je viditelná pouze jedna. To ale může být způsobeno provedením malého počtu iterací. Nepochybně se však jedná o nejlepší metodu pro tento účel.

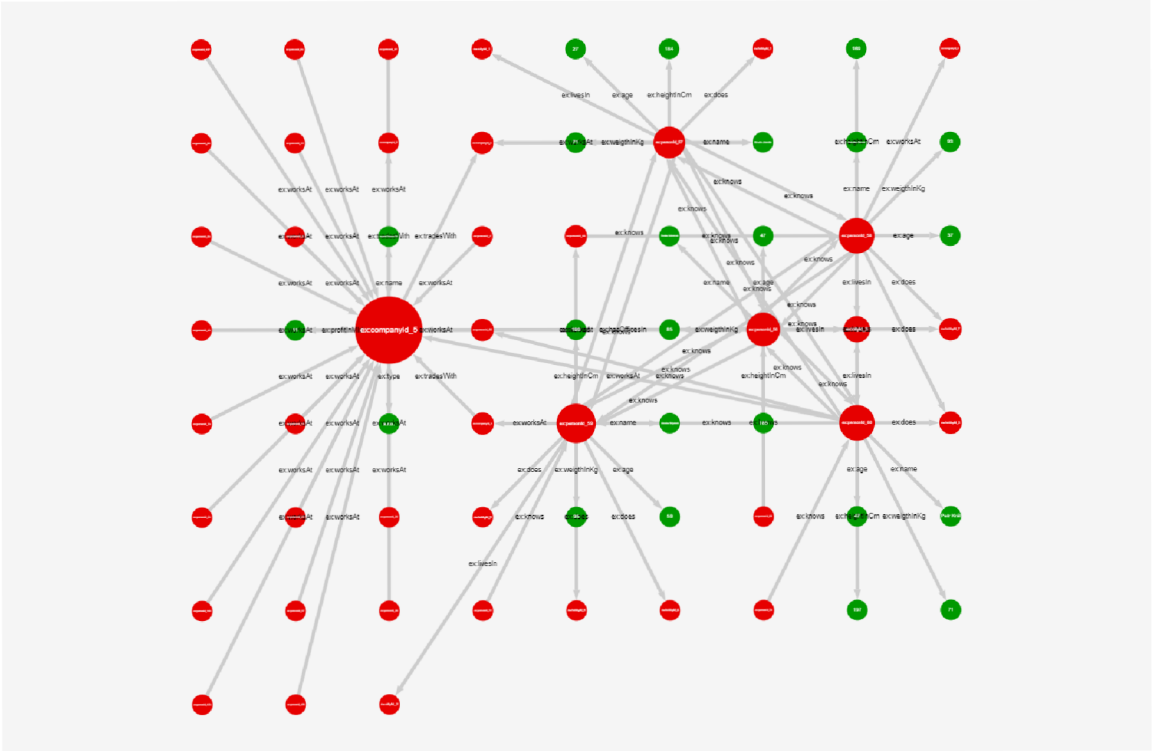
Nyní bude následovat zhodnocení přehlednosti. To bude provedeno na podčásti datasetu *AllFeatures*. Výsledky jednotlivých metod jsou ukázány na následujících obrázcích 7.12, 7.13 a 7.14.

Z nich je patrné, že metody greedy a greedy-swap v ohledu přehlednosti čelí stejným problémům, které jsou způsobeny použitím mříže. Protože jsou uzly rovnoměrně rozprostřeny po ploše, je obtížné se při prohlížení zaměřit na konkrétní část. Hlavní problém ale způsobují překryvy propojení a popisů. Často nastává situace, kdy propojení leží na stejné přímce a splývají do sebe. Pozice jejich názvů pak způsobují problémy kvůli tomu, že jsou umístěny v jejich středu, takže vycházejí přesně do míst, kde se nacházejí uzly. Greedy-swap je ale celkově o trochu přehlednější díky kratším propojením. Poslední force-directed žádné předchozí problémy nemá a opět převyšuje oba předchozí algoritmy svým výsledkem. Také sice obsahuje několik nešťastných překryvů, jinak je ale výsledná vizualizace vynikající.

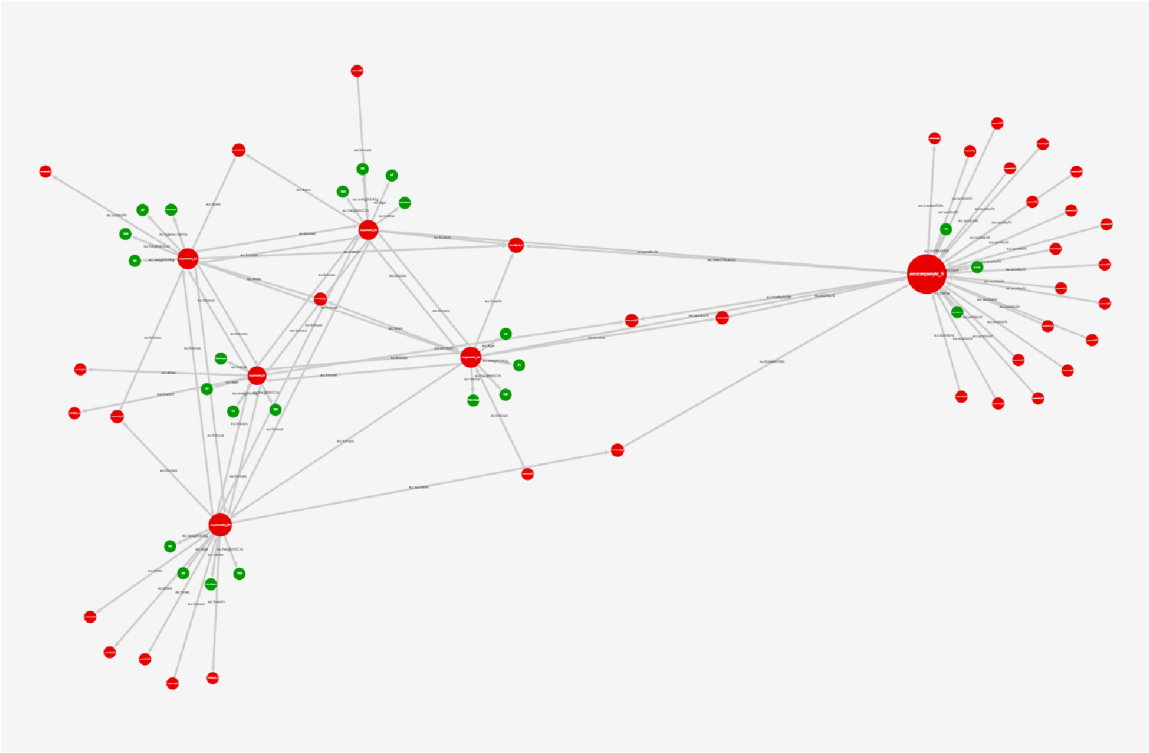
Pokud tedy jde o obecnou kvalitu vizualizace, je pořadí jasné: Nejlepší je force-directed, poté greedy-swap a nakonec greedy. Nyní tedy zbývá otestovat výkon jednotlivých metod a zjistit, zda metody poskytují tyto výsledky v rozumném čase.



Obrázek 7.12: Vizualizce datasetu *AllFeatures* algoritmem greedy.



Obrázek 7.13: Vizualizce datasetu *AllFeatures* algoritmem greedy-swap.



Obrázek 7.14: Vizualizce datasetu *AllFeatures* algoritmem force-directed (2000 iterací).

7.3 Testování výkonu

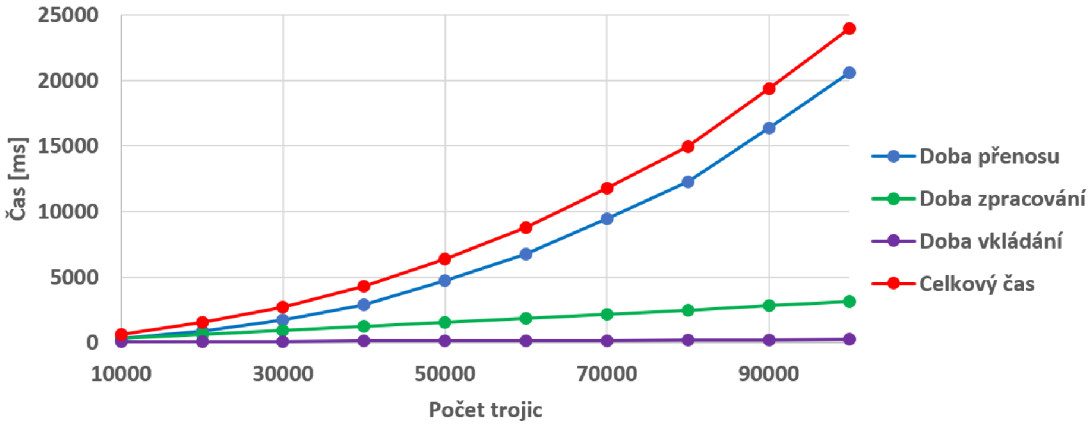
Výkon všech aspektů aplikace byl testován v prohlížeči Opera 68.0. Byl zde využit pouze jediný dataset *PerformanceSet_200k*, který byl uložen na lokálním serveru v uložišti typu memory store. Aplikační limit načtených trojic poté simuloval datasety s nižšími počty trojic. Testování bylo primárně zaměřeno za časovou složitost, která byla měřena pomocí funkce *performace.now()* dostupné v jazyce JavaScript. Paměťová složitost byla také měřena, v žádném případě však nebylo využito více než 200 MB paměti RAM, což je velmi malá hodnota pro dnešní počítače. Dále tedy nebude podrobněji analyzována.

Prvním testovaným aspektem byl čas potřebný k načtení dat. To zahrnuje přenos daných trojic ze serveru na klienta, jejich zpracování a následné vložení do grafu.

Z měření (tabulka 7.2 a graf 7.15) je vidět, že nejdelší dobu zabírá přenos trojic ze serveru do klienta, jehož hodnoty se pohybují v řádech sekund, a to již při počtech trojic v řádu desetitisíců. Překvapivě se tedy také jedná o limitující faktor při snaze vizualizovat rozsáhlé datasety. Přenosová rychlost by neměla představovat problém, je ale možné, že tento čas navyšuje server samotný, který běží lokálně a nemusí tedy mít dostatek výkonu pro rychlejší provedení požadavků. Doba potřebná ke zpracování již přijatých dat poté zabírá jen zlomek celkového času, není ale zanedbatelná. To může být způsobeno použitím regulárních výrazů – ty byly použity kvůli jejich rozsáhlým možnostem při práci s řetězci, nemusí být ale nejrychlejší. Nejkratší čas poté zabírá vložení trojic do lokální grafové struktury. To dosahuje perfektních výsledků a potvrzuje fakt, že vložení jedné trojice do grafu by mělo mít konstantní časovou složitost. Pro určení limitu vizualizace je poté nutné určit čas, který je uživatel ochoten počkat před načtením dat – pokud se předpokládá, že se pohybuje okolo desíti sekund, tak bude výsledný limit této časti vizualizace stanoven na 60000 trojic.

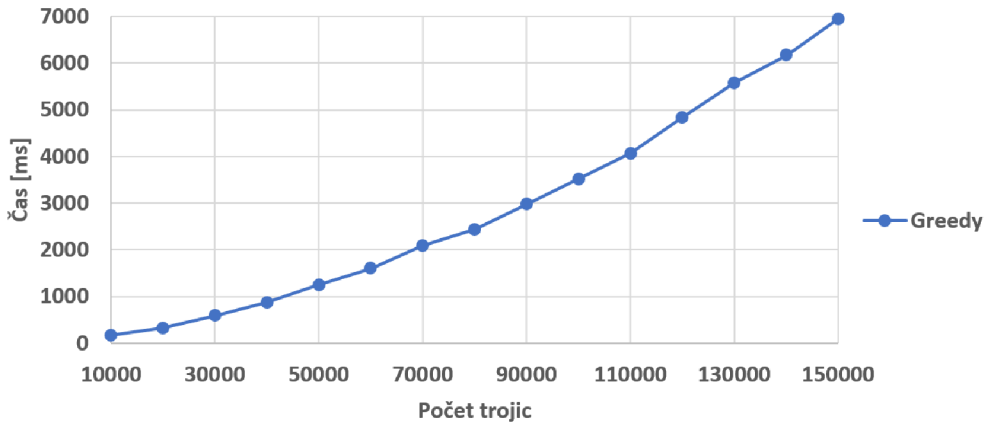
Tabulka 7.2: Časy potřebné k načtení dat ze serveru do klienta v milisekundách.

Počet trojic	Doba přenosu	Doba zpracování	Doba vkládání	Celkový čas
10000	293,2	312,6	45,1	650,9
20000	874,1	623,8	69,8	1567,7
30000	1701,1	925,3	93,8	2720,2
40000	2922,3	1241,6	125,4	4289,3
50000	4721,8	1550,0	134,0	6405,8
60000	6761,7	1861,3	138,1	8761,1
70000	9431,6	2177,7	163,2	11772,5
80000	12258,3	2490,5	190,3	14939,1
90000	16351,0	2806,5	215,1	19372,6
100000	20584,6	3125,9	234,5	23945,0



Obrázek 7.15: Graf ukazující časy potřebné k načtení dat ze serveru do klienta.

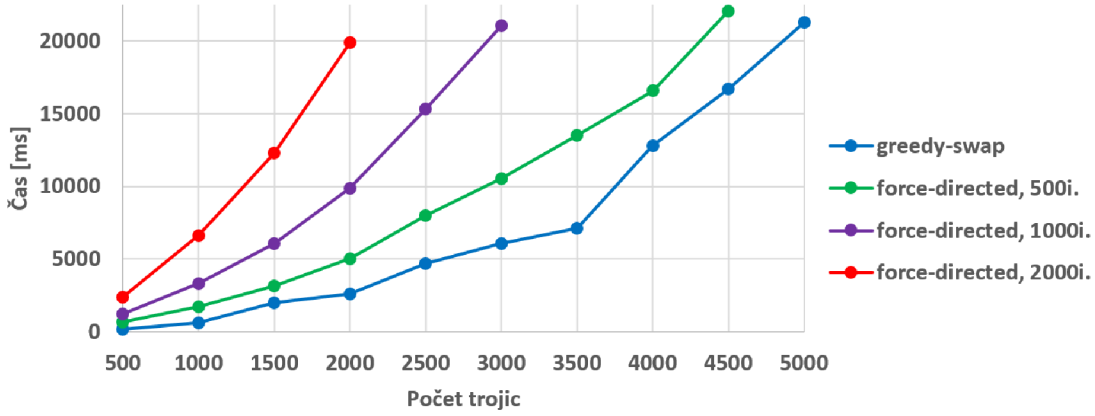
Dalším aspektem, který byl testován je doba potřebná pro provedení jednotlivých algoritmů.



Obrázek 7.16: Graf ukazující časy potřebné k provedení metody greedy.

Tabulka 7.3: Časy potřebné k provedení metody greedy v milisekundách.

Počet trojic	greedy
10000	168,7
20000	321,9
30000	588,5
40000	871,4
50000	1246,3
60000	1600,9
70000	2094,4
80000	2434,2
90000	2973,9
100000	3509,3
110000	4068,4
120000	4840,4
130000	5581,5
140000	6168,0
150000	6941,8



Obrázek 7.17: Graf ukazující časy potřebné k provedení metod greedy-swap a force-directed.

Nejrychlejší je podle očekávání algoritmus greedy (tabulka 7.3 a graf 7.16). Překvapil však mírou velikosti rozdílu – bez problému zvládl umístit 150000 uzlů do 7s a byl tak rychlejší něž samotné načtení dat. Greedy-swap a force-directed poté dosahují řádově horších výsledků (tabulka 7.4 a graf 7.17). Čas provedení obou metod byl podobný, greedy-swap byla ale výsledku o trochu rychlejší. U metody force-directed také samozřejmě záleží na počtu iterací, experimentálně však bylo zjištěno, že jich je potřeba alespoň 500 pro obstojný výsledek. Od toho čísla se tedy odrazilo testování. Z implementace poté vyplývá, že čas potřebný pro výpočet je přímo úměrný jejich počtu.

Suverénně nejlepší v této kategorii byl tedy algoritmus greedy. Pokud je opět vzato v úvahu, že doba čekání uživatele by neměla přesáhnout 10s, tak limit zpracování touto metodou přesahuje 100000 trojic. U zbylých dvou metod se pohybuje v řádech tisíců.

Další aspekt, který byl testován je čas potřebný provedení všech částí programové smyčky. Do těch spadá zpracování vstupů, provedení logiky a samotné vykreslení grafu

Tabulka 7.4: Časy potřebné k provedení metod greedy-swap a force-directed v milisekundách.

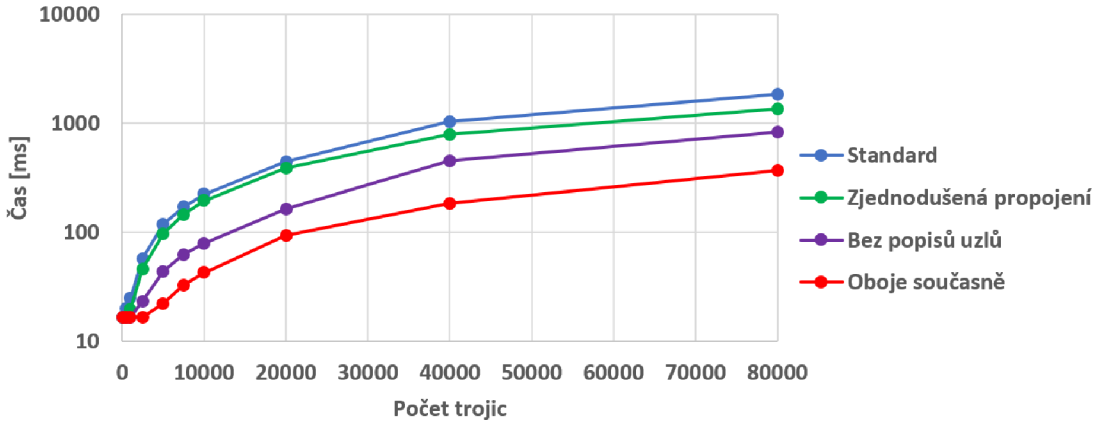
Počet trojic	greedy-swap	force-directed, 500 iterací	force-directed, 1000 iterací	force-directed, 2000 iterací
500	172,4	639,2	1208,5	2357,6
1000	583,6	1677,6	3291,5	6611,0
1500	1997,9	3132,8	6036,9	12270,7
2000	2558,7	4995,8	9876,2	19907,3
2500	4686,5	7993,1	15309,1	
3000	6059,9	10509,0	21045,9	
3500	7126,1	13502,9		
4000	12805,7	16569,1		
4500	16698,5	22036,3		
5000	21265,5			

do prohlížeče. Protože se aplikace snaží běžet na 60 fps, mělo by se vše stihnout provést do 1/60 s (16,67 ms). V tom případě poběží aplikace perfektně plynule. Hranice 30 fps (33,3 ms) je přijatelná, pokud se však aplikace dostane pod 15 fps (66,6 ms), tak práce s ní začíná být velice obtížná. Také je důležité zmínit fakt, že díky principu frustrum culling prvku canvas se podle míry přiblížení může obnovovací frekvence zvýšit až několikanásobně. Jak již bylo řečeno u implementace, aplikace také umožňuje vypnout popisy uzlů a zjednodušit vizualizaci propojení, což povede ke zvýšení výkonu. Všechny varianty tedy budou otestovány.

Tabulka 7.5: Časy potřebné k provedení všech částí programové smyčky v milisekundách.

Počet trojic	Standard	Zjednodušená propojení	Bez popisů uzlů	Oboje současně
100	16,67	16,67	16,67	16,66
500	20,00	16,67	16,68	16,67
1000	24,65	19,64	16,67	16,67
2500	56,72	45,99	23,12	16,67
5000	118,51	96,42	43,47	22,07
7500	170,07	145,83	61,82	32,49
10000	223,26	195,18	78,94	42,43
20000	445,78	388,26	163,87	93,63
40000	1029,45	792,47	452,67	184,47
80000	1836,63	1343,75	829,41	367,12

Jak ukazuje tabulka 7.5 a graf 7.18, tak v případě, kdy jsou zapnuty všechny prvky vizualizace je dosáhnuto limitu 15 fps již při 2500 trojicích, což je o dost méně, než u načtení a zpracování. Vypnutím některých částí vizualizace se poté dá dosáhnout až na limit 20000 trojic. Jedná o výrazné zlepšení, bylo ale vyměněno za nižší přehlednost a stále není dostačující. Je tedy zřejmé, že samotné vykreslení je nejvíce limitující faktor a pokud bude nutné vizualizovat maximální počet trojic, který dokáže zpracovat algoritmus greedy, tak bude nevyhnutelné, že aplikace poběží na nízké obnovovací frekvenci.

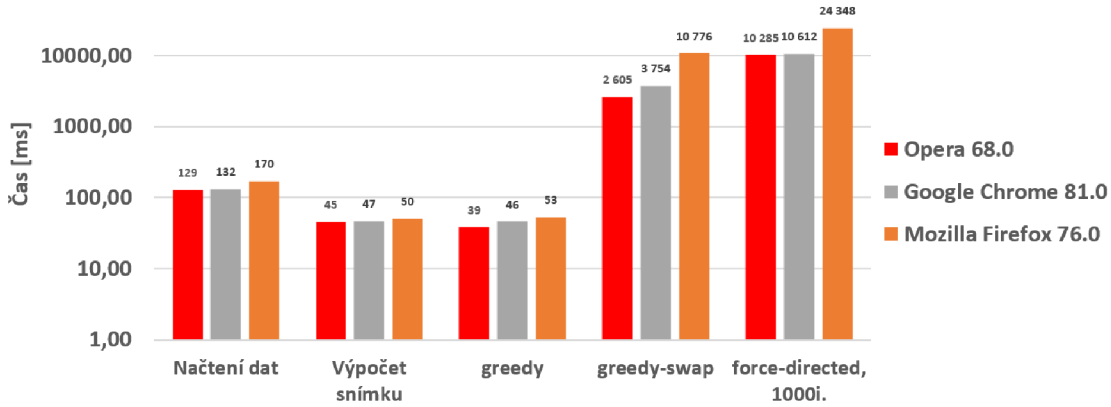


Obrázek 7.18: Graf ukazující časy potřebné k provedení všech částí programové smyčky.

Posledním aspektem, který byl testován je porovnání výkonu v různých prohlížečích. Ty byly následující: již zmíněná Opera 68.0, Google Chrome 81.0 a Mozilla Firefox 76.0. Microsoft Edge do testu zahrnut nebyl, protože nepodporuje všechny funkce použité v tomto projektu. Bylo provedeno srovnání doby načtení, vykreslení i provedení jednotlivých algoritmů. Testování proběhlo s dvěma tisíci načtenými trojicemi.

Tabulka 7.6: Časy potřebné k provedení několika částí aplikace v různých prohlížečích v milisekundách.

Webový prohlížeč	Načtení dat	Výpočet snímku	greedy	greedy-swap	force-directed, 1000 iterací
Opera 68.0	129,08	45,43	38,6	2605,3	10285,3
Google Chrome 81.0	132,27	46,59	46,1	3754,3	10611,6
Mozilla Firefox 76.0	169,70	50,20	53,0	10776,0	24348,0



Obrázek 7.19: Graf ukazující časy potřebné k provedení několika částí aplikace v různých prohlížečích.

V tomto ohledu byla nejlepší Opera těsně následovaná prohlížečem Google Chrome (tabulka 7.6 a graf 7.19). To je dáno tím, že používají stejné jádro, Opera ale disponuje odlehčenější implementací. Na třetím místě se poté se znatelným rozdílem umístil prohlížeč Mozilla Firefox 76.0. Z výkonnostního hlediska je tedy nejlepší použít pro běh aplikace prohlížeč Opera.

7.4 Shrnutí

Primárním cílem této práce byla vizualizace grafových databází a následné nalezení limitu trojic, které jsou webové prohlížeče schopny vizualizovat. Po otestování a zhodnocení jednotlivých částí bude nyní stanoven limit pro aplikaci jako celek, ten se ale bude měnit v závislosti na požadavcích uživatele.

Nejprve bude stanoven limit pro případ, kdy je cílem vizualizovat absolutní maximum trojic za jakoukoliv cenu. Ten se pohybuje okolo počtu 150000 trojic, výsledná kvalita však bude velice nízká. Na výsledek se bude čekat půl minuty a musí být použita metoda greedy, která dopadla z hlediska vzhledu nejhůře. Také musí být zapnuty jednoduchá propojení, vypnuty popisy a i poté se bude obnovovací frekvence aplikace pohybovat okolo 2fps. Pokud bude vyžadována alespoň nízká plynulost aplikace (větší než 15 fps) nutná pro zajištění interaktivity, tak limit rapidně spadne na 10000 trojic ve zjednodušeném režimu. Při nutnosti vykreslení všech informací je dále snížen na 2500 trojic. I v těchto případech je stále nutné použít algoritmus greedy.

Pokud je cílem uživatele vizualizovat data v nejlepší kvalitě a při plynulém běhu aplikace (více než 30 fps), je možné použít algoritmus greedy-swap nebo force-directed. Oba poskytují lepší vizualizaci než algoritmus greedy, pokud se však zohlední výkon i vzhled současně, dopadá force-directed lépe. Když bude poté zvolena střední možnost s tisíci iteracemi, tak kvalita výsledku značně převýší druhou metodu a vynahradí tak i čtyřnásobný požadavek na výkon. Při použití této varianty se limit pohybuje okolo 2000 trojic.

Poslední případ je poté ten, že uživatel chce použít aplikaci pouze k jednoduchému prohlížení grafových dat a nezabývat se přitom výkonem. Pro tento účel je doporučena metoda force-directed s 2000 iteracemi a limit je pak stanoven na 1000 trojic. Při tomto objemu dat poběží aplikace plynule nehledě na zvolené parametry.

Kapitola 8

Závěr

Cílem této práce bylo vizualizovat grafové databáze a přitom zjistit, jak velký objem grafových dat zvládnou webové prohlížeče při aktuálním výkonu vizualizovat. K tomuto účelu byl vytvořen generátor grafových dat a interaktivní webová aplikace, která data vykresluje. Datasety byly nejen vygenerovány ale i staženy z externích zdrojů. Ty jsou uloženy na databázovém serveru, se kterým klient komunikuje pomocí REST API. Samotná vizualizace je poté implementována pomocí tří rozdílných metod nazvaných greedy, greedy-swap a force-directed, které byly speciálně navrženy pro tento projekt. Ty se liší výkonem i kvalitou výsledku. Vykreslení do webové stránky bylo poté realizováno pomocí prvku canvas. Testování a vyhodnocení bylo primárně zaměřeno na časovou náročnost všech aspektů aplikace, jednotlivé vizualizační metody však byly vyhodnoceny i z hlediska vzhledu a přehlednosti jejich výsledků.

Nejrychlejší byla metoda greedy, poskytovala však nejhorší vizualizaci. Jedná se tedy o metodu, pomocí které se dá vykreslit nejvíce trojic najednou – limit se pohybuje okolo 150000 trojic. Takový počet ale prohlížeče nestíhají dostatečně rychle vykreslit a pokud je tedy nutné, aby aplikace běžela plynule, tak počet trojic nesmí přesáhnou 20000. Jestliže je prioritou maximální kvalita vykreslení, tak by měla být použita metoda force-directed. Ta dosahuje v tomto ohledu nejlepších výsledků, limit objemu dat se poté pohybuje okolo 2000 trojic. Poslední metoda greedy-swap se umístila mezi oběma předchozími, její výsledný vzhled ale nevynahradí výkon potřebný pro výpočet.

Všechny stanovené cíle tedy byly splněny. Aplikaci by šlo dále rozšířit o další vizualizační algoritmy, které by mohli dosáhnout lepších výsledků, samotné vykreslení pomocí standardních metod prvku canvas by však bylo možné zefektivnit jen obtížně. K většímu navýšení stanovených limitů by se tedy musela změnit technologie nebo navýšit výkon hardwaru.

Literatura

[1] *Apache Tomcat resources*. Dostupné z: https://www.mulesoft.com/tcat/understanding-apache-tomcat.

[2] *What is SPARQL - Semantic Search Query Language - Ontotext*. 2018. Dostupné z: https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/.

[3] *Canvas API*. Prosinec 2019. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[4] *HTML Canvas Reference*. 2019. Dostupné z: https://www.w3schools.com/tags/ref_canvas.asp.

[5] *JavaScript HTML DOM*. 2019. Dostupné z: https://www.w3schools.com/js/js_htmldom.asp.

[6] *What Is a Graph Database and Property Graph | Neo4j*. Říjen 2019. Dostupné z: https://neo4j.com/developer/graph-database/.

[7] *What is REST? | Codecademy*. 2019. Dostupné z: https://www.codecademy.com/articles/what-is-rest.

[8] Barrasa, J. *RDF Triple Stores vs. Labeled Property Graphs: What's the Difference? - DZone Big Data*. Leden 2018. Dostupné z: https://dzone.com/articles/rdf-triple-stores-vs-labeled-property-graphs-whats.

[9] Bouška, P. *Základy jazyka HTML < články -> SAMURAJ-cz.com*. 2020. Dostupné z: https://www.samuraj-cz.com/clanek/zaklady-jazyka-html/.

[10] Chao, J. *Graph Databases for Beginners: Native vs. Non-Native Graph Technology*. Prosinec 2018. Dostupné z: https://neo4j.com/blog/native-vs-non-native-graph-technology/.

[11] Guindon, C. *The RDF4J REST API | The Eclipse Foundation*. Dostupné z: https://rdf4j.org/documentation/reference/rest-api/.

[12] Guindon, C. *RDF4J Server and Workbench | The Eclipse Foundation*. Dostupné z: https://rdf4j.org/documentation/tools/server-workbench/.

[13] Guindon, C. *The Repository API | The Eclipse Foundation*. Dostupné z: https://rdf4j.org/documentation/programming/repository/.

[14] Harris, S. a Seaborne, A. *SPARQL 1.1 Query Language*. Březen 2013. Dostupné z: www.w3.org/TR/sparql11-query/.

[15] Manola, F. a Miller, E. *RDF Primer*. 2014. Dostupné z: https://www.w3.org/TR/rdf-primer/.

[16] McCreary, D. *How to Explain Index-free Adjacency to Your Manager*. Medium, březen 2018. Dostupné z: https://medium.com/@dmccreary/how-to-explain-index-free-adjacency-to-your-manager-1a8e68ec664a.

[17] Poudel, S. *Consuming REST APIs with Python*. Faun, říjen 2019. Dostupné z: https://medium.com/faun/consuming-rest-apis-with-python-eb86c6b724c5.

[18] Raimond, Y. a Schreiber, G. *RDF 1.1 Primer*. červen 2014. Dostupné z: https://www.w3.org/TR/rdf11-primer/.

[19] Sasaki, B. M. *Graph Databases for Beginners: Why Graph Technology Is the Future*. Červenec 2018. Dostupné z: https://neo4j.com/blog/why-graph-databases-are-the-future/.

[20] Vilhena, T. *Index-free adjacency*. 2019. Dostupné z: https://thomasvilhena.com/2019/08/index-free-adjacency.

[21] Wikipedia. *Graph database*. Wikimedia Foundation, listopad 2019. Dostupné z: https://en.wikipedia.org/wiki/Graph_database.

Příloha A

Obsah přiloženého CD

- /Code/ – složka obsahující zdrojové soubory klienta a generátoru
- /Datasets/ – složka obsahující externí a vygenerované datasety
- /LaTeX/ – složka obsahující zdrojové soubory technické zprávy
- /documentation.pdf – technická zpráva
- /readme.txt – soubor obsahující návod ke spuštění řešení